Supermax System V

Pascal C Interface

Dansk Data Elektronik A/S
23. Jun 1989

Version 1.2

Supermax System V

Manual Addendum for pta Version 2.2

Dansk Data Elektronik A/S
23. June 1989

To achieve the full documentation for the Pascal-assembler Programming Package Issue 2 Version 2, please replace the front page in Your Pascal C Interface Manual and the pages 1.1 - 1.2 and 4.13 - 4.17 with the pages enclosed.

Table of Contents.

1. Introduction.

This manual describes how to write external procedures and functions to Pascal programs. In addition it describes the standard external procedures and functions. The same external rutines can be used both from interpreted and compiled Pascal (Pascal to Assembler).

2. Writing External Procedures and Functions.

When writing Pascal programs it is possible to use, that is declare and call, external subroutines written in the language C. A set of standard external subroutines is supplied with the Pascal systems.

This chapter explains how to write and use your own external subroutines written in C.

2.1 Naming external subroutines.

An external subroutine must always be declared in the Pascal program in which it is called. This declaration is used by the Pascal compiler to perform syntaxcheck on the call of the subroutine. The Pascal declaration is similar to a usual declaration of a PROCEDURE or FUNCTION except that the body of the routine is substituted by the special symbol EXTERNAL (analogous to a FORWARD declaration).

The name of the subroutine will be recognized by the Pascal compiler using 14 significant characters. As C-subroutine names, when compiled by the C-compiler, are recognized using all characters as significant it is important that the name of the routine in the C-module does not exceed 14 characters.

Furthermore the Pascal compiler generates the names of external subroutines using small letters whereas the C-compiler generates the names using the actual characters.

Thus the name of an external subroutine used in the C-module may consist of at most 14 small letters.

2.2 Parameters to External Procedures and Functions.

When writing external subroutines it is necessary to know how the different types of parameters are given to the external subroutines. The least significant bit of a word is bit 0, the most is bit 15. The most significant byte is in the lowest address. The proper declaration to be used in C is given.

Boolean: One word. Bit 0 indicates the value - false=0,true=1. Use the type short.

Integer: One word, two's complement, capable of representing values in the range -32768..32767. Use the type short.

Longint: Two word, two's complement, capable of representing values in the range -2147483648..2147483647. Use the type int.

Scalar (user-defined): One word, in range 0..32767. Use the type short.

Char: One word with the least significant byte containing the character. Use the type short and use the least significant byte as an unsigned char. Using the type unsigned char will cause an error as this type only takes up one byte in C.

Real: Four words in IEEE representation. Use the type double.

Pointer: Two words containing an address. The value NIL is the address 1.

Set: 0..255 words. Sets are implemented as bit vectors, always with a lower index of zero. A set variable declared as SET OF M..N is allocated (N+15) DIV 16 words. If the bits and the words are numbered starting from 0 we have: Bit B of word W is 1 if the element W*16+B is a member of the set. Using this information the user should be able to find an appropriate type in C.

Records and arrays: Any number of words. Arrays are stored in row-major order (last index varying most rapidly). Packed arrays have an integral number of elements in each word as there is no packing across word boundaries (it is acceptable to have unused bits in each word). Using this information the user should be able to find an appropriate type in C.

Strings: 1..128 words. Strings are a flexible version of packed arrays of characters. A STRING(n) occupies (n div 2)+1 words. Byte 0 of a string is the current length of the string, and bytes 1..length(string) contain valid characters. Use the type unsigned char *.

Longstrings: 1..32767 bytes. A LONGSTRING(n) occupies ((n+1) div 2)+1 words. Byte 0 and 1 of a longstring is the current length of the longstring, and bytes 2..length(longstring) contain the valid characters. Use the type unsigned char *.

Cstrings: 1..32767 bytes. A CSTRING(n) occupies (n+1) div 2 words. Bytes 0..n-1 can contain valid characters. In C the last valid character is followed by the NULL character. Use the type unsigned char *.

Note: to match the C way of handling parameters all short types (16 bit) parameters will be sign extended to long (32 bit) when pushed onto the stack in the same way as C does. A formal parameter of type set to an external subroutine must be a variable parameter (var declared). Actual parameters of type record, array, string, longstring or string, and actual parameters, whose corresponding formal parameter is a variable parameter, are always passed to an external subroutine as a pointer to the variable.

Due to the difference in the internal handling of parameters in C and Pascal the order in which the parameters are written in the C-declaration must be the opposite of the one given in the declaration in the Pascal program.

2.3 Altering the value of IORESULT in external subroutines.

A C language subroutine can alter the value of IORESULT if the following declaration is included in the C-module containing the subroutine:

```
extern short iorslt;
```

iorslt is the name of the variable in the Pascal run-time system containing the IORESULT value.

There exists a subroutine make_res in the library containing the standard external subroutines (/lib/libpext.a). This subroutine can be used to set the value of iorslt according to the result of a system call. Make_res has two parameters:

```
short make_res(res,p_iorslt)
  int res;
  short *p_iorslt;
```

res is the result from a system call and p_iorslt is a pointer to a short. If the system call was succesfully completed (res greater than or equal to zero) *p_iorslt is set to EOK. If res is equal to -2, *p_iorslt is set to -2 indicating EOF otherwise if res is equal to -1, *p_iorslt is set to the errorcode contained in smoserr. Make_res returns *p_iorslt. In order to use the mnemonics for the smoserr codes, these must be included by:

```
#include <smoserr.h>
```

2.4 Examples of external subroutines.

This section gives some examples of external subroutines. Note example 2.4.3 where files of different types are passed as parameters to the same external subroutine.

2.4.1 Example: XOR.

The procedure XOR declared
PROCEDURE XOR(VAR I: INTEGER; J: INTEGER); EXTERNAL;
may be written in the following manner:

```
xor(j,i)
  short j;
  short *i;
begin
  *i ^= j;       /* perform exclusive or */
end
```

Note that the parameters i and j in the C declaration are reversed compared with the external Pascal declaration.

2.4.2 Example: DELBOX

The standard external procedure used to delete a box, DELBOX, is declared:

```
PROCEDURE DELBOX( BOXNAME: STRING ); EXTERNAL;
```

and it may be written in the following manner:

```
#include <std.h>

extern short iorslt;

delbox(boxname)
  char *boxname;

begin
  register short lgth;
  char       name[80];

  lgth = boxname[0];          /* the length of the box name */
  memcpy(name,&boxname[1],lgth); /* the name of the box is copied
                                    to the variable 'name' */
  name[lgth] = '\0';  /* the name-string is null-terminated */
  make_res(unlink(name),&iorslt);
                            /* the box is deleted by unlink */
end;
```

2.4.3 Example: Passing a File as Parameter.

Consider a procedure involving a parameter:

```
VAR F1: FILETYPE;   (FILETYPE is a direct access file)
or
VAR F2: TEXT;
```

If a file is variable declared in the declaration of an external procedure, the following structures must be declared in the C subroutine:

```
struct dcb begin
  short ioud, /* i/o unit descriptor, -1 for not open */
        recl, /* record length */
        eof;  /* end-of-file flag */
end;

struct txtbuf begin
  struct dcb txtdcb;
  unsigned char term, /* TRUE for terminals and printers */
                linelength,
                buffer [255];
                      /* buffer 0 is current position */
end;
```

These structures are used by the Pascal runtime system to hold information about files.

The direct access file F1 is declared as

```
struct dcb *f1;
```

and the sequential file F2 is declared as

```
struct txtbuf *f2;
```

It is possible to write external procedures that can be called using both kinds of files as parameters (and any type of direct access file) by declaring the parameter UNIV in the Pascal declaration (see Supermax Pascal User's Guide section 2.6.3).

The following example illustrates this. The function uses the i/o unit descriptor for a box and tests whether the box is empty or not:

```
/*************************************************************/
/*  Declaration in Pascal :                                  */
/*  FUNCTION FEMPTYBOX(UNIV BOX: TEXT): BOOLEAN; EXTERNAL;   */
/*  Before call:                                             */
/*  BOX     : TEXT or FILE OF xxx                            */
/*  After call:                                              */
/*  IORESULT: The result-code from call of fstat un          */
/*  RETURN  : TRUE if the box is empty  (no readable bytes)  */
/*            if IORESULT <> 0 RETURN value is FALSE         */
/*************************************************************/

#include <std.h>
#include <smoserr.h>
#include <sys/types.h>
#include <sys/stat.h>

extern short iorslt;

short femptybox(fileid)
  struct txtbuf *fileid;
begin
  struct stat staun;
  short res;

  if ((make res(fstat(fileid->ioud,&staun),&iorslt)) == EOK) then
    if ((staun.st_mode & S_IFMT) == S_IFIFO) then
      res = (staun.st_size == 0);
      return(res);
    otherwise
      iorslt = EILOP;
    end_if
  end_if
  return(FALSE);
end
```

3. Using External Procedures and Functions.

This section describes in section 3.1 how to use external procedures and functions from Compiled Pascal (Pascal-Assembler) and in section 3.2 how to use external procedures and functions from Interpreted Pascal.

3.1 Using External Procedures and Functions from Compiled Pascal.

The program pac (see the manual Running Pascal Assembler Compiler) is used to compile and link Pascal programs. When using standard external procedures and functions the program pac will look for these subroutines in the archive /lib/libpext.a. User-defined external procedures and functions must be given explicitly to the program pac.

Assume that the program tst.p declares a user-defined external procedure, XOR for example. The C-code for XOR is found in the file xor.c.

First the XOR subroutine is compiled by using the pac program in the following way:

```
$ pac -v -c xor.c
```

and hence the relocatable file xor.o exists. (The C-module could also have been compiled using the cc program).

Now the loadmodule, tst, is created by using the pac program as follows:

```
$ pac -v -o tst tst.p xor.o
```

As indicated by the example above relocatable files containing user-defined external procedures or functions are simply listed following the Pascal source-code file, and the pac program will pass these files to the linker when it is called.

The loadmodule tst in the preceeding example can also be created by activating the program pac in the following way:

```
$ pac -v -o tst tst.p xor.c
```

The user-defined relocatable files can be inserted in libraries by using the program ar (see Supermax Operating System User's Manual Section 1). These libraries can then be given to the program pac.

For instance the relocatable file xor.o can be inserted in /lib/libextern.a by:

```
$ ar rv /lib/libextern.a xor.o
```

The loadmodule tst can then be created in this way:

```
$ pac -v -o tst tst.p -lextern
```

3.2 Using External Rutines from Interpreted Pascal.

In connection with the interpretation of a Pascal program an environment file may be supplied. This file is really a load module produced from

- an assembly language code originating from the compiler (the type j file) and possibly modified
- a set of subroutines written in C, being the external procedures and functions of the Pascal program

The environment file is linked to segment 8 and 9.

3.2.1 Contents of Environment File.

An environment file contains the following:

. load information

size of data area for variables
size of p-code area
size of heap

debug flag

address of external procedure no. 1, zero for none
address of external procedure no. 2, zero for none
.
.
address of external procedure no. 128, zero for none

the code for the external routines, if any

The external procedures and functions are numbered in the order the first calls of the respective routines occur in the programtext. The maximum number of external procedures and functions is 128 in interpreted Pascal. See the manual Supermax Running Interpreted Pascal for further information about the contents of an environment file.

3.2.2 Linking Environment File.

The Pascal compiler creates an assembly language program (the type j file), which is the source code for the environment file. If the compiled program contains external routines, this file must be assembled and linked together with the the external C routines. The resulting file is the environment file (type e). This file can be created by the program pasenvr.

How to link an environment file is shown by an example:

In the Pascal program tst.p the external procedure XOR is declared and called. The C code for this subroutine is found in the file xor.c. The xor subroutine is compiled by using the cc program in the following way (see Supermax Operating System User's Manual Section 1).

```
$ cc -v -c xor.c
```

After that the relocateable file xor.o exists.

3.4 Pascal C Interface

The Pascal compiler creates the p-code and the source code for the environment file:

```
$ pascal +q
Source code file:          tst.p
P-code file:               tst
Environment file:          tst.j
List option (t/l/ /q):
Conditions:
```

Now the enviromnent file tst.e can be linked by using the program pasenvr:

```
$ pasenvr +q
Enter source name:                tst.j
Enter dest name:                  tst.e
Enter ofiles (separated by ;):    xor.o
Enter libraries (sparated by ;):
```

The program pasenvr can also be executed in parameter form:

```
$ pasenvr -i tst.j -o tst.e -O xor.o
```

The option for libraries is -l.

The user-defined external rutines can be inserted in libraries by using the program ar (see section 3.1 and Supermax Operating System User's Manual Section 1) and these libraries can be given to the program pasenvr. When using standard external rutines, pasenvr will automatically look for the external rutines in the archive /lib/libpext.a. Only userdefined external rutines must be given explicitly to the program pasenvr.

Furthermore the program ppc can be used to execute both the compiler and pasenvr (see the manual "Supermax Running Interpreted Pascal").

4. Standard External Procedures and Functions.

This section describes the standard external procedures and functions. The declarations of the standard external procedures and functions are found in the file /pbin/extdecl.p. The relocateable code for the subroutines is contained in the library /lib/libpext.a. Section 3 describes how to use external procedures and functions.

4.1 Direct input.

These procedures enable the programmer to override the normal Pascal console input conversions. In the so-called direct input mode keystrokes are stored in the direct input buffer of the terminal. 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 buffer is used cyclically and has a size of 123 characters.

The subroutines described in section 4.1.1 to 4.1.5 concern direct input from the terminal from which the Pascal program is executed. Sections 4.1.6 to 4.1.8 describe procedures and functions which enable the programmer to get input from another terminal.

4.1.1 SETUP - Initiate direct input.

```
PROCEDURE SETUP; EXTERNAL;
```

A call of SETUP places a terminal in direct input mode. In this mode the program may read the keys pressed on the keyboard one by one as they are entered. Echoing of input characters does not occur automatically. When operating in direct input mode the READ, READLN, and EDIT routines may be used. Only one process in the computer may use the direct input mode for a given terminal at a given time. IORESULT is affected.

4.1.2 AVAIL - Test for availability of keybord input.

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

This function tests whether the direct input buffer contains one or more unprocessed characters. It returns TRUE if unprocessed characters exist otherwise FALSE. IORESULT is affected.

4.1.3 NEXT - Get next character from terminal.

```
FUNCTION NEXT: INTEGER; EXTERNAL;
```

This function fetches the next character from the direct input buffer and returns the ordinal value of the character.

If the next character is 0x01 the pressed key is a function key. Thus the function next fetches the next character which is the value of the function key. The value returned is the sum of the ordinal value of this character and 255 (giving a value between 320 and 367).

If the input buffer is empty, the program waits until a key is pressed. IORESULT is affected.

4.1.4 NEXT1 - Get next character from terminal.

```
FUNCTION NEXT1: CHAR; EXTERNAL;
```

This function fetches the next character from the direct input buffer and returns it. If the input buffer is empty, the program waits until a key is pressed. IORESULT is affected.

4.1.5 FINIS - Terminate direct input.

```
PROCEDURE FINIS; EXTERNAL;
```

This procedure clears direct input mode and restores the normal input mode. IORESULT is affected. Observe that direct input mode is automatically cleared when the terminal is closed (for example, when a process terminates).

4.1.6 SETDI - Initiate direct input from another terminal.

```
PROCEDURE SETDI(VAR FILEID: TEXT); EXTERNAL;
```

FILEID identifies an open terminal.

The procedure SETDI places the given terminal in direct input mode. For the standard input iounit the external procedures for direct input described in section 3 must be used. IORESULT is affected. The direct input mode is cleared when the given terminal is closed.

4.1.7 AVAILDI - Test for availability of input from another terminal.

```
FUNCTION AVAILDI(VAR FILEID: TEXT): BOOLEAN; EXTERNAL;
```

FILEID identifies an open terminal in direct input mode.

This function returns TRUE, if there are unprocessed characters in the direct input buffer for the other terminal, otherwise FALSE. The given terminal must have been placed in direct input mode by a call of SETDI. IORESULT is affected.

4.1.8 GETDI - Read from another terminal in direct input mode.

```
PROCEDURE GETDI(VAR FILEID: TEXT; VAR CONTENTS: STRING;
                                  COUNT: INTEGER); EXTERNAL;
```

FILEID identifies an open terminal in direct input mode.

The procedure inputs up to COUNT characters currently available in the direct input buffer of the given terminal. If the direct input buffer is empty GETDI waits until a character is input. The input characters are placed in the CONTENTS parameter. The length of the string is the number of input characters. The given terminal must have been placed in direct input mode by a call of SETDI. IORESULT is affected.

4.2 FUNC and FUNCZ - Function key values.

Two external functions returning the value of the last pressed function key exist, these are FUNC and FUNCZ.

```
FUNCTION FUNC: CHAR; EXTERNAL;

FUNCTION FUNCZ: CHAR; EXTERNAL;
```

Both FUNC and FUNCZ return the value of the last pressed function key. FUNCZ in addition resets the internal variable containing the value of the last pressed function key. Thus, when using FUNCZ a pressed function key can only be read once.

4.3 TRMNR - Get the terminal number.

```
PROCEDURE TRMNR(VAR UNIT, CHANNEL, WINDOW: INTEGER); EXTERNAL;
```

If the standard input iounit is a terminal, the unit number, channel number, and window number of this terminal is returned, otherwise -1 is returned. IORESULT is affected.

4.4 Printer reservation.

These procedures enable the Pascal programmer to reserve and release a printer.

4.4.1 GETPR - Reserve a printer.

```
FUNCTION GETPR(PR: STRING): BOOLEAN; EXTERNAL;
```

PR identifies a printer.

A call of the function GETPR reserves the printer identified by PR. It returns TRUE if the printer can be reserved otherwise FALSE. After a successful call the error device (ERROR) is the specified printer.

4.4.2 FREEPR - Release a printer.

```
PROCEDURE FREEPR; EXTERNAL;
```

This procedure releases the printer used as the error device. After a call of FREEPR the error device is the standard output device (OUTPUT).

4.5 File information.

This procedure enables the Pascal programmer to get the size of a Unix file.

```
PROCEDURE SFILEINF(FILENAME: STRING;VAR SM, ST, S: INTEGER);
                                                   EXTERNAL;
```

FILENAME identifies a Unix file, e.g. '/usr/an/tst.p'.

This procedure places information about the size of the given file in the parameters SM, ST og S. The size is (SM*1000+ST)*1000+S. IORESULT is affected.

4.6 REDBAK - Read one record backwards.

```
PROCEDURE REDBAK(VAR FILEID:TEXT; VAR CONTENTS: xxxx); EXTERNAL;
```

FILEID identifies an open sequential file, e.g. '/usr/an/tst.p'. xxxx is a string type declared: TYPE xxxx = STRING(255);

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 across which the system backspaced are placed in the parameter CONTENTS. The length of the string used as the CONTENTS parameter is set according to the length of the record read. IORESULT is affected. If begin-of-file is reached IORESULT returns 0, but the standard function EOF returns TRUE.

4.7 RENAMF - Rename a file.

```
PROCEDURE RENAMF(OLD, NEW: STRING); EXTERNAL;
```

This procedure renames the file specified in the parameter old to the name specified in the parameter new. IORESULT is affected.

4.8 Boxes.

The following procedures and functions operate on boxes. The procedure DELBOX can also be used to delete a closed file.

4.8.1 Creation of box from Pascal.

```
PROCEDURE PMAKEBOX(NAME: STRING); EXTERNAL;
```

This procedure creates a socalled box. Hence this box can be opened using REWRITE.

The NAME parameter is a Supermax file name possibly followed by a colon and one or more indicators of parameters.

The indicators are:

:s followed by the size of the box in bytes. This size will be rounded to the next higher multiple of 512. If :s is omitted or if the size is greater than 16384 the box will be created with size 2048.

:a followed by the access rights in octal notation. If :a is omitted the box will be created with access 0764 (that is rwxrw-r--). Note that the access rights also depend on the umask of the user calling the procedure.

IORESULT is affected by the procedure and the user should always check the value of IORESULT following a call of PMAKEBOX.

Example:

```
PMAKEBOX('abcbox:s200:a777');
```

will cause the creation of the box abcbox in current directory with a size of 512 (next higher multiple of 512) bytes and access rights rwxrwxrwx.

4.8.2 Test for readable bytes in an (open) box.

```
FUNCTION EMPTYBOX(BOXNAME: STRING): BOOLEAN; EXTERNAL;
```

or

```
FUNCTION FEMPTYBOX(FILETYPE: BOOLEAN; UNIV BOX: TEXT):
                                              BOOLEAN; EXTERNAL;
```

BOXNAME is the name of a box, e.g. '/dev/box/buf'.
FILETYPE must be TRUE if the next parameter is of type TEXT and FALSE if it is of type FILE OF XXX. BOX is a variable of type TEXT or FILE OF XXX

Both functions return TRUE if the box is empty (no readable bytes), otherwise FALSE is returned. When using FEMPTYBOX the box must be opened in advance. IORESULT is affected. If IORESULT <> 0 the function returns FALSE.

4.8.3 Test for writeable bytes in an (open) box.

```
FUNCTION FULLBOX(BOXNAME: STRING): BOOLEAN; EXTERNAL;
```

or

```
FUNCTION FFULLBOX(FILETYPE: BOOLEAN; UNIV BOX: TEXT): BOOLEAN;
                                                      EXTERNAL;
```

BOXNAME is the name of a box, e.g. '/dev/box/buf'.
FILETYPE must be TRUE if the next parameter is of type TEXT and FALSE if it is of type FILE OF XXX. BOX is a variable of type TEXT or FILE OF XXX

Both functions returns TRUE if the box is full (no writeable bytes), otherwise FALSE is returned. When using FFULLBOX the box must be opened in advance. IORESULT is affected. If IORESULT <> 0 the function returns FALSE.

4.8.4 Get information about an (open) box.

```
PROCEDURE BOXSTATUS(BOXNAME: STRING; VAR RBYTES, WBYTES:
                                          INTEGER); EXTERNAL;
```

or

```
PROCEDURE FBOXSTATUS(FILETYPE: BOOLEAN; UNIV BOX: TEXT;
                     VAR RBYTES, WBYTES: INTEGER); EXTERNAL;
```

BOXNAME is the name of a box, e.g. '/dev/box/buf';
FILETYPE must be TRUE if the next parameter is of type TEXT and FALSE if it is of type FILE OF XXX. BOX is a variable of type TEXT or FILE OF XXX.

Both procedures give the number of readable and writeable bytes in a box. After a call RBYTES/WBYTES contains the number of readable/writeable bytes in the given box. When using FBOXSTATUS the box must be opened in advance. IORESULT is affected.

4.8.5 DELBOX - Delete a box.

```
PROCEDURE DELBOX(BOXNAME: STRING); EXTERNAL;
```

BOXNAME is the name of a box, e.g. '/dev/box/buf';

A call of this procedure deletes a box. If some process has the box open, deletion will be postponed until the box is closed. The box is deleted even if it is not empty. Note that empty boxes are automatically deleted when closed. IORESULT is affected.

The procedure may as well be used to delete a closed file. BOXNAME is then a file name, e.g. '/usr/an/test.p'.

4.9 CLOCK - Get the system time.

```
PROCEDURE CLOCK(VAR DATE, TIME: STRING); EXTERNAL;
```

A call of this procedure returns the date in the form dd.mm.yyyy in the parameter corresponding to DATE and the time in the form hh.mm.ss in the parameter corresponding to TIME. The length of the string of the first parameter is set to 10 and the length of the second is set to 8.

4.10 PROCNO - Get the process number for the current process.

```
PROCEDURE PROCNO(VAR PRCNUM: INTEGER); EXTERNAL;
```

This procedure places the process number of the current process in the parameter PRCNUM. IORESULT is affected.

4.11 GETUSER - Get the user name.

```
PROCEDURE GETUSER(VAR USERNAME: STRING); EXTERNAL;
```

The procedure reads the name of the user who started the calling process in the file /etc/passwd. The user name is placed in the USERNAME parameter. The length of this string is 8. IORESULT is affected. If the user name of the current process is not contained in the file /etc/passwd, IORESULT is EILLPARM (302).

4.12 SETPRIO - Set the priority of a process.

```
PROCEDURE SETPRIO(PRCNO, PRIO: INTEGER); EXTERNAL;
```

PRCNO is a process number. NOTE: this parameter has no significance under system V, as one can only change the priority of the calling process. The PRCNO parameter is ignored!.

PRIO is a priority.

The procedure changes the priority of the calling process. IORESULT is affected.

4.13 PROCSTA - Get a process status.

```
TYPE STATUS=
     RECORD
       PRCNO,          (* process ID *)
       PRCGRP: INTEGER;(* process group ID *)

       PRCNAME: PACKED ARRAY (.1..16.) OF CHAR;
                                        (* process name *)
       PRCRGRN,        (* real group ID *)
       PRCRUSN,        (* real user ID *)
       PRCEGRN,        (* effective group ID *)
       PRCEUSN,        (* effective user ID *)
       PRCKIND,        (* process kind *)
       PRCPRIO: INTEGER;(* process priority *)
```

```
    PRCUTIM,          (*user mode time in ticks (clock interrupts)*)
    PRCSTIM,          (* supervisor mode time in ticks *)
    PRCBTIM: LONGINT;(* birth time in seconds*)

    PRCPPID: INTEGER;(* parent process id *)

    PRCPRIV,         (* process is privileged *)
    PRCACT,          (* process is active *)
    PRCRUNN,         (* process is running *)
    PRCESUSP,        (* process externally suspended *)
    PRCISUSP,        (* process internally suspended *)
    PRCWAIT,         (* process is waiting for box I/O *)
    PRCABO: BOOLEAN;(* process is dying *)

    PRCSUSR: INTEGER;(* reason for internal suspension *)
  END;

PROCEDURE PROCSTA(PRCNO: INTEGER; VAR BLOCK: STATUS); EXTERNAL;
```

This procedure fetches information about the specified process. If process number -1 is specified the process status of the calling process is returned. IORESULT is affected.

The declaration of the type STATUS is contained in the file /pbin/extdecl.p.

4.14 PWAIT - Wait for dead offspring process.

```
TYPE PI = ^INTEGER;
PROCEDURE PWAIT(VAR PID: PI); EXTERNAL;
```

PID identifies a process spawned by the CHAIN procedure. The process id is returned in the third parameter of the Pascal standard procedure CHAIN.

A call of PWAIT makes the calling program wait until the identified process dies. IORESULT is the condition code from the dead process. If the calling program has no offspring processes IORESULT is EDEADPNX (114).

4.15 Exception handlers for attention, interrupt and quit.

```
PROCEDURE AKEY(ANSWER: BOOLEAN; VAR ATT: INTEGER);

PROCEDURE IKEY(ANSWER: BOOLEAN; VAR INTR: INTEGER);

PROCEDURE QKEY(ANSWER: BOOLEAN; VAR QUIT: INTEGER);
```

The procedures AKEY, IKEY and QKEY are used to set up exception handlers for attention, interrupt and quit, and to get the number of times the attention, interrupt and quit keys have been pressed.

When ANSWER is FALSE in a call of one of the procedures the belonging exception handler is set up. When ANSWER is TRUE the number of times the belonging key was pressed since the exception handler was set up is returned.

4.16 PIPEOPEN and PIPECLOSE - Open and close a pipe.

```
FUNCTION PIPEOPEN(PROGNAME: STRING; VAR F1, F2: TEXT): INTEGER;
                                                       EXTERNAL;

FUNCTION PIPECLOSE(PID: INTEGER): INTEGER; EXTERNAL;
```

The function PIPEOPEN is used to create a pipe (an anonymous box) between the process calling the function PIPEOPEN and the process which executes the program specified in the parameter PROGNAME. The pipe will connect the standard I/O files for the two processes in one of the following ways:

- standard INPUT for the new process is connected to standard OUTPUT for the calling process

- standard INPUT for the new process is connected to standard ERROR for the calling process

- standard OUTPUT for the new process is connected to standard INPUT for the calling process

- standard ERROR for the new process is connected to standard INPUT for the calling process

The parameters F1 and F2 indicate the connection between the standard I/O files for the two processs. The parameter F1 must be a standard I/O file for the calling process and F2 must be a standard I/O file for the new process. F1 must be INPUT, OUTPUT or ERROR. F2 must be OUTPUT or ERROR. If F1 is OUTPUT or ERROR, F2 is assumed to be INPUT.

Upon successfull completion the function PIPEOPEN returns the process number of the new process. Otherwise, a value of -1 is returned and an errorcode is returned in IORESULT.

Example:

```
RESULT := PIPEOPEN('lp',ERROR,INPUT);
```

This call of PIPEOPEN will start the execution of the program lp. A pipe will be created between the process which executes lp and the calling process. When the calling process writes data to standard ERROR the data will be stored in an anonymous box. When the new process reads data from standard INPUT the data stored in the box by the calling process will be read.

The function PIPECLOSE takes a process number as parameter. It disconnects the pipe between the calling process and the process given by the parameter. PIPECLOSE re-etablishes standard INPUT, OUTPUT or ERROR of the calling process. When the process with the given process number is dead, PIPECLOSE returns the completion code of the dead process. IORESULT is affected.

4.17 XCHAIN and XPWAIT - Start and wait for new process.

```
FUNCTION XCHAIN(COMM: STRING; TYP: INTEGER): INTEGER; EXTERNAL;
```

COMM contains a command, which will be interpreted by shell. TYP contains an integer, that defines what kind of process will be created.

TYP = 0 will create a child process of the calling process. XCHAIN returns immediately and the return value will be the process id of the new process. The process id may be used as parameter to the external procedure XPWAIT described below.

TYP = 1 will create a child of process 1 (normally init). XCHAIN returns immediately and the return value will be the process id of the new process.

TYP = 2 will create a sibling of the calling process. This means that the parent process id of the new process will be the same as the parent process id of the calling process. XCHAIN returns immediately and the return value will be the process id of the new process.

TYP = 3 will not create another process. The calling process will transform into the new process. XCHAIN will not return. The process id of the new process is the same as the process id of the calling process.

TYP = 4 will create a child process of the calling process. XCHAIN will not return until after the termination of the child process. The return value is the exit code from the child process.

If XCHAIN fails, the return value will be -1 and IORESULT will indicate the error.

```
PROCEDURE XPWAIT(VAR PID: INTEGER); EXTERNAL;
```

PID identifies a child process of the calling process. The process may be started using the standard function XCHAIN. The process id is returned by the function XCHAIN. A call of XPWAIT makes the calling program wait until the identified process dies. IORESULT is the condition code from the dead process. If the calling program has no offspring processes IORESULT is EPROCNX (106).

4.18 PSETPGRP - Set process group number.

```
PROCEDURE PSETPGRP; EXTERNAL;
```

A call of this procedure makes the calling process a process group master, that is, the process group ID of the calling process is set to the value of the process ID of the calling process.

Examples.

```
TYPE NAMEREC = RECORD
                 FIRSTNAME: STRING(20);
                 SURNAME:   STRING(20)
               END;

VAR NAMEFILE: FILE OF NAMEREC;
```

Let 'namefile' be a file containing records of type NAMEREC. The file contains 100 records and the first 7 records are not to be sorted. The records should be sorted after increasing SURNAME. The parameters must then be:

```
FILENAME    = 'namefile'
BUF SIZE   >= 524
REC SIZE    = 44
REC PR UNIT = 1
NOT SORT    = 7
NO OF UNITS = 93
REC NO      = 1
KEY NO      = 24
KEY SIZE    = 20
```

Let 'namepair' be a file containing records of type NAMEREC. The file contains 100 records which all should be sorted. A sort unit contains two record and the sort units should be sorted after increasing SURNAME in the second record:

```
FILENAME    = 'namepair'
BUF SIZE   >= 824
REC SIZE    = 44
REC PR UNIT = 2
NOT SORT    = 0
NO OF UNITS = 50
REC NO      = 2
KEY NO      = 24
KEY SIZE    = 20
```

4.18 PSETPGRP - Set process group number.

```
PROCEDURE PSETPGRP; EXTERNAL;
```

A call of this procedure makes the calling process a process group master, that is, the process group ID of the calling process is set to the value of the process ID of the calling process.