First Edition - March 1985

This manual contains sample VAXELN programs for use and reference in designing VAXELN applications.

VAXELN
Application Design Guide

Document Order Number: AA-EU41A-TE
Software Version: 2.0

digital equipment corporation
maynard, massachusetts

First Edition, March 1985

The postage-paid READER'S COMMENTS form on the last page of this document requests your critical evaluation to assist us in preparing future documentation.

Contents

Application 9: Multiple Circuit Server

Application 10: Self-Defining Data Structures

Application 11: VAXELN Interface to VAX/VMS

Application 12: VAXELN Time Routines

Preface

The *VAXELN Application Design Guide* provides sample programs for your reference in designing applications using the VAXELN toolkit.

Manual Objectives

This manual contains solutions to several programming problems you may have. Each section's example program can be used as written to solve your problem, or it can be used merely as a guide in designing your own application.

Intended Audience

This manual is designed for programmers and students who have a working knowledge of Pascal or the C programming language. Knowledge of the fundamental principles of the VAX/VMS operating system, as well as knowledge of VAXELN, is required.

Structure of this Document

This manual consists of 13 sections. The first section provides an overview of the considerations you face when designing your VAXELN applications. The next 12 sections each consist of a simple statement of a problem, a description of the program that solves that problem, and an example program.

Associated Documents

The following documents are relevant to designing VAXELN applications:

- *VAXELN Installation Manual (AA-EU37A-TE)*
- *VAXELN V2.0 Release Notes (AA-Z454C-TE)*
- *VAXELN User's Guide (AA-EU38A-TE)*
- *VAXELN Pascal Language Reference Manual (AA-EU39A-TE)*
- *VAXELN C Run-Time Library Reference Manual (AA-EU40A-TE)*

Overview

Structuring VAXELN Applications

When designing VAXELN applications, you must first decide how the application will be structured; there are four ways:

- As a single job with a single process
- As a single job with multiple processes
- As multiple jobs, each with a single process
- As multiple jobs with multiple processes

For simple applications not requiring concurrency within the application, a single job with a single process is best because the application can be broken into small functional units, each a callable procedure. For very complex applications, multiple jobs with one or more processes per job may be needed.

In many cases, the efficiency of communicating between concurrently executing parts of the application is the determining factor in the overall performance of the application. For most applications, this concern with efficiency leads to a choice between two configurations: single-job/multi-process, and multi-job/single-process.

Multiple Jobs

Multiple jobs have these advantages:

- The application can be distributed over several VAXELN nodes in a network. This distribution of jobs is transparent to the user.

- Each job has its own address space. Therefore, bugs that occur in one part of the application will not propagate to other parts of the application.
- Since each job is a separate functional entity, and communication between jobs is more formal than between processes, it may be easier to distribute the design and implementation of the application among several members of a programming team.

Multiple jobs have these disadvantages:

- Each job consumes more system resources than would a separate process within a single job.
- Synchronization and data passing between jobs can affect performance.

Communication between jobs can be accomplished by using either areas or messages. Areas are the most efficient method of communication. However, areas may only be used when all jobs using the area are running on the same node. This removes the advantage of the application being distributable over several nodes in a network.

Message passing may be used to communicate between jobs even in a distributed network. However, the overhead associated with message passing may be prohibitive, depending on the application.

For an example of the multi-job/single-process method, see Application 6, "Interjob Communication."

Single Job

A single job with multiple processes has these advantages:

- Memory sharing makes communication and synchronization between processes fast and easy; heap and static memory are shared by all processes within the job; interprocess communication using simple job-wide structures, such as queues and data structures synchronized by mutexes, provides better overall performance.
- Individual processes consume very few system resources.
- Creating a new process is significantly faster than creating a new job.

A single job with multiple processes has these disadvantages:

- Since the entire application is contained in one job, the application cannot be distributed in a network.
- Since heap and static memory are shared by all processes, corruption of the heap or static memory affects all processes. Only stacks are protected among processes.
- Due to the availability of data sharing between processes, it may be more difficult to ensure "clean" interfaces to procedures, especially for an application being written by a team of programmers.

For an example of the single-job/multi-process method, see Application 1, "Asynchronous I/O."

Designing Communication Protocols

If, after planning the partitioning of your application, you've decided to use message passing for interjob communication, you must choose whether to use datagrams or circuits. You must also design both the format of these messages and the communication protocol.

Whether to use datagrams or circuits is usually an easy decision: for most applications you should use circuits; datagrams should only be used for single-message transactions. Circuits are best for continuous connections because circuits are much more reliable than datagrams.

Having chosen whether to use datagrams or circuits, you must now design a communication protocol; the following paragraphs offer guidelines.

When using datagrams:

- An application-level acknowledgment and timeout should be used to detect lost messages.
- A sequence number should be contained in each message to ensure that retransmissions do not result in duplicate requests, and that acknowledgments can be properly paired with requests.

When using circuits:

- An application-level acknowledgment should only be used when a request MUST be confirmed; "ping-pong" protocols should always be avoided, particularly because the virtual circuit already acknowledges each message when necessary.
- Small messages should be packed into larger messages whenever possible. The overhead for each message is almost always the limit to throughput, and virtual circuit protocols have

access to information to perform the most efficient segmentation and reassembly.

- An application-level acknowledgment should always be used when terminating a connection to ensure that the receiver completed the request. The virtual circuit protocol only makes a best-effort attempt to deliver all the messages; if it could not deliver them, the application would never know. Alternatively, the sender of the last message can wait on the port for the receiver to disconnect. This also ensures that the final message was actually received before the circuit was disconnected.
- After circuit connection, the applications should exchange version number and configuration messages; this allows applications and protocols to be upgraded over time and to provide subset and superset functionality.

Application 1
Asynchronous I/O

Problem

How do you program asynchronous behavior, such as asynchronous I/O, while computation is occurring?

Solution

VAXELN does not have the concept of ASTs (asynchronous system traps) as VAX/VMS does, but the concept of concurrently executing processes in VAXELN can be used to create the features of ASTs. In fact, the VAXELN mechanism is more flexible since multiple processes can function as prioritized ASTs.

The example in this section shows how to use multiple VAXELN processes to perform asynchronous operations. In the example (a simple checksum operation) one process is reading data from a file, and the other process is performing a calculation on the data.

The master process starts the sequence by opening the data file and setting up the synchronization objects that will be used to protect access to the data buffers. Then the master process creates the subprocess that will read the data from the file into the buffers. The subprocess simply reads the file using a typical double buffer method. As the data is available in a buffer, the master process computes the checksum. When the file is completely read, the checksum is displayed.

The buffers are synchronized by using two mutexes per buffer. One mutex indicates that the buffer is full of data and the other indicates that the buffer is empty. The reader process uses a transition of the empty mutex to indicate that the computational process is finished with the checksum calculation. When data is read into the buffer, the reader process sets the full mutex to indicate that the buffer is ready to process.

To build the sample application, use the following commands:

```
$ epascal application1 + eln$:rtlobject/lib
$ link application1 + eln$:rtlshare/lib + rtl/lib
$ ebuild /noedit application1
```

The sample application can then be loaded into a target machine and executed. The data file must contain information for EBUILD, as follows:

```
characteristic /nofile
program application1
```

Example

The following is a listing of the example written in Pascal (application1.pas).

```
module asynchronous_io_example;

{++
{
{ Abstract:
{
{       This is a simple program to show how asynchronous
{       activity is performed using the VAXELN multitasking
{       facilities.
{
{       The master process creates a subprocess to perform
{       the I/O operations.  As each buffer is filled,
{       the master process computes a simple checksum on
{       the data.  When all the data is read, the checksum
{       is displayed.  The subprocess asynchronously reads
{       data from a file using a straightforward double
{       buffering scheme that is synchronized with the
{       master process by using EVENT objects.
{
{--}

include
        $mutex;

{
{       Job-wide declarations.
{}

{
{       Define a record that contains both data and the
{       mutex to protect that data from multiple access.
{}

type
        file_record = packed array[1..512] of char;

        data_record = record;
            full: mutex;
            empty: mutex;
            last_block: boolean;
            data: file_record
```

```
            end;

{
{       Declare a "double buffer" of data_records.
{}

const
        first = 0;
        second = 1;

var
        data_blk: array[first..second] of data_record;

{
{       Declare the input file.
{}

var
        data_file: file of file_record;




[inline] function other(index: integer): integer;

{++
{
{ Functional description:
{
{       This is an inline routine to "flip" the
{       buffer index to the other buffer index.
{
{ Inputs:
{
{       index - Buffer index.
{
{ Outputs:
{
{       Index of the other buffer.
{
{--}

begin

if index = first
then
        other := second
```

```
else
        other := first
end;


program asynchronous_io(output);

{++
{
{ Functional description:
{
{       This is the master process that creates a
{       subprocess to asynchronously read the data blocks.
{       As the data blocks are read, a checksum is computed.
{       When all the data is processed, the checksum is
{       displayed.
{
{ Inputs:
{
{       A data file.
{
{ Outputs:
{
{       The simple checksum is displayed.
{
{--}

{
{       Master-process-local variable declarations.
{}

var
        reader_process: process;
        checksum: integer;
        i, j, k: integer;
        id: integer;
        status: integer;
        checksum_done: boolean;

begin

{
{       Open the data file.  If the open fails, exit using
{       the failure status as the job exit status.
{}

open(data_file,
     file_name := '10.172::gathered.dat',
     history := history$old,
     status := status);
```

```
if not odd(status)
then
        exit(exit_status := status);

reset(data_file);

{
{       Initialize both data_blk structures.
{       Set mutexs to indicate that both buffers are empty.
{}

create_mutex(data_blk[first].full);
lock_mutex(data_blk[first].full);
create_mutex(data_blk[first].empty);
data_blk[1].last_block := false;

create_mutex(data_blk[second].full);
lock_mutex(data_blk[second].full);
create_mutex(data_blk[second].empty);
data_blk[second].last_block := false;

{
{       Create the subprocess to read the file.
{}

create_process(reader_process,
               reader_process_code,
               status := status);

{
{       Initialize the variables used during
{       the checksum computation.
{}

checksum := 0;
id := first;
checksum_done := false;

{
{       Checksum computation loop:
{       Pass over each buffer in turn, locking it
{       while the data is being processed.
{}

repeat
        lock_mutex(data_blk[id].full);
        if not data_blk[id].last_block
        then
```

```
                for i := 1 to 512 do
                        checksum := checksum +
                                    ord(data_blk[id].data[i])
        else
                checksum_done := true;
        unlock_mutex(data_blk[id].empty);
        id := other(id)

        until checksum_done;

{
{       Close file and display the computed checksum.
{}

close(data_file);
writeln('Data file checksum is: ', checksum)
end;




process_block reader_process_code;

{++
{
{ Functional description:
{
{       This process reads the data file using a
{       double buffer scheme.  The buffers are "locked,"
{       filled with data, and unlocked.  This locking
{       protocol will synchronize this process with the master
{       process, which is computing the checksum.
{
{       A boolean is set in the buffer to indicate
{       end-of-file.
{
{ Inputs:
{
{       Data_file is open.
{       The first buffer's lock is set.
{
{ Outputs:
{
{       <No direct outputs.>
{--}
```

```
var
        id: integer;

begin

{
{       Initialize local variables.
{}

id := first;

{
{       File read loop.
{}

repeat
        lock_mutex(data_blk[id].empty);
        if not eof(data_file)
        then
                read(data_file,data_blk[id].data)
        else
                data_blk[id].last_block := true;
        unlock_mutex(data_blk[id].full);
        id := other(id)

        until data_blk[other(id)].last_block;
end;
end;
```

Application 2
C Device Driver

Problem

How do you write a device driver in C?

Solution

One of the first steps in designing a device driver is deciding what the interface to the driver will be. Three major alternatives exist:

- Providing the driver in the form of callable procedures. Any program wishing to perform I/O to the device links with the driver module and, once running in the VAXELN system, calls the appropriate I/O procedure. The ADV, DRV, DLV, and KWV drivers provided with VAXELN use this method.
- Using your own programs through the DAP message protocol provided with VAXELN. In this case, the driver is its own job with a separate process or processes for each device unit. These unit processes pass addresses of service routines to the DAP server routine, which in turn communicates with the user program through DAP messages. When the DAP server routine receives a request, it calls the appropriate action routine supplied by the driver to perform the actual I/O. The major advantage to this method is that support for Pascal and C I/O is transparent; a user program can use OPEN,

READ, WRITE, and CLOSE in Pascal, or their C equivalents, plus standard I/O routines to access the device. The disk drivers and terminal drivers provided with VAXELN use this method.

- A modified version of the DAP-driver method mentioned above. In this interface, the driver is still its own job, but the interface between the driver and a user program is a direct message-passing scheme where both the driver and the user program require knowledge of the format and content of the messages passed between them. The datalink drivers (QNA and UNA) provided with VAXELN use a method similar to this.

The example in this section uses the third method (described in the immediately preceeding paragraph). In the example, messages passed between the driver and the user program contain:

An operation type (such as read or write)
An error code
The length of the data to be read or written
A data buffer

Only three operations are supported:

READ BLOCK (read a fixed number of characters from the device)
WRITE BLOCK (write a fixed number of characters to the device)
DONE (indicating the user program has completed its I/O to or from the device)

Because drivers are usually long and complex, many simplifications were made to this example driver to make it as small as possible. These simplifications, and possible enhancements you can make to the example driver, are described below.

- The example driver does not support power-fail recovery. The *VAXELN C Run-Time Library Reference Manual* describes the basic theory behind writing a power recovery routine and provides an example that can be adapted to the example driver. Since the interrupt service routines (ISRs) are performing all of the I/O, the power recovery routines (one each for the receive and transmit devices) should only reinitialize the device and continue any I/O that may have been in progress when the power fail occurred.

- Most of the kernel procedure calls in the example driver pass NULL as the status argument; should an error occur, an exception would be raised. Normally, drivers either provide exception handling routines or request status for all kernel procedures.

- The DLV device returns more information on read errors than is passed back to the user by the driver. Additional error codes could be defined to indicate the reason for the read failure.

- The example driver supports only one DLV line. Some DLV devices provide multiple serial line support but, for simplicity, the example driver supports just one. Adding support for a multi-line DLVJ1 is a fairly simple enhancement; a separate process is created for each line. Each process connects to a port whose name uniquely identifies both the device and the line. The process then services I/O requests in the same manner as the example driver does.

- There is no support for flow control; the driver transmits characters as fast as the DLV interface can take them. If the device connected to the DLV is slower than the interface, some provision would have to be made for controlling the flow (such as XOFF/XON support).

The device used in this example driver is a DLV11-A single-line serial interface that connects a Q-bus based computer with a serial device, such as a terminal.

To include the example driver in a VAXELN system, the lines below must be in the EBUILD data file:

```
program application2 /initialize /kernel_stack=8 -
  /mode=kernel /job_priority=5 /argument=("DLVA")
device DLVA /register=%o776500 /vector=%o300 /noautoload
```

Example

The following is a listing of the example written in C (application2.c).

```
#module dlv_driver

/*
 *      This is a sample DLV device driver written in C.  This
 *      driver does not support UNIX or stdio-style I/O; rather,
 *      it provides a message-based form of I/O requests.  Since
 *      C I/O is not supported, the normal C run-time library
 *      interpretation of program arguments is not used.
 *      Instead, the program assumes the first program argument
 *      is the device name.
 *
 *
 *      The interface to the driver behaves as follows:
 *
 *      •  The program wishing to perform I/O to and/or from the
 *         DLV device makes a circuit with the DLV$DRIVER_PORT
 *         port.
 *
 *      •  Over this circuit, the program sends requests to
 *         do reads and writes to/from the DLV.  The driver
 *         services the request and sends back an appropriate
 *         response.
 *
 *      •  The program sends a special "I'm done" message when
 *         it has completed its I/O.
 *
 *      The messages passed between the driver and the user
 *      program contain a request type (read_block, write_block,
 *      or done), a place for an error code (set by the driver),
 *      the number of bytes to be read or written, and a buffer
 *      into which the data will be read, or from which the data
 *      will be written.  See the structure definition below for
 *      the exact message format.  Note that there is a maximum
 *      size for data.
 */

#include $vaxelnc
#include descrip
```

```
/*
 *      Define the size of the receive and transmit buffers
 *      in the communications region.
 */

#define RBUFFER_LENGTH  512
#define XBUFFER_LENGTH  512

/*
 *      Define the supported function codes (used in the
 *      operation field of the dlv_packet message structure).
 */

#define DONE_FUNCTION           0
#define READ_BLOCK_FUNCTION     1
#define WRITE_BLOCK_FUNCTION    2

/*
 *      Define the bit locations and mask used in the
 *      transmit and receive CSR and buffer registers.
 */

#define RCSR$V_INT_ENA  6
#define RCSR$M_INT_ENA  (1<<RCSR$V_INT_ENA)
#define RBUF$V_CHAR     0
#define RBUF$M_CHAR     (0xFF<<RBUF$V_CHAR)
#define RBUF$V_ERROR    15
#define RBUF$M_ERROR    (1<<RBUF$V_ERROR)
#define XCSR$V_BREAK    0
#define XCSR$M_BREAK    (1<<XCSR$V_BREAK)
#define XCSR$V_INT_ENA  6
#define XCSR$M_INT_ENA  (1<<XCSR$V_INT_ENA)

/*
 *      Define the COPY_BYTES macro.
 *          This macro copies the specified number of
 *          bytes from one string to another without
 *          any character interpretation.
 */

#define COPY_BYTES(src,dst,cnt)             \
            {                               \
            char *s = (src);                \
            char *d = (dst);                \
            int   c;                        \
            for(c=(cnt);c;c--)              \
                  *d++ = *s++;              \
            }
```

```
/*
 *      Define and allocate a pointer to the DLV
 *      device registers.
 */

struct register_def
            {
            unsigned short  rcsr;
            unsigned short  rbuf;
            unsigned short  xcsr;
            unsigned short  xbuf;
            }                        *register_ptr;

/*
 *      Define and allocate a pointer to the receive
 *      and transmit communications regions.
 */

struct rx_region_def
            {
            char    rbuffer[RBUFFER_LENGTH];
            int     read_count;
            int     buf_ptr;
            BOOLEAN read_in_progress;
            BOOLEAN error;
            }                        *rx_region_ptr;

struct tx_region_def
            {
            char    xbuffer[XBUFFER_LENGTH];
            int     write_count;
            int     buf_ptr;
            BOOLEAN write_in_progress;
            }                        *tx_region_ptr;

/*
 *      Define the format of the dlv_packet message.
 */

struct dlv_packet
            {
            int     operation;
            int     error;
            int     length;
            char    buffer[];
            };

/*
 *      Master process:  This function will be that which is
 *      started as the job and, therefore, must come first.
```

```
 *      MAIN_PROGRAM is not used so that the program arguments
 *      are not interpretted by the C run-time library.
 */

dlv_driver()
{

VARYING_STRING(32)      device_name_string;
DEVICE                  dlv_receive_device,dlv_transmit_device;
PORT                    dlv_driver_port;
MESSAGE                 dlv_message;
NAME                    dlv_name;
void                    receive_service_routine(),
                        transmit_service_routine();
struct dlv_packet       *dlv_request;
BOOLEAN                 done;
int                     *adapter,*vector,ipl,status;
int                     request_size;

/*
 *      These macros allocate string descriptors.
 */

static $DESCRIPTOR(dlv_port_name,"DLV$DRIVER_PORT");
static $DESCRIPTOR(device_name,"");

/*******************************************
 *                                         *
 *         Driver Initialization           *
 *                                         *
 *******************************************/

/*
 *      Obtain the device name from the program argument
 *      list and put it into the device_name string
 *      descriptor.
 */

eln$program_argument(&device_name_string, 1);
device_name.dsc$a_pointer = device_name_string.data;
device_name.dsc$w_length  = device_name_string.count;

/*
 *      Create the receive DEVICE object.
 */

ker$create_device(
                &status, &device_name,
                1, receive_service_routine,
                sizeof(struct rx_region_def),
```

```
                &rx_region_ptr, &register_ptr,
                &adapter, &vector, &ipl,
                &dlv_receive_device,
                sizeof(DEVICE),
                NULL);

/*
 *      Create the transmit DEVICE object.
 */

ker$create_device(
                &status, &device_name,
                2, transmit_service_routine,
                sizeof(struct tx_region_def),
                &tx_region_ptr, &register_ptr,
                &adapter, &vector, &ipl,
                &dlv_transmit_device,
                sizeof(DEVICE),
                NULL);

/*
 *      Initialize the device by setting the receiver's
 *      interrupt enable bit.
 */

write_register(RCSR$M_INT_ENA,&register_ptr->rcsr);

/*
 *      Get the driver's job port, create a string
 *      descriptor pointing to the desired name for
 *      the port (DLV$DRIVER_PORT), and create the name.
 */

ker$job_port(NULL, &dlv_driver_port);
ker$create_name(NULL, &dlv_name, &dlv_port_name,
                &dlv_driver_port, NAME$LOCAL);

/*
 *      Initialization complete; inform the kernel.
 */

ker$initialization_done(NULL);

/********************************************
 *                                          *
 *         Driver Normal Operation          *
 *                                          *
 ********************************************/
```

```
/*
 *      Continuously wait for connection requests to
 *      perform I/O to and from the DLV device.
 */

for (;;)
      {

      /*
       *      Accept a circuit connection with another
       *      job that wants to perform I/O with the
       *      DLV device.
       */

      ker$accept_circuit(NULL, &dlv_driver_port, NULL,
                        TRUE, NULL, NULL);

      /*
       *      Service I/O requests until a done
       *      packet is sent.
       */

      for (done = FALSE; !done;)
            {

            /*
             *      Wait on the port until a message
             *      has been sent, then receive it.
             */

            ker$wait_any(NULL, NULL, NULL, &dlv_driver_port);
            ker$receive(NULL, &dlv_message, &dlv_request,
                        &request_size, &dlv_driver_port,
                        NULL, NULL);

            /*
             *      Case on requested operation.
             */

            switch(dlv_request->operation)
                  {

                  /*
                   *      Service the read request.
                   */

                  case READ_BLOCK_FUNCTION:
```

```
/*
 *      Disable interrupts for
 *      the device.
 */

ELN$DISABLE_INTERRUPT(ipl);

/*
 *      Initialize the communications
 *      region for this request.
 */

rx_region_ptr->read_count =
                dlv_request->length;
rx_region_ptr->buf_ptr = 0;
rx_region_ptr->error = FALSE;
rx_region_ptr->read_in_progress =
                TRUE;

/*
 *      Re-enable interrupts and wait
 *      for the read to be performed
 *      by the interrupt service
 *      routine (ISR).
 */

ELN$ENABLE_INTERRUPT();
ker$wait_any(NULL, NULL, NULL,
             dlv_receive_device);

/*
 *      Check for read errors; if
 *      an error occurred, set the
 *      error flag in the DLV packet,
 *      and set the buffer length to
 *      the number of characters
 *      successfully read.
 */

if (rx_region_ptr->error)
        {
        dlv_request->error = -1;
        dlv_request->length =
            rx_region_ptr->buf_ptr;
        }
else
        dlv_request->error = 0;
```

```
        /*
         *      Copy the received bytes from
         *      the communications buffer
         *      into the DLV message packet.
         */

        COPY_BYTES(rx_region_ptr->rbuffer,
                   dlv_request->buffer,
                   dlv_request->length);

        /*
         *      Send the response back to
         *      the requestor.
         */

        ker$send(NULL, dlv_message,
                 request_size,
                 &dlv_driver_port,
                 NULL, FALSE);
        break;

    /*
     *  Service write request.
     */

    case WRITE_BLOCK_FUNCTION:

        /*
         *      Copy the packet buffer data
         *      to the communications region
         *      buffer.
         */

        COPY_BYTES(dlv_request->buffer,
               tx_region_ptr->xbuffer,
               dlv_request->length);

        /*
         *      Initialize the communications
         *      region for this request.
         */

        tx_region_ptr->buf_ptr = 0;
        tx_region_ptr->write_count =
                 dlv_request->length;

        /*
         *      Disable interrupts from the
         *      device and set the interrupt
         *      enable bit in the CSR; this
```

```
         *      causes the device to inter-
         *      rupt the processor (since the
         *      ready bit should be set), and
         *      the ISR can then perform the
         *      output.
         */

        ELN$DISABLE_INTERRUPT(ipl);
        write_register(XCSR$M_INT_ENA,
                       &register_ptr->xcsr);
        tx_region_ptr->write_in_progress =
                       TRUE;

        /*
         *      Re-enable interrupts and wait
         *      for the I/O to complete.
         */

        ELN$ENABLE_INTERRUPT();
        ker$wait_any(NULL, NULL, NULL,
                     dlv_transmit_device);

        /*
         *      Send the response back to
         *      the requestor, indicating
         *      the buffer was output.
         */

        dlv_request->error = 0;
        ker$send(NULL, dlv_message,
                 request_size,
                 &dlv_driver_port, NULL,
                 FALSE);
        break;

/*
 *      Service done request.
 */

case DONE_FUNCTION:

        /*
         *      Send a message back to the
         *      requestor indicating the
         *      done request was received,
         *      then set the done flag to
         *      exit from the loop.
         */
```

```
                        dlv_request->error = 0;
                        ker$send(NULL, dlv_message,
                                request_size,
                                &dlv_driver_port, NULL,
                                FALSE);
                        done = TRUE;

                    }
                }

        /*
         *      Since the user program is the last to receive
         *      a message on the circuit, wait on the port until
         *      it disconnects, then disconnect at this end;
         *      this avoids having the circuit disconnected
         *      before the user program receives the last
         *      message.
         */

        ker$wait_any(&status, NULL, NULL, &dlv_driver_port);
        ker$disconnect_circuit(NULL, &dlv_driver_port);
        }
}

/*
 *      Receiver ISR.
 */

void receive_service_routine(int_registers, int_region)

struct register_def             *int_registers;
struct rx_region_def            *int_region;

{

unsigned short  receive_input;

/*
 *      Read the receive buffer register.
 */

receive_input = read_register(&int_registers->rbuf);

/*
 *      If the driver is waiting for input, put the
 *      character in the communications region buffer,
 *      otherwise drop it.
 */
```

```
if (int_region->read_in_progress)

      /*
       *      Check for errors on the read.
       */

      if (receive_input&RBUF$M_ERROR)
            {

            /*
             *      If an error occurred, set the
             *      error bit in the communications
             *      region and signal the device.
             */

            int_region->error = TRUE;
            int_region->read_in_progress = FALSE;
            ker$signal_device(NULL, 0);
            }
      else
            {

            /*
             *      Otherwise, put the received
             *      character in the communcations
             *      region buffer and bump up the
             *      buffer pointer.  If this character
             *      satisfies the request, signal
             *      the device.
             */

            int_region->rbuffer[int_region->buf_ptr++] =
                                    receive_input&RBUF$M_CHAR;
            if (int_region->buf_ptr >= int_region->read_count)
                  {
                  ker$signal_device(NULL, 0);
                  int_region->read_in_progress = FALSE;
                  }
            }
}

/*
 *      Transmitter ISR.
 */

void transmit_service_routine(int_registers, int_region)

struct register_def           *int_registers;
struct tx_region_def          *int_region;
```

```
{

/*
 *      If the driver is waiting for output, output
 *      characters to the DLV until done.
 */

if (int_region->write_in_progress)
      if (int_region->write_count > int_region->buf_ptr)

            /*
             *      More characters to output so
             *      output the next one and bump up
             *      the buffer pointer.
             */

            write_register(
                    int_region->xbuffer[int_region->buf_ptr++],
                    &int_registers->xbuf);
      else

            /*
             *      All characters output; clear
             *      the write_in_progress flag,
             *      clear interrupt_enable on the
             *      transmitter, and signal the
             *      device.
             */

            {
            ker$signal_device(NULL, 0);
            int_region->write_in_progress = FALSE;
            write_register(0, &int_registers->xcsr);
            }

}
```

Application 3
C Interface to Disk and File Utilities

Problem

How do you implement the C interface to the disk utility and file utility procedures, described in the *VAXELN User's Guide*, and the *VAXELN C Run-Time Library Reference Manual*?

Solution

The example in this section is designed to show as many of the disk utility and file utility procedures as possible.

The example is also designed to show:

- How the data types not normally found in C code written for UNIX can be integrated with the generic C data types and standard UNIX extensions. For example, notice the example's use of the RTL routine **sprintf** to concatenate one C string to two VARYING_STRING data items, yielding a VARYING_STRING result.
- How bit mask definitions are used in C to take the place of PASCAL sets. For example, see the volume, file, and record protection parameters passed to ELN$INIT_VOLUME. The masks deny a particular type of access and, therefore, the bitwise complement (~) operator is used to cast them into the more familiar positive-logic format. Also note the use of the address-of operator (&) to pass these constant values to the

procedure by reference, rather than by value; this extension to the C language is unique to the VAX C compiler.

- How parameters are passed by reference to the disk utility and file utility procedures. A common mistake when coding C for VAXELN is to omit an ampersand (&) on a function parameter.
- How to implement a construction necessitated by the status code conventions of VAXELN (and VAX/VMS). The UNIX status code convention is: 0 (= C "false") return status indicates success; a nonzero (= C "true") return status indicates an error. VAXELN and VAX/VMS use the low bit of a status code to denote success(=1) or failure(=0); this is the basis for the almost idiomatic test in the example:

```
if (!(status&1))
     statement...          /* failure */

-or-

if (status&1)
     statement...          /* success */
```

To build the sample application, use the following commands:

```
$ cc application3 + eln$:vaxelnc/lib
$ link application3 + eln$:crtlshare/lib + rtlshare/lib +-
  rtl/lib
$ ebuild/noedit application3
```

The System Builder data file used to build this program to be run on an RX50 drive on a MicroVAX I system is:

```
characteristic /emulator=both
```

```
program application3
device DUA /register=%O772150 /vector=%O154
```

When run, this program produces the following output:

```
"Initialized disk in drive 'DUA1:' as volume name 'SAMPLE'.
Mounted disk.

Created directory 'DISK$SAMPLE:[TEST_DIR]'.
Created DISK$SAMPLE:[TEST_DIR]TEST_FILE.DAT;1.
Created DISK$SAMPLE:[TEST_DIR]TEST_FILE.DAT;2.
Created DISK$SAMPLE:[TEST_DIR]TEST_FILE.DAT;3.
Created DISK$SAMPLE:[TEST_DIR]TEST_FILE.DAT;4.
Created DISK$SAMPLE:[TEST_DIR]TEST_FILE.DAT;5.
Created DISK$SAMPLE:[TEST_DIR]TEST_FILE.DAT;6.
Created DISK$SAMPLE:[TEST_DIR]TEST_FILE.DAT;7.
Created DISK$SAMPLE:[TEST_DIR]TEST_FILE.DAT;8.
Created DISK$SAMPLE:[TEST_DIR]TEST_FILE.DAT;9.
Created DISK$SAMPLE:[TEST_DIR]TEST_FILE.DAT;10.

Copied   'DISK$SAMPLE:[TEST_DIR]TEST_FILE.DAT;10'
to       'DISK$SAMPLE:[TEST_DIR]TEST_COPY_FILE.DAT;100'.
Renamed  'DISK$SAMPLE:[TEST_DIR]TEST_COPY_FILE.DAT;100'
to       'DISK$SAMPLE:[TEST_DIR]TEST_RENAME_FILE.DAT;1234'.

Contents of data file =  0
Contents of data file =  1
Contents of data file =  2
Contents of data file =  3
Contents of data file =  4
Contents of data file =  5
Contents of data file =  6
Contents of data file =  7
Contents of data file =  8
Contents of data file =  9

Changed protection of
  'DISK$SAMPLE:[TEST_DIR]TEST_FILE.DAT;10'.

Deleted 'DISK$SAMPLE:[TEST_DIR]TEST_FILE.DAT;10'.
Deleted 'DISK$SAMPLE:[TEST_DIR]TEST_FILE.DAT;9'.
Deleted 'DISK$SAMPLE:[TEST_DIR]TEST_FILE.DAT;8'.
Deleted 'DISK$SAMPLE:[TEST_DIR]TEST_FILE.DAT;7'.
Deleted 'DISK$SAMPLE:[TEST_DIR]TEST_FILE.DAT;6'.
Deleted 'DISK$SAMPLE:[TEST_DIR]TEST_FILE.DAT;5'.
Deleted 'DISK$SAMPLE:[TEST_DIR]TEST_FILE.DAT;4'.
Deleted 'DISK$SAMPLE:[TEST_DIR]TEST_FILE.DAT;3'.
Deleted 'DISK$SAMPLE:[TEST_DIR]TEST_FILE.DAT;2'.
```

```
Deleted 'DISK$SAMPLE:[TEST_DIR]TEST_FILE.DAT;1'.
Deleted 'DISK$SAMPLE:[TEST_DIR]TEST_RENAME_FILE.DAT;1234'.
Dismounted the disk.

End of sample program."
```

Example

The following is a listing of the example written in C (application3.c).

```
#include $disk_utility
#include $file_utility
#include descrip
#include stdio

/*
 * Abstract:
 *
 *      This example shows typical calls from a C program to
 *      the disk utility and file utility procedures.
 *
 *      WARNING.  This program initializes the disk mounted
 *      in the drive named by the preprocessor constant
 *      TARGET_DRIVE, defined below; no other warning will be
 *      given.  The device must be readied for writing
 *      before the program is started.
 */

/*
 *      Preprocessor definitions:
 */

#ifndef TARGET_DRIVE
#define TARGET_DRIVE "DUA1:"    /* Default drive to use */
#endif

/*
 *      File specification definitions:
 */

#define DIRECTORY "DISK$SAMPLE:[TEST_DIR]"
#define DATAFILE  "DISK$SAMPLE:[TEST_DIR]TEST_FILE.DAT;"
#define COPYFILE                                                \
        "DISK$SAMPLE:[TEST_DIR]TEST_COPY_FILE.DAT;100"
#define COPYFILE2                                               \
        "DISK$SAMPLE:[TEST_DIR]TEST_RENAME_FILE.DAT;1234"

/*
 *      Define shorthand versions of volume and file
 *      protection masks:
 */
```

```
#define RWED     (DSK$M_READ |                           \
                  DSK$M_WRITE |                          \
                  DSK$M_EXEC |                           \
                  DSK$M_DELETE)
#define R        (DSK$M_READ)
#define RE       (DSK$M_READ |                           \
                  DSK$M_EXEC)

#define NOGROUP ((FILE$DENY_READ_ACCESS          |       \
                   FILE$DENY_WRITE_ACCESS        |       \
                   FILE$DENY_EXECUTE_ACCESS      |       \
                   FILE$DENY_DELETE_ACCESS) << FILE$GROUP)

/*
 *      VARYING_STRING constant declarations:
 */

VARYING_STRING_CONSTANT(drive_name,TARGET_DRIVE);
VARYING_STRING_CONSTANT(dir_fs,DIRECTORY);
VARYING_STRING_CONSTANT(data_fs,DATAFILE);
VARYING_STRING_CONSTANT(copy_fs,COPYFILE);
VARYING_STRING_CONSTANT(copy_fs2,COPYFILE2);
VARYING_STRING_CONSTANT(search_fs,
                        "DISK$SAMPLE:[TEST_DIR]*.*;*");
VARYING_STRING_CONSTANT(username,"USER");
VARYING_STRING_CONSTANT(volume,"SAMPLE");

/*
 *      Define a macro used to output VARYING_STRING data
 *      items:
 */

#define PRINT_VARYING(text1, vs, text2)                  \
        printf("%s%.*s%s", text1, vs.count, vs.data, text2)

/*
 * Routine description:
 *
 *      This code performs the following steps:
 *
 *          1.  Initializes the target disk with label =
 *              "SAMPLE".
 *          2.  Mounts the disk.
 *          3.  Creates the directory [TEST_DIR] on the
 *              disk.
 *          4.  Writes 10 data files named
 *              [TEST_DIR]TEST_FILE.DAT;* to the disk.
 *          5.  Copies [TEST_DIR]TEST_FILE.DAT;10 to
 *              [TEST_DIR]TEST_COPY_FILE.DAT;100.
```

```
 *          6.  Renames [TEST_DIR]TEST_COPY_FILE.DAT;100 to
 *              [TEST_DIR]TEST_RENAME_FILE.DAT;1234.
 *          7.  The renamed file in step 6 is typed on the
 *              console.
 *          8.  Changes the protection of
 *              [TEST_DIR]TEST_FILE.DAT;9 to exclude all
 *              GROUP access.
 *          9.  Uses eln$directory_open and eln$directory_list
 *              to visit all files in [TEST_DIR].  Each file
 *              visited is deleted.
 *          10. Dismounts the disk.
 */

main()
{

DSK$_BADBLOCK            bad_blocks[2];
DSK$_BADLIST             bad_block_list = {2,&bad_blocks};
char                     buffer[132];
ELN$DIR_FILE             *directory;
FILE$ATTRIBUTES_RECORD   *file_attr;
FILE                     *fp;
VARYING_STRING(255)      delete_fs,old_fs,new_fs,dirtmp_fs;
int                      status,i,j;

/*
 *      Initialize imaginary bad block information to mark
 *      LBNs 100-119 and 222-231, inclusive, as bad blocks.
 */

bad_blocks[0].type.logical.start_lbn = 100;
bad_blocks[0].type.logical.lbn_count = 20;
bad_blocks[0].pbn_format = FALSE;

bad_blocks[1].type.logical.start_lbn = 222;
bad_blocks[1].type.logical.lbn_count = 10;
bad_blocks[1].pbn_format = FALSE;

/*
 *      1.  Initialize the disk using reasonable values:
 */

eln$init_volume(&drive_name,    /* device name          */
                &volume,        /* volume name          */
                5,              /* default extension    */
                &username,      /* username             */
                0x00010001,     /* owner                */
                                /* volume protection    */
                                /* [RWED,RWED,RWED,]    */
                &~( RWED<<DSK$V_SYSTEM |
```

```
                        RWED<<DSK$V_OWNER  |
                        RWED<<DSK$V_GROUP),
                                    /* default file prot.   */
                                    /* [RWED,RWED,RE,]      */
                    &~( RWED<<DSK$V_SYSTEM |
                        RWED<<DSK$V_OWNER  |
                        RE  <<DSK$V_GROUP),
                                    /* default record prot. */
                                    /* [RWED,RWED,R,]       */
                    &~( RWED<<DSK$V_SYSTEM |
                        RWED<<DSK$V_OWNER  |
                        R   <<DSK$V_GROUP),
                    3,              /* accessed directories */
                    0,              /* maximum files        */
                    0,              /* user_directories     */
                    0,              /* file headers         */
                    7,              /* windows              */
                    0,              /* cluster size         */
                    DSK$_MIDDLE,    /* index position       */
                    DSK$_NOCHECK,   /* data check           */
                    TRUE,           /* share                */
                    FALSE,          /* group                */
                    TRUE,           /* system               */
                    TRUE,           /* verified             */
                    &bad_block_list,/* bad list             */
                    NULL);          /* status               */

    PRINT_VARYING("Initialized disk in drive '", drive_name,"'");
    PRINT_VARYING(" as volume name '", volume, "'.\n");

    /*
     *      2.  Mount the disk.
     */

    eln$mount_volume(&drive_name,
                     &volume,
                     NULL);

    printf("Mounted disk.\n\n");

    /*
     *      3.  Create the directory used by this test.
     */

        eln$create_directory(&dir_fs,
                             NULL,
                             0x00010001,
                             &new_fs);

        PRINT_VARYING("Created directory '", new_fs, "'.\n");
```

```
/*
 *      4.  Create a series of simple text files with the same
 *          filename and file type, but with version numbers
 *          ranging from 1 through 10.
 *
 *          The number of records in each file is the same as
 *          the file's version number.  Each record consists
 *          of the records numbered from 0 to the record
 *          version number -1.
 */

for(i = 1; i <= 10; i++)
        {
        fp = fopen(DATAFILE, "w");
        for(j = 0; j < i; j++)
                fprintf(fp, "%d\n", j);
        fclose(fp);
        printf("Created %s%d.\n", DATAFILE, i);
        }

/*
 *      5.  Copy the last file to another file.
 */

eln$copy_file(&data_fs,
              &copy_fs,
              NULL,
              NULL,
              NULL,
              NULL,
              &old_fs,
              &new_fs);

PRINT_VARYING("\nCopied\t'", old_fs, "'\n");
PRINT_VARYING("to\t'", new_fs, "'.\n");

/*
 *      6.  Rename the file just copied.
 */

eln$rename_file(&copy_fs,
                &copy_fs2,
                NULL,
                &old_fs,
                &new_fs);

PRINT_VARYING("Renamed\t'", old_fs, "'\n");
PRINT_VARYING("to\t'", new_fs, "'.\n\n");
```

```
/*
 *      7.  Open the renamed file for reading and type it on
 *          the console.
 */

fp = fopen(COPYFILE2, "r");
while(fgets(buffer, sizeof(buffer), fp) != NULL)
        printf("Contents of data file =\t%s", buffer);
fclose(fp);

/*
 *      8.  Protect the data file with the highest version
 *          number from all group access.
 */

eln$protect_file(&data_fs,
                 NULL,
                 &NOGROUP,
                 NULL,
                 &new_fs);

PRINT_VARYING("\nChanged protection of '", new_fs, "'.\n\n");

/*
 *      9.  Start a directory listing of this directory.
 */

directory = calloc(1, sizeof(*directory));
file_attr = calloc(1, sizeof(*file_attr));

eln$directory_open(&directory,
                   &search_fs,
                   &new_fs,
                   &dirtmp_fs,
                   NULL,
                   NULL,
                   &file_attr);

/*
 *      Loop for each file in the directory and delete them.
 */

for(;;)
        {
        eln$directory_list(&directory,
                           &dirtmp_fs,
                           &new_fs,
                           &status,
                           &file_attr);
        if (!(status&1))
```

```
                break;

        /*
         *      Concatenate volume + directory + filename.
         */

        delete_fs.count = sprintf(delete_fs.data,
                                  "DISK$SAMPLE:%.*s%.*s",
                                  dirtmp_fs.count,
                                  dirtmp_fs.data,
                                  new_fs.count,
                                  new_fs.data);

        eln$delete_file(&delete_fs, NULL, &new_fs);
        PRINT_VARYING("Deleted '", new_fs, "'.\n");
        }

/*
 *      10.  Dismount the disk.
 */

eln$dismount_volume(&drive_name, NULL);
printf("Dismounted the disk.\n\n");

printf("End of sample program.");

}
```

Application 4
Fast Device-Handling

Problem

How do you perform device I/O while avoiding the overhead incurred using standard Pascal I/O?

Solution

Most realtime device handling should be performed using procedures and/or processes within the job requiring the I/O. The Pascal I/O interface is only useful for distributed, record-and-file oriented I/O. Very efficient device I/O can be accomplished using VAXELN, because a program can directly initiate I/O requests, without going through a runtime system.

The example in this section shows a set of procedures and an interrupt service routine (ISR) that can be used to gather data from the AXV11C or ADV11C analog-to-digital converter. The example's ISR is called by the VAXELN kernel upon receiving an interrupt from the converter. The example's two procedures are used to create and initialize data structures that control the converter and to initiate conversion and read the resulting data.

The AXV11C and ADV11C are typical real-time devices. The basic strategy in the driver routines is to write to the device's control/status register (CSR), initiating I/O, and then wait on the DEVICE object. When the I/O is complete, the physical device interrupts the processor, causing the ISR to read the input data from the device's data buffer register. The ISR then signals the DEVICE object, allowing the main driver routine to complete.

Device drivers written in this fashion have at least one limitation: programs calling the procedures must run in kernel mode because the drivers will almost always need to call CREATE_DEVICE, and may also need to change the interrupt priority level of the processor.

The routines shown in this section are actually much simpler in function than a real device driver typically is; in addition to not supporting the full functionality of this particular device, these routines don't do several things usually done by real-time drivers:

- Polling. Many devices can be driven by polling rather than interrupts. That is, the driver writes to the CSR to initiate I/O, and then does not wait for an interrupt but rather goes into a loop, reading the CSR repeatedly until the I/O request is completed. Polling usually results in higher throughput but, since polling is done at a raised interrupt priority level, it prevents other processes from executing during the polling.
- Multiple I/O operations per call. Higher overall throughput can also be achieved by allowing the driver to read or write more than one piece of data in each call.

The AXV11C driver supplied in your kit does, in fact, implement both polling and multiple I/O operations per call. After becoming familiar with the example in this section, it's a good idea to study this and other VAXELN real-time device drivers to learn more about writing VAXELN real-time drivers.

The AXV11C hardware itself is discussed in detail in the *LSI-11 Analog System User's Guide.*

To build the sample application, use the following commands:

```
$ epascal application4 + eln$:rtlobject/lib
$ link/nosysshr application4 + eln$:rtlshare/lib +-
  eln$:rtl/lib
$ ebuild/noedit application4
```

The sample application can then be loaded into a target machine and executed. The data file must contain information for EBUILD, as follows:

```
characteristic /noconsole /nofile /noserver /emulator=both
program application4 /debug /mode=kernel
device AXV1 /register=%O170400 /vector=%O400 /noautoload
```

If your device has its dip switches set to values different from those above, you may have to use the System Builder to change the vector address, register address, or both.

Example

The following is a listing of the example written in Pascal (application4.pas).

```
module axv11;

{
{     This module contains some simple procedures to read
{     converted analog data from an AXV11 device, and a
{     program to demonstrate the procedures' use.
{}

type

      {
      {     Input/output data and gain data.
      {}

      axv_data = -%o3777..%o7777;

      {
      {     Identifiers, one for each physical device.
      {}

      axv = ^anytype;

      {
      {    AXV control/status register (CSR) record definition:
      {
      {    start:                  Initiates a conversion.
      {    gain_setting:           Controls gain.
      {    ext_start_enable:       Permits external start of
      {                            conversion.
      {    clock_start_enable:     Permits external start of
      {                            conversion.
      {    done_int_enable:        Enables interrupt at end of
      {                            conversion.
      {    a_d_done:               Is set when conversion is
      {                            complete.
      {    multiplexer_address:    Channel being addressed.
      {    error_int_enable:       Enables interrupt at an error
      {                            condition.
      {    a_d_error:              Set when error detected;
      {                            can't happen in this program.
      {}
```

```
axv_csr_type = [word] packed record
    start: [pos(0)] boolean;
    gain_setting: [pos(2)] 0..3;
    ext_start_enable: [pos(4)] boolean;
    clock_start_enable: [pos(5)] boolean;
    done_int_enable: [pos(6)] boolean;
    a_d_done: [pos(7)] boolean;
    multiplexer_address: [pos(8)] 0..15;
    error_int_enable: [pos(14)] boolean;
    a_d_error: [pos(15)] boolean
    end;

{
{     Result of A/D conversion.
{}

axv_dbr_type = [word] axv_data;

{
{     AXV register layout in controller:
{
{     csr -   Control/status register.
{     dbr -   Data buffer register.
{}

axv_registers = [aligned(1)]  packed record
    csr: axv_csr_type;
    dbr: axv_dbr_type
    end;

{
{     AXV interrupt communication region:
{
{     dbr_read - Temporary repository for data read
{                in an interrupt service routine.
{}

axv_done_interrupt_region = record
    dbr_read: axv_dbr_type
    end;

{
{     Data area for each device:
{
{     done_device:      Device object for completed
{                       interrupt.
{     registers:        Address of device's registers.
{     done_region_ptr:  Address of completed interrupt
{                       region.
```

```
    {}

    axv_data_area = packed record
        done_device: device;
        registers: ^axv_registers;
        done_region_ptr: ^axv_done_interrupt_region
        end;

interrupt_service done_interrupt(reg: ^axv_registers;
                                 com: ^axv_done_interrupt_region);

{++
{
{ Routine description:
{
{     Called upon receipt of an interrupt that indicates a
{     conversion is complete, this routine reads the data
{     from the just-completed conversion into the communication
{     region, then signals the device.
{
{ Inputs:
{
{     reg - Address of the device registers.
{     com - Address of the done interrupt communication region.
{
{ Outputs:
{
{     The conversion data is stored in the communication
{     region and the device is signaled.
{
{--}

begin

{
{     Read the new data.
{}

com^.dbr_read := read_register(reg^.dbr);

{
{     Signal the device to enable axv_read to continue.
{}

signal_device

end;
procedure axv_initialize(device_name: [readonly]
varying_string(30);
                       var identifier: axv);
```

```
{++
{
{ Routine description:
{
{     This procedure is called to allocate and initialize
{     the data area for an ADV11C or AXV11C, and it also
{     creates the necessary DEVICE objects.  This procedure
{     must be called once for each physical device.
{
{ Inputs:
{
{     device_name -   1-to-30-character string giving the name
{                     of the device. It must match the name
{                     established with the System Builder.
{
{ Outputs:
{
{     identifier -    Longword identifier to be used in
{                     subsequent calls to axv_initialize,
{                     axv_read, and axv_write to identify
{                     this device.
{
{     An identifier is allocated and returned, a device object
{     is created, and the physical device is initialized.
{
{--}

begin

{
{     Get a new identifier.
{}

new(identifier::^axv_data_area);

with identifier^::axv_data_area do
      begin

      {
      {     Create a device object for the completed interrupt.
      {}

      create_device(device_name,
                    done_device,
                    vector_number := 1,
                    service_routine := done_interrupt,
                    region := done_region_ptr,
                    registers := registers);
```

```
      {
      {     Initialize the device's CSR.
      {}

      write_register(registers^.csr)
      end;
end;
procedure axv_read(identifier: axv;
                   channel: integer;
                   var converted_data: axv_data);

{++
{
{ Routine description:
{
{     This routine is called to initiate a conversion and
{     gather the resulting datum from an AXV11C or ADV11C
{     device on the specified channel.
{
{ Inputs:
{
{     identifier -     Expression of type AXV giving the
{                      value of an identifier (which was
{                      returned by axv_initialize) of the
{                      device to be read.
{
{     channel -        Integer expression giving the analog
{                      channel to be read.
{
{ Outputs:
{
{     converted_data - Received resultant datum from the
{                      requested conversion.
{
{--}

{
{     Local-variable declarations:
{}

var
      csr_template: axv_csr_type;

begin

with identifier^::axv_data_area do
      begin
```

```
        {
        {     First, set up the CSR templates;
        {     begin with the contents of the CSR.
        {}

        csr_template := read_register(registers^.csr);

        {
        {     Now set the following fields:
        {
        {     start -                 Necessary for initiating
        {                             conversions from the program.
        {     done_int_enable -       Lets the device interrupt.
        {     multiplexer_address - Sets the channel.
        {}

        with csr_template do
              begin
              start := true;
              done_int_enable := true;
              multiplexer_address := channel
              end;

        {
        {     Write to the device register to initiate
        {     the conversion and wait on the device.
        {     The wait will be satisfied when the ISR
        {     has read the converted data and signals the device.
        {}

        write_register(registers^.csr, csr_template);
        wait_any(done_device);

        {
        {     Finally, move the converted data into
        {     the user-supplied variable.
        {}

        converted_data := done_region_ptr^.dbr_read
        end;
end;

program test(input,output);

{
{     This test program tests the above driver routines.
{     It prompts the user for a channel to sample and
{     samples the channel 5 times, printing the resulting
{     voltage from each sample.  This program assumes
{     that the device is configured to give bipolar,
```

```
{      offset-binary outputs; if this is not the case,
{      you can change the statement marked '**'.
{}

var
       ident: axv;              { the device's identifier }
       channel: integer;        { channel to be read }
       result: axv_data;        { resultant data from a conversion }
       voltage: real;           { result converted to a voltage }
       i: integer;

begin

{
{      Initialize the device.
{}

axv_initialize('AXV1', ident);

{
{      Loop to read data.
{}

while true do
       begin

       {
       {      Obtain the channel number from the user.
       {      Exit if the channel number is negative.
       {}

       write('channel? ');
       readln(channel);

       if channel < 0
       then
              goto finished;

       for i := 1 to 5 do
              begin
              axv_read(ident, channel, result);
{'**'}          voltage := (result - %o4000) * (10.0 / %o4000);
              writeln(voltage:5:2, ' volts')
              end;
       end;

finished:

end;
end.
```

Application 5
FORTRAN Routine Inclusion

Problem

How do you include FORTRAN routines in a VAXELN system without rewriting them in Pascal or C?

Solution

The example in this section demonstrates how two FORTRAN functions are called in VAXELN Pascal and C; it demonstrates appropriate interfaces for passing strings, scalars, and multidimensional arrays.

Several considerations (demonstrated in this section's example) must be kept in mind when calling FORTRAN subroutines and functions from VAXELN programs:

- Not all FORTRAN is suitable. FORTRAN that calls any VAX/VMS services or runtime routines not included in the VAXELN libraries cannot be used. For example, FORTRAN routines that perform I/O cannot be used because there is no FORTRAN I/O system included in VAXELN.
- When linking to produce an image for a VAXELN system in which a FORTRAN routine is being called, specify the NOSYSSHR qualifier to prevent the linker from searching the VAX/VMS default shareable-image library for unresolved references.

- By default, FORTRAN passes parameters by reference. Therefore, in the VAXELN Pascal declaration of a FORTRAN routine, the [REFERENCE] attribute must normally be specified on all parameters that VAXELN Pascal would not otherwise pass by reference. In C, this can be handled by being sure to pass the address of the argument.
- FORTRAN stores array elements differently from both VAXELN Pascal and C. In FORTRAN, elements are stored such that the leftmost subscript varies the most rapidly as one traverses the array in memory. In Pascal and C, the *rightmost* subscript varies the most rapidly. This means that, for example, in a two dimensional array, rows become columns and columns become rows.

To build the Pascal sample application, use the following commands:

```
$ fortran application5c
$ epascal application5b + eln$:rtlobject/lib
$ link/nosysshr application5b + application5c + -
  eln$:rtlshare/lib + eln$:rtl/lib
$ ebuild/noedit applicationb
```

To build the C sample application + use the following commands:

```
$ fortran applicationc
$ cc application5a + eln$:vaxelnc/lib
$ link/nosysshr application5a+application5c + -
  eln$:crtlshare/lib + eln$:rtlshare/lib + _$eln$:rtl/lib
$ ebuild/noedit application5a
```

The sample application can then be loaded into a target machine and executed. The data file must contain information for EBUILD, as follows:

```
characteristic /noconsole /nofile /noserver /emulator=both
program application5 /debug
```

C Example

The following is a listing of the example written in C (application5a.c).

```
#module application5
#include descrip

/*
 *      This module demonstrates the calling of
 *      FORTRAN functions from C.
 */

float  a1[5][10] = { 1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0,
                     1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0,
                     1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0,
                     1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0,
                     1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0 };

float  a2[10][5] = { 1.0,2.0,3.0,4.0,5.0,
                     1.0,2.0,3.0,4.0,5.0,
                     1.0,2.0,3.0,4.0,5.0,
                     1.0,2.0,3.0,4.0,5.0,
                     1.0,2.0,3.0,4.0,5.0,
                     1.0,2.0,3.0,4.0,5.0,
                     1.0,2.0,3.0,4.0,5.0,
                     1.0,2.0,3.0,4.0,5.0,
                     1.0,2.0,3.0,4.0,5.0,
                     1.0,2.0,3.0,4.0,5.0 };

float  vec1[5],vec2[10];

/*
 *       The declarations of FORTRAN routines in C involve
 *       the only thing that C "knows" about the function:
 *       the function's return type.  When actually coding
 *       calls to the FORTRAN routines, you are responsible
 *       for mapping the FORTRAN language semantics of the
 *       arguments into C semantics, for example putting the
 *       correct data in the argument list.
 */

/*
 *       Below is the declaration of the first function.
 *       By default, FORTRAN passes arguments by reference,
 *       and the array dimensions are reversed from what they
```

```
 *      are in C.  Therefore, although this function sums
 *      the FORTRAN array's columns, it will sum the C array's
 *      rows.  Note that the declaration's extents are reversed
 *      from the way they appear in the FORTRAN declaration;
 *      that is, it's the number of rows that's passed before
 *      the number of columns, in the FORTRAN sense.
 */

/* 1st arg = pointer to matrix to sum */
/* 2nd arg = pointer to output sum vector */
/* 3rd arg = pointer to number of rows (FORTRAN) in matrix */
/* 4th arg = pointer to number of columns (FORTRAN) in matrix */

float   sum();

/*
 *      Declaration of second function:
 */

/* 1st argument = pointer to string descriptor */
/* FORTRAN calls this a "passed length character argument" */

int     icmax();

main()
{

float   result;
int     i;
static  $DESCRIPTOR(str, "abcdefghij");
static  $DESCRIPTOR(str2, "zyxwvutsrq");

result  = sum(&a1, &vec1, &10, &5);
result += sum(&a2, &vec2, &5, &10);
result += icmax(&str);
result += icmax(&str2);

/*
 *      Check the result of the calls.  Result itself should be
 *      436.  Each of the elements of vec1 should be 55,
 *      and each of the elements of vec2 should be 15.
 */

printf("The value of Result is %g (Should be 436)\n\n", result);

printf("\nThe values of vec1 should all be 55.  They are:\n");
for(i = 0; i < 5; i++)
      printf("%g\n", vec1[i]);

printf("\nThe values of vec2 should all be 15.  They are:\n");
```

```
for(i = 0; i < 10; i++)
            printf("%g\n",vec2[i]);

}
```

Pascal Example

The following is a listing of the example written in Pascal (application5b.pas).

```
module application5;

{
{     This module demonstrates the calling of
{     FORTRAN functions from VAXELN Pascal.
{}

type
      real_array(m,n: integer) = array[1..m,1..n] of real;
      real_vector(m: integer) = array[1..m] of real;

var
      a1: real_array(5,10) := (5 of (1,2,3,4,5,6,7,8,9,10));
      a2: real_array(10,5) := (10 of (1,2,3,4,5));
      vec1: real_vector(5);
      vec2: real_vector(10);

{
{     Below is the declaration of a FORTRAN function.
{     By default, FORTRAN passes arguments by reference,
{     and the array dimensions are reversed from what they
{     are in Pascal.  Therefore, although this function sums
{     the FORTRAN array's columns, it will sum the Pascal
{     array's rows.  Note that the declaration's extents are
{     reversed from the way they appear in the FORTRAN
{     declaration; that is, it's the number of rows that's
{     passed before the number of columns, in the FORTRAN sense.
{}

function sum(ary: real_array(<m>,<n>);
          vec: real_vector(m);
          n,m: [reference] integer): real;
external;

{
{     Declaration of another FORTRAN function; this one
{     demonstrates the passing of a string by descriptor.
{     (FORTRAN calls this a "passed length character argument".)
{}

function icmax(cvar: string(<n>)): integer;
```

```
external;

program test(output);

var
      result: real;
      i: integer;
      str: string(10) := 'abcdefghij';

begin

result := sum(a1,vec1);
result := result + sum(a2,vec2);
result := result + icmax(str);
result := result + icmax('zyxwvutsrq');

{
{     Check the result of the calls.  The result itself
{     should be 436.  Each of the elements of vec1 should
{     be 55, and each of the elements of vec2 should be 15.
{}

writeln('The value of Result is ',
        result:5:1,
        ' (Should be 436)');
writeln;

writeln('The values of vec1 should all be 55.  They are:');
for i := 1 to 5 do
      writeln(vec1[i]:4:1);
writeln;

writeln('The values of vec2 should all be 15.  They are:');
for i := 1 to 10 do
      writeln(vec2[i]:4:1)

end;
end;
```

FORTRAN Subroutines

The following is a listing of the example subroutines written in FORTRAN (application5c.for).

```
C
C This module defines some FORTRAN functions to be
C called from VAXELN Pascal or C in a VAXELN process
C

C
C This function sums each column in the array,
C places the sum into the vector, and returns
C the sum of all elements as the function result.
C
      FUNCTION SUM(ARRAY, VECTOR, M, N)

            DIMENSION ARRAY(M,N), VECTOR(N)

            INTEGER COL,ROW

            SUM = 0.0

            DO 20 COL = 1,N
                  VECTOR(COL) = 0.0

                  DO 10 ROW = 1,M
                      VECTOR(COL) = VECTOR(COL) + ARRAY(ROW,COL)
                      SUM = SUM + ARRAY(ROW,COL)
10                    CONTINUE

20                CONTINUE

            RETURN
            END

C
C This function returns the position in a string of
C the character with the highest ASCII code value.
C
      FUNCTION ICMAX(CVAR)

      CHARACTER*(*) CVAR

            ICMAX = 1
            DO 10 I = 2, LEN(CVAR)
```

```
                IF (CVAR(I:I) .GT. CVAR(ICMAX:ICMAX)) ICMAX = I
10            CONTINUE
          RETURN
          END
```

Application 6
Interjob Communication

Problem

How do you perform interjob communication in VAXELN?

Solution

MESSAGE objects manipulated with the SEND and RECEIVE kernel procedures over a circuit give you the most flexible and dependable method to transfer data between VAXELN jobs.

This method gives you the most dependability because a circuit, which guarantees in-order delivery of your messages, is used.

This method gives you the most flexibility because, with message passing, portions of your application can be moved to any VAXELN node in your local network without change to your program. (However, this flexibility affects performance; for a discussion of the performance cost and alternatives to message passing, see the "Overview" section.)

The example in this section shows interjob data communication using a circuit associated with a named port. There are two jobs in the test system.

The first job, named APPLICATION6A, is the owner of the port with the associated local name. The port is named INITIAL_JOB_PORT. Using a NAME object allows your programs to identify themselves symbolically to their "peers" and allows you the option of making the ports visible on a local (per processor) or universal (per local network) basis; this means your applications can serve as system-wide or network-wide resources with very few changes in your code required. (Typically, all that is required is changing the "table" argument you supplied to the CREATE_NAME kernel procedure in one source module.)

The port name used in this section's example must be established before the second job in the example system is started, or the example system might fail when it attempts to connect a circuit to the named port. Correspondingly, the *Init required* attribute is specified for the first job in this system and it invokes the INITIALIZATION_DONE kernel procedure when it has finished establishing the port name.

The action in this example is controlled by the second job in this example, named APPLICATION6B, which transmits a data message to the first job and waits for it to be returned by the first job. The data messages are set up in this example so that they gradually shrink in size until they match a particular value the two jobs have both defined as being the end-of-dialogue indicator.

To build the sample application, use the following commands:

```
$ epascal application6a
$ link application6a + eln$:rtlshare/library + rtl/library
$ epascal application6b
$ link application6b + eln$:rtlshare/library + rtl/library
$ ebuild application6/noedit
```

The sample application can then be loaded into a target machine and executed. The data file must contain information for EBUILD, as follows:

```
program application6a /initialize
program application6b
```

Example

The following is a listing of the example written in Pascal (application6a.pas, application6b.pas).

```
module application6a;

{++
{
{ Abstract:
{
{       This module contains the first job (named APPLICATION6A)
{       of a two-job system designed to show two jobs passing
{       data back and forth using the SEND and RECEIVE kernel
{       procedures on a circuit based on a named port.
{
{--}

{
{       Job-wide declarations:
{}

var
        io_port: port;
        data_message: message;
        identifier: name;
        string_data: ^varying_string(32);
        done: boolean := false;




program application6a;

{++
{
{ Functional description:
{
{       This program creates a port with an associated name
{       of INITIAL_JOB_PORT as part of an initialization
{       action.
{
{       After the initialization is performed, the program
{       simply waits for incoming messages (in ASCII) to be
```

```
{       received through a circuit established on that port.
{       The program transmits the data back to the sender,
{       stripping off the first and last characters, until it
{       receives a message with the value '***END***', which
{       causes this program to terminate.
{
{ Inputs:
{
{       Incoming ASCII string messages directed to the global
{       port named INITIAL_JOB_PORT.
{
{ Outputs:
{
{       All incoming data is transmitted back to the sender
{       with the first and last characters removed.
{
{--}

begin

{
{       Initialization section:
{
{       Create the port and name it INITIAL_JOB_PORT.
{}

create_port(io_port);
create_name(identifier, 'INITIAL_JOB_PORT', io_port);
initialization_done;

{
{       Wait here to accept the incoming circuit request.
{}

accept_circuit(io_port);
writeln('Job APPLICATION6A accepted the circuit.');

{
{       Loop for each message received and process it.
{}

while not done do
        begin

        {
        {       Wait for a message to arrive on the port.
        {       When it arrives, receive the message and output
        {       it's contents to the standard output file.
        {}
```

```
        wait_any(io_port);
        receive(data_message, string_data, io_port);
        writeln('Job APPLICATION6A received "',
                string_data^,
                '".');

        if string_data^ = '***END***'
        then
                done := true
        else
                begin
                string_data^ := substr(string_data^,  2,
                                       length(string_data^)-2);
                send(data_message, io_port)
                end;
        end;

writeln('Job APPLICATION6A exited.')
end;
end;
```

```
module APPLICATION6B;

{++
{
{ Abstract:
{
{       This module contains the second job (named
{       APPLICATION6B) of a two-job system designed to show two
{       jobs passing data back and forth using the SEND and
{       RECEIVE kernel procedures on a circuit based on a named
{       port.
{
{--}

{
{       Job-wide declarations:
{}

var
        io_port: port;
        data_message: message;
        string_data: ^varying_string(32);
        done: boolean := false;
```

```
program application6b;

{++
{
{ Functional description:
{
{       This program starts a message dialogue with job
{       APPLICATION6A.  The messages transmitted back and
{       forth start out as the string:
{
{               '**********END**********'
{
{       which APPLICATION6A whittles down to the final
{       string, '***END***', by deleting the first and last
{       characters of the messages it receives.
{
{       This program initially connects the circuit between
{       the two jobs, then transmits data and receives
{       the modified data for retransmission.
{
{ Inputs:
{
{       ASCII string messages directed from APPLICATION6A
{       through its port named INITIAL_JOB_PORT.
{
{ Outputs:
{
{       All incoming data is transmitted back to the sender
{       without modification.
{
{--}

begin

{
{       Create the message packet and initialize the data
{       string.
{}

create_message(data_message, string_data);
string_data^ := '**********END**********';
```

```
{
{       Create a port and connect to the other program.
{}

create_port(io_port);
connect_circuit(io_port,
                destination_name := 'INITIAL_JOB_PORT');
writeln(
    'Job application6b connected circuit to INITIAL_JOB_PORT.');


{
{       Loop transmitting and receiving data until '***END***'
{       is seen.
{}

while not done do
        begin
        done := (string_data^ = '***END***');
        send(data_message, io_port);

        if not done
        then
                begin

                {
                {       Wait for the modified string to be
                {       sent back to our port.  When it arrives,
                {       receive it and output it's contents to
                {       the standard output file.
                {}

                wait_any(io_port);
                receive(data_message, string_data, io_port);
                writeln('Job APPLICATION6B received "',
                        string_data^,
                        '".')
                end;
        end;

writeln('Job APPLICATION6B exited.')
end;
end;
```

Application 7
Intra-Job Synchronization

Problem

How do you efficiently synchronize two processes running in the same job?

Solution

VAXELN provides the semaphore data type, which synchronizes processes within the same job. A semaphore may be thought of as a gate that will let only a given number of processes through to a certain resource. (For more information about semaphores, see the *VAXELN User's Guide*.)

The simplest semaphore is called a binary semaphore; this semaphore allows only one process at a time to access the resource the semaphore protects.

VAXELN also provides a more efficient implementation of a binary semaphore, a *mutex*. A mutex is more efficient as a process synchronization mechanism because calls to ELN$LOCK_MUTEX and ELN$UNLOCK_MUTEX do not usually incur the overhead inherent in calling a kernel procedure; the semaphore routines, WAIT and SIGNAL, are kernel procedures. (For more information about mutexes, see either the *VAXELN Pascal Language Reference Manual*, or the *VAXELN C Run-Time Library Reference Manual*.)

In the example in this section, mutexes are used to synchronize two processes writing to the console. Two processes are created, each running the same code. Each process locks the mutex, writes to the console, then unlocks the mutex, thus preventing both processes from writing to the console at the same time.

To build the sample application, use the following commands:

```
$epascal application7 + eln$:rtlobject/lib
$link/nosysshr application7 + eln$:rtlshare/lib +-
  eln$:rtl/lib
$ebuild/noedit application7
```

The sample application can then be loaded into a target machine and executed. The data file must contain information for EBUILD, as follows:

```
characteristic /noconsole /nofile /noserver
program application7 /debug
```

If you are using EDEBUG to run this program (which we recommend), it's a good idea to first issue the CANCEL CONTROL command so that the debugger will not pause at the beginning of each process's execution.

Example

The following is a listing of the example written in Pascal (application7.pas).

```
module mutex_test;

{
{      This module demonstrates the use of mutexes to
{      synchronize two processes writing to the console.
{}

include
       $mutex;

const
       write_limit = 10;    { Writes performed by each process. }

var
       gate : mutex;

program test(input,output);

var
       first_process, second_process: process;

begin

{
{      Create the mutex and the processes.  Lock the mutex
{      as soon as it's created so that both of the created
{      subprocesses will be forced to wait on it.
{}

create_mutex(gate);
lock_mutex(gate);
create_process(first_process, two_flavors, 1);
create_process(second_process, two_flavors, 2);
write('Hit <CR> to start the program: ');
readln;

{
{      Now, start the two processes by unlocking the mutex;
{      wait until both are finished before exiting.
{}
```

```
unlock_mutex(gate);
wait_all(first_process,second_process)
end.

process_block two_flavors(process_number : integer);
{
{     This is the process that will be created in two
{     different versions. One version will have the
{     value 1 for process_number, the other will have
{     the value 2 for process_number.
{}

var
      process_name: string(3);
      write_count: integer := 0;

begin

{
{     Loop once to write a message to the console.
{}

while write_count < write_limit do
      begin

      {
      {     Wait on the mutex.
      {}

      lock_mutex(gate);

      {
      {     Write the message.
      {}

      if (process_number = 2)
      then
            begin
            process_name := 'two';
            write(' ':30)
            end
      else
            process_name := 'one';

      writeln('This is from process ', process_name);
      write_count := write_count + 1;
      {
      {     Unlock the gate so that the other process
      {     can continue.
      {}
```

```
          unlock_mutex(gate)
          end;
      end;
      end;
```

Application 8
Making a Bootable Floppy Disk

Problem

How do you make a VAXELN system bootable from a floppy disk if you do not have MicroVMS running on your MicroVAX?

Solution

You need a VAXELN program to initialize a floppy disk and make it bootable. The example in this section initializes a disk (presumably a floppy disk), mounts it, creates all required directories, and provides three methods for copying the bootable system file from a host system to the floppy. The copying can be performed by:

- The DCL COPY command on the host system
- The VAXELN COPY_FILE utility
- The program itself, using the GET and PUT functions

The example program takes 2 program arguments: the drive specification (such as DUA1:), and the desired volume label for the disk. The program prompts the user for missing parameters.

To build the sample application, use the following commands:

```
$ epascal application8 + eln$:rtlobject/lib
$ link application8 + eln$:rtlshare/lib+rtl/lib/include=(-
                  eln$msgdef_text,-
                  ker$msgdef_text,-
                  pas$msgdef_text)
$ ebuild application8
```

The sample application can then be loaded into a target machine and executed. The data file must contain information for EBUILD, as follows:

```
PROGRAM application8
DEVICE DUA /register=%o772150 /vector=%0154
```

Example

The following is a listing of the example written in Pascal (application8.pas).

```
module APPLICATION8;

{++
{
{ Abstract:
{
{       This program initializes an RX50 floppy disk, creates
{       required directories on it, and copies a bootable
{       VAXELN system image on it.  A MicroVAX may then be
{       booted from the floppy.
{
{--}

include
        $disk_utility, $file_utility, $get_message_text,
        $elnmsg, $kernelmsg, $pascalmsg;

program make_bootable_floppy;

{
{       Local constant definitions:
{}

const
        boot_file = '[SYS0.SYSEXE]sysboot.exe';

{
{       Local type declarations:
{}

type
        blocks = packed array [1..128] of integer;

{
{       Local variable declarations:
{}

var
        copy_method, file_size, status: integer;
        bad_block_list: dsk$_badlist(1);
```

```
        source_file_var, destination_file_var: file of blocks;

        answer: varying_string(10);
        drive_name: varying_string(30);
        status_text, system_file_spec: varying_string(255);
        volume_label: varying_string(12);

        bootable: boolean;




procedure error_exit(status_message: varying_string(80);
                     status_value: integer);
{++
{
{ Routine description:
{
{       This procedure accepts a status value and some
{       accompanying text, translates the status value
{       to VAXELN message text, and outputs both text strings
{       to the console.  This procedure then causes the program
{       to terminate.
{
{ Inputs:
{
{       status_message - Supplies a varying string which is
{                        output to the console prior to the
{                        status message.
{
{       status_value -   Supplies the status value whose
{                        associated text will be output to
{                        the console.
{
{ Outputs:
{
{       Status code text information is written to the standard
{       output file (usually the console).
{
{--}

{
{       Local variable declarations:
{}

var
        status_text: varying_string(255);
```

```
begin

{
{       Obtain the text associated with this status code and
{       output both the input string and the generated string
{       to the console.
{}

eln$get_status_text(status_value, [status$text], status_text);
writeln;
writeln(status_message);
writeln(status_text);

{
{       Dismount the specified disk volume.
{}

dismount_volume(drive_name, status:=status);
exit

end;




function get_source_spec: varying_string(255);

{++
{
{ Routine description:
{
{       This function prompts the user for the file
{       specification of the source system file on
{       the host machine.
{
{ Inputs:
{
{       None.
{
{ Outputs:
{
{       Function returns user-entered file specification.
{
{--}
```

```
{
{       Local variable declarations:
{}

var
        source_spec : varying_string(255);

begin

{
{       Prompt the user for the file specification.  Note that
{       a multi-line prompt is used.  Read the specification.
{}

writeln;
writeln('Enter full filename (including node number and access',
        'control string if necessary)');
write('of system file: ');
readln (source_spec);

{
{       Return the file specification input.
{}

get_source_spec := source_spec

end;




[inline] procedure dcl_method;

{++
{
{ Routine description:
{
{       This procedure simply prompts the user to type the
{       appropriate DCL commands for copying the system file to
{       the disk.
{
{ Inputs:
{
{       None.
{
{ Outputs:
{
```

```
{       None.
{
{--}

begin

{
{       Write an appropriate prompt string and wait for the user
{       to indicate that the file has been copied.
{}

writeln;
writeln('From the host system, use');
writeln('      $ COPY systemfile.sys node::', drive_name,
        '[SYS0.SYSEXE]sysboot.exe/CONTIGUOUS');
writeln('to copy the system file');
writeln;
write('Hit RETURN when complete: ');
readln (answer)

end;




[inline] procedure copyfile_method;

{++
{
{ Routine description:
{
{       This procedure copies the system image using the
{       copy_file utility procedure.  Note that the file
{       being copied is assumed to be contiguous.
{
{ Inputs:
{
{       None.
{
{ Outputs:
{
{       None.
{
{--}

begin
```

```
{
{       Get source-file specification.
{}

system_file_spec := get_source_spec;

{
{       Copy the file to this node.
{}

copy_file(system_file_spec,
          drive_name+boot_file,
          status := status);
if not odd(status)
then
        error_exit('copy_file failed', status)

end;




[inline] procedure get_put_method;

{++
{
{ Routine description:
{
{       This procedure copies the system file directly, using
{       GET and PUT.  This method is used instead of the
{       copy_file procedure if the source system file is not
{       contiguous.
{
{ Inputs:
{
{       None.
{
{ Outputs:
{
{       None.
{
{--}

{
{       Local variable declarations:
{}
```

```
var
        file_attributes: ^file$attributes_record;

begin

{
{       Get the source file specification.
{}

system_file_spec := get_source_spec;

{
{       Open the source file.
{}

open(source_file_var,
     file_name := system_file_spec,
     history := history$old,
     file_attributes := file_attributes,
     status := status);

if not odd(status)
then
        error_exit('Open source file failed', status);

reset(source_file_var);

{
{       Compute the size of the file.
{}

file_size := file_attributes^.end_of_file_block;
if file_attributes^.first_free_byte = 0
then
        file_size := file_size - 1;

{
{       If the file is not of zero length, continue.
{}

if file_size > 0
then
        begin

        {
        {       Open the destination file.  Make it contiguous.
        {}

        open(destination_file_var,
             file_name := drive_name+boot_file,
```

```
			history := history$new,
			contiguous := true,
			filesize := file_size);

		if not odd(status)
		then
			error_exit('Copy_file failed', status);
		rewrite(destination_file_var);

		{
		{	Copy the source to the destination.
		{}

		while not eof(source_file_var) do
			begin
			destination_file_var^ := source_file_var^;
			put(destination_file_var);
			get(source_file_var)
			end;
		end;
end;
```

```
{
{	Main program starts here.
{}

begin

{
{	Get the drive name and desired volume label as program
{	arguments 1 and 2, respectively.  If no drive or volume
{	label is specified, ask the user.
{}

drive_name := program_argument(1);
if drive_name = ''
then
	begin
	write('Enter drive name   : ');
	readln(drive_name);
	write('Enter volume label : ');
	readln(volume_label)
	end
else
```

```
        begin
        volume_label := program_argument(2);
        if volume_label = ''
        then
                begin
                write('Enter volume label : ');
                readln(volume_label)
                end;
        end;

{
{       Set name and label to defaults, if not specified.
{}

if drive_name = ''
then
        drive_name := 'DUA1:';
if volume_label = ''
then
        volume_label := 'SCRATCH';

if substr(drive_name, length(drive_name), 1) <> ':'
then
        drive_name := drive_name + ':';

{
{       Ask the user if all is correct.
{}

writeln;
writeln('***** Initializing ', drive_name, ' *****');
writeln('This will destroy all information on this disk');
write('Do you wish to continue (Y or N [Y])? ');
readln (answer);

if (answer = '') or
   (substr(answer, 1, 1) = 'Y') or
   (substr(answer, 1, 1) = 'y')
then
        begin

        {
        {       Initialize the volume.
        {}

        init_volume(drive_name,
                    volume_label,
                    verified := false,
                    bad_list := bad_block_list::dsk$_badlist(0),
                    status := status);
```

```
if not odd(status)
then
        error_exit('init_volume failed', status);

{
{       Mount the floppy.
{}

mount_volume(drive_name, volume_label, status);
if not odd(status)
then
        error_exit('mount_volume failed', status);

{
{       Create the necessary directories.
{}

create_directory(drive_name + '[SYS0]', status);
if not odd(status)
then
        error_exit('create_directory failed', status);
create_directory(drive_name + '[SYS0.SYSEXE]', status);
if not odd(status)
then
        error_exit('create_directory failed', status);

{
{       The floppy is now initialized and is bootable.
{       The system image may be copied in one of three
{       ways, shown below.
{}

writeln;
writeln('Ready to copy system image.');
writeln('    1  Copy from host using DCL copy');
writeln('    2  Copy via VAXELN COPY_FILE utility');
writeln('    3  Copy with GETs and PUTs');
writeln;
writeln('Enter desired copy method (note: method 2');
write(
'requires a contiguous file on the host system) 1-3: ');
readln(copy_method);

bootable := true;
case copy_method of

        1:      dcl_method;

        2:      copyfile_method;
```

```
                3:      get_put_method;

                otherwise
                        bootable := false;

                end;

        dismount_volume(drive_name, status := status);
        if not odd(status)
        then
                error_exit('dismount_volume failed', status);

        writeln;
        if bootable
        then
                writeln('Operation complete - disk bootable')
        else
                writeln('Operation complete - disk initialized')
        end
else
        begin
        writeln;
        writeln('Initialization aborted')
        end;
end;
end;
```

Application 9
Multiple Circuit Server

Problem

How do you build and use a "server" in VAXELN? A server is a process that performs a particular processing function for other processes; the other processes are referred to here as "applications."

Description

A server can be used to describe many data processing and control problems, especially those problems that require one or more of the following characteristics:

- Resource Control. If a central resource, a disk data file for example, must be protected in one of your systems, access to that resource can be metered by a server.
- Complex Synchronization. The example in this section is of a multi-thread server, but a single-thread server can also be useful for forcing all operations of a particular kind through a single gateway. For example, in the case of a central application database where a data file must be protected against concurrent access, the server could be used to perform an intelligent GET/PUT operation, with additional application-specific record processing performed as a side-effect. This capability is an extension of the extant VAXELN synchronization features.

- Modularity. The server model is the epitome of modularity. Writing code under VAXELN to communicate with a server process is not much harder than writing code to call a subroutine. An additional benefit of the server model is a natural consequence of VAXELN: the server and application programs can be moved around the local network without any changes. This freedom allows you to easily balance the resource requirements of your applications program across the nodes in your network. However, servers need not be network-wide resources; implementing node-local servers under VAXELN is a trivial modification to the more common network-wide server. In the server example shown in this section, the only thing making the server node-local rather than network-wide is the scope of the NAME variable, SERVER$PORT.
- Reliability. The server and application communicate with each other through a circuit, a reliable communication mechanism.

Solution

The example programs in this section show the design of a simple network-wide multi-thread server. In order to keep the emphasis on the basic framework of the server, the example's function is simple: records of ASCII text sent to the server are converted to upper case and are sent back to the application.

The server job has the *Init required* attribute set in it's System Builder data file; this allows the master process to create the global NAME object (used by applications to locate the server's input port) before any application has a chance to execute.

After initialization, the server's master process simply waits for incoming connection requests on the port associated with the global name SERVER$PORT. For each such request it receives, the master process creates a process to handle the complete server dialogue and connects the circuit from the application to the new subprocess.

The code for the subprocess is also quite simple. In the example server, the logical end-of-dialogue is defined as the receipt of a null record by the server. The code is a basic structure of looping until a null record is received, translating records and retransmitting them back to the application.

Also in the example server is an additional bit of logic to implement a rudimentary timeout capability. If the server does not receive a record from the application before the timeout expires, the server assumes that the application has implictly terminated the dialogue. In an actual application however, a timeout's lapse should probably cause the output of an error message or some other abnormal event; the correct behavior in this situation is highly dependent upon the application.

What follows on the next several pages are a Pascal example, a Pascal sample application, a C example, and a C sample application.

Note that either of the server examples can be used with either of the sample applications; the examples implement identical capabilities.

The following is a listing of the System Builder data file used to build a system containing the sample server and sample application pair:

```
program application9a /initialize
program application9b
```

C Example

The following is a listing of the example written in C (application9a.c).

```
#module multiple_circuit

#include $vaxelnc
#include ctype
#include descrip

/*
 *
 * Abstract:
 *
 *      This module shows an example of how a typical server is
 *      implemented using an individual process to send each
 *      incoming circuit request to the server's global port,
 *      named SERVER$PORT.  This module demonstrates how
 *      the master process dispatches incoming circuit requests
 *      to the subordinate server processes.
 *
 */

/*
 *      Job-wide declarations:
 */

LARGE_INTEGER timeout_interval;

multiple_circuit()
{

/*
 * Functional description:
 *
 *      This is the master process for the server example.
 *      It simply listens for circuit requests from remote
 *      processes and creates a subprocess to handle each
 *      request.
 *
 * Inputs:
 *
 *      Incoming circuit connection requests to
 *      the global port, named SERVER$PORT.
 *
```

```
 * Outputs:
 *
 *      All incoming requests are handled by creating
 *      a subprocess to satisfy each request.
 *
 */

/*
 *      Master-process-local variable declarations:
 */

PORT    *circuit_port;
NAME    global_port_name;
PORT    master_process_job_port;
static  $DESCRIPTOR(server_name,"SERVER$PORT");
void    server$process();
int     status;
static  $DESCRIPTOR(timeout_string,"   0 00:10:00.00");
PROCESS subprocess;

/*
 *      M A S T E R   P R O C E S S   I N I T I A L I Z A T I O N:
 *
 *      Begin by creating a name for this job's port.
 *      If the name already exists, a server process
 *      already exists; simply exit.
 */

ker$job_port(NULL, &master_process_job_port);

ker$create_name(&status,
          &global_port_name,
          &server_name,
          &master_process_job_port,
          NAME$UNIVERSAL);

if (!(status&1))
      ker$exit(NULL, 1);

/*
 *      Compute 10-minute timeout constant used by subprocesses.
 */

timeout_interval = eln$time_value(&timeout_string);


/*
 *      The initialization is done; inform VAXELN.
 */
```

```
ker$initialization_done(NULL);


/*
 *      M A S T E R   P R O C E S S   M A I N L I N E   C O D E:
 *
 *      Loop indefinitely waiting for a remote circuit request.
 *      When one is received, create a port to handle the
 *      circuit and try to establish the circuit with the
 *      sender.
 *
 *      If the circuit can be established, create a process
 *      to service this circuit and pass this newly created port
 *      to the process as a parameter.
 *
 *      If the circuit cannot be established, simply delete
 *      the new port and continue looping, waiting for requests.
 */

for(;;)
      {

      /*
       *      Wait for any requests on the job port.
       */

      ker$wait_any(NULL, NULL, NULL, &master_process_job_port);

      /*
       *      Allocate a new port and create the PORT object.
       */

      circuit_port = calloc(1,sizeof (PORT));
      ker$create_port(NULL, circuit_port, 4);

      /*
       *      Setup the circuit using the new port.
       */

      ker$accept_circuit(&status,
                    &master_process_job_port,
                    circuit_port,
                    FALSE,
                    NULL,
                    NULL);
      if (status&1)
            {
```

```
        /*
         *      Start the server process and
         *      pass it to the circuit port.
         */

        ker$create_process(NULL,
                    &subprocess,
                    server$process,
                    NULL,
                    circuit_port);

        /*
         *      Note that it is now the responsibility of
         *      the subprocess to delete the PORT object
         *      and deallocate the port variable memory at
         *      the completion of the server's dialogue
         *      with the remote application. Of course,
         *      this house-cleaning must also be done if
         *      the circuit is broken due to error.
         *
         *      Now lower the process priority of the
         *      created subprocess to be just BELOW the
         *      priority of the master process; this
         *      ensures that none of the created
         *      subprocesses ever prevent the master
         *      process from servicing connection requests.
         */

        ker$set_process_priority(NULL, subprocess, 9);
        }
    else
        {

        /*
         *      The connect failed; delete and
         *      deallocate the PORT object.
         */

        ker$delete(NULL, circuit_port);
        cfree(circuit_port);
        }
    }
}

void server$process(circuit_port) PORT *circuit_port;
{

/*
 *    S U B P R O C E S S   M A I N L I N E   C O D E
 *
```

```
 * Routine description:
 *
 *      This is the entry routine for a separate process that
 *      is created to handle an incoming connection request.
 *
 *      In this example, the service performed by the server,
 *      and the protocol observed by the two circuit partners,
 *      is vastly simplified to keep the example small and
 *      understandable.
 *
 *      The protocol is simple: Messages containing text strings
 *      are sent from the "application" (the other half of the
 *      circuit) to this process (the "server").  The server
 *      processes each message by converting all the lower-case
 *      letters in the string to upper-case, and transmiting
 *      the converted text back to the application.  The
 *      application terminates the exchange by sending a record
 *      consisting of the null string.
 *
 *      A receive timeout is built into this server to add a
 *      little realism to a simplified example.  If the timeout
 *      expires, the server abandons the circuit as if the
 *      exchange had been terminated normally.  In an actual
 *      application, some further application-specific error
 *      processing, such as printing a diagnostic message, would
 *      most likely occur.
 *
 * Inputs:
 *
 *      circuit_port - Circuit upon which a request
 *                     has been accepted.
 *
 * Outputs:
 *
 *      The incoming request is handled.
 *
 */

/*
 *      Process-local variable declarations:
 */

BOOLEAN                 done = FALSE;
int                     i,status,wait_result,message_size;
MESSAGE                 message_id;
VARYING_STRING(80)      *message_ptr;

/*
 *      Loop until:
 *
```

```
 *              Receive timeout occurs
 *                      or
 *              Receive error occurs
 *                      or
 *              Null string is received from application
 *
 */

while(!done)
      {

      /*
       *      Wait for the port or a timeout.
       */

      ker$wait_any(NULL,
              &wait_result,
              &timeout_interval,
              circuit_port);

      /*
       *      If the result of the wait service was 0,
       *      the wait terminated because of a timeout.
       */

      if (wait_result == 0)
            done = TRUE;
      else
            {

            /*
             *      Otherwise, a message has been sent to
             *      the port.  Receive the message.
             */

            ker$receive(&status,
                    &message_id,
                    &message_ptr,
                    &message_size,
                    circuit_port,
                    NULL,
                    NULL);
            if (!(status&1))
                  done = TRUE;
            else
                  {
                  if (message_ptr->count == 0)
                        done = TRUE;
                  else
                        {
```

```
                                        /*
                                         *      A nonzero-length string has
                                         *      successfully been received.
                                         *      Convert the string to upper
                                         *      case.
                                         */

                                        for(i=0; i<message_ptr->count; i++)
                                            message_ptr->data[i] =
                                                 _toupper(message_ptr->data[i]);
                                        ker$send(NULL,
                                               message_id,
                                               message_size,
                                               circuit_port,
                                               NULL,
                                               FALSE);
                                        }
                                  }
                            }
                      }

      /*
       *      The exchange has terminated; delete the port,
       *      deallocate the local port storage, and exit.
       */

      ker$delete(NULL, circuit_port);
      cfree(circuit_port);

      }
```

Sample Application

The following is a listing of a sample application written in C (application9b.c).

```
#module multiple_circuit_sender

#include $vaxelnc
#include descrip
#include stdio

/*
 *
 * Abstract:
 *
 *      This module shows an example of a simple terminal-driven
 *      application that makes use of the server example program
 *      described above.
 *
 *      The application reads a line from the terminal and
 *      passes the line to the server for processing.  The
 *      processed line is read back from the server and
 *      displayed at the terminal.
 *
 *      The process continues until the user enters a blank
 *      line, which is the protocol established in the server as
 *      the "end-of-dialogue" marker.
 *
 */

multiple_circuit_sender()
{

/*
 *      Variable declarations:
 */

PORT                    circuit_port;
static                  $DESCRIPTOR(destination_name,"SERVER$PORT");
int                     discard;
BOOLEAN                 done = FALSE;
MESSAGE                 message_id;
VARYING_STRING(80)      *message_ptr;
```

```
/*
 *      Start by connecting our job port to the sample server,
 *      using the job port's universal name, SERVER$PORT.
 *      On error, exit the job with appropriate status.
 */

ker$job_port(NULL, &circuit_port);
ker$connect_circuit(NULL,
                &circuit_port,
                NULL,
                &destination_name,
                FALSE,
                NULL,
                NULL);

/*
 *      Print the prompt for user input.
 */

printf("\nEnter your input data.\n");
printf("Terminate your input by entering a blank line.\n");

/*
 *      Loop for each nonblank line entered.  Send it
 *      to the server for processing, read it back from
 *      the server, and print it.
 */

while(!done)
      {
      ker$create_message(NULL,
                         &message_id,
                         &message_ptr,
                         sizeof (*message_ptr));
      gets(message_ptr->data);
      message_ptr->count = strlen(message_ptr->data);
      if (message_ptr->count == 0)
             done = TRUE;

      ker$send(NULL,
               message_id,
               sizeof (*message_ptr),
               &circuit_port,
               NULL,
               FALSE);
```

```
        if (!done)
                {
                ker$wait_any(NULL,
                             &discard,
                             NULL,
                             &circuit_port);
                ker$receive(NULL,
                            &message_id,
                            &message_ptr,
                            &discard,
                            &circuit_port,
                            NULL,
                            NULL);
                printf("%.*s\n",
                       message_ptr->count,
                       message_ptr->data);
               }
        }
    }
```

Pascal Example

Below is a listing of the example written in PASCAL (application9c.pas).

```
module multiple_circuit;

{++
{
{
{ Abstract:
{
{      This module shows an example of how a typical server is
{      implemented, in Pascal, using an individual process to
{      service each incoming circuit request sent to the server's
{      global port, named SERVER$PORT.  This module
{      demonstrates how the master process dispatches incoming
{      circuit requests to the subordinate server processes.
{
{
{--}

{
{      Job-wide declarations:
{}

var
       timeout_interval: large_integer;
```

```
program multiple_circuit;

{++
{
{ Functional description:
{
{      This is the master process for the server example.
{      It simply listens for circuit requests from remote
{      processes and creates a subprocess to handle each
{      request.
{
```

```
{ Inputs:
{
{     Incoming circuit connection requests to the global port,
{     named SERVER$PORT.
{
{ Outputs:
{
{     All incoming requests are handled by creating a
{     subprocess to accomplish each request.
{
{--}

{
{     Master-process-local variable declarations:
{}

var
      master_process_job_port: port;
      circuit_port: ^port;
      global_port_name: name;
      status: integer;
      subprocess: process;
begin

{
{     M A S T E R   P R O C E S S   I N I T I A L I Z A T I O N:
{
{     Begin by creating a name for this job's port.  If the
{     name already exists, there is already a server process
{     in existence; simply exit.
{}

job_port(master_process_job_port);

create_name(global_port_name,
            'SERVER$PORT',
            master_process_job_port,
            table := name$universal,
            status := status);

if not odd(status)
then
      exit(exit_status := 1);

{
{     Compute 10-minute timeout constant used by subprocesses.
{}

timeout_interval := time_value('   0 00:10:00.00');
```

```
{
{     The initialization is done; inform VAXELN.
{}

initialization_done;
{
{     M A S T E R   P R O C E S S   M A I N L I N E   C O D E:
{
{     Loop indefinitely waiting for a remote circuit request.
{     When one is received, create a port to handle the
{     circuit and try to establish the circuit with the
{     sender.
{
{     If the circuit can be established, create a process
{     to service this circuit and pass this newly created port
{     to the process as a parameter.
{
{     If the circuit cannot be established, simply delete
{     the new port and continue looping, waiting for requests.
{
{}

while true do
      begin

      {
      {     Wait for any requests on the job port.
      {}

      wait_any(master_process_job_port);

      {
      {     Allocate a new port and create the port object.
      {}

      new(circuit_port);
      create_port(circuit_port^,limit := 4);

      {
      {     Setup the circuit using the new port.
      {}

      accept_circuit(master_process_job_port,
                     connect := circuit_port^,
                     status := status);

      if odd(status)
      then
            begin
```

```
        {
        {     Start the server process and
        {     pass it to the circuit port.
        {}

        create_process(subprocess,
                       server$process,
                       circuit_port);

        {
        {     Note that it is now the responsibility
        {     of the subprocess to delete the PORT
        {     object and deallocate the port variable
        {     memory at the completion of the server's
        {     dialogue with the remote application.
        {     Of course, this house-cleaning must also
        {     be done if the circuit is broken due to
        {     error.
        {
        {     Now, lower the process priority of the
        {     created subprocess to just BELOW the
        {     priority of the master process; this
        {     ensures that none of the created
        {     subprocesses ever prevent the master
        {     process from servicing connection
        {     requests.
        {}

        set_process_priority(subprocess, 9)
        end
    else
        begin

        {
        {     The connect failed; delete and
        {     deallocate the PORT object.
        {}

        delete(circuit_port^);
        dispose(circuit_port)
        end
    end;
end.
```

```
process_block server$process(circuit_port: ^port);

{
{     S U B P R O C E S S   M A I N L I N E   C O D E
{
{ Routine description:
{
{     This is the entry routine for a separate process that
{     is created to handle an incoming connection request.
{
{     In this example, the service performed by the server,
{     and the protocol observed by the two circuit partners,
{     is vastly simplified to keep the example small and
{     understandable.
{
{     The protocol is simple: Messages containing text strings
{     are sent from the "application" (the other half of the
{     circuit) to this process (the "server").  The server
{     processes each message by converting all the lower-case
{     letters in the string to upper-case and then transmits
{     the converted text back to the application.  The
{     application terminates the exchange by sending a record
{     consisting of the null string.
{
{     A receive timeout is built into this server to add a
{     little realism to a simplified example.  If the timeout
{     expires, the server abandons the circuit as if the
{     exchange had been terminated normally.  In an actual
{     application, some further application-specific error
{     processing, such as printing a diagnostic message, would
{     most likely occur.
{
{ Inputs:
{
{     circuit_port - Circuit on which a request
{                    has been accepted.
{
{ Outputs:
{
{     The incoming request is handled.
{
{--}

{
{     Process-local variable declarations:
{}

var
     done: boolean := false;
     status, wait_result: integer;
```

```
      message_id: message;
      message_ptr: ^varying_string(80);

begin

{
{     Loop until:
{
{                 Receive timeout occurs
{                         or
{                 Receive error occurs
{                         or
{                 Null string is received from application
{
{}

while not done do
      begin

      {
      {     Wait for the port or a timeout.
      {}

      wait_any(circuit_port^,
               result := wait_result,
               time := timeout_interval);

      {
      {     If the result of the wait service was 0,
      {     the wait terminated because of a timeout.
      {}

      if wait_result = 0
      then
            done := true
      else
            begin

            {
            {     Otherwise, a message has been sent to
            {     the port.  Receive the message.
            {}

            receive(message_id,
                    message_ptr,
                    circuit_port^,
                    status := status);
            if not odd(status)
            then
                  done := true
```

```
                else
                      begin
                      if length(message_ptr^) = 0
                      then
                            done := true
                      else
                            begin

                            {
                            {     A nonzero-length string
                            {     has successfully been
                            {     received.  Convert the
                            {     string to upper case.
                            {}

                            message_ptr^ :=
                                translate_string(message_ptr^,
                                'ABCDEFGHIJKLMNOPQRSTUVWXYZ',
                                oldchars :=
                                    'abcdefghijklmnopqrstuvwxyz');
                            send(message_id,
                                 circuit_port^)
                            end;
                      end;
                end;
          end;

    {
    {     The exchange has terminated; delete the port, deallocate
    {     the local port storage, and exit.
    {}

    delete(circuit_port^);
    dispose(circuit_port)

    end;
    end;
```

Sample Application

The following is a listing of a sample application written in PASCAL (application9d.pas).

```
module multiple_circuit_sender;

{++
{
{
{ Abstract:
{
{      This module shows an example of a simple terminal-driven
{      application that makes use of the server example program
{      described above.
{
{      The application reads a line from the terminal and passes
{      the line to the server for processing.  The processed line
{      is read back from the server and displayed at the
{      terminal.
{
{      The process continues until the user enters a blank line,
{      which is the protocol established in the server as the
{      "end-of-dialogue" marker.
{
{
{--}

program multiple_circuit_sender;

{
{      Variable declarations:
{}

var
      circuit_port: port;
      done : boolean := false;
      message_id: message;
      message_ptr: ^varying_string(80);
begin

{
{      Start by connecting our job port to the sample server,
{      using the job port's universal name, SERVER$PORT.
{      On error, exit the job with appropriate status.
{}
```

```
job_port(circuit_port);
connect_circuit(circuit_port,
                destination_name := 'SERVER$PORT');

{
{     Print the prompt for user input.
{}

writeln;
writeln('Enter your input data.');
writeln('Terminate your input by entering a blank line.');
writeln;

{
{     Loop for each nonblank line entered.  Send it to the
{     server for processing, read it back from the server,
{     and print it.
{}

while not done do
      begin
      create_message(message_id, message_ptr);
      readln(message_ptr^);
      if length(message_ptr^) = 0
      then
            done := true;

      send(message_id, circuit_port);

      if not done
      then
            begin
            wait_any(circuit_port);
            receive(message_id, message_ptr, circuit_port);
            writeln(message_ptr^)
            end;
      end;
end;
end;
```

Application 10
Self-Defining Data Structures

Problem

How do you neatly access self-defining data structures using VAXELN Pascal? A self-defining data structure is one in which the content of one field determines the size of one or more following fields.

Solution

The VAXELN Pascal concept of flexible types, together with the WITH-AS statement, provides a powerful tool to easily access self-defining data structures. The general strategy is to define a flexible "template" type that consists of a variable number of fill bytes followed by a series of data items known to occur together.

The example in this section shows the use of this technique to access the contents of a structure consisting of variably sized strings, along with some data pertaining to each string. The example shows the construction of a routine to walk through such a structure and access all the data.

To build the sample application, use the following commands:

```
$ epascal application10 + eln$:rtlobject/lib
$ link/nosysshr application10 + eln$:rtlshare/lib +-
  eln$:rtl/lib
$ ebuild/noedit application10
```

The sample application can then be loaded into a target machine and executed. The data file must contain information for EBUILD, as follows:

```
characteristic /noconsole /nofile /noserver
program application10 /debug
```

Example

The following is a listing of the example written in Pascal (application10.pas).

```
PROGRAM test(output);

{
{       This program demonstrates accessing a self-defining
{       data structure using a flexible type and a WITH-AS
{       statement.  This self-defining data structure
{       consists of a block of bytes containing repeated
{       instances of name and age data.
{
{       The first byte is an unsigned integer giving the
{       count of bytes in the immediately following string,
{       which is a person's name.  The name string is followed
{       by an unsigned byte giving the person's age.
{       The last byte in the data block is a zero length
{       for a name string.  (The name string is, of course,
{       nonexistent.)
{}

{
{       Below is an example data block.  It would be more
{       common to have this data block read from a disk.
{}

VAR
        data_block : array[1..57] of char := (
          chr(4), 'F','r','e','d', chr(19),
          chr(3), 'B','o','b', chr(26),
          chr(6), 'M','a','r','t','h','a', chr(32),
          chr(4), 'J','a','c','k', chr(14),
          chr(6), 'V','i','c','t','o','r', chr(52),
          chr(4), 'D','a','w','n', chr(17),
          chr(6), 'M','a','r','c','i','a', chr(29),
          chr(7), 'B','a','r','b','a','r','a', chr(5),
          chr(0));

TYPE
        unsigned_byte = [BYTE]0..255;

{
{        Below is the template type that will be used
{        to access data in data blocks.
```

```
{}

      template(m,n : integer) = packed record
          fill : byte_data(m);
          name : string(n);
          age : unsigned_byte;
          next_length : unsigned_byte;
          end;

PROCEDURE print_block(blk_ptr : ^anytype);

{
{       This procedure prints out the contents of a
{       data block whose address is given by blk_ptr.
{}

var
       skip_count: integer;
       string_length: integer;

begin

{
{       Set the length of the first name string and
{       initialize the number of bytes of data to skip.
{}

string_length := blk_ptr^::unsigned_byte;
skip_count := 1;

{
{       Output a header.
{}

writeln('Contents of data block:');
writeln;

while string_length > 0 do
      begin
      with x as blk_ptr^::template(skip_count,string_length) do
            begin

            {
            {       First, write the name and age.
            {}

            write(x.name);
            writeln(', age ', x.age:1);
```

```
            {
            {       Increment the skip_count to skip over
            {       the name string, as well as the count
            {       byte and age byte.  Then, get the
            {       string length of the next name string.
            {}

            skip_count := skip_count + string_length + 2;
            string_length := x.next_length
            end;
      end;
end;

{
{      Main program starts here.
{}

begin

print_block(address(data_block))

end;
```

Application 11
VAXELN Interface to VAX/VMS

Problem

How does your VAXELN system communicate with a VAX/VMS system?

Solution

First, the following command procedure (filename time.com) must be present in the default DECnet directory on the VAX/VMS machine:

```
$ open/write fred sys$net
$ time = f$time()
$ write fred time
$ close fred
```

Then, when the VAXELN system connects to the target machine, this command file is executed; this execution causes the VAXELN system to receive a message containing the current time. Another way to accomplish this is having the command file run a program that opens the file and writes to it.

To build the sample application, use the following commands:

```
$ epascal application11 + eln$:rtlobject/lib
$ link/nosysshr application11 + eln$:rtlshare/lib +-
  eln$:rtl/lib
$ ebuild/noedit application11
```

The sample application can then be loaded into a target machine and executed. The data file must contain information for EBUILD, as follows:

```
characteristic /noconsole /nofile /noserver
program APPLICATION11 /debug
```

Example

The following is a listing of the example written in Pascal (application11.pas).

```
module time_req_test;

{
{     This is a VAXELN application that initiates a connection
{     with a COM file on a remote VMS system to request the
{     time of day.  Note the following:
{
{         Change 10.172 to the node address
{         of the VMS system.
{
{         The command procedure TIME.COM must be present in the
{         default DECnet directory of the machine running VMS.
{}

var
      this_port: port;
      this_message: message;
      these_data: ^string(32);
      actual_time: string(32);
      current_time: large_integer;

program time_request(input, output);

begin

{
{     First, create the port that will be used to communicate
{     with the VMS system.
{}

create_port(this_port);

{
{     Executing the connect_circuit causes TIME.COM in the
{     default DECnet directory on the VMS system to be run.
{     10.172 is the number of the node on which VMS was running.
{     TIME.COM itself looks like this:
{
{     $ open /write fred sys$net
{     $ time = f$time()
{     $ write fred time
```

```
{       $ close fred
{}

connect_circuit(this_port, destination_name := '10.172::time');
writeln('Connected to the VMS system');
wait_any(this_port);

{
{       Read the message and display it.
{}

receive(this_message, these_data, this_port);
writeln('The message was "', these_data^, '"');
disconnect_circuit(this_port);

{
{       Set the time; then get the time to
{       double check that everything worked.
{}

actual_time := substr(these_data^, 1, 23);
current_time := time_value(actual_time);
set_time(current_time);
get_time(current_time);
actual_time := time_string(current_time);
writeln('The current time is ', actual_time);
writeln('Done')

end.
end;
```

Application 12
VAXELN Time Routines

Problem

How do you manipulate time data in VAXELN?

Solution

VAXELN provides the SET_TIME and GET_TIME routines to set and retrieve the system time; they use large integers to manipulate times. The TIME_VALUE, TIME_STRING, and TIME_FIELDS routines are also provided; they convert LARGE_INTEGERs, representing time, to and from strings. The example in this section demonstrates the use of all of these routines.

To build the sample application, use the following commands:

```
$ epascal application12 + eln$:rtlobject/lib
$ link/nosysshr application12 + eln$:rtlshare/lib +-
  eln$:rtl/lib
$ ebuild/noedit application12
```

The sample application can then be loaded into a target machine and executed. The data file must contain information for EBUILD, as follows:

```
characteristic /noconsole /nofile /noserver
program application12 /debug
```

Example

The following is a listing of the example written in Pascal (application12.pas).

```
module timer_test;

{
{       This module demonstrates the use of
{       the VAXELN time routines.
{}

var
        elapsed_time, actual_time, current_time: large_integer;
        result: integer;
        time_rec: time_record;
        a_time_string: varying_string(80);
        elapsed_time_string: varying_string(12);

program timer(input,output);

var
        i: integer;

begin
writeln('Program starting');

{
{       Set the date and time.
{}

write('Enter today''s date and time: ');
readln(a_time_string);
current_time := time_value(a_time_string);
set_time(current_time);
writeln('The date and time have been set');

{
{       Use the time_fields function to convert
{       current_time back to a string.
{}

time_rec := time_fields(current_time);
with time_rec do
writeln(day:1, '/', month:1, '/', year:1, ' ',
        hour:2, ':', minute:2, ':', second:2, '.', hundredth:2);
```

```
{
{     Loop 5 times to display the time every 5 seconds.
{}

for i := 1 to 5 do
      begin

      {
      {     Set up for a delay of 5 seconds.
      {}

      elapsed_time := time_value('0 ::5');
      get_time(actual_time);

      {
      {     Wait for 5 seconds to go by.
      {}

      wait_any(time := elapsed_time, status := result);
      get_time(current_time);

      {
      {     Compute the elapsed time.
      {     Display the actual and elapsed time.
      {}

      elapsed_time := current_time - actual_time;
      a_time_string := time_string(elapsed_time);
      elapsed_time_string := substr(a_time_string, 12);
      writeln('The actual time was ', time_string(actual_time));
      writeln;
      writeln('The elapsed time is ', elapsed_time_string);
      writeln;
      writeln
      end;
end;
end.
```