MIKADOS

Pascal Standard Assembler Package

Dansk Data Elektronik ApS
9 May 1980

Author: Claus Tøndering

Table of contents

1. Introduction.

This manual describes the Pascal Standard Assembler Package which is a set of procedures and functions that make certain facilities in the MIKADOS Operating System available to the Pascal programmer.

As assembler programmers will know, the MIKADOS Operating System offers a number of facilities that normally cannot be accessed from programs written in Pascal. The Pascal Standard Assembler Package contains a number of interface routines which make many of the advanced MIKADOS assembler facilities available to the Pascal programmer.

The Pascal Standard Assembler Procedures and Functions fall into six groups:

- Direct input procedures.
- Message exchange procedures.
- Process related procedures.
- Memory allocation procedures.
- Resource reservation procedures.
- File handling procedures.

2. Overview.

2.1. Direct input procedures.

These procedures enable the programmer to override the normal Pascal console input conventions. In the so-called direct input mode keystrokes are transmitted directly to a user defined cyclic input buffer. The system does not by itself output any information to the terminal display in response to keyboard input. Thus the user is in full control of the meaning of the input keys and the resulting terminal output.

The direct input procedures may be used to construct full screen editors, input forms with protected fields and input checking, etc.

The direct input procedures are described in detail in chapter 3.

2.2. Message exchange procedures.

In many applications a programmer finds that a certain problem may be solved in a simpler way if he has two or more programs running simultaneously. This is particularly true in many real time control applications where console operator communication is required while process monitoring continues.

MIKADOS Pascal provides a standard subroutine (CHAIN) which can be used to initiate parallel programs. However, in many applications parallel cooperating programs have to exchange information (new alarm limits, valve settings etc.). The need therefore arises for a means by which simultaneous programs may communicate. The message exchange procedures enable parallel programs to exchange information.

The message exchange procedures are described in detail in chapter 4.

2.3. Process related procedures.

These procedures enable the programmer to change the priority of the calling program (process) and to get the address of the process control block of his program.

The process related procedures are described in detail in chapter 5.

2.4. Memory allocation procedures.

In a Pascal program the user may allocate memory areas dynamically using the NEW procedure. However, this procedure allocates memory in an area which is not accessible from other programs. The memory allocation procedures enable the user to allocate and deallocate memory areas which are accessible from all user programs. The memory thus allocated may be used for communication between parallel processes, particularly if large amounts of information are to be exchanged.

The memory allocation procedures are described in detail in chapter 6.

2.5. Resource reservation procedures.

In systems where many users share resources the need arises for a means to reserve and later release such resources so that the different users will not interfere with each other. The resource reservation procedures are designed to meet this need. They are particularly useful in controlling access to printers, data areas, file elements etc.

The resource reservation procedures are described in detail in chapter 7.

2.6. File handling procedures.

A number of MIKADOS file operations are not available using the standard Pascal procedures. The file handling procedures in the assembler package enable the user to rename files, read or write sequential (TEXT) files backwards, and purge existing files.

The file handling procedures are described in detail in chapter 8.

2.7. How to use the procedures.

Whenever one of the Pascal Standard Assembler Package procedures is used in a Pascal program, the user must declare this procedure with the attribute EXTERNAL in his Pascal program as shown in the appropriate procedure description in this manual. He must then compile the Pascal program using the E (external) option and link the resulting relocatable code together with the relocatable code for the Pascal Standard Assembler Package procedures to get a special interpreter which is used to execute the Pascal program.

Linking to external procedures is described in the Pascal User´s Guide section 8.3.

3. Direct input procedures.

These procedures can be used to perform character-by-character input from a terminal.

In certain applications the programmer may wish to override the standard input conventions of Pascal. He may wish to change the meaning of certain keys, or he may wish to perform a whole series of operations upon depression of a single key on the terminal keyboard. Such operations cannot be carried out using the normal Pascal input routines (READ, READLN, and EDIT), as they always require that a "RETURN" be entered before any input is made available to a Pascal program.

The "direct input" facility is designed to meet this need. The system normally operates in what is called "normal input mode". By issuing a call to the SETUP routine, as described below, the system is put in "direct input mode", where the program may read the keys depressed on the keyboard one by one as they are entered. In direct input mode echoing of input characters does not occur automatically.

When operating in direct input mode the READ, READLN, and EDIT routines should not be used. The WRITE and WRITELN routines may be used without restrictions.

In direct input mode the system uses a cyclic buffer to store the value of each key as it is entered from the keyboard. The keys are not echoed on the screen; this must be done by the program. The direct input routines can get the key values one by one from the cyclic buffer. The length of the cyclic buffer determines the number of keystrokes the keyboard operator may be ahead of the program. Buffer wraparound is handled automatically by the system.

If the keyboard operator is ahead of the program by more keystrokes than the cyclic buffer can contain, erroneous characters may be returned to the program. No error condition is reported by the direct input routines.

3.1. SETUP - Initiate direct input.

This procedure is used to put the system in direct input mode.

The calling program should contain the following declarations:

```
CONST LNGTH=10;
TYPE DIRECTBUF = PACKED ARRAY (1..LNGTH) OF CHAR;
PROCEDURE SETUP(VAR BUFFER: DIRECTBUF;
                    LENGTH: INTEGER);
EXTERNAL;
```

A call of SETUP places the system in direct input mode. The user must supply the system with a character array, which will be used as the cyclic input buffer described above. This buffer is specified as a parameter to the procedure. The user must decide upon a suitable buffer length which must be used in the buffer declaration (the constant LNGTH in the declaration above) and in the procedure call.

During execution of this procedure a dynamic data area of 6 bytes is allocated for internal system use. If this space allocation fails, the procedure returns with IORESULT set to -10.

Note 1: READ, READLN, and EDIT should not be used while the system operates in direct input mode.

Note 2: The system uses three bytes of the buffer for

overhead. Thus in the declaration above the actual length of the cyclic input buffer will be only 7 bytes.

3.2. AVAIL - Test for availability of keyboard input.

This function tests if the cyclic input buffer contains one or more unprocessed characters.

The calling program should contain the following declaration:

```
FUNCTION AVAIL: BOOLEAN;
EXTERNAL;
```

This function returns the value TRUE if the cyclic input buffer contains one or more unprocessed characters. More specifically TRUE is returned if the cyclic input buffer is not empty. Otherwise FALSE is returned.

AVAIL may be used to test if a call of NEXT would cause the calling process to be suspended.

3.3. NEXT - Get next character from terminal.

This function returns the next character from the cyclic input buffer.

The calling program should contain the following declaration:

```
FUNCTION NEXT: CHAR;
EXTERNAL;
```

A call of NEXT will return the next key value in the input buffer to the program. If the input buffer is empty the program waits until a key is depressed.

3.4. FINIS - Terminate direct input.

This procedure causes the system to revert to normal input mode.

The calling program should contain the following declaration:

```
PROCEDURE FINIS;
EXTERNAL;
```

Calling FINIS terminates the direct input mode, and restores the normal input mode.

Upon return from FINIS the program may use the cyclic input buffer for other purposes.

4. Message exchange procedures.

Before using any of the message exchange routines the user must declare two types, one which is used to identify the other participant in the communication and one which describes the kind of information to be exchanged.

```
TYPE PCB = ^INTEGER;
     MESSAGE = STRING(40);
```

The type PCB describes a pointer to the process control block of a process with which messages are exchanged.

The type MESSAGE defines the structure of the information to be exchanged. In this example strings of length 40 bytes are to be exchanged, but in other applications the user may want to exchange, for instance, integers, arrays of Booleans, or records (structured data). In such cases the definition of the type MESSAGE should be changed accordingly.

Note: The length of the message must not exceed 255 bytes. If the length is less than or equal to 10 bytes the message exchange procedures work faster.

The programmer must ensure that a program has received all messages sent to it before its execution is terminated. Stray (undelivered) messages may reappear at odd places and will cause system malfunction. In such situations it will be necessary to restart the system.

Assembler programmers should note that inter-process communication in Pascal always occurs using the main process semaphore.

4.1. SENDM - Send a message.

This procedure is used to send a message to another process.

The calling program should contain the following declaration:

```
PROCEDURE SENDM(     RECEIVER: PCB;
                VAR CONTENTS: MESSAGE;
                    LENGTH: INTEGER;
                VAR STATUS: INTEGER );
EXTERNAL;
```

This procedure sends a message to another process. If the receiving process is waiting for a message, its execution is resumed. Otherwise the message is put into a queue waiting for the receiving process to fetch it.

The parameters are:

- RECEIVER: a pointer to the PCB of the receiving process.
- CONTENTS: the message to be sent. Note that this parameter is declared as a VAR parameter even though no result is returned in it. This is necessary to ensure a uniform transfer of parameters to the assembler procedure independent of the actual type of the message.
- LENGTH: the length of the message in bytes. LENGTH should not exceed 255.
  Note 1: The function SIZEOF may be used here to compute the length of the message.
  Note 2: If the message is of type STRING the value supplied in this parameter must be one greater than the actual (dynamic) length of the string, as system overhead occupies one extra byte.

- STATUS: This variable will receive information about the outcome of the call of SENDM. During the operation of the procedure, memory space is allocated for the message (unless its length does not exceed 10 bytes). If this allocation fails an error is reported in the STATUS parameter:

  0 - operation completed successfully.
  1 - memory allocation failure; no message sent.
  2 - LENGTH is greater than 255.
  3 - No message buffer available in the system.

4.2. RECEIV - Receive a message.

The procedure RECEIV is used to receive a message.

The calling program should contain the following declaration:

```
PROCEDURE RECEIV( VAR SENDER: PCB;
                  VAR CONTENTS: MESSAGE;
                  VAR LENGTH: INTEGER );
EXTERNAL;
```

This procedure causes the calling process to wait for a message that has been sent to it by another process. If one or more messages are waiting in the message queue, the first message is made available to the calling process and execution continues immediately. If no messages are waiting, the execution of the process is suspended until a message is sent to it from another process.

The contents of the parameters on entry to RECEIV are not significant.
Upon return from RECEIV the parameters will contain:

- SENDER: a pointer to the PCB of the process from which the message was sent (and to which a possible answer message should be sent).
- MESSAGE: the message.
- LENGTH: the length in bytes of the message received.
  Note: The value returned in this parameter for a STRING variable is one greater than the actual number of characters received due to system overhead.

WARNING: Normally the type of the message in the sending and the receiving processes should be the same.
If the (static) length of the variable that receives the message is less than the length of the message unpredictable errors may occur, as the system does not check the length of the receiving variable.

4.3. WAITNG - Look for a message.

The function WAITNG is used to check the possible presence of a waiting message.

The program should include the following declaration:

```
FUNCTION WAITNG: BOOLEAN;
EXTERNAL;
```

This function returns FALSE if no messages are waiting for the calling process. If one or more messages are waiting, the value TRUE will be returned.

This function may be used if a process has useful work to do while waiting for a message and therefore cannot afford to be suspended as would be the case with RECEIV if no message was waiting.

4.4. Example.

The following is an example of the utilization of the message exchange procedures:

A key operator enters lines of text which are output to a disk as each line is entered. The program works thus:

Step 1: Read one line from terminal.
Step 2: Write line to disk.
Step 3: Go to step 1.

However, with this approach the operator may have to pause between input lines waiting for the file system to complete output of a line before the next line may be entered. This disadvantage can be avoided by using two parallel Pascal programs instead of one to perform the job:

Routine 1: Step 1: Read one line from terminal.
Step 2: Send line to routine 2.
Step 3: Go to step 1.

Routine 2: Step 1: Receive one line from routine 1.
Step 2: Write line to disk.
Step 3: Go to step 1.

The parallelism ensures that the key operator will always be able to enter lines. If the file system routine (routine 2) is

not ready to receive a line, the line is simply queued until routine 2 is ready to process it, and routine 1 is free to continue execution. The only requirement is that the average write time of the file system routine (routine 2) is less than the read time of routine 1, so that the queue of lines will not grow indefinitely.

The following two Pascal programs perform this task:

Routine 1:

```
PROGRAM READLIN;
TYPE PCB = ^INTEGER;
     MESSAGE = STRING(80);
VAR MES    : MESSAGE;
    R2     : PCB;
    STATUS: INTEGER;
PROCEDURE SENDM(     RECEIVER:PCB;
                VAR CONTENTS:MESSAGE;
                    LENGTH:INTEGER;
                VAR STATUS:INTEGER);
EXTERNAL;

BEGIN
    CHAIN('WRITDISK*2','',R2); (* START FILE SYSTEM ROUTINE *)
    REPEAT
        READLN;
        READ(MES);     (* GET ONE LINE OF INPUT *)
        REPEAT
            SENDM(R2,MES,LENGTH(MES)+1,STATUS)
                    (* SEND THE LINE TO THE WRITDISK ROUTINE *)
        UNTIL STATUS=0
    UNTIL MES(1)='>'          (* A > IN COLUMN 1 OF INPUT
                               IS USED TO TERMINATE EXECUTION *)
END.
```

Routine 2:

```
PROGRAM WRITDISK;
TYPE PCB = ^INTEGER;
     MESSAGE = STRING(80);
VAR MES    : MESSAGE;
    FINISH: BOOLEAN;
    LNG    : INTEGER;
    R1     : PCB;
PROCEDURE RECEIV(VAR SENDER:PCB;
                 VAR CONTENTS:MESSAGE;
                 VAR LENGTH: INTEGER);
EXTERNAL;

BEGIN
    FINISH:=FALSE;
    REPEAT
        RECEIV(R1,MES,LNG);
                (* RECEIVE MESSAGE FROM THE READLIN ROUTINE *)
        IF MES(1) <> '>' THEN
            (* write MES on disk *)
        ELSE
            FINISH:=TRUE;
    UNTIL FINISH;
END.
```

5. Process related procedures.

5.1. SETPR - Alter priority.

Every running program (process) has a priority assigned to it. This priority normally lies in the range from 4 to 7 (with 4 designating the highest priority and 7 designating the lowest priority). When a Pascal program is started a priority of 6 is automatically assigned to it. If the user requires the program to respond immediately to terminal input or other I/O activities he may wish to change this priority to 5 or even 4. On the other hand he may wish to have certain very CPU consuming programs running at priority 7, where they won't hamper the operation of the other system activities too much. The procedure SETPR is used to change the priority of the calling process dynamically.

The calling program should contain the following declarations:

```
TYPE PRIORANGE = 4..7;
PROCEDURE SETPR(PRIORITY: PRIORANGE);
EXTERNAL;
```

This procedure will change the priority of the calling process to the value specified by the parameter PRIORITY.

If the value of the parameter does not lie in the interval from 4 to 7, calling SETPR has no effect.

5.2. PROBLK - Determine address of Process Control Block.

In certain applications the user may wish to know the address of the Process Control Block (PCB) of a running program. It may be that he wants to pass this information on to other programs so that these may communicate with the original program.

The function PROBLK returns a pointer to the Process Control Block of the calling program.

The calling program should contain the following declarations:

```
TYPE PCB = ^INTEGER;
FUNCTION PROBLK: PCB;
EXTERNAL;
```

6. Memory allocation procedures.

The procedures ALLOCA and DEALLO are used to allocate and deallocate a data memory area for a program.

The memory area is always allocated outside the bank switch area, i.e. in a part of memory that is accessible from all running processes. This distinguishes the ALLOCA procedure from the NEW procedure as the latter allocates memory in the bank in which the process is running. This means that memory allocated by one process (using the ALLOCA procedure) may be referenced by another process, and thus we have an efficient method for passing large amounts of information from one process to another.

The total size of the allocatable area is, however, rather limited (currently approximately 270 bytes), so these procedures should not be used too extravagantly.

6.1. ALLOCA - Allocate memory.

The procedure ALLOCA allocates a data memory area.

The calling program should contain the following declarations:

```
TYPE STRING20 = STRING(20);
     STRINGPTR = ^STRING20;
     LENGTHRANGE = 1..2044;
PROCEDURE ALLOCA( VAR ADDRESS: STRINGPTR;
                      LENGTH: LENGTHRANGE;
                  VAR STATUS: INTEGER );
EXTERNAL;
```

The parameters are:

- ADDRESS: a pointer variable that receives the address of the allocated memory area. In the declaration above this parameter is specified as ^STRING20, but this is only one way of doing it. The user must decide for what purpose he intends to use the allocated memory and then declare the parameter accordingly.
- LENGTH: the length (in bytes) of the requested memory area.
- STATUS: a variable that receives information about the outcome of the call of ALLOCA. Upon return from the procedure STATUS will have the value
  - 0, if the allocation was successful.
  - 1, if a data area of the desired size is currently not available.
  - 2, if the size of the data area is illegal (less than 1 or greater than 2044).

6.2. DEALLO - Deallocate memory.

The procedure DEALLO deallocates a memory area that has been previously allocated by the procedure ALLOCA.

The calling program should contain the following declarations:

```
TYPE STRING20 = STRING(20);
     STRINGPTR = ^STRING20;
PROCEDURE DEALLO(     ADDRESS: STRINGPTR;
                  VAR STATUS: INTEGER);
EXTERNAL;
```

The parameters are:

- ADDRESS: a pointer variable containing the address of a memory area to be deallocated. In the declaration above this parameter is specified as ^STRING20, but this is only one possibility. The user must ensure that the type of the parameter corresponds to the type used when the area was allocated.
- STATUS: a variable that receives information about the outcome of the call of DEALLO. Upon return from the procedure STATUS will have the value
  - 0, if the deallocation was successful.
  - 1, if the data area was not allocated using the ALLOCA procedure.

7. Resource reservation procedures.

The resource administration system makes it possible for co-operating programs to establish a scheme for controlled access to serially reusable resources (e.g. line printers, main storage areas, or information in a data base) by providing procedures for the reservation and release of symbolic resources.

A resource may be reserved exclusively or non-exclusively. If a resource has been reserved exclusively, no other program is allowed to reserve that resource before it has been released. If a resource has been reserved non-exclusively other programs are allowed non-exclusive reservations of the resource.

The resources are identified by resource names consisting of 10 ASCII characters.

The system does not establish any connection between a symbolic resource and an actual physical resource in the system. The resource administration does not provide any protection in the case where a program uses a resource which it has not reserved previously.

Before using any of the symbolic resource reservation procedures the user must make the following type declaration:

```
TYPE NAME = STRING(10);
```

The type NAME specifies the name of a symbolic resource.

7.1. RESERV - Reserve a symbolic resource.

The procedure RESERV is used to reserve a symbolic resource.

The calling program should contain the following declaration:

```
PROCEDURE RESERV(     RESOURCE: NAME;
                      EXCLUSIVE, WAIT: BOOLEAN;
                  VAR STATUS: INTEGER);
EXTERNAL;
```

The parameters are:

- RESOURCE: the name of the symbolic resource to be reserved.
- EXCLUSIVE: TRUE if exclusive reservation is desired, FALSE otherwise.
- WAIT: TRUE if the calling program should wait until the resource is available, FALSE if the procedure should return immediately to the calling program with an appropriate status code if the resource is not available.
- STATUS: a variable that receives information about the outcome of the call of RESERV. Upon return from the procedure STATUS will contain
  - 0, if reservation was successful.
  - 1, if the resource was reserved by someone else.
  - 4, if no resource element is available. (A resource element is the internal system representation of a symbolic resource. Only a limited number of such elements are available.)

7.2. RELEAS - Release a symbolic resource.

The procedure RELEAS releases a symbolic resource that has been reserved previously by a RESERV call.

The calling program should contain the following declaration:

```
PROCEDURE RELEAS(    RESOURCE: NAME;
                 VAR STATUS: INTEGER);
EXTERNAL;
```

The parameters are:

- RESOURCE: the name of the symbolic resource to be released.
- STATUS: a variable that receives the information about the outcome of the RELEAS call. Upon return from the procedure STATUS contains
  - 0, if the release was successful.
  - 2, if the resource had not been reserved.

8. File handling procedures.

This chapter describes a number of MIKADOS file system procedures.

The procedures communicate I/O error conditions through the IORESULT function (see section 5.2.4 in the Pascal User´s Guide).

8.1. RENAMF - Rename a file.

The procedure RENAMF is used to give a MIKADOS file a new name.

The calling program should contain the following declaration:

```
PROCEDURE RENAMF(OLDNAME,NEWNAME: STRING);
EXTERNAL;
```

OLDNAME must contain

MIKADOS filename : disk identification : file type

and NEWNAME must contain

MIKADOS filename

(see section 5.2 in the Pascal User´s Guide).

The effect of calling RENAMF is that the file with the name and type specified in OLDNAME residing on the specified disk, will have its file name changed to the name specified in NEWNAME.

It is not possible to change the type of a file.

Renaming a file does not change the file contents.

Example:

```
VAR OLD,NEW:STRING;
BEGIN
.
.
OLD:='ALFA:P2:K';
NEW:='BETA'
RENAMF(OLD,NEW);
.
.
RENAMF('GAMMA:P5:P','DELTA');
.
.
END.
```

8.2. REDBAK - Read one record backwards.

This procedure is used to backspace a sequential file one record while reading the record.

The calling program should contain the following declaration:

```
PROCEDURE REDBAK( VAR FILEID: PHYLE;
                  VAR CONTENTS: STRING );
EXTERNAL;
```

FILEID identifies an open sequential file. Calling REDBAK will cause the file to be backspaced one record, that is, the record preceding the current record becomes the new current

record. The contents of the record accross which the system backspaces is placed in the parameter CONTENTS. The length of the string that is used as the CONTENTS parameter is set according to the length of the record read. If this length exceeds 80 bytes the call of REDBAK has no effect and IORESULT is set to 19.

8.3. WRIBAK - Write one record backwards.

This procedure is used to write a record backwards in a sequential file.

The calling program should contain the following declaration:

```
PROCEDURE WRIBAK( VAR FILEID: PHYLE;
                      CONTENTS: STRING );
EXTERNAL;
```

FILEID identifies an open sequential file. Calling WRIBAK will cause the contents of the parameter CONTENTS to be written backwards into the specified file, that is, a subsequent READLN(FILEID,...) call will read the record just written. The current length of the string CONTENTS determines the length of the record written. If this length exceeds 80 bytes, the call of WRIBAK has no effect and IORESULT is set to 19.

Note: If an attempt is made to write backwards a record that is longer than the available space in the file, the structure of the file is destroyed and IORESULT is set to 12.

8.4. PURGEF - Delete a file.

The procedure PURGEF is used to purge (delete) an existing MIKADOS file.

The calling program should contain the following declaration:

```
PROCEDURE PURGEF(FILENAME: STRING);
EXTERNAL;
```

FILENAME must contain

MIKADOS filename : disk identification : file type

(see section 5.2 in the Pascal User's Guide).

The effect of calling PURGEF is that the file with the name and type specified in FILENAME is purged from the specified disk.

An open file cannot be purged. The space occupied by the file is not released until the disk is compressed.

Example:

```
VAR PURGEFILE:STRING;
BEGIN
.
.
PURGEFILE:='GOAWAY:P4:K';
PURGEF(PURGEFILE);
.
.
PURGEF('BADFILE:P1:P');
.
.
END.
```