Supermax System V

Running Pascal Assembler Compiler

Dansk Data Elektronik A/S
15. Sep 1987

Version 1.1

Table of Contents.

1. Introduction.

This manual describes the use of the Pascal to Assembler compiler pta and the program pac, which creates a load module from a Pascal source code and external procedures and functions written in Pascal or C. See the Pascal C Interface Manual for writing external procedures and functions in C.

2. The Compiler.
2.1 Running the Compiler.

There exist two Pascal compilers which translate from Pascal source code to assembler source code. The compilers are located in the files /pbin/pta and /pbin/pta20. /pbin/pta is generates code to run on a Supermax with a mc68000 processor, (which can also run on a mc68020 processor). /pbin/pta20 generates code to run on a mc68020 processor (which cannot run on a mc68000 processor). Both compilers check the value of the environment TARGETMC which must be set to either 68000 or 68020. If the value of TARGETMC differs from 68000 /pbin/pta terminates without compiling; if the value is different from 68020 /pbin/pta20 terminates without compiling.

Both compilers may execute in either parameter mode or in dialog mode.

When +q is given as parameter the compilers ask questions (user input shown underlined):

```
$ pta +q
Source code file:          tst.p
Asm-code file:             tst.s
List option (t/l/ /q):     l
List file:                 tst.list
Conditions:                d
Statistics (y/n):          y
```

Source code file:
The source code file is the file containing the source code of the Pascal program. It will normally (but not necessarily) be of type p.

Asm-code file:
The asm-code file is the file in which the translated program i.e. the assembler source code should be stored. It must be of type s.

List option:
The listoption may be used to control the list output produced by the compiler. The listoptions recognized by the compiler are T (t), L (l), and Q (q). The listoption may be changed during a compilation using the L and Q compile time options (see section 2.2).

If L, l, T, or t is specified as the listoption, and the compiler is executed in dialog mode the compiler will ask for the name of the file in which to store a source listing of the compiled program.

If L (l) is specified as the listoption, the compiler will produce a source listing of the compiled program including the number of each source line compiled, as well as the block level (the number of RECORDs, BEGINs, REPEATs, and statement CASEs minus the number of ENDs, UNTILs, and OTHERWISEs processed) before the line is compiled. Error messages will appear in the list file just after the line in which the error was detected.

If T (t) is specified as the listoption, the compiler will produce the same output as mentioned under L above, but after each record declaration the size of the record will be output.

The Q (q) listoption suppresses all output except error messages.

If an illegal listoption is specified or if no listoption is specified, the compiler will use the output device to keep the operator informed about the progress of the compilation by displaying the source line number and the name of the procedure currently being compiled plus a dot for each line compiled. Error messages will appear on the output device together with a printout of the line in which the error was detected.

Conditions:
Conditions is simply a number of capital letters that control the conditional compilation, see section 2.2. If the program contains compiler directives such as (*$XH*).....(*$X-*), the code between the two compiler directives will be compiled only if the letter H is among the letters given in the conditions parameter.

Statistics:
If the statistics question is answered 'y' or 'Y' the generated code will contain instructions that count the number of times each procedure/function is called. The result will be placed in a file called profile.out after normal termination of the compiled program.

The parameter mode corresponding to the dialog mode shown in the start of the section is:

```
$ pta -i tst.p -o tst.s -Ll -T tst.list -Cd -py
```

The default values are:

for asmcode - the name of the source code file with type s instead of p

for listop - blank

for cond - no conditions

for stat - n

Compilation may be aborted at any time by entering a ctrl c at the terminal that was used to start the compilation. Using ctrl c will not kill other Pascal programs running in the background on the same terminal, and no trace dump will be produced.

2.2 Compile Time Options.

The compilers may be instructed to generate code according to certain options; in particular, it may be requested to insert or omit run-time test instructions, and it may be requested to include files. Compiler directive are written as comments and are designated as such by a $-character as the first character of the comment:

```
(*$<option sequence>*)
```

The option sequence is a sequence of instructions separated by commas. Each instruction consists of a letter, designating the option, followed either by a plus (+) if the option is to be activated or a minus (-) if the action is to be passivated, or by a digit, or by a filename. Example:

```
(*$C+,Ideclfile*)
```

Illegal syntax in a compiler directive is not reported by the compiler, but the results are unpredictable.

The following options are currently available:

B causes the compiler to generate line numbers referring to the lines in the listing of the program produced by the compiler and not (as default) relative to the source code file, the line originates from. (Also see option D).

C causes the compiler to generate I/O check instructions after each statement which performs any I/O. The instruction checks to see if the I/O operation was accomplished successfully. In the case of an unsuccessful I/O operation the program will be terminated with a Supermax I/O error message. Note that no I/O check instructions are generated after RESET, REWRITE, LOCK, or CHAIN procedure calls.

When automatic I/O checking is off, the user must check the value of the IORESULT function after each I/O operation.

C+ I/O check instructions are generated (default).
C- I/O check instructions are not generated.

D causes the compiler to generate line numbers in the assembler code. If a run time error occurs the program will print the number for the source line corresponding to the code that was executed when the error occured, along with the name of the file containing the source code. Each line number generated occupies 6 bytes in the final code.

D+ line number are generated (default).
D- line number are not generated.

F includes a file containing global declarations (to be used in several modules of one program) into the compilation. The characters between 'F' and the terminating '*)' are taken as the file name of the source file to be included. The comment must be

closed at the end of the file name, therefore no other options can follow the file name.

Example: (*$F/usr/abc/decl.p*)

G determines whether Pascal GOTO statements are allowed within the program. This option may be used to restrict novice programmers from using the GOTO statement in situations where structured constructs like FOR, WHILE, REPEAT, and CASE statements would be more appropriate

G+ allows the use of the GOTO statement.
G- causes the compiler to generate a syntax error upon encountering a GOTO statement (default).

I includes a source file into the compilation. The characters between 'I' and the terminating '*)' are taken as the file name of the source file to be included. The comment must be closed at the end of the file name, therefore no other options can follow the file name.

Example: (*$I/usr/an/test.p*)

The compiler cannot keep track of nested inclusions, that is, an included file is not allowed to have an include file compiler directive. This would result in a fatal syntax error. If nested inclusions are wanted, the Pascal pre-processor can be used. Note that files included by $I are included during the compilation, while files included by the pre-processor are included before compilation.

L controls whether the compiler should generate a program listing of the following source text. This directive is analogous to the L listoption discussed in section 2.1. The default value of this option is set by the listoption when the compiler is started.

L+ start output of source listing in the list file.
L- stop output of source listing.

M This option only has meaning if the heapsize of the pta compiler is different from zero (otherwise the heap is only limited by the size of memory, see section 4). The option causes the compiler to output the number of bytes remaining in the heap of the compiler. This number gives the user a hint as to whether a compiler run time error 'Heap overflow' may be expected if the program is expanded. The heap grows with each declaration of an identifier. The heap size can be changed by the program setheap (see section 4).

N This letter must be followed by two digits. These digits are taken as an integer, and the number of lines per page in a program listing is set to this number.

For example (*$N30*)

O This option controls the optimization of the final assembler code. A variety of peephole optimization is performed including; jump to next ins.; jump to jump; constant propagation; etc.

O+ turns optimization on (default).
O- turns optimization off.

Programs compiled with the O option set will run slightly faster and require less code.

P causes the compiler to skip to a new page on the printer if the compiler is generating a source listing on the printer at the time when the P compiler directive is encountered.

Q is the 'quiet compiler' option which can be used to suppress the output to the output device of procedure names, line numbers, and dots detailing the progress of the compilation. This compiler directive is analogous to the Q listoption discussed in section 2.1. The default value of this option is set by the listoption when the compiler is started.

Q+ causes the compiler to suppress output to the output device.
Q- causes the compiler to output procedure names, line numbers, and dots to the console device.

R This option controls whether the compiler should output additional code to perform checking on array and string subscripts and assignments to variables of subrange types.

R+ turns range checking on (default).
R- turns range checking off.

Programs compiled with the R option set will run slightly faster and require less code; however, if an invalid index occurs or an invalid assignment is made, the program will not be terminated with a run time error. Until a program has been completely tested and is known to be correct, it is usually best to compile with the R+ option set. Note that certain string indexing errors (index<0 or >255) are detected even if range checking is disabled.

S This option makes it possible to use arbitrary ascii characters in strings and as character constants. S must be followed by a character. This character is denoted an escape character. In strings and character constants this escape character can be given followed by two hexadecimal digits. The two hexadecimal digits specify an ascii character. When a character no longer should be used as escape character the escape character can be deleted by the compile time option S followed by -. Default escape character is '-'. Example:

```
(*$S\*)
writeln('error\07');  (* writes error and beeps *)
(*$S-*)
```

U,V both options are generated by the Pascal Pre-processor making the compiler able to print correct line numbers and filenames on discovering syntax errors. These options should not be used by the programmer.

X specifies code which is to be compiled only under certain conditions. This directive takes the form (*$Xn*), where n is either a capital letter or a minus sign. If n is a capital letter the following code is compiled only if that letter was given in

the conditions parameter when the compiler was started (see section 2.1). If n is a minus sign the following code is compiled unconditionally. This conditional compilation facility is useful if two almost identical versions of a program are desired; for example, certain statements producing test output may be conditionally compiled so that it is easy to switch from a test version of a program to a non-test version. Conditional compilation can also be obtained using Pre-processor commands.

Example:

Assume that the following program resides in the file tst.p:

```
PROGRAM TEST;
BEGIN
  WRITELN('COMMON');
  (*$XT*) WRITELN('TTT'); (*$X-*)
  WRITELN('COMMON');
  (*$XQ*) WRITELN('QQQ'); (*$X-*)
  WRITELN('COMMON');
END.
```

If this program is compiled in the following manner:

```
$ pta -i tst.p
```

it will be equivalent with the following program:

```
PROGRAM TEST;
BEGIN
  WRITELN('COMMON');
  WRITELN('COMMON');
  WRITELN('COMMON');
END.
```

If the program is compiled as follows:

```
$ pta -i tst.p -CQ
```

it will be equivalent with the following program:

```
PROGRAM TEST;
BEGIN
  WRITELN('COMMON');
  WRITELN('COMMON');
  WRITELN('QQQ');
  WRITELN('COMMON');
END.
```

If, finally, the program is compiled this way:

```
$ pta -i tst.p -CQT
```

it will be equivalent with the following program:

```
PROGRAM TEST;
BEGIN
  WRITELN('COMMON');
  WRITELN('TTT');
  WRITELN('COMMON');
  WRITELN('QQQ');
  WRITELN('COMMON');
END.
```

3. The pac Program.

pac is a program which depending on what types of files and what options are given executes on or more of the programs ppp, pta (or pta20), as (or as20), and ld in order to compile a pascal program. In addition to this pac can take c-routines as input and using the appropriate c-compiler create a relocatable file.

pac will check if the environment TARGETMC is set to 68000 or 68020. If not pac terminates without compiling. Otherwise pac uses the value of the environment in selecting which programs to run and which standard libraries to use in the link fase.

pac works as follows:

- a file of type p is sent through the preprocessor before output from the preprocessor is sent on to the compiler.

- a file of type pp is given directly to the compiler.

- if a file of type p and the option -P is given to pac the file will be given directly to the compiler.

The following options can be given to pac:

-o after -o the name of the resulting file is given.

-L may be followed by l, L, t, T, q or space. If space is written the preprocessor and the compiler will output a dot for each line. If q is written output will be suppressed and if l,L,T or t is given the compiler will produce a listing of the program. If -T option is not given the listing will be placed in <xxx>.list given the name of the sourcefile is <xxx>.p.

-T must be followed by a space and the name of the file in which the program listing should be placed, if listoption t, T, l, or L has been given.

-C the letters written following C are given to the compiler as conditions.

3.2 Running Supermax Pascal-Assembler Compiler

-P type p files are given directly to the compiler

-p the compiler generates statistical code

-S pac stops after having generated the assembly language code

-c pac stops after having produced the relocatable file (type o)

-O the optimizer is evoked after compilation af C-modules

-v verbose: pac keeps score on standard output what fase it is currently executing.

Example:

```
$ pac -o tst tst.p tst1.p ext.c -CM
```

First pac will send tst.p and tst1.p to the preprocessor which will be executed with listoption q. Output from the preprocessor will be used as input for the compiler and the modules will be compiled using condition M. Then the assembler is evoked and the relocatable modules tst.o and tst1.o are created.

Next the cc program will be evoked in order to compile ext.c and ext.o is created.

Finally the linker will be evoked to link together the created relocatable files and the runtime libraries.

Note: when pac executes the linker one and only one of the relocatable modules must contain a main program.

4. Stack and Heap.

All loadmodules contain a stacksize which can be changed by the program chstack (see Supermax Operating System User's Manual Section 8). In addition a Pascal loadmodule contains a heapsize which can be changed by the program setheap (se section 4.3).

4.1 How to determine stacksize.

Every loadmodule - be it a module originating fra a Pascal, C, or other language source code - makes use of a socalled stack. The stack is used by a running program to store values concerning the specific subroutines currently executed, for instance parameters, local variables, the return address, and for functions also the return value.

A running process uses segment 13 as stack area. Segment 2 contains the actual code of the program and segment 3 contains data. Data consists of all globally declared variables and the static structures of those relocatable modules that were linked together to form the loadmodule. What Pascal programs are concerned segment 3 also contains the heap area.

Every time a subroutine is invoked stack area is reserved for this call of the routine, and when the program returns from the routine the area is given free. If for instance a routine requires N bytes of stack area and it is called recursively, the amount of bytes available on the stack prior to the initial call of the routine must be at least N times the number of levels in the recursion.

The necessary size of the stack depends on the specific program. Pascal programs containing a call of the standard procedure CHAIN always require a stack of at least 8 K (0x2000) bytes due to the demands of the routine used by the pascal run time system to make a new proces.

A review of the routines a program is built of often makes it possible to determine a relevant stack size for the program. Consider the following pieces of a pascal program:

```
.
.
type large = array (1..10,1..1000) of integer;
var  big : large;
.
procedure eat_stack(big:large);
begin
.
.
end;
begin (* main program *)
  .
  .
  eat_stack(big);
  .
  .
end.
```

When eat_stack is called the program requires stack area to store the call-by-value parameter big - this parameter alone takes up 10*1000*2 = 20000 bytes and using a stack size less than this will cause an error during run time.

If big had been declared as a call-by-reference (var) parameter it would have taken up only 4 bytes on the stack (being the address of big).

The stack size of a program can be set/read/changed by the program chstack:

```
$ chstack tst
```

outputs the current amount of bytes allocated as stack when the program tst is called.

```
$ chstack -s 0x4000 tst
```

sets/changes the stack size to 0x4000 bytes.

4.2 Dynamic stack allocation.

Some Supermax machines support socalled dynamic allocation of stack space, meaning that a running program can allocate more stack space when needed. Thus the size set by chstack is no more an ultimate upper limit on the stack.

The routines that utilize dynamic stack allocation have been implemented in the pascal assembler system analogous to the way it works for C programs.

The routines used depend on the environment STACKCHECK. STACKCHECK can be set to three different values:

```
STACKCHECK=ON    (default)
STACKCHECK=OFF
STACKCHECK=TEST (only mc68020)
```

If STACKCHECK=ON or the environment is not set at all or it is set to something different than the three above mentioned values the compiler /pbin/pta (or /pbin/pta20) will generate code, that uses the C-routine spgrow to allocate stack space. If STACKCHECK=TEST the compiler will generate code using a special quicker facility in the operating system to allocate stack space (this facility is of now only implemented in operating systems for mc68020 processors). If STACKCHECK=OFF no code for dynamic stack space allocation will be generated.

If the machine does not support dynamic stack space allocation the stack size of the program should be set using chstack. For further information on stack sizes see "Supermax Operating System, System V, Reference Manual", exec(2), bottom of page 3.

4.3 The heap.

If a Pascal program uses heap space (run time storage allocation), that is if the program contains call(s) of the standard subroutine NEW, it can either be allowed to allocate storage up to a limit set in the loadmodule or unlimited (only limited by the storage available).

The limit set in the loadmodule can be 0 (zero) which is default or a value greater than 0. The default value 0 causes the program to dynamically allocate storage when it is needed, and the allowed amount is unlimited. If the limit is greater than 0 the allowed amount of heap space is allocated when the program is started. Programs with no limit run slightly slower than programs with a limit.

The program /pbin/setheap can be used to change the limit in the loadmodule. It can be run in either parameter or dialog mode. When +q is given to the program it asks questions (user input showned underlined):

```
$ setheap +q
Enter file name: tst
Enter heap size: 0x00a000
```

Thereafter the heapsize in the loadmodule tst is a000 hexadecimal.

The analogous parameter mode is:

```
$ setheap -i tst -h 0xa000
```

The heapsize can be given in octal, decimal or hexadecimal notation. An octal number must be preceded by '0', and a hexadecimal by '0x' or '0X'.

5. Implementation Size Limits.

The following is a list of limitations imposed upon the user by the current implementation of the Supermax Pascal Assembler Compiler:

1) The maximum number of characters in a STRING variable is 255.

2) The maximum number of characters in a LONGSTRING variable is 32767.

3) The maximum number of characters in a CSTRING variable is 32767.

4) The maximum number of procedures or functions is 512.

5) The maximum number of nested blocks is 13.

6) The maximum number of internal object code in a procedure or function is 16000. Approx. 50k final code.

7) Local variables in a procedure or function can occupy a maximum of 1024K.

8) A set element must be in the range from 0 to 4079.

Appendix A. Compile Time Error Messages.

Errors with numbers > 400 cause the compiler to terminate.

1: error in simple type
2: identifier expected
4: ')' expected
5: ':' or '..' expected
6: symbol illegal in context (may be missing ';' on the line above or ';' in front of ELSE)
7: error in parameter list
8: 'OF' expected
9: '(' expected
10: error in type
11: '(.' expected
12: '.)' expected
13: 'END' expected
14: ';' expected
15: integer expected
16: '=' expected
17: 'BEGIN' expected
18: error in declaration part
19: error in <field list>
20: ',' expected
21: '.' expected
50: error in constant
51: ':=' expected
52: 'THEN' expected
53: 'UNTIL' expected
54: 'DO' expected
55: 'TO' or 'DOWNTO' expected in FOR-statement
56: 'EXITIF' expected
57: 'ENDLOOP' expected
58: error in <factor> (bad expression)
59: error in variable

101: identifier declared twice
102: low bound exceeds high bound
103: identifier is not of the appropriate class (may be a type identifier used where a variable is required)

104: undeclared identifier
105: sign not allowed
106: number expected
107: incompatible subrange types
108: file not allowed here
109: type must not be real
110: <tagfield> type must be scalar or subrange
111: incompatible with <tagfield> part
113: index type must be a scalar or a subrange
114: base type must not be real
115: base type must be a scalar or a subrange
116: error in type of standard procedure parameter
117: unsatisfied forward reference
119: re-specified parameters not ok for a forward or global declared procedure
120: function result type must be scalar, subrange or pointer
121: file value parameter not allowed
122: a forward declared function's result type cannot be respecified
123: missing result type in function declaration
125: error in type of standard function parameter
126: number of parameters does not agree with declaration
127: illegal parameter substitution
128: result type does not agree with declaration
129: type conflict of operands
130: expression is not of set type
131: tests on equality allowed only
132: strict inclusion not allowed
133: file comparison not allowed
134: illegal type of operand(s)
135: type of operand must be boolean
136: set element type must be scalar or subrange
137: set element types must be compatible
138: type of variable is not array
139: index type is not compatible with the declaration
140: type of variable is not record
141: type of variable must be file or pointer
142: illegal parameter solution
143: illegal type of loop control variable
144: illegal type of expression
145: type conflict

146: assignment of files not allowed
147: label type incompatible with selecting expression
148: subrange bounds must be scalar
149: index type must not be integer
150: assignment to standard function is not allowed
152: no such field in this record
153: type error in read
154: actual parameter must be a variable
155: control variable cannot be formal or non-local
156: multidefined case label
158: no such variant in this record
159: real or string tagfields not allowed
160: previous declaration was not forward
161: procedure has allready been forward declared
162: parameter size must be constant
163: missing variant in declaration
165: multidefined label
166: multideclared label
167: undeclared label
168: undefined label
169: error in base set
173: externaloption not specified when compilation was started
174: parameter universal declared in non external procedure
175: only files and unpacked array may be universal declared
176: parameter declared as cstring in non-external procedure
177: parameter of type power must be variable declared in external procedure declaration
178: comparison not allowed on cstring
180: constant outside range
181: division by zero in constant expression
182: overflow in constant expression
183: case constant to large

190: inclusion of global declarations not allowed in external module
191: inclusion of global declarations must be before other declarations
192: only one inclusion of global declarations allowed
193: only procedure/function declarations allowed in subprograms
194: global procedure/function declaration not allowed in external module

195: only procedure/function declarations allowed after const, type or var declarations in external module or only procedure/function declarations allowed in subprograms
196: external declaration of routines allowed only on level 1
201: error in real number - digit expected
202: string constant must not exceed source line
203: integer constant exceeds range
204: illegal hexadecimal character

250: too many scopes of nested identifiers
251: too many nested procedures or functions
253: procedure to long
254: CASE statement to long
258: var declaration to big

397: implementation restriction
398: implementation restriction
399: implementation restriction

400: illegal character in text
401: unexpected end of input
403: 'PROGRAM' expected
408: include control comment not allowed in inclusion file
409: error in parameters to the Pascal compiler omitted

500: too many generated identifiers (This maximum is well beyond half a million!)

10xxx: error during open of inclusion file
11xxx: error during open of source file
12xxx: error during create/open of assembler-code file
15xxx: error during output to assembler-code file
16xxx: error during input from source file
18xxx: error during open of list file

In the error messages with numbers >= 10000 the last three digits represent the Supermax Operating System error code (see Supermax System Operation Guide Appendix A).

Appendix B. Run Time Error Messages & Their Handling.

When a Pascal-Assembler program is run, run time exception may occur. If an exception condition is detected it is reported in the following way:

```
Run-time error - dump:

Exception cause <exception text>
Execution stopped at line <no> in (sub)program <name>

Call trace:

line number   procedure name   (sub)program name
```

After this heading a trace of the called procedures will occur.

<exception text> will be replaced by one of the following depending on the detected exception:

- ILL. INS. (illegal instruction)
- FLOAT (float error)
- BUSERROR
- ADDRESS
- ALPHABETIC COMPARISON
- ZERODIV (division by zero)
- RANGE (invalid index or range of variable exceeded)
- OVERFLOW (arithmetic overflow)
- I/O
- COPY (more than 255 characters in record in sequential file)
- CONCAT
- EXIT
- HEAP CREATION
- HEAP OVERFLOW
- HEAP ALLOCATION
- HEAP DEALLOCATION
- LONG TO SHORT STRING ERROR
- STRING LENGTH TOO LONG
- TEXT LIBRARY HAS NOT BEEN LOADED
- UNKNOWN

<no> is the source line number where the exception occurs. However if the D option has been toggled off the line number is the line of the routine where the exception occurs.

<name> is the name of the program, subprogram or external Pascal routine where the exception occurs.

A Supermax I/O error code appears when an I/O operation fails, and I/O checking has not been disabled through a (*$C-*) compiler option.

Note that when a run time error occurs, all open files will be closed, and if the files are sequential ones that have been opened with a REWRITE call, end-of-file will be at the current file position.

User handling of run time errors.

The Pascal-assembler system's run time error handling is performed by an internal subroutine, postmortem, written in C. This routine consists of three calls of other subroutines:

```
preerr();       /* dummy routine - delivered empty */
dump(outfile); /* writes run time error dump */
posterr();      /* dummy routine - delivered empty */
```

preerr and posterr are dummy routines that can be rewritten by users, who wish to handle runtime errors differently. If f.ex. the C-routine exit is called in preerr, this will cause the programs in which this version of preerr is linked to terminate with no dump, when a run time error occurs.

The dump routine is called with a parameter of type FILE (a stream). The pascal system uses stderr.

Three internal variables are available to the user:

```
int Pexcno              - contains the number of the exception which
                          caused the run-time error

struct buseinf *Pinfo   - contains additional information on buserrors
                          and address errors
```

(see Supermax Operation System, section 2, signal(2))

If these variables are used the c-module must contain the following inclusion:

```
#include <signal.h>
```

The following example is taken from the run time routine for the compiler pta itself (the compilers are written in Pascal):

```
/*****************************************************/
/*       module: p-prepost.c                         */
/*   written by: abc   date: 01.08.1987              */
/* copyright (c) 1987 by Dansk Data Elektronik A/S   */
/*****************************************************/

static char *-v=" (#) p-prepost.c 01.08.1987";

#include <stdio.h>
#include <pta.h>
#include <signal.h>

extern struct iob LIST;
extern int Pexcno;

preerr()
{
  closf(3,&LIST);   /* close LIST file when exiting */
  if (Pexcno == SIGINT)
  {
    fprintf(stderr,"\ncompilation terminated - interrupt key pressed\n");
    exit(Pexcno);
  }
}

posterr()
{
  if (Pexcno)
    fprintf(stderr,"\ncompilation terminated - completion code %d\n",Pexcno
  exit(Pexcno);
}
```

These routines cause the compiler to skip writing a dump when it was terminated by pressing the ctrl c keys.

B.4 Running Supermax Pascal-Assembler Compiler

When the compiler is linked the module p_prepost.o is given explicitly to the linker causing this module to be linked to the compiler instead of the empty preerror and posterr routines found in the library /lib/libpa.a (or /lib20/libpa.a).

Appendix C. Necessary files.

The Pascal Assembler System consists of the following files:

/pbin/pta	- Pascal to Assembler compiler (mc68000)
/pbin/pta20	- Pascal to Assembler compiler (mc68020)
/pbin/pac	- control program for compilation
/pbin/ppp	- pascal pre-processor
/pbin/setheap	- used to change the heapsize
/lib/libpa.a	- run time routines for Pascal (mc68000)
/lib20/libpa.a	- run time routines for Pascal (mc68020)
/lib/libpext.a	- library containing the external standard routines
/lib/libopen.a	- routines simulating Mikfile modes on files
/lib/liblingua.a	- routines for language independence (mc68000)
/lib20/liblingua.a	- routines for language independence (mc68020)
/pbin/extdecl.p	- the declarations of the external standard routines

Furthermore the following files are required:

mc68000:

/bin/as
/bin/ld
/lib/crt0.o
/lib/libc.a
/lib/libm.a

C.2 Running Supermax Pascal-Assembler Compiler

mc68020:

/bin/as20
/bin/ld
/lib20/crt0.o
/lib20/libc.a
/lib20/libm.a

Besides the above mentioned files, it is of course required to have a C-compiler system if one wants to write and use own C-routines.

Along with the Pascal-Assembler System the following files are supplied:

/nlslib/pta/uk - text file containing the syntax error messages

/usr/lib/alphabet/dk - table on the Danish alphabet used for alphabetic comparison on strings.