TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

The PAP preprocessor : a precompiler for a language for concurrent processing on a multiprocessor system

Citation for published version (APA):
Lemmens, W. J. M. (1982). *The PAP preprocessor : a precompiler for a language for concurrent processing on a multiprocessor system*. (EUT report. E, Fac. of Electrical Engineering; Vol. 82-E-130). Technische Hogeschool Eindhoven.

Document status and date:
Published: 01/01/1982

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 05. Oct. 2023

Eindhoven
University of Technology
the Netherlands

Department of
Electrical Engineering

The PAP preprocessor: A precompiler for a language for concurrent processing on a multiprocessor system.

By
W.J.M. Lemmens

EUT Report 82-E-130
ISBN 90-6144-130-7
ISSN 0167-9708
October 1982

Eindhoven University of Technology Research Reports

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Department of Electrical Engineering

Eindhoven The Netherlands

THE PAP PREPROCESSOR:

A precompiler for a language for concurrent processing on a multiprocessor system

By

W.J.M. Lemmens

EUT Report 82-E-130

ISBN 90-6144-130-7

ISSN 0167-9708

Eindhoven

October 1982

CIP-gegevens

Lemmens, W.J.M.

The PAP preprocessor : a precompiler for a language for concurrent processing on a multiprocessor system / by W.J.M. Lemmens ; [publ. of the] Department of electrical engineering, Eindhoven University of Technology. - Eindhoven : University of technology. - Fig. - (Eindhoven University of Technology research reports ; 82-E-130)
Met lit. opg., reg.
ISBN 90-6144-130-7
ISSN 0167-9708
SISO 365.3 UDC 681.3.066 UGI 650
Trefw. : programmeertalen.

Summary.

This report describes the design, implementation and use of a language for concurrent processing on a multiprocessor computer system. It contains a user's manual and implementation notes and ends with a discussion of the advantages and disadvantages of using a preprocessor in addition to a compiler for the translation of programs written in such a language.

Lemmens, W.J.M.
THE PAP PREPROCESSOR: A precompiler for a language for concurrent processing on a multiprocessor system.
Department of Electrical Engineering, Eindhoven University of Technology, 1982.
EUT Report 82-E-130

Address of the author:

Group Measurement and Control,
Department of Electrical Engineering,
Eindhoven University of Technology,
P.O. Box 513,
5600 MB EINDHOVEN,
The Netherlands

Contents:

Summary.

Contents.

Preface. 3

Part I. PAP user's manual.

1. Introduction. 5
2. PAP features. 6
3. PAP syntax. 8
4. Structure of the list file. 11
5. Operating environment. 12
6. Error messages. 12

Part II. Preprocessor design and implementation.

1. Introduction. 16
2. The parser. 18
3. Output text generation. 20
4. Some considerations on building a preprocessor. 22

Literature. 24

Appendices.

A. Insertion and replacement of texts.

B. The text generation software.

C. Other preprocessor subprograms and data structures.

Preface.

At the measurement and control group of the department of electrical engineering of the Eindhoven University of Technology we have built a multiprocessor computer configuration for measurement and control applications. For this system we have created an advanced language for concurrent programming, named PAP (PAscal Plus real-time extensions), and built the facilities to use it. This language and the hardware mentioned have so far been used for practical work assignments for M. Sc. students of several specialisations and as an object of study for M. Sc. theses. The report presented here discusses the design and implementation of PAP. A report covering the whole project will be published separately.

This report is divided in two parts: Part I constitutes a user's manual, aimed at those people who want to use the system, but do not intend to change it or otherwise want to be concerned with its inner workings. It mainly describes how the system should be used and what it should be used for. However, those who are interested in the development and construction of the system will need it as an introduction to its features and use.

In part II, some aspects that have been mentioned in part I are elaborated. This part deals specifically with the preprocessor which forms part of the PAP system. It contains a description of the construction and operation of the program and some considerations on which these are based.

The appendices document some aspects of the system in more detail, facilitating the alterations and additions that may be needed for maintenance and extension of the package.

Part I.

PAP user's manual.

I.1. Introduction.

PAP is a programming language based on Pascal and designed for real-time multiprocessor applications. Programs written in PAP will run on a PDP-11 single- or multiprocessor system. Therefore the PAP package consists of three parts:

1. A preprocessor for the conversion of PAP programs to standard Pascal programs.

2. A sligthly modified Pascal compiler to translate the output of the preprocessor to machine code.

3. A run-time system specifically built for the PAP system.

This report is mostly concerned with the preprocessor mentioned above. The preprocessor constitutes an extra translation step before the compiler. It checks the correct use of the features that have been added to standard Pascal and produces Pascal programs as output. To be more specific, it has the following tasks:

1. To insert declarations for PAP standard types, variables, functions and procedures.

2. To check the syntactical correctness of PAP programs, or at least of the PAP-specific elements of PAP programs.

3. To perform conversion of PAP language elements to Pascal constructs, e.g. PROCESS translates to PROCEDURE, CYCLE to WHILE TRUE DO BEGIN, etc.

4. To establish links to the run-time system, e.g. by insertion of system routine calls for starting up of processes or for checking stack overflow.

5. To insert initialisation statements for the PAP variables that need initialisation.

6. To carry out type compatibility checks of PAP variables used in PAP operations.

7. To investigate possible deadlock situations when using critical regions.

8. To compact the output text in order to speed up the Pascal compilation.

9. To produce a listing of the program, including input and output line numbers, error messages and summaries of declared identifiers.

The following chapters provide more information on PAP features and PAP use.

I.2. PAP features.

PAP as a language is nothing more than standard PAscal Plus some added features to allow real-time programming:

The reserved word PROCESS designates the following program unit as a software module that may be executed concurrently with, and independently of, other modules (see lit. 1).

The reserved word CYCLE is equivalent to the Pascal construct WHILE TRUE DO BEGIN and marks the beginning of a block that should be executed for an indefinite period of time.

PAP features three different communication and/or synchronisation primitives: Critical regions, semaphores and message buffers with their corresponding messages (see lit. 2). These are available as predeclared types. Variables of these types must be declared using the reserved words REGION, SEMAF, MSGBUF and MESSAGE respectively.

1. Critical regions may be used to restrict access to certain devices or groups of data to only one process at a time. A region may be entered by using REGENTER <regname> and left through REGEXIT <regname>.

2. Semaphores are used as a synchronising mechanism between processes. The only operations to be performed upon variables of this type are SIGNAL(SEM) and WAIT(SEM). SEM may be any identifier of type SEMAF.

3. Messages and message buffers are a means of communication among processes. Messages are passed from one process to another by way of message buffers. They are arrays of CHAR (or ASCII or BYTE) or INTEGER, declared in a special way to allow type checking with SEND(MSG,MBUF) and RECEIVE(MSG,MBUF) operations, whereby MSG is a message and MBUF is a message buffer. These operations constitute the only admissible way to access message buffers.

Besides semaphores, messages and message buffers, arrays of elements of these types may be declared, using the reserved word VECTOR. Vectors are limited to one dimension only, unlike conventional arrays.

Procedure declarations follow Pascal rules, but there are two extensions:

1. Declarations of procedures and functions may be preceded by the reserved word SYSTEM. They are thereby declared as system procedures, which means that the preprocessor automatically issues

some process identification with every procedure or function call. This feature is mostly used by operating system procedures and its use is transparent to the normal user.

2. The type specification of a procedure or function parameter may be preceded by the word symbol UNIV. This suppresses type compatibility checks between actual and formal parameters at a subprogram call. Any parameters declared UNIV must also be declared VAR in the procedure/function declaration. This feature allows type conversion between any two types that can be mapped upon the same memory segment. It should however be used with the utmost care as erroneous use affects the integrity of the whole program.

Apart from the standard procedures SIGNAL, WAIT, SEND and RECEIVE and the standard Pascal procedures, PAP offers some more predeclared procedures:

1. ATTACH(DEV,SEM), which associates semaphore SEM with interrupt vector (adress) DEV. It enables a specific device to signal the associated semaphore by generating an interrupt.

2. START(PROCQ,PRI,HL,SL,PROCS,..) activates process PROCS so it may compete for processor time concurrently with other processes. PROCQ indicates the queue in which the process is to reside while being active (see below), but not running, PRI is the process priority, HL and SL are the length of the hardware and software stack respectively. The dots after the process name indicate any parameters the process will have. All process parameters must be given a value at the invocation of the start procedure.

3. BYE deactivates the process from which it is called, effectively removing it from the system (but not from memory).

4. READREG(PART,REG) is a function of type WORD, which supplies the content of memory location REG within partition PART. WORD is a predeclared type meaning the contents of a memory word without conversion to any specific type.

5. WRITEREG(PART,REG,CONT) transfers the value of CONT to location REG in partition PART.

6. STRINGTOINTEGER(S,I) converts string S of ASCII-coded figures to an integer value that is assigned to I.

7. INTEGERTOSTRING(I,S) accomplishes the reverse: It is a procedure that converts integer value I to string S.

For a more detailed description of these subprograms, see lit. 6.

There are three system dependent predeclared variables: PROCAQ, PROCBQ and READYQ that are used as a parameter in the START procedure. They indicate

whether a process is to be run exclusively on one specific processor, and if so, on which one.

PAP offers possibilities for code sharing among processes. In fact, this is the only reason why processes may have parameter lists. Any process may be started more than once, each time with different values of the parameters and in a different queue.

I.3. PAP syntax.

The syntax graphs depicted here are intended as an addition to the Pascal graphs (see lit. 3). Together with the Pascal graphs and the following notes they define the syntax of the PAP language.

Notes:

1. <declaration part> is identical to the Pascal <block> excluding BEGIN <statement> {; <statement>} END .
   Procedure and function declarations may be preceded by the reserved word SYSTEM (see chap. I.2).

2. The PAP statement is identical to the Pascal statement with the addition of | REGENTER <region identifier> <statement> {; <statement>} REGEXIT <region identifier> .

3. The definition of the PAP <parameter list> given here replaces the Pascal definition. It is identical to this except for UNIV.

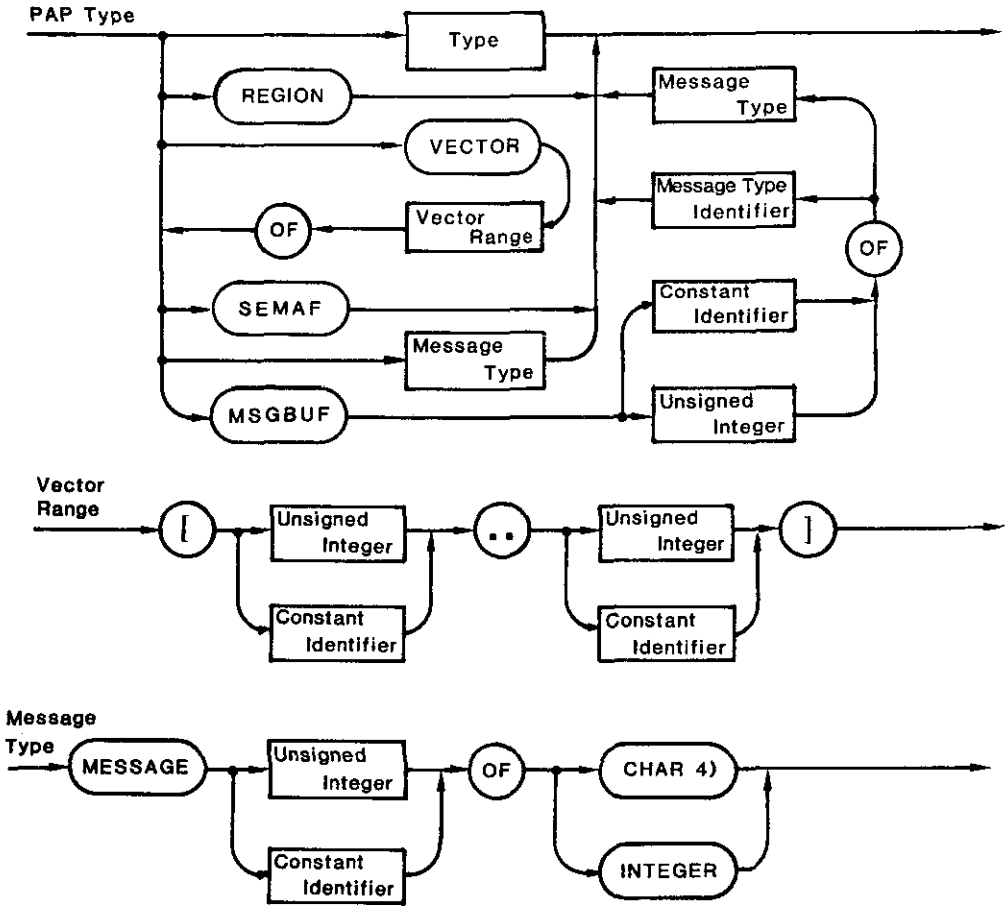
4. BYTE or ASCII are also allowed message element types.

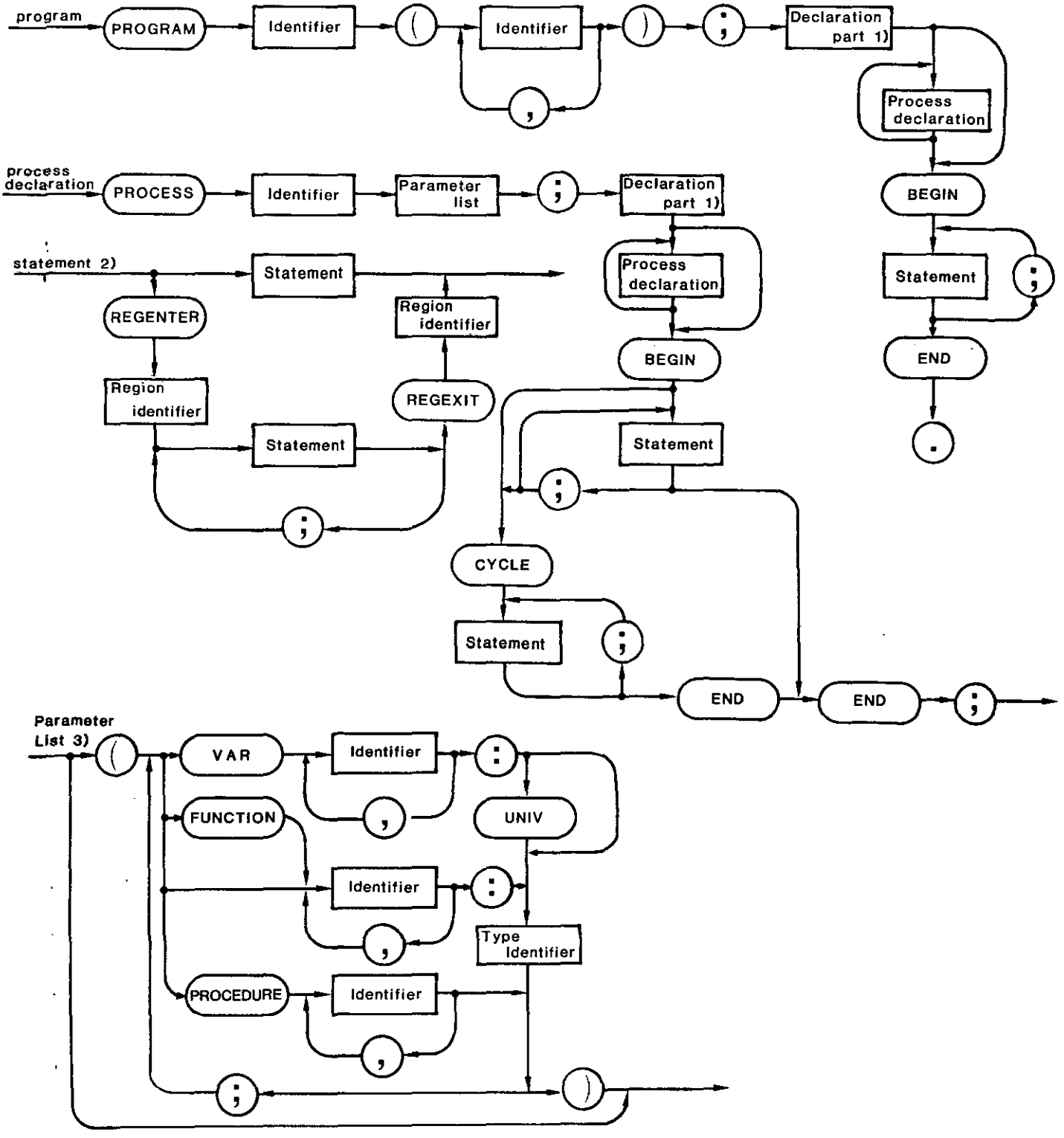
The Pascal scope rules apply also to processes declared within other processes. These processes are considered local to the enclosing process, so the START procedure calls for these processes should be situated within the statement part of that process.

In addition to the word delimiters of Pascal, PAP has the following reserved words: ASCII, BYTE, CHAR, CYCLE, INTEGER, MESSAGE, MSGBUF, PROCESS, REGENTER, REGEXIT, REGION, SEMAF, SYSTEM, VECTOR. These should never be used as identifiers anywhere within a PAP program. The word UNIV has a special meaning only if it precedes a type identifier within a parameter list of a process, procedure or function.

As the user program and part of the system software are merged into one Pascal program by the preprocessor, a number of reserved identifiers should not occur in a user declaration at the program and/or process level because they are used internally by the system. INITIALISE, INITTOS, NUMMER, PHOENIX, R5, TEMPQPNTR, TOPOFSTACK, ERRORBUF should not be declared at the program level. INIT, NEWCHK, NEWREG, NEWSTK, OVFLCHK, PDPOINTER, STARTREG

should occur neither within a program nor in a process declaration part. The same goes for some of the system identifiers that are accessible to the user: BYE, PROCAQ, PROCBQ, READYQ, SIGNAL and WAIT. SEND and RECEIVE should never be redeclared as system procedures or functions, and NEW and START never as normal subprograms because the preprocessor scans the program for subprograms having these names in order to do certain checks or insertions.




program
PROGRAM
Identifier
(
Identifier
)
;
Declaration part 1)
,
Process declaration
process declaration
PROCESS
Identifier
Parameter list
;
Declaration part 1)
BEGIN
statement 2)
Statement
Process declaration
Statement
;
REGENTER
Region identifier
END
Region identifier
REGEXIT
BEGIN
Statement
Statement
;
CYCLE
Statement
;
END
END
;
Parameter List 3)
(
VAR
Identifier
:
,
FUNCTION
UNIV
Identifier
:
,
Type Identifier
PROCEDURE
Identifier
,
;
)

I.4. Structure of the list file.

Upon request the preprocessor produces a listing of the source program, which contains the original program text plus information on identifiers declared and errors detected within the program. The source text lines in the list file are preceded by two numbers: first a line number of the input text and second a line number of the output text. In this way the Pascal compilation that eventually follows does not need to produce a listing of its own. This would be rather illegible anyhow, as the preprocessor compresses the output text and any lay-out from the source text is totally disturbed. So if the Pascal compiler displays an error message, the corresponding line can be found in the preprocessor listing.

In between the source text lines overviews of identifiers declared within the program unit concerned are given. These are listed in alfabetical order, followed by an indication of their kind (constant, type, variable or process identifier), their type (for type or variable identifiers) or value (for integer constant identifiers) and some information that depends on their type. The kind of the procedure and function identifiers is taken to be variable. The type of type or variable identifiers is indicated by a number:

0 = Semaphore.
1 = Message.
2 = Message buffer.
3 = Critical region.
4 = Vector.
5 = Procedure or function.
6 = The rest, i.e. all Pascal types.

For type identifiers this is all that is displayed. Variable identifiers that are declared in the parameter list of the process concerned will have PARAMETER VARIABLE displayed after their type indication. Of the other variables more information is given:

1. For messages the number of elements and the type of the elements (word type or byte type).

2. For message buffers the maximum number of messages they may contain and the corresponding message type identifier.

3. For vectors the lower boundary, the upper boundary, and an element type identifier.

4. And for procedures/functions a list of critical regions and subprograms called from within the procedure/function body.

These overviews are given each time immediately after the first BEGIN of the statement part of the program or process.

If the preprocessor detects an error somewhere in the program text, the

corresponding line in the list file is closed at that point and an error message is inserted at the next line. The program text is continued on the line thereafter, following a string of '*'-characters that replaces the line numbers. For possible error messages, see I.6.

Each page of the listing has a header in which the name of the list file and the date and time of production are mentioned.

I.5. Operating environment.

PAP, or more specifically the PAP run-time system is originally designed to run on a multiprocessor system composed of two or more LSI-11 processors connected to one or more memory units by way of a crossbar system. Each memory unit has an arbiter that decides which processor may have access to the unit and sets the crossbar switches accordingly. Peripheral devices are directly connected to one of the processor busses, so any process that needs a device must be active on the corresponding processor exclusively. Hence the different processor queues.
Processes that are running on a certain processor keep doing so until they become waiting through execution of a WAIT operation. Until that occurs, they may only be temporarily suspended because of a hardware interrupt.
A processor that becomes idle, e.g. because of a wait operation performed by the process that was being executed on it, first searches its own queue for new processes to execute. if that queue is empty, the CPU inspects the READYQ. In the queues processes are sorted according to their priority, so high priority processes are handled before processes having a lower priority. This goes for both the active queues and the semaphore queues, in which processes reside while waiting.

The run-time system is made exclusively for the configuration described above. If the software is to be transferred onto an other hardware system the run-time system should be rewritten for that system. The preprocessor may run on any system that accepts the Pascal in which it is written and the only system-specific aspect of its output is the use of ready and processor queues.

For more information on the hardware configuration, see Lit. 4. The run-time system is described in Lit. 5 and 6.

I.6. Error messages.

If during preprocessing an error is encountered, the current line in the list file is closed and an error message is inserted. After preprocessing is finished, the number of errors encountered is displayed on the terminal. Error messages are:

OUTPUT BUFFER OVERFLOW. Indicates the presence of a string that is

too long and without spaces or TAB-characters. May be an indication of missing closing symbol ("*)" or "!") in a comment or literal string.

INTEGER VALUE TOO LARGE. Integers must have a value between +32767 and -32768.

UNDECLARED IDENTIFIER. Non-standard identifiers must be declared before they are used.

IDENTIFIER DECLARED TWICE. No two identifiers within the same scope may have the same name, except field identifiers of different records or field identifiers and other identifiers.

ILLEGAL VECTOR TYPE. Only semaphores, messages and buffers are allowed as VECTOR elements.

TYPE NOT ALLOWED IN MESSAGE. Messages may only be of CHAR, ASCII, BYTE or INTEGER type.

ERROR IN BUFFER TYPE. Buffers may only be declared for legal message types.

WRONG DECLARATION ORDER. LABEL declarations should come first. Next come CONST declarations, next TYPE declarations, then VAR declarations, then PROCEDURE and FUNCTION declarations and thereafter PROCESS declarations. Statements should be last.

NO REGION IDENTIFIER. REGENTER should be followed by a REGION identifier.

WRONG NESTING OF REGIONS. No REGION identifier or the wrong REGION identifier after REGEXIT.

DEAR USER,
I AM SORRY, BUT I'M NOT QUITE SURE WHAT YOU ARE TRYING TO ACHIEVE. YOU SEE, IN MY HUMBLE OPINION, YOU JUST TRIED TO ENTER A REGION THAT WAS ALREADY OCCUPIED BY THE CURRENT PROCESS. PERHAPS, IN YOUR INFINITE WISDOM, YOU HAVE REASONS TO CREATE A MASSIVE DEADLOCK THIS WAY, BUT MAYBE IT'S JUST A SILLY MISTAKE. NEVER MIND THEN, ANY HUMAN IS ENTITLED TO ITS OCCASIONAL ERRORS, EVEN THOUGH HE DOESN'T EXPECT THEM FROM ME. IN CASE IT WAS INTENDED, PLEASE FORGIVE MY RUDE INTERFERENCE.

THIS IS NOT A CONSTANT IDENTIFIER. And probably it should be.

ILLEGAL USE OF PROCESS NAME. A process name may only occur in a process declaration or in a START procedure call.

THIS IS NOT A MESSAGE. The first operand in a SEND or RECEIVE operation should be of MESSAGE type.

THIS IS NOT A BUFFER. The second operand in a SEND or RECEIVE operation should be of MSGBUF type.

INCOMPATIBLE MESSAGE AND BUFFER TYPES. The message type in a SEND or RECEIVE operation should match the type for which the buffer is declared.

IDENTIFIER EXPECTED.
"BEGIN" EXPECTED.
"OF" EXPECTED.
"REGEXIT" EXPECTED.
"END" EXPECTED.
UNSIGNED INTEGER OR INTEGER CONSTANT EXPECTED.
RIGHT PARENTHESIS EXPECTED.
COLON EXPECTED.
SEMICOLON EXPECTED.
PERIOD EXPECTED.
FINAL PERIOD (".") EXPECTED.
These messages are more or less self-explanatory.

UNEXPECTED END OF INPUT. Probably too little ENDs or too many BEGINs.

Apart from the error messages in the list file, the following messages may appear on the terminal during preprocessing:

PAP PREPROCESSOR. OUTPUT = ooo, INPUT = iii. This message is generated at the start of the preprocessing if all goes well. ooo and iii are the output and the input file name, respectively.

OPEN FAILURE ON INPUT FILE indicates that a file with the name given for the input file does not exist or is otherwise inaccessible. If this message is generated, the part INPUT = iii is omitted in the previous string.

PREPROCESSING ABORTED AFTER nnnn LINES OF INPUT is generated if the end of the input file is encountered before regular completion of the program is detected.

END OF PREPROCESSING.
eeeeee ERRORS DETECTED, nnnn LINES READ. This message concludes preprocessing if the preprocessor program reached its regular termination. eeeeee Indicates the number of errors encountered during preprocessing, nnnn is the number of lines processed.

Part II.

Preprocessor design and implementation.

II.1. Introduction.

The PAP preprocessor has many of the features of a Pascal compiler. It comprises a lexical scanner, a parser, a kind of code generator and software to produce a listing of the program text. In fact, the choice of implementing the PAP language by building a preprocessor in stead of a full blown compiler causes many things to be done twice while preparing a program for execution on the system. Constructing a new compiler or converting an existing compiler to PAP however would have cost many more man hours.

1. The lexical scan identifies strings of characters and replaces them with a symbol code plus attribute values, like the name of an identifier or the magnitude of an unsigned integer. Spaces, TABs, carriage returns and other delimiters are skipped. However, because the preprocessor output should be a file suitable for compilation by a standard Pascal compiler, the symbol codes are not exported to the compiler and are used only internally. Most of the program text is passed unchanged to the output text generator. Only superfluous delimiters, used for enhancement of readability of the program, are discarded, as are comments.

2. The parser should examine programs as to their accordance to the PAP-specific syntax rules, but it does not necessarily need to analyse the standard Pascal constructs of the programs. So, in contrast to parsers for other languages, it does not pass judgment over all language constructs used, but it transfers part of the program unanalysed to the output file. Thus in the declaration part only region, vector, semaphore, message and message buffer declarations are processed and the statement part of program units is treated as an unstructured stream of symbols and identifiers with BEGIN, END, CYCLE, REGENTER and REGEXIT-markers standing out. From a certain point on parsing is limited to merely looking for certain identifiers in the program until a certain symbol is reached. Upon finding one of these identifiers - SEND, RECEIVE, NEW, START and all process and system subprogram names - certain actions are performed after which scanning proceeds.
   Besides violations of the context free syntax rules the parser also detects any deviations from the scope rules or the declaration conventions, as far as typical PAP variables are concerned, and the use of illegal types in PAP operations or expressions.
   Because the parser analyses the program it is able to decide where certain texts should be inserted or which part of the input text should be replaced by a different text. So another task of the parser is the execution of control over the output text generator.

3. Most of the input text appears at the output essentially unaltered, with unnecessary delimiters removed and certain declarations and statements inserted. But only Pascal symbols and identifiers, delimiters that are indispensable and certain compiler directives are passed on to the output file, so specific PAP symbols have to

be replaced by a legal Pascal text. The replacement and insertion of texts is directed by the parser. The texts to be inserted or to be used instead of those input symbols are contained in a file on disk, together with the text of the error messages that may appear in a listing. Every text on disk has an unique number. With this number it may be accessed through a table that is loaded from disk during initialisation of the preprocessor. The randomly accessible texts may contain one or more insertion symbols ('@'-characters) that are to be be substituted with any other text from the text file, or a name from the identifier table, or the current symbol in the input text, or any number expressed in ASCII characters. A text as it appears in the output file may thus be composed of several texts from the text file, combined with symbols or names from the input and numbers generated by the preprocessor.

4. The program listing is assembled from a number of components:

   1. The original program text. This is put into the list file by the lexical scan software.

   2. Page heading and input and output line numbers. Every time a new line is to be started a procedure is called that checks first if the current page is full and starts a new page if necessary. After that the new line is started and this is provided with line numbers if it is an input text line.

   3. Error messages, if necessary. These are inserted by the parser, using the text insertion subprograms that are also used by the code generation software.

   4. An overview of all identifiers declared within a process or in the main program. This is produced by the procedure that also takes care of the initialisation of PAP variables, as it scans the identifier table.

   So there is no specific listing part of the preprocessor. All parts contribute to the composition of the list file.

Added to all this are some procedures for input of file names via the user terminal and initialisation of the preprocessor itself.
The parts mentioned here all are active simultaneously, but not concurrently. Ours is a one-pass preprocessor, contained in one sequential program, with procedures and functions that are called when they are needed.

The compiler that is used to convert the preprocessor output to machine code is the so-called Ericsson compiler (lit. 7), but in fact any reliable Pascal compiler would have been adequate. Only, it proved necessary for the

implementation of SEND and RECEIVE in PAP and in order to create a PAP file handling system (lit. 8), to be able to suppress type checking at some points. Therefore provisions have been made to switch off the compiler type compatibility check for procedure and function parameters. As descibed in I.2 the word UNIV has been introduced for this purpose. A VAR parameter of which the type identifier is preceded by UNIV in the subprogram declaration, is marked as universal in the compiler identifier table and no compatibility check is made on this parameter when the procedure or function concerned is called. There are compilers with this feature built in (lit. 9), but we chose to adapt the Ericsson compiler because that was better suited for our purpose.
The programmer is now able to make implicit type conversions between a subprogram and its environment: The variables in question may be of one type outside the subprogram, but may be treated as of different type inside the subprogram body. Also, variables of different lengths may be accommodated by one procedure or function.

II.2. The parser.

The parser is of recursive descent type (lit. 10), so the syntax graphs of chap. I.3 may be viewed as a blueprint for the structure of the parser: there is a procedure for the syntactical item <program> which calls a procedure for the declaration part, which among others calls a procedure for parameter lists, etc. until finally the presence of a specific symbol is verified.

The primary task of the parser is to check whether the program contains errors. At every point in the syntax graph there is a limited number of allowed continuation symbols, for example in a FOR statement, after the assignment only TO or DOWNTO are legal. If the symbol detected at a certain point in the program text does not belong to the current set of continuation symbols or is an identifier of the wrong kind, we have an error condition. This is notified in the list file in the form of an error message which, if possible, mentions the symbol expected at that point. The boolean variable ERROR becomes TRUE and thus suppresses further error messages.
During the error state attempts are made to restore the situation to normal: If a specific symbol was expected a search is made for that symbol by inspecting the next symbol in the program. In this way for every symbol to be evaluated two symbols are inspected: the current symbol and the next. This method deals effectively with erroneously inserted symbols and incorrectly spelled symbols. There is however no guarantee that this will lead to a return to the normal state. For one thing, the parser does not always call the routine that performs the actions described above. It also has other ways to deal with specific symbols.
Therefore some key procedures are constructed in such a way that they return control to the calling program upon reaching one of a group of closing symbols, regardless of their inner state at the moment. This deals with the inadvertent omission of necessary symbols from the source program. Any symbols between the occurrence of the error and the closing symbol are

ignored.
During an error state the output generation is not interrupted, but heuristical attempts are made to produce an output with as little errors as possible. As soon as the parser finds a symbol that matches the syntax definition at that point or an identifier of the correct kind, the ERROR variable becomes FALSE and the situation reverts to normal.

The preprocessor maintains a table of most of the identifiers declared in a user program. The structure of this table reflects the program structure, so adherence to the scope rules can be verified rather easily. An other purpose of this table is to enable type checking:

1. Every time a message, a buffer or a vector is declared, a check is made to ensure that the elements are of the right type.

2. The operands in a send or receive operation are examined for their compatibility: the length of the buffer elements should be the same as that of the messages transmitted and the types of the constituents of both should tally. Hereby CHAR-types are taken to be compatible with ASCII and BYTE-types.

3. The identifier that follows a REGENTER or REGEXIT should have been declared as a variable of type REGION. Further the identifiers that follow corresponding REGENTER and REGEXIT delimiters should be identical and no REGENTER should be done on a critical region that has been entered before and has not been left since. All this is checked by the preprocessor, even inside the body of procedures that are called from within a region.

The identifier table is implemented as a stack of trees: For every block that is entered a new table is started that is linked to the previous one. These subtables take the form of a tree of identifier records in which the nodes are ordered lexicographically, according to the names of the identifiers. That is, every node of the tree has two subtrees, a left one and a right one. The left subtree only contains records of identifiers of which the names lexicographically precede the name of the node identifier, while the right tree contains identifiers with names that succeed it in lexicographical order. Alternatively, one or both of the subtrees may be empty.
The records of which the trees are built contain all information that the preprocessor may need on the identifier concerned, such as its name, whether it is a constant, type, variable or process identifier, whether it has been declared formally in a parameter list or actually in the declaration part of a program or subprogram block. For integer constants the value is stored and for type and variable identifiers the type is registered, with for each type some characteristic data, like the number of elements it contains and a reference to the corresponding message type for a message buffer or element type and boundaries for a vector. Also for each record in the tree the number of records by which it is referenced is noted (see below). Procedures and functions are taken to be variables.
After a block is left the corresponding tree is broken down and the records

are returned to the pool of free records, but with a few exceptions. The stacked tree structure harbours an other structure of linked records, namely the lists of regions entered and procedures and functions called from inside a block. Every procedure and function record has such a list attached and the records of the list should not be discarded until the owner of the list disappears from the identifier table. So even after the parser has left the block where they have been declared, some identifier records may continue their existence. This goes specifically for procedures and functions declared within other procedures or functions.
The list described here is set up to create the possibility for a deadlock test as mentioned above. It is scanned every time the procedure or function to which it belongs is called. The regions in the list are inspected to see whether they are occupied and a deadlock warning is issued if that is the case.

Besides error detection and report generation, the parser executes control over the output text generation by calling the approriate procedure of the output generator part when needed.

II.3. Output text generation.

If the source program is a correct PAP program, then the preprocessor output should be a correct Pascal program. So, first specific PAP symbols should in the output file be replaced with their Pascal equivalents, or, if they are preprocessor directives, be discarded altogether. We already encountered the conversion of PROCESS to PROCEDURE and CYCLE to WHILE TRUE DO in I.1. Other examples are:

1. MESSAGE n OF .. translates to ARRAY[0..m] OF.., with m=n-1.

2. REGENTER r translates to BEGIN WAIT(r);

3. START(a,b,c,d,e,..) to NEWSTK(a,b,c,d); e(..)

4. SYSTEM is skipped.

But apart from such direct translations there are quite a number of texts that are added to the output for a variety of reasons. New types that are standard in PAP must be declared in Pascal, such as SEMAF, REGION and WORD. The same goes for the standard procedures and functions that are mentioned in I.2. Then some variables that are used internally by the system are added and the statements for the initialisation of PAP types are inserted in the Pascal text.
The initialisation of variables declared in the declaration part of a program or a process takes place immediately after the initialisation of that module itself. After the BEGIN that follows the declaration part first some statements are inserted for initialisation of certain system data structures and then the procedure PRTREE(ROOTOFLOCALIDTREE) is called. This

procedure recursively scans the local identifier tree and performs two acts: It passes a description of the identifiers it encounters to the list file and it inserts the statements needed for the initialisation of any variables it finds that require initialisation in the output file. The result of all this is an alfabetical list of identifiers in the list file, with their kind, their type or their value, and any other relevant data mentioned. In the output file there will appear a group of statements that establish initial values for semaphore counters, buffer indices and so on, that form part of the variables declared within the process in which the initialisation occurs.
Finally, there are special constructs to be built, e.g. critical regions around START and NEW, a process that does not contain a CYCLE should be terminated after completion, etc.
For more details, see appendix A.

The decision to discard a piece from the input or to pass it on will only be made after the part concerned has been analysed by the parser. So the lexical scan software may not transfer the text it reads directly to the output. An output buffer is used instead, where pieces of text reside while they are being analysed. After that they may be discarded or put into the output file.
The output buffer is a circular buffer of 132 characters. It will never contain more than one PAP symbol plus leading delimiters. These are (strings of) characters that have no special meaning for the preprocessor but they may have a meaning in Pascal. Associated with the buffer are two pointers: a putpointer that points to the position where new characters may be inserted, and a getpointer that indicates the position from where the next character should be taken when emptying the buffer. Discarding the contents of the buffer is accomplished by making the putpointer equal to the getpointer. Text to be inserted is put directly into the output file. It will therefore appear before the current symbol in the Pascal text produced, if that symbol is not discarded, in which case we have a replacement of the input text.
The texts to be inserted are stored in a file on disk from which they have to be retrieved in random order. This file is prepared from a file made by the person that built the PAP system by using a special program. This program creates the file in which the texts are directly accessible and a table in which the text number is associated with the position of the text in the text file. Insertion of texts, using this table, is done by:
PROCEDURE OUTXT(IFIL:DESTIN; NTXT:INTEGER; ILST:INLIST)
The parameters of this procedure indicate successively:

1. The file to which the text is to be transported. IFIL=LISF indicates the list file, IFIL=OUTF the output file.

2. NTXT is the text number, the index in the text table.

3. ILST is a pointer that indicates the list of elements that should take the place of the '@'-characters in the indicated text. The elements are characterised by a code and an attribute, such as a numerical value or a name string, if necessary. If the element to be inserted is an other text from the text file, it will have its

own insert list as an attribute.

Appendix B offers a full account on how the insertion of texts into texts that themselves are to be inserted into other texts is accomplished.

II.4. Some considerations on building a preprocesssor.

As we noted in II.1, the use of a preprocessor instead of a specially developed compiler creates a certain amount of overhead and duplication. The chosen solution represents a trade-off between excessive development cost and utilisation cost.

The decision to build a preprocessor that does not fully duplicate some compiler functions poses considerable limitations on the language design. Especially the variable and type declarations may take time to evaluate, if we allow Pascal and PAP types to be fully mixed. In that case namely, we have to have all ARRAY, RECORD, FILE and pointer type declarations evaluated in addition to the PAP types, because of possible combinations of these types with PAP types. Therefore, declarations of types like

```
ARRAY[BOUNDS] OF RECORD P:POINTER;
                        I:INTEGER;
                        S:SEMAF
                 END
```

are not allowed in PAP. PAP types should not be mixed with Pascal types and that is the reason why they stand beside the Pascal types in the syntax graphs. That is also the reason why we needed to introduce the VECTOR keyword to declare arrays of PAP types.
The same considerations apply to the message concept. Ideally one would like to transfer any variable by way of message buffers. Just to make that possible we have introduced the UNIV symbol. This leaves type compatibility checks of the operands in SEND and RECEIVE operations to the preprocessor. But to allow all variable types in a message buffer, the preprocessor should fully analyse all types declared within the program. That is why the message type is introduced and the range of types of message elements is limited to CHAR, ASCII, BYTE or INTEGER.
In such a way duplication of compiler functions in the preprocessor is kept within reasonable bounds: The analysis of Pascal type declarations is left to the Pascal compiler. These declarations are only scanned for the detection of the closing symbol, after which the search for PAP types is resumed. However, in order to detect the closing symbol in a record declaration, which is END, the preprocessor must be aware that the type is a record type, and therefore must be able to detect the starting word symbol RECORD. The same applies to the parentheses surrounding enumerations.

A comparable situation occurs during the processing of statements. The preprocessor does not analyse Pascal constructs like WHILE <expression> DO <statement>, but it should distinguish the cyclic part of a process from the rest of the statement part and the statement part from the rest of the process body. Therefore it should detect the enclosing CYCLE and END symbols

and BEGIN and END symbols, respectively. But in order to associate the correct END with the BEGIN or CYCLE, it should be able to detect CASE and END pairs as well. So the CASE symbol has to be detected by the preprocessor, like the RECORD symbol, although they have no special meaning in PAP.

These are only a few examples to demonstrate that a certain duplication of actions between the preprocessor and the Pascal compiler is unavoidable. As we have shown, care has been taken to limit this duplication to the least possible. But there are even more disadvantages of using a preprocessor, such as the fact that PAP variables have to be initialised by statements that are inserted into the Pascal text. These statements raise the load on the compiler and the amount of memory occupied by the compiled program. A compiler would have initialised the variables itself, without expanding the object program. A PAP compiler, however, would have been almost ten times as big as the preprocessor, with accompanying extra cost in manpower.

Literature

(1) Dijkstra, E.W.
CO-OPERATING SEQUENTIAL PROCESSES.
In: PROGRAMMING LANGUAGES. NATO Advanced Study Institute Summer School, Villard-de-Lans, 1966. Ed. by F. Genuys.
London: Academic Press, 1968. P. 43-112.

(2) Brinch Hansen, P.
OPERATING SYSTEM PRINCIPLES.
Englewood Cliffs, N.J.: Prentice-Hall, 1973.

(3) Jensen, K. and N. Wirth
PASCAL USER MANUAL AND REPORT. 2nd ed.
New York: Springer, 1975.

(4) Kanters, M.J.
HARDWARE FOR A 'MULTIPROCESSOR SYSTEM' WITH LSI-11.
In: Proc. European DECUS Symp., Monte Carlo, 4-6 Sept. 1979, Proc. of the Digital Equipment Computer Users Society, Vol. 6 (1979), No. 1, p. 39-41.
Maynard, Mass.: Digital Equipment Corp., 1979.

(5) Dekker, W.P.M. den
DEVELOPMENT OF A SOFTWARE PACKAGE FOR A MULTIPROCESSOR SYSTEM (in Dutch).
Project report. Group Measurement and Control, Department of Electrical Engineering, Eindhoven University of Technology, 1979.

(6) Meulenbroeks, F.H.J.M.
PASCAL FOR PARALLEL PROCESSES: An implementation on a multiprocessor system.
M.Sc. Thesis. Group Measurement and Control, Department of Electrical Engineering, Eindhoven University of Technology, 1982.

(7) Torstendahl, S.
PASCAL USER MANUAL: PASCAL for PDP 11 under RSX/IAS. 1980.
A report intended as a supplement to (3). Available from the author at: Telefonaktiebolaget L.M. Ericsson, S-126 25 Stockholm (Sweden).

(8) Kruysdijk, H.J.M. van
A FILE SYSTEM FOR A MULTIPROCESSOR CONFIGURATION (in Dutch).
M.Sc. Thesis. Group Measurement and Control, Department of Electrical Engineering, Eindhoven University of Technology, 1981.

(9) Hartmann, A.C.
A CONCURRENT PASCAL COMPILER FOR MINICOMPUTERS.
Berlin: Springer, 1977.
Lecture notes in computer science, Vol. 50.

(10) Gries, D.
COMPILER CONSTRUCTION FOR DIGITAL COMPUTERS.
New York: Wiley, 1971.

APPENDIX A

Insertion and replacement of texts.

This appendix presents an overview of all texts that may be inserted into the output file.

1. TYPE @ is inserted if the program does not have any type declarations.

2.
```
WORD=INTEGER;
ADDRESS=WORD;
PDPT=PRODESC;
QPOINTER=QUEUE;
PRODESC=RECORD SP:WORD;
                PROCXQ:QPOINTER;
                PS:INTEGER;
                NR:INTEGER;
                PRI:INTEGER;
                HST:INTEGER;
                SST:INTEGER;
                NEXT:PDPT
        END;
QUEUE=RECORD SLOT:WORD;
                GET:PDPT
END;
SEMAF=RECORD SLOT:WORD;
                CTR:INTEGER;
                GET:PDPT
        END;
REGION=SEMAF;
MSGBUF=RECORD EMPTY:SEMAF;
                FULL:SEMAF;
                MUTEX:SEMAF;
                LENGTH:INTEGER;
                NBYT:INTEGER;
                GETPT:INTEGER;
                PUTPT:INTEGER;
                BUF:ARRAY[0..1]OF ARRAY[0..1] OF CHAR
        END;
ERRORMES=ARRAY[0..29] OF BYTE;
ERRORBUFTYPE=RECORD EMPTY:SEMAF;
```

```
                        FULL:SEMAF;
                        MUTEX:SEMAF;
                        LENGTH:INTEGER;
                        NBYT:INTEGER;
                        GETPT:INTEGER;
                        PUTPT:INTEGER;
                        BUF:ARRAY[0..9] OF ERRORMES
                END;
```

These are the standard types that are inserted before the user type declarations at the program level or that replace the ' ' in the previous text.

3. ARRAY[0..m] replaces MESSAGE n, whereby m=n-1.

4.
```
RECORD EMPTY:SEMAF;
       FULL:SEMAF;
       MUTEX:SEMAF;
       LENGTH:INTEGER;
       NBYT:INTEGER;
       GETPT:INTEGER;
       PUTPT:INTEGER;
       BUF: ARRAY[0..m] OF <message type>
END
```
replaces MSGBUF n OF <message type>. Here too m=n-1.

5. ARRAY replaces VECTOR.

6. VAR @ is inserted if the program or process does not have VAR declarations. Instead of '@' comes the following single declaration if it concerns a process, or all of the following variable declarations if it concerns the program declaration part.

7.
```
       PDPOINTER:PDPT;
       R5:INTEGER;
       PROCAQ, PROCBQ, READYQ, BYEQUEUE:QPOINTER;
       NUMMER:INTEGER;
       TEMPQPNTR:PDPT;
       TOPOFSTACK:INTEGER;
       NEWREG:REGION;
       STARTREG:REGION;
       ERRORBUF:ERRORBUFTYPE;
```

8.
```
PROCEDURE INITIALISE; EXTERN;
PROCEDURE PHOENIX(VAR PROCAQ:QUEUE;VAR PROCBQ:QUEUE;
       VAR READYQ:QUEUE); EXTERN;
PROCEDURE SIGNAL(VAR SEM:SEMAF); EXTERN;
PROCEDURE WAIT(VAR PDPOINTER:PDPT;VAR SEM:SEMAF); EXTERN;
PROCEDURE ATTACH(DEV:ADDRESS;VAR SEM:SEMAF); EXTERN;
PROCEDURE INIT(VAR PDPOINTER:PDPT); EXTERN;
PROCEDURE BYE(VAR PDPOINTER:PDPT); EXTERN;
PROCEDURE SEND(VAR PDPOINTER:PDPT;
```

```
        VAR MSG:UNIV ARRAY[INTEGER] OF CHAR;
        VAR MBUF:UNIV MSGBUF); EXTERN;
PROCEDURE RECEIVE(VAR PDPOINTER:PDPT;
        VAR MSG:UNIV ARRAY[INTEGER] OF CHAR;
        VAR MBUF:UNIV MSGBUF); EXTERN;
PROCEDURE STRINGTOINTEGER(STRING S;VAR I:INTEGER);EXTERN;
PROCEDURE INTEGERTOSTRING(I:INTEGER;STRING S);EXTERN;
FUNCTION READREG(PART:INTEGER; REG:ADDRESS):WORD; EXTERN;
PROCEDURE WRITEREG(PART:INTEGER; REG:ADDRESS; CONT:WORD);
        EXTERN;
PROCEDURE NEWSTK(QID:QPOINTER;PRI:INTEGER;HL:INTEGER;
        SL:INTEGER); EXTERN;
PROCEDURE OVFLCHK; EXTERN;
PROCEDURE NEWCHK(TOPOFSTACK:INTEGER); EXTERN;
FUNCTION INITTOS:INTEGER; EXTERN;
(* Standard processes *)
PROCEDURE TTIN(VAR CHARBUF:UNIV MSGBUF); EXTERN;
PROCEDURE TTUIT(VAR CHARBUF:UNIV MSGBUF); EXTERN;
PROCEDURE ERRORLOG(VAR BUF:UNIV MSGBUF); EXTERN;
```

These are the standard procedure and function declarations that will appear before the global user subprogram declarations in the output file.

9. PROCEDURE comes instead of PROCESS.

10. INIT(PDPOINTER); First statement in a process.

11. WHILE TRUE DO BEGIN replaces CYCLE.

12. WITH @ DO BEGIN SLOT:=1; CTR:=@; GET:=NIL END;
    Initialisation of semaphores and critical regions. First '@' is replaced by SEMAF or REGION identifier, second '@' becomes 1 for REGIONs and 0 for SEMAFs.

13. WITH @ DO BEGIN PUTPT:=0; GETPT:=0; MUTEX.SLOT:=1; MUTEX.CTR:=1; MUTEX.GET:=NIL; EMPTY.SLOT:=1; EMPTY.CTR:=@; EMPTY.GET:=NIL; FULL.SLOT:=1; FULL.CTR:=0; FULL.GET:=NIL; LENGTH:=@; NBYT:=@ END;
    Initialisation of message buffers. On the first '@' the buffer name is inserted. The second an third are replaced by the maximum number of messages in the buffer and the fourth becomes the number of bytes in a message.

14. @[@] inserted for first '@' in initialisation of VECTOR of SEMAF, MSGBUF or MESSAGE. First '@' is replaced by VECTOR identifier, second by index value.

15. BEGIN WAIT(PDPOINTER,@); replaces REGENTER@.

16. ; SIGNAL(@) END replaces REGEXIT@.

17. PDPOINTER Inserted as first parameter in a SYSTEM subprogram call.

18. (PDPOINTER) Used if SYSTEM subprogram call has no parameters.

19. BEGIN WAIT(PDPOINTER,NEWREG); <user call of NEW>
    ;SIGNAL(NEWREG);NEWCHK(TOPOFSTACK) END
    Critical region created around call of NEW.

20. ;BYE(PDPOINTER) Last statement in a process having no CYCLE ... END part and in main program, as the main program is treated like a normal process by the run-time system.

21. INITIALISE;
    TOPOFSTACK:=INITTOS-40; (* 20 words reserved for NEWSTK *)
    NEWSTK(READYQ,10000,30,300);PDPOINTER:=TEMPQPNTR;
    PHOENIX(PROCAQ,PROCBQ,READYQ);
    First statements of the main program.

22. BEGIN WAIT(PDPOINTER,STARTREG) and
    ;SIGNAL(STARTREG) END
    form a critical region around invocation of START procedure.

23. ;NEWSTK(Q,PRI,HL,SL); PROCS(...)
    replaces START(Q,PRI,HL,SL,PROCS,...)

APPENDIX B

The text generation software.

The file from which the texts to be inserted are taken is organised as a direct access file of 8-byte chunks, in which texts are stored in consecutive chunks, with each new text starting in a new chunk. So one text may occupy more than one chunk, but one chunk only contains (part of) one text. As described in II.1, the texts may be retrieved by number via a table that contains a reference to the first chunk of every text. The number of chunks for one text is calculated by subtracting the table entry for that text from the succeeding entry. The two files, one containing the text chunks and one containing the index table, are prepared by program FILTXT. This program uses as input a text file in which the texts are stored consecutively, with each text terminated by a '$' character. Anything after that character on the same line up to and including the carriage return and line feed characters is considered as a comment. This is usefull for adding text numbers and other information to the different texts. The file is terminated by a double '$'.

Replacement and insertion of texts is performed by:

```
PROCEDURE REPLACE(NTXT:INTEGER; ILST:INLIST); and
PROCEDURE OUTXT(IFIL:DESTIN; NTXT:INTEGER; ILST:INLIST);
```

REPLACE discards the current contents of the output buffer and invokes OUTXT. OUTXT transfers text NTXT to the file indicated by IFIL.
As the number of '@'-characters in a text varies from one text to another it is not possible to indicate the strings that have to be inserted at those positions by parameters of the OUTXT procedure. Moreover, the string to be inserted may itself originate in the text file and have its own '@'s. Therefore a list construction, consisting of records linked by pointers is used:

```
INLIST=^TXTLIST;
MKIND=(TXT,INT,NME,PSY,MTY);
TXTLIST=RECORD NEXT:INLIST;
                CASE KIND:MKIND OF
                TXT: (TXTNR:INTEGER; SLIST:INLIST);
                INT: (I:INTEGER);
                NME: (NM:IDNAME)
        END;
```

Here TXT indicates a text from the text file, INT means an integer number, NME an identifier (max. 10 char.), PSY is the next symbol from the input file and MTY indicates an empty string - nothing is filled in at the '@'-position. The records are created by:

```
FUNCTION INSTX(NTXT:INTEGER; ILIST:INLIST): INLIST; This function
produces a record of kind TXT. ILIST is the list of inserts for text
NTXT.
FUNCTION INSI(I:INTEGER): INLIST; generates an INT record.
FUNCTION INSNM(NM:IDNAME): INLIST; generates a NME record.
FUNCTION INSSY: INLIST; for PSY records.
FUNCTION INSO: INLIST; for MTY records.
```

The result of these functions is a pointer to the record created. So the function call may be used as OUTXT parameter for lists of only one element. The record generation functions use:

```
FUNCTION ILNEW: INLIST;
```

to obtain a new record from the pool of free records or to have it created if none is available there. Lists of more elements are built by:

```
FUNCTION CONC(LST1,LST2:INLIST): INLIST;
```

This function joins two lists LST1 and LST2, head to tail, and delivers a pointer to the first list element (being the first element of LST1 if that is a non-empty list).

The codes TXT, INT, NME, PSY and MTY of each succeeding record are interpreted by OUTXT, one each time a '@'-character is encountered, and the appropriate string is produced. Therefore the following procedures are declared within OUTXT:

```
PROCEDURE PUTO(C:ASCII);
PROCEDURE SPLICE(NTXT:INTEGER; ILST:INLIST);
```

PUTO is used to transfer texts to the output file, thereby discarding any superfluous TABs or spaces. SPLICE takes care of the text generation and is called recursively every time a text from the file of chunks is to be inserted. After use the records are disengaged from the chain and disposed, that is, added to the pool of free records to be used again for a new text to be generated.

Now we are able to generate output texts by using statements like:

```
LP:=INSTX(43,CONC(INSNM(REPR),INSI(I)));
OUTXT(OUTF,11,CONC(LP,CONC(INSI(VSUB^.MULT),INSI(NBYT))));
```

These yield a text string consisting of text 11 with at its first '@' text 43 inserted, in which the first '@' is replaced by identifier REPR and the second by number I. The second and third '@'s of text 11 are filled in with

numbers VSUB.MULT and NBYT respectively.

APPENDIX C

Other preprocessor subprograms and data structures.

C.1 Data types and variables used by preprocessor:

Selection of type declarations:

```
SYMBOL=(ASCIISYM,BEGINSYM,BYTESYM,CASESYM,CHARSYM,CONSTSYM,
CYCLESYM,ENDSYM,EXTSYM,FWDSYM,FUNCTSYM,INTGRSYM,MESSGSYM,
MBUFSYM,OFSYM,PROCDSYM,PROCSSYM,RECDSYM,RENTRSYM,ROUTSYM,REGSYM,
SEMSYM,SYSTSYM,TYPESYM,VARSYM,VECTSYM,IDENTIF,UNSINT,STRNGSYM,
COMTSYM,OPEN,CLOSE,COLON,SEMICOLN,PERIOD);
```

Enumeration of all special PAP symbols and Pascal symbols that have to be processed.

```
IDKIND=(CONSTIDENT,TYPIDENT,VARIDENT,PRCSIDENT);
```

Kind of identifier: Constant, Type, Variable or Process.

```
IDCLASS=(SEMIDENT,MSGIDENT,BUFIDENT,REGIDENT,VECTIDENT,
        PROCFUNCID,RESTIDENT);
```

Identifier class: Semaphore, Message, Message buffer, Critical region, Vector of PAP types, Procedure or Function, Pascal type.

```
IDNAME=ARRAY[0..MAXIDNAME] OF CHAR;
```

Identifier name (maximum of 10 characters).

```
MSGTYP=(BYTE,WORD);
```

Message element type: Byte = CHAR, ASCII or BYTE; Word = INTEGER.

```
SPREF=(FWD,EXT,DCL);
```

FORWARD, EXTERNAL or direct declaration of procedure or function.

```
REGPT=^REGENTRY;
IDPT=^IDSPECS;
REGENTRY=RECORD REG:IDPT;
                NEXT:REGPT
        END;
```

Record of region/subprogram list to be attached to identifier record.

```
        IDSPECS=RECORD
                        REPR:IDNAME;
                        FORMAL:BOOLEAN; REFS:INTEGER;
                        (*THESE 2 FIELDS ONLY SIGNIFICANT FOR VARIDENT*)
                        LEFTPT,RIGHTPT:IDPT;
                        CASE KIND:IDKIND OF
                        CONSTIDENT: (CASE INTC:BOOLEAN OF
                                TRUE: (VALUE:INTEGER));
                        TYPIDENT,VARIDENT: (CASE CLASS:IDCLASS OF
                                REGIDENT: (INUSE:BOOLEAN);
                                MSGIDENT: (ELTYP:MSGTYP; NEL:INTEGER);
                                BUFIDENT: (SUBPT:IDPT; MULT:INTEGER);
                                VECTIDENT: (VSUB:IDPT; LL,UL:INTEGER);
                                PROCFUNCID: (OCCUR:SPREF; REGLIST:REGPT;
                                                SYSP:BOOLEAN))
                END;
```

Identifier record with full description of identifier.
FORMAL is TRUE if identifier is specified in parameter list.
REFS is number of references to this record from other identifier records or region/subprogram list elements.
INTC is TRUE if constant is of INTEGER type.
INUSE is TRUE during evaluation of region. Used for deadlock detection.
SYSP is TRUE for SYSTEM PROCEDURES or SYSTEM FUNCTIONS.

```
        SCOPT=^SCOPE;
        SCOPE=RECORD CONTENTS:IDPT;
                        FATHER:SCOPT
                END;
```

Used to build scope stack.

```
        MODUL=(MAIN,PRCSS,SUBP);
```

Identification of software module: Main program, Process or Subprogram.

Some variables used by the preprocessor:

```
        WDEL:ARRAY[0..MAXWDEL] OF RECORD
                        RWORD:IDNAME;
                        SPSYM:SYMBOL
                END;
```

Table of reserved words with corresponding symbol.

```
        OUTBUF:ARRAY[0..MAXOUTB] OF ASCII; GI,PI:0..MAXOUTB;
```

Output buffer with getpointer and putpointer.

C.2 Overview of subprograms not mentioned in previous appendix.

```
FUNCTION IPNEW:IDPT;
DELIVERS NEW IDENTIFIER RECORD

PROCEDURE INSERTREE(NEWPT:IDPT);
ADDS IDENTIFIER RECORD TO IDENTIFIER TREE

PROCEDURE DUMP;
TRANSFERS CONTENTS OF OUTPUT BUFFER TO OUTPUT FILE

PROCEDURE LISLIN;
 CLOSES LINE OF LISTING. STARTS NEW PAGE IF NECESSARY.

PROCEDURE FAULT(MSGNR:INTEGER);
HANDLES ERROR LOGGING

PROCEDURE COPY(C:ASCII);
ADDS CHARACTER TO OUTPUT BUFFER

PROCEDURE INSCHAR(C:ASCII);
TRANSMITS C TO OUTPUT FILE

PROCEDURE REPLACE(I:INTEGER; ILST:INLIST);
DELETES CONTENTS OF OUTPUT BUFFER AND TRANSMITS TEXT I TO OUTPUT FILE

PROCEDURE RINPUT;
READS INPUT CHARACTER; TAKES CARE OF LISTING

PROCEDURE FETCH;
GETS NEXT CHARACTER FROM INPUT FILE; CONVERTS (STRINGS OF) CONTROL
CHARACTERS PLUS SPACES TO ONE SPACE AND LOWER CASE TO UPPER CASE

PROCEDURE NEXTSYM;
EVALUATES NEXT INPUT SYMBOL
Procedures declared within NEXTSYM:

        PROCEDURE COMMENT;
        PROCESSES COMMENTS: COMMENTS ARE SKIPPED IF THEY DON'T START
        WITH '$'. ONLY OPTION IDENTIFIERS FROM LEGAL SUBSET ARE
        TRANSFERRED TO OUTPUT FILE.

        PROCEDURE NEXTCHR;
        EVALUATES NEXT INPUT CHARACTER

        PROCEDURE WORDSYM;
        EVALUATES TEXT SYMBOLS

        PROCEDURE NUMBER;
        EVALUATES NUMBERS
```

```
        PROCEDURE STRING;
        PROCESSES STRING CONSTANTS

FUNCTION TERMSYM(Y:SYMBOL):BOOLEAN;
CHECKS INPUT SYMBOL

PROCEDURE TESTSYM(Y:SYMBOL);
TEST FOR SYMBOL Y

FUNCTION RPNEW:REGPT;
DELIVERS NEW REGION/SUBPROGRAM LIST ELEMENT

PROCEDURE ENTERSCOPE;
SETS UP NEW SCOPE STRUCTURE: PUSHES NEW SCOPE ELEMENT ON SCOPE STACK
AND INITIALISES IDENTIFIER TREE

PROCEDURE LEAVESCOPE;
LEAVE CURRENT SCOPE, RELEASE IDENTIFIER TREE ELEMENTS

        PROCEDURE IDREL(IDTREE:IDPT);
        TRANSFERS IDENTIFIER RECORDS TO FREE POOL IF ALLOWED

FUNCTION CHECKID(VAR IP:IDPT; IK:IDKIND; IC:IDCLASS): BOOLEAN;
LOOKUP IDENTIFIER, TEST ATTRIBUTES

        FUNCTION FOUND(VAR IDTREE:IDPT);
        SEARCHES IDENTIFIER DATA STRUCTURE FOR RECORD OF IDENTIFIER WITH
        GIVEN NAME.

FUNCTION INTCONST(VAR I:INTEGER):BOOLEAN;
EVALUATES INTEGER CONSTANT

PROCEDURE TYPESPEC(DECLPTR:IDPT);
EVALUATES TYPE OF IDENTIFIER DECLPTR AND UPDATES IDENTIFIER TREE

        PROCEDURE DECLARE(DECLPTR:IDPT; DECLASS:IDCLASS);
        COMPLETES RECORDS OF IDENTIFIER LIST AND ADDS THEM TO IDENTIFIER
        TREE

        FUNCTION DCLSUBTYP(DECLPTR:IDPT): IDPT;
        SUBTYPE DECLARATION

        PROCEDURE LIST(CLSYM:SYMBOL);
        DIGESTS INPUT TEXT UP TO CLSYM

PROCEDURE PRTREE(P:IDPT);
INSERTS INITIALISATION STATEMENTS OF IDENTIFIERS IN OUTPUT FILE
AND LISTS ATTRIBUTES
```

```
PROCEDURE STATEMT;
PROCESSES BLOCK OF STATEMENTS AND UPDATES REGION/SUBPROGRAM LIST

        PROCEDURE REGTREE(VAR RLPT:REGPT);
        INSPECTS REGION/SUBPROGRAM LIST FOR OCCUPIED REGIONS

        PROCEDURE SCLOSE;
        DIGESTS PARAMETER LIST OF SEND OR RECEIVE AFTER ERROR

        PROCEDURE SKIPDL;
        SKIPS DELIMITERS

PROCEDURE VARDECL(FML:BOOLEAN);
HANDLES VARIABLE DECLARATIONS

PROCEDURE PARLIST;
HANDLES PARAMETER LIST IN PROCEDURE, FUNCTION OR PROCESS DECLARATIONS.

PROCEDURE DECLPART(MKIND:MODUL);
EVALUATES CONSTANT, TYPE, VAR AND PROCEDURE/FUNCTION DECLARATIONS
AND CREATES IDENTIFIER DATA STRUCTURE FOR TYPE CHECKING

PROCEDURE PROCDEF;
HANDLES PROCESS DEFINITIONS

PROCEDURE INIPREP;
TAKES CARE OF STANDARD DECLARATIONS

PROCEDURE BODY;
HANDLES PROGRAM BODY

FUNCTION STARTPREP:BOOLEAN;
HANDLES FILE SPECIFICATIONS AND INITIALISES VARIABLES

        PROCEDURE GCC;
         C:=NEXT CHARACTER FROM COMMAND LINE. C:='\' IF END OF LINE.

        PROCEDURE NAMIN(EXTNAME:FLN);
        EVALUATES FILE SPECIFICATIONS

        PROCEDURE DPLFIL;
        DISPLAYS FILE SPECIFICATIONS
```

C.3 Overview of global identifiers from the preprocessor:

Name	Kind	If kind = constant, is it an integer, and if so, what is its value? If kind = subprogram (procedure or function), what other subprograms are called from this one?				
BLKSTSYM	VARIABLE					
BODY	SUBPROGR	FETCH	INSCHAR	TESTSYM	STATEMT	LISLIN
		PRTREE	OUTXT	DUMP	FAULT	PROCDEF
		DECLPART	ENTERSCOPE	INIPREP	NEW	NEXTSYM
BUFLENGTH	CONSTANT	INTEGER, 132				
C	VARIABLE					
CH	VARIABLE					
CHECKID	SUBPROGR	FOUND				
CHUNK	TYPE					
CONC	SUBPROGR					
COPY	SUBPROGR	FAULT				
CSCOPE	VARIABLE					
DAT	VARIABLE					
DCLENDSYM	VARIABLE					
DECLPART	SUBPROGR	LEAVESCOPE	STATEMT	PARLIST	ENTERSCOPE	FAULT
		CHECKID	VARDECL	INSTX	TYPESPEC	OUTXT
		TESTSYM	INSERTREE	INTCONST	IPNEW	TERMSYM
		NEXTSYM				
DESTIN	TYPE					
DIGITS	VARIABLE					
DUMP	SUBPROGR					
ENDF	VARIABLE					
ENTERSCOPE	SUBPROGR	NEW				
EOM	CONSTANT					
ERROR	VARIABLE					
FAULT	SUBPROGR	OUTXT	LISLIN			
FETCH	SUBPROGR	RINPUT				
GI	VARIABLE					
IDCLASS	TYPE					
IDKIND	TYPE					
IDNAME	TYPE					
IDPT	TYPE					
IDSPECS	TYPE					
ILFREE	VARIABLE					
ILNEW	SUBPROGR	NEW				
IMTYP	VARIABLE					
IMUL2	VARIABLE					
IMULT	VARIABLE					
INIPREP	SUBPROGR					
INLIST	TYPE					
INSO	SUBPROGR	ILNEW				
INSCHAR	SUBPROGR	OUTRY				
INSERTREE	SUBPROGR					

Name	Kind	Integer constant or subprograms called.				
INSI	SUBPROGR	ILNEW				
INSNM	SUBPROGR	ILNEW				
INSPC	VARIABLE					
INSSY	SUBPROGR	ILNEW				
INSTX	SUBPROGR	ILNEW				
INTCONST	SUBPROGR	FAULT	CHECKID			
INTVAL	VARIABLE					
INTXT	VARIABLE					
IOSELECT	VARIABLE					
IOSPEC	TYPE					
IPFREE	VARIABLE					
IPNEW	SUBPROGR	NEW				
IRP	VARIABLE					
ISUBPT	VARIABLE					
LC	VARIABLE					
LEAVESCOPE	SUBPROGR	IDREL				
LEGALSW	VARIABLE					
LETTERS	VARIABLE					
LFIL	VARIABLE					
LI	VARIABLE					
LISI	VARIABLE					
LISLIN	SUBPROGR					
LISNAM	VARIABLE					
LNR	VARIABLE					
LSIGNS	VARIABLE					
MAXCHN	CONSTANT	INTEGER,	7			
MAXIDNAME	CONSTANT	INTEGER,	9			
MAXOUTB	CONSTANT	INTEGER,	131			
MAXSRC	CONSTANT	INTEGER,	132			
MAXWDEL	CONSTANT	INTEGER,	25			
MKIND	TYPE					
MODUL	TYPE					
MSGTYP	TYPE					
MXINTD10	CONSTANT	INTEGER,	3276			
MXINTLD	CONSTANT	INTEGER,	7			
NAME	VARIABLE					
NERR	VARIABLE					
NEWL	VARIABLE					
NEXTSYM	SUBPROGR	FAULT	FETCH	COMMENT	STRING	COPY
		NUMBER	WORDSYM	DUMP	NEXTCHR	
NOMAIN	VARIABLE					
OUTBUF	VARIABLE					
OUTRY	SUBPROGR					
OUTXT	SUBPROGR	SPLICE				
PAGELT	CONSTANT	INTEGER,	57			
PARLIST	SUBPROGR	TESTSYM	VARDECL	TERMSYM	NEXTSYM	
PI	VARIABLE					
PROCDEF	SUBPROGR	LEAVESCOPE	STATEMT	LISLIN	PRTREE	OUTXT
		DUMP	DECLPART	TERMSYM	TESTSYM	PARLIST
		ENTERSCOPE	INSERTREE	IPNEW	FAULT	NEXTSYM
		REPLACE				

Name	Kind	Integer constant or subprograms called.				
PRTREE	SUBPROGR	INSTX	INSI	INSNM	CONC	OUTXT
		LISLIN				
PTABL	VARIABLE					
QM	CONSTANT					
REGENTRY	TYPE					
REGPT	TYPE					
REPLACE	SUBPROGR	OUTXT				
RINPUT	SUBPROGR	LISLIN				
RPFREE	VARIABLE					
RPNEW	SUBPROGR	NEW				
SCFREE	VARIABLE					
SCOPE	TYPE					
SCOPT	TYPE					
SP	CONSTANT					
SPREF	TYPE					
SRCI	VARIABLE					
STARTPREP	SUBPROGR	GCC	DPLFIL	NAMIN	GCML	
STATEMT	SUBPROGR	REGTREE	SKIPDL	SCLOSE	INSTX	INSCHAR
		OUTXT	TERMSYM	DUMP	FAULT	RPNEW
		CHECKID	INSSY	REPLACE	TESTSYM	NEXTSYM
SYM	VARIABLE					
SYMBOL	TYPE					
TABLE	TYPE					
TERMSYM	SUBPROGR	NEXTSYM				
TESTSYM	SUBPROGR	INSNM	INSTX	OUTXT	LISLIN	NEXTSYM
		TERMSYM				
TIM	VARIABLE					
TXTLIST	TYPE					
TYPESPEC	SUBPROGR	LIST	OUTXT	DCLSUBTYP	CHECKID	DECLARE
		TERMSYM	TESTSYM	FAULT	INSI	INTCONST
		NEXTSYM	REPLACE			
TYPSTSYM	VARIABLE					
VARDECL	SUBPROGR	TYPESPEC	NEXTSYM	TESTSYM	IPNEW	
WDEL	VARIABLE					

EINDHOVEN UNIVERSITY OF TECHNOLOGY
THE NETHERLANDS
DEPARTMENT OF ELECTRICAL ENGINEERING

Reports:

EUT Reports are a continuation of TH-Reports.

116) Versnel, W.
THE CIRCULAR HALL PLATE: Approximation of the geometrical correction factor for small contacts.
TH-Report 81-E-116. 1981. ISBN 90-6144-116-1

117) Fabian, K.
DESIGN AND IMPLEMENTATION OF A CENTRAL INSTRUCTION PROCESSOR WITH A MULTIMASTER BUS INTERFACE.
TH-Report 81-E-117. 1981. ISBN 90-6144-117-X

118) Wang Yen Ping
ENCODING MOVING PICTURE BY USING ADAPTIVE STRAIGHT LINE APPROXIMATION.
EUT Report 81-E-118. 1981. ISBN 90-6144-118-8

119) Heijnen, C.J.H., H.A. Jansen, J.F.G.J. Olijslagers and W. Versnel
FABRICATION OF PLANAR SEMICONDUCTOR DIODES, AN EDUCATIONAL LABORATORY EXPERIMENT.
EUT Report 81-E-119. 1981. ISBN 90-6144-119-6.

120) Piecha, J.
DESCRIPTION AND IMPLEMENTATION OF A SINGLE BOARD COMPUTER FOR INDUSTRIAL CONTROL.
EUT Report 81-E-120. 1981. ISBN 90-6144-120-X

121) Plasman, J.L.C. and C.M.M. Timmers
DIRECT MEASUREMENT OF BLOOD PRESSURE BY LIQUID-FILLED CATHETER MANOMETER SYSTEMS.
EUT Report 81-E-121. 1981. ISBN 90-6144-121-8

122) Ponomarenko, M.F.
INFORMATION THEORY AND IDENTIFICATION.
EUT Report 81-E-122. 1981. ISBN 90-6144-122-6

123) Ponomarenko, M.F.
INFORMATION MEASURES AND THEIR APPLICATIONS TO IDENTIFICATION (a bibliography).
EUT Report 81-E-123. 1981. ISBN 90-6144-123-4

124) Borghi, C.A., A. Veefkind and J.M. Wetzer
EFFECT OF RADIATION AND NON-MAXWELLIAN ELECTRON DISTRIBUTION ON RELAXATION PROCESSES IN AN ATHMOSPHERIC CESIUM SEEDED ARGON PLASMA.
EUT Report 82-E-124. 1982. ISBN 90-6144-124-2

125) Saranummi, N.
DETECTION OF TRENDS IN LONG TERM RECORDINGS OF CARDIOVASCULAR SIGNALS.
EUT Report 82-E-125. 1982. ISBN 90-6144-125-0

126) Królikowski, A.
MODEL STRUCTURE SELECTION IN LINEAR SYSTEM IDENTIFICATION: Survey of methods with emphasis on the information theory approach.
EUT Report 82-E-126. 1982. ISBN 90-6144-126-9

EINDHOVEN UNIVERSITY OF TECHNOLOGY
THE NETHERLANDS
DEPARTMENT OF ELECTRICAL ENGINEERING

Eindhoven University of Technology Research Reports (ISSN 0167-9708)

(127) Damen, A.A.H., P.M.J. Van den Hof and A.K. Hajdasiński
THE PAGE MATRIX: An excellent tool for noise filtering of Markov parameters, order testing and realization.
EUT Report 82-E-127. 1982. ISBN 90-6144-127-7

(128) Nicola, V.F.
MARKOVIAN MODELS OF A TRANSACTIONAL SYSTEM SUPPORTED BY CHECKPOINTING AND RECOVERY STRATEGIES. Part 1: A model with state-dependent parameters.
EUT Report 82-E-128. 1982. ISBN 90-6144-128-5

(129) Nicola, V.F.
MARKOVIAN MODELS OF A TRANSACTIONAL SYSTEM SUPPORTED BY CHECKPOINTING AND RECOVERY STRATEGIES. Part 2: A model with a specified number of completed transactions between checkpoints.
EUT Report 82-E-129. 1982. ISBN 90-6144-129-3

(130) Lemmens, W.J.M.
THE PAP PREPROCESSOR: A precompiler for a language for concurrent processing on a multiprocessor system.
EUT Report 82-E-130. 1982. ISBN 90-6144-130-7

(131) Eijnden, P.M.C.M. van den, H.M.J.M. Dortmans, J.P. Kemper and M.P.J. Stevens
JOBHANDLING IN A NETWORK OF DISTRIBUTED PROCESSORS.
EUT Report 82-E-131. 1982. ISBN 90-6144-131-5