Web: www.CAM-postwriter.com
E-mail: info@CAM-postwriter.com
Fax: +31 (0)84 756 0953

# CAM-postwriter

# User Manual

**version 3.1**
**date: 17 april 2005**

Document: http://CAM-postwriter.com/manuals/User_Manual_vs3-1.pdf

# Contents

# 1. Document history

## 1.1. Document location

This document is only valid on the day it is printed. The original version of this document is stored on the location mentioned on the first page.

## 1.2. Version history

| Version date | Version number | Change notes | Changes marked |
|---|---|---|---|
| 3-06-96 | 1.0 | First issue. | |
| 9-10-03 | 2.0 | Conversion of picture source and format.<br>Review of document. | |
| 21-11-04 | 3.0 | English translation. | |
| 17-04-2005 | 3.1 | Chapter "Error messages" added. | |

## 1.3. Distribution

This document is distributed to:

| Name | Title | Date of issue | Version |
|---|---|---|---|
| | | | |
| | | | |
| | | | |

# 2. What is CAM-postwriter?

CAM-postwriter is a special tool for development of postprocessors for CNC-machines.
The postprocessor build by CAM-postwriter gets its input from a CAM system, CAD/CAM system or NC programming system. The only restriction for the CAM-postwriter system is that the CAM system has to output a, standardized, Cutter Location file. (Other names are CL-file or CLDATA.)

The core of CAM-postwriter is a specification language, which allows you to describe the functionality of a postprocessor on a high abstraction level. In this specification language specific elements are provided to describe the input from a CLDATA file and the output of NCDATA. Besides a number of checks build in, for example syntax checks of postprocessor statements, you are able to build your one checks and logic into the postprocessor. The specification language is very powerful and yet simple to understand. In the past fifteen years the concept of the specification language has proved to be able to specify postprocessors for very complex machines.

Postprocessors build by CAM-postwriter can meet a very high degree of automation and production process checking. With a high quality postprocessor the part programmer doesn't have to worry about the specific details of the machine; the postprocessor will solve them. Yet CAM-postwriter enables you to build a very simple postprocessor that simply translates CLDATA to NC machine codes. The complexity of the postprocessor depends on the logic you put into it and CAM-postwriter enables you to put in as many as you want. You don't need high level programming expertise to accomplish this. The level of automation and checking depends on the level you want to reach and satisfies you.

## 2.1. Post-processing in general.

Post-processors can do many other things besides translating CLDATA to NC machine codes (NCDATA). For example a post-processor may summarize maximum axes travel, feed and speed limits, job runtime and tool usage information, which enables better selection and scheduling of resources.

More sophisticated postprocessors may validate the program before it is run by the machine tool. This means that the program can be corrected at the CAM-system instead of correction at the machine tool in the workshop where it implies loss of production time. There are many simple rules that can be checked, with warning or error messages displayed when these rules are violated. Some examples are:

- Warning if a tool is not selected at the start of the program.
- Warning if a table is not indexed at the start of the program.
- Warning if no workpiece offset is activated at the start of the program or after indexing of the table.
- Warning if axes travel beyond their limits.
- And so on…

With CAM-postwriter you can define your own rules and put them in the logic of the postprocessor.

Postprocessors give a higher degree of automation in the process of part programming. Take for example the tool change sequence. The possible steps, which may be automated in the logic of the postprocessor, are:

- Stop coolant.
- Retract to tool change position. (Z-retract and XY-retract or XYZ-retract.)
- Tool change.
- Select range and start spindle.
- Activate diameter and length compensation in next motion(s)
- Pre-select next tool.

This tool change sequence may be activated by one, simple, LOADTL-statement.
With CAM-postwriter you can define your own level of automation and specify this in the logic of the postprocessor.

Post-processors can also work around limitations and bugs in the CAM system or in the machine tool. It is generally far easier to change the post-processor than it is to get a new revision of the CAM system or a new revision for the NC controller.
An example of extend possibilities is a machine tool equipped with an adjustable square head. On this machine tool the coordinate system couldn't be tilted by the machine control while the tool could. In this case the postprocessor took care of XYZ axis transformation.

The important point to be made here is that the NC programmer should not be concerned about specific details and peculiar characteristics of the machine tool that do not directly affect the production of a job. A good postprocessor should hide these details, as much as possible, within. Enabling the NC programmer to focus entirely on the job.

Standard CAM systems, standard NC machines, standard CLDATA and standard postprocessor vocabulary can not all be simple mixed together to instantly produce a working system. There are too many variables in the real world to achieve integration with off-the-shelf components.
Postprocessors put it all together, and good postprocessors can do this with a minimum of effort of the part programmer. Post-processing works best when it is "transparent".

With CAM-postwriter you have the perfect tool to put this all together.

# 3. How do you build a postprocessor?

Before you start building a postprocessor you have to consider several aspects. First of all you build a postprocessor for a specific machine. You have to consider the capabilities of this machine and answer the question if and in which way you want to control these capabilities by means of a postprocessor.

Next there is the machine control. All the available features are documented in the "Programming Manual" of the machine control. Some of these features may not apply to your machine. E.g. your machine may not be able to perform a helical interpolation while the subject is documented in the "Programming Manual". Perhaps you are lucky and you have a "Programming Manual" written by the manufacturer of the machine. Such a "Programming Manual" is written with the possibilities of the machine and control combination in mind.

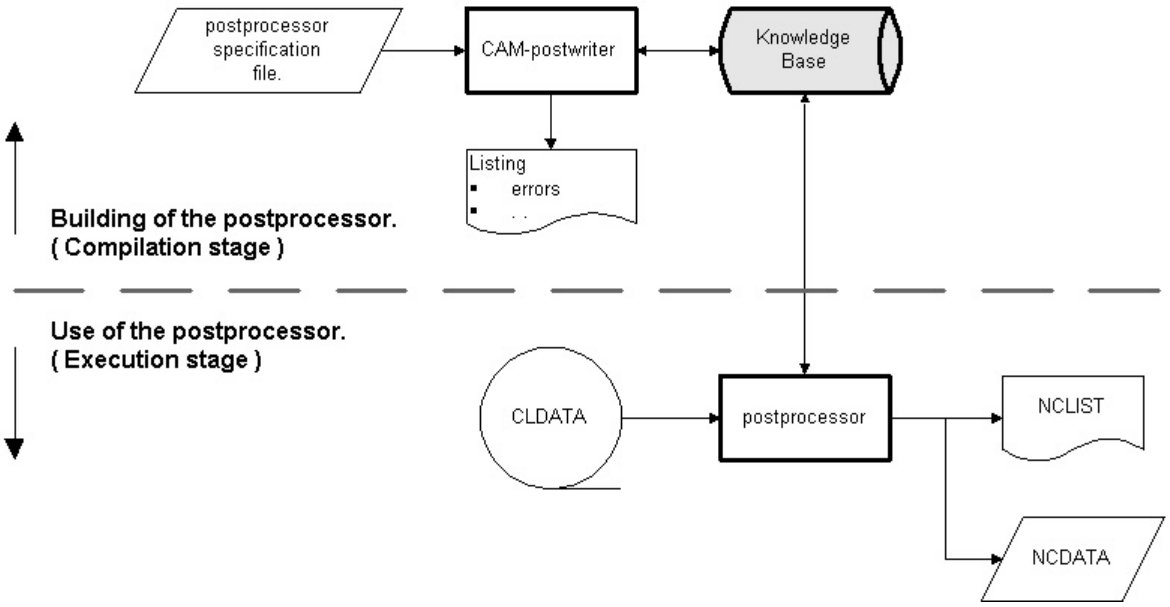
Another aspect is the NC programming system, CAM system or CAD/CAM system. The output of the system has to be a Cutter Location file (CL-file). All these systems will, more or less automatically, generate a "tool path". Important matter is in which way "machine commands" can be inserted in this "tool path". These "machine commands" are defined in the postprocessor as "postprocessor statements". Examples are setting a feed rate for a motion, the loading of a tool, starting the machine spindle, activating a cutter compensation, indexing a rotary table and selecting a new workpiece offset before machining a new "tool path".

The type of parts you are machining is another aspect. One example is where you manufacture a "shape" by just one tool and a very long "tool path". The part in this example is called "geometry driven". The opposite is for example a part where you use many tools and relatively less geometry in the tool paths. The part in this example is called "technology driven".
The contribution of a postprocessor in the part programming of the last example is in general much more than in the first example. However, parts are in general not completely "geometry driven" or " technology driven". There may be a mix of these two types in the parts you have to machine.
Other aspect may be whether you machine unique parts (one of) or when the parts are machined regularly, in small or larger batches, in time. The accuracy of the parts and the procedures and protocols to follow may be another aspect.
You have to consider how much effort (money) you invest in a postprocessor, the higher the quality the more expensive the postprocessor. How does this effort contribute to your production process?

All these aspects, and maybe more, have to be taken in consideration when you build your postprocessor.

# 4. How does CAM-postwriter work?

The next, simplified, schema illustrates the main components of CAM-postwriter. In the building stage there is a postprocessor specification file. (This is an ASCII text file.) The postprocessor specification file is processed by CAM-postwriter and information is stored in the knowledge base.




The knowledge base contains information about how the postprocessor has to operate. It tells the postprocessor **how** information in the CL-file, tool path information and machine information, has to be translated into NCDATA, which checks has to be made and in case of errors which messages has to be generated.

The NCDATA can be sent directly to the machine control to produce the workpiece. The file NCLIST contains the same NCDATA, presented in a more human readable form and possible warnings or error messages for the part programmer.

# 5. Starting the postprocessor.

In most cases the postprocessor will be triggered to execute when the CAM / NC programming system has finished the calculation of all tool paths. The last statement of a part program is the FINI statement. This statement stops the generation of the Cutter Location file (CL-file) and is also the last statement in the CL-file.

Next step is to start the postprocessor. The postprocessor will open the CL-file and read the CL-records one by one until the MACHIN-statement is met. The machine statement contains the name of the specific machine. Once this name is known, the specific postprocessor knowledge files will be opened. The CL-file is rewind, the records will be read again but now they will be processed as specified in the postprocessor specification. This process will end when the FINI statement is read from the CL-file.

The postprocessor can also be started separately. The only condition is there has to be a CL-file.

It is also possible to start the postprocessor before the CL-file is finished. This means that every record generated by the CAM / NC programming system in the CL-file will be processed immediately by the postprocessor. This means that, besides some other restrictions, the machine statement has to be one of the first statements.

The command file "campost" starts the postprocessor. So if you want to start the postprocessor automatically from your NC-programming system, this file has to be activated. See also the **Installation Manual**.

# 6. CAM-postwriter specification language.

## 6.1. General.

The specification file contains the specification of the postprocessor. The extension of this file has to be “spc”. So, the input file is <filename>.spc. This file is only used as input and will not be altered by CAM-postwriter. When the specification file is processed a new file is created with the extension “lst”. The output file is <filename>.lst. This second file is a copy of the specification file except that page headers and line numbers have been added. In case of errors, error messages are listed in this file describing the type of error in detail. At the end of this file a summary will be given of the total number of errors, if any.

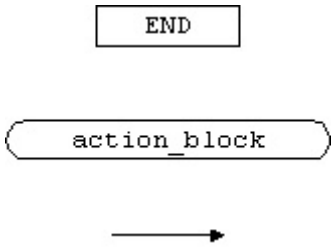
## 6.2. Syntax and Syntax Diagrams.

In this document only parts of the syntax will be given for explanation of specific details. The complete syntax of the CAM-postwriter language is described in the document “CAM-postwriter Syntax Diagrams”. For a full, detailed description of the syntax see this document.

In this document the specification language of CAM-postwriter is described by means of syntax diagrams. A syntax diagram consists of three types of elements:

- a rectangle,
- a bar with round ends and,
- an arrow.

See the example below.




The rectangle represents a terminal. This is an element in the language that cannot be split into smaller parts. You will recognize these elements one on one in the specification language. In the above example, the rectangle with the word "END" means you will find the keyword "END" in the specification language.

A bar with round ends represents a non-terminal. Non-terminals are elements that can be split into smaller parts. In other words, a non-terminal consists of elements on a lower level. When you meet a non-terminal in a syntax diagram this means that there has to be a syntax diagram that describes the non-terminal. This syntax diagram, however, may contain other non-terminals. On the lowest level only terminals exist.

The arrows give the paths and direction in the syntax diagram. The elements one encounters along a path are a syntactically correct construction in the specification language.

## 6.3. Comments.

Comments in the specification file are placed between "{" and "}". The comment starts with "{" and ends with "}". A comment can cover several lines. A new line doesn't mean the end of a comment. Comments can be nested. This means that after, for example, two comment opening characters there have to be two comment closing characters to end the comment.

All characters in a comment are ignored by CAM-postwriter.

## 6.4. Statements.

A new line does **NOT** terminate a statement. This means that a statement can cover multiple lines or that there can be more than one statement on one line.

# 7. Error messages.

Error messages generated by CAM-postwriter can be divided in compile errors and runtime errors. The first three error types, syntax errors, semantic errors and system errors, are compiling errors and are detected during the compilation of the specification file.

Runtime errors are detected during runtime of the postprocessor.

For a complete list of error messages see Appendix 3.

## 7.1. Compile errors.

Compile errors are signaled during the process of scanning the specification file and transforming the specification into the knowledge base.

### 7.1.1. Syntax errors.

A syntax error in the specification file is reported to the postwriter. The message is listed on the screen and in the output file <filename>.lst. The line number and the incorrect item will be listed in the syntax error message. An attempt will be made to restart the scanning of the specification file for possible more errors. Syntax errors (and other errors) between the point of the signaled syntax error and the point where the process comes in track again are not signaled. They will however be found after correction of the first syntax error. An example of a syntax error report is given below.

```
31 *
32 *            VAR  record_number          : INTEGER;
33 *                 stat_nr                : INTEGER;
34 *                 card_id1 ,card_id2     : CHAR*4;
38 *                 $_illegal_identifier   : INTEGER;

*** SYNTAX ERROR ***
Unexpected item in specification file.
On line :        38
Item    : $_illegal_identifier
```

On line 38 is attempted to declare a variable with the name "$_illegal_identifier". However, an identifier has to start with a letter, so the item is not recognized as an identifier.

### 7.1.2. Semantic errors.

A semantic error refers to the meaning of the found item. In the example below, the declaration of the two variables, too_short_string and too_long_string, is syntactically correct. However in CAM-postwriter the minimum length of a string is 1 and the maximum length of a string is 128.

The description of the error, the line number and the item will be listed in the semantic error message.

```
89 *      VAR    too_short_string : CHAR*0;

*** SEMANTIC ERROR ***
Length specification out of rang.
On line:        89
Item   : 0

90 *             too_long_string  : CHAR*129;

*** SEMANTIC ERROR ***
Length specification out of rang.
On line:        90
Item   : 129
```

### 7.1.3. System errors.

System errors are errors due to the restrictions of the system. For example there may not be enough memory in the system to hold all the variables specified by the user. In case of a system error contact CAM-postwriter to adjust the system.

Below an example of a system error. There is no memory available to hold the variable "string2".

```
10989 *      VAR string1 : CHAR*128;
10990 *          string2 : CHAR*128;

*** SYSTEM ERROR ***
Full operand_file.
On line:     10990
Item   : string2
```

## 7.2. Runtime errors.

Runtime errors are detected during runtime of the postprocessor. This means that runtime errors are detected during processing the CLDATA and performing the actions specified in the specification file of the postprocessor.
Actions that result in a runtime error are **NOT** executed. CAM-postwriter will continue processing subsequent actions, which however may result in other runtime errors.

**Important**
When runtime errors occur this means that the output of the postprocessor is not what you expected it to be. You have to analyze and correct the problem!

Runtime errors are listed in the output of the postprocessor, NCLIST, and on the screen of your workstation. At the end of NCLIST you will find a summary of the number of errors, if any.

### 7.2.1. Input errors.

The input of the postprocessor is read from the CLDATA, so input errors refer to the information in the CLDATA. In case of an input error a reference is made to the line number of the part program. This is the statement in your part program that caused the error.

```
*** INPUT ERROR ***
Overflow of operand in postprocessor statement.
On line :        13  of part program.
Item    : MCODE


*** INPUT ERROR ***
Unknown postprocessor statement format.
On line :        15  of part program.
```

### 7.2.2. Runtime errors.

Runtime errors are errors that occur while performing an action, or sub action, in the postprocessor specification. An example is the assignment of the value of an expression to a variable. The expression may result in a value that lies outside the range of the variable.

Content of the specification:

```
989 *   VAR icount : 1..5;
990 *       string : CHAR*128;
991 *   ACTION
992 *     icount := 6;
993 *
```

Example of the resulting error message:

```
*** RUNTIME ERROR ***
Operand overflow.
On line :       992
Item    : ICOUNT
```

Notice that in this case the line number of action in the specification file is printed in which the error occurred.

### 7.2.3. Fatal runtime errors.

Fatal runtime errors are much like normal runtime errors except that they lead to impossible situations. Examples are an expression, which result in a value greater than max_integer when it is evaluated or the reference of a not existing array element.

```
*** FATAL RUNTIME ERROR ***
Array out of bounds.
On line :       483
Item    : I_ARRAY
```

# 8. The Cutter Location File (CL-file).

## 8.1. General.

A Cutter Location file, CL-file or CLDATA, is output from a CAM-system. The CL-file consists of a sequence of records, CL-records. A CL-file contains hundreds, sometimes thousands of CL-records. Most of these records give the position or **location** of the cutting tool. (This explains the name: Cutter Location File. Another name is CLDATA.)

The CL-file is a binary file, it is not man readable. (Some systems generate a man readable CL-file. In fact this is an APT file with only simple APT statements, like GOTO/ <x, y, z>. This type of CL-file can be transformed into a binary CL-file that in its turn can be read by a postprocessor.)

The content of the CL-file is described in standard ISO 3592 "Numerical control of machines – NC processor output – Logical structure (and major words)".
Implementation of this standard may differ between manufacturers of CAD/CAM or NC programming systems. However, the differences are most of the time minor. They can be solved in the interface. This interface reads the contents of the CL-file and inputs information into the postprocessor. Once this interface is build for a specific system it is available for everyone. So if you want to use CAM-postwriter for your CAM-system contact us.

## 8.2. CL-records.

A Cutter Location Record is a sequence of (binary) values. Each value representing a value for a specific parameter. A value can be one of the following types: **integer**, **real** or **character**. The CL-records, with their specific parameters are described in the standard ISO 3592.

An example of a CL-record is given below:

| number of bytes | type | value | meaning | |
|---|---|---|---|---|
| 4 | integer | - | record sequence number | |
| 4 | integer | 5000 | type 5000 record (linear motion) | |
| 4 | integer | 5/6 | 5-first motion record, 6-continuation record. | |
| 4 | char | - | name of geometry in programming system. | |
| 4 | integer | - | index of geometry in programming system. | |
| 4 | real | - | X-coordinat endpoint. | First endpoint. |
| 4 | real | - | Y-coordinat endpoint. | |
| 4 | real | - | Z-coordinat endpoint. | |
| 4 | real | - | X-coordinat endpoint. | Second endpoint. |
| 4 | real | - | Y-coordinat endpoint. | |
| 4 | real | - | Z-coordinat endpoint. | |
| | | | | |
| | | | | |
| 4 | real | - | X-coordinat endpoint. | Endpoint in case of multi axis machining. |
| 4 | real | - | Y-coordinat endpoint. | |
| 4 | real | - | Z-coordinat endpoint. | |
| 4 | real | - | I-vector endpoint. | |
| 4 | real | - | J-vector endpoint. | |
| 4 | real | - | K-vector endpoint. | |

| | | | |
|---|---|---|---|
| | | | |

The second item in a CL-record is always indicating the type of record. In this case a type 5000 record, linear motion record is shown. This type of record contains an implementation dependent, maximum number of endpoints. (The interface of CAM-postwriter breaks this record down to records containing just one endpoint.)

Some other types of CL-records are:

| type | kind of record. |
|---|---|
| 1000 | Identification record. |
| 2000 | Postprocessor statement (machin command) record. |
| 3000 | Circle geometry record. |
| 5000 | Linear motion record. |
| 6000 | Don´t CUT / CUT record. |
| 14000 | Fini record. |
| 15000 | Circulair motion record. |

Possible occurences of sequences of CL-records.

| | | | | |
|---|---|---|---|---|
| 1000 | 1000 | 1000 | 1000 | 1000 |
| 2000 | 3000 | 6000 | 14000 | 15000 |
| | 5000 / 5 | | | |
| | 5000 / 6 | | | |
| | 5000 / 6 | | | |

Each new sequence of records is started with a type 1000 record. (However there are implementations of the CL-file that don't use a type 1000 record at all. This is solved in the interface of CAM-postwriter by generating a type 1000 record. In this way the definition of CL-records in CAM-postwriter becomes independent of the NC-programming system's implementation of the CL-file.

A special sequence of records is shown in the second column. First a type 1000 record. The type 1000 record contains the statement number of the part program. Next a type 3000 record, which contains the circle geometry, center point and radius. The next record is a type 5000, subtype 5 record. The movement of the tool is split into short linear motions along the circle. (The number of end coordinates in the record is implementation dependent, e.g. 35 endpoints.) If necessary a continuation of linear motions along the circle will be given in a type 5000, subtype 6 records. The number of linear motions depends on the tolerance, the radius of the circle and the part of the circle that has to be machined.

The last column shows the, compact, alternative without linear motions along the circle. The type 15000 record contains all the information of the circular motion, start point, center point, end point, radius and direction of movement.
In both cases the circular interpolation mode of the machine control can be used. In the case of the sequence with type 3000 record, the linear motions are not output to the machine control. In the case the circle radius is too large to use the circular interpolation mode of the machine control only the sequence with type 3000 record can be used.

# 9. CL-record specification.

## 9.1. General.

In a CL-record specification a connection is made between the value of the parameter in the CL-record and the parameter in the postprocessor. The name of the parameter can be chosen freely. However, a significant name makes the specification readable and easier to understand.

**The type and position of the parameters are determined by the structure of the CL-record.**

The structure of the CL-records in the CL-file doesn't change. (Provided the CL-file is generated by the same CAM-system.) So this part of the specification is for each postprocessors the same.

Example of a CL-record in the CL-file:

| number of bytes | type | value | meaning |
|---|---|---|---|
| 4 | integer | - | record sequence number |
| 4 | integer | 5000 | type 5000 record (linear motion) |
| 4 | integer | 5/6 | 5-first motion record, 6-continuation record. |
| 4 | char | - | name of geometry in programming system. |
| 4 | integer | - | index of geometry in programming system. |
| 4 | real | - | X-coordinat endpoint. |
| 4 | real | - | Y-coordinat endpoint. |
| 4 | real | - | Z-coordinat endpoint. |

This CL-record contains a sequence of 3 integer numbers, a character string, an integer number and 3 real numbers. The declaration of this CL-record is described in next chapter.

## 9.2. Definition of CL-records in CAM-postwriter.

Example of a CL-record specification:

```
 1
 2  CL_REC
 3    record_number, 5000, sub_type, surf_name, surf_index,
 4      x_apt, y_apt, z_apt;
 5
 6  VAR
 7    record_number : integer;
 8    sub_type      : integer;
 9    surf_name     : char*6;
10    surf_index    : integer;
11
12  ACTION
13    N := N + block_incr;
14    if fedrat_motion then Gcode := 1 else Gcode := 0;
15    NC_BLOCK( N, Gcode, x_apt, y_apt, z_apt )
16  END;
17
```

**NOTE:** The line numbers in this example are not part of the specification but are only used for explanation.

The CL-record declaration is started with the keyword "CL_REC" on line 2. On line 3 and 4 the CL-record declaration is found, a sequence of parameters. The declaration of the parameters in the CL-record starts at line 6 with the keyword "VAR". The actual parameters are declared on line 7 to10. (The parameters may have been declared in another part of the specification file, before the declaration of the CL-record.)

The action, which will be executed by the CL-record, is defined on line 13 to 15 proceeded by the keyword "ACTION" on line 12. After the keyword "ACTION" one can specify which specific actions will be executed after the CL-record is read from the CL-file. For example an action to generate a NC block which contains the coordinates to move towards.

The declaration of the CL-record is closed by the keyword "END" on line 16.

The number "5000" is only present in the CL-record declaration. It is recognized as an integer constant with a value equal 5000. The second parameter in the CL-record is always used to recognize the type of CL-record. In some cases also the third parameter is used as CL-record identification. So, the second parameter has to be a constant and the third parameter may be a constant.

The following requirements must be fulfilled:

- The first three parameters in the CL-record must be of type integer.
- Only the second and third parameter may be constants.
- A CL-record specification cannot contain more than 20 parameters.
- All parameters must have been declared at the end of the CL-record declaration.

## 9.3. Execution of CL-records by the postprocessor.

When the postprocessor is started it will read CL-records from the CL-file one by one. For each CL-record will be determined whether there exist a CL-record specification or not. If no specification exists an error message will be given and the next CL-record will be read. If there is a CL-record specification the value of the parameters in the postprocessor will be updated with the value of the same parameters in the CL-record.

The next step is to determine whether a CL-record action is defined. (This action is defined after the keyword "ACTION" in previous example.) If a CL-record action is defined this action will be executed. A CL-record action can consist of many sub actions. After the CL-record action is completed, or if there was no CL-record action, the next CL-record will be read.

The processing of CL-records will stop after a type 14000 record, a FINI record, has been read from the CL-file.

Next an example of the processing of a CL-record by CAM-postwriter. This example refers to the previous given example in "Definition of CL-records in CAM-postwriter". First the APT statements are given which result in output in the CL-file. (Your CAM system may produce this result in a different way.)

## APT-statements:

STRPNT = POINT / 0, 50, 100
GOTO / STRPNT

## Result in CL-file:

(here man readable displayed.)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 13 | 5000 | 5 | STRPNT | 0 | 0.000 | 50.000 | 100.000 |

## Actions taken by postprocessor:

### 1. Cl-record is read and values are stored in parameters.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 13 | 5000 | 5 | STRPNT | 0 | 0.000 | 50.000 | 100.000 |
| ⌡ | | ⌡ | ⌡ | ⌡ | ⌡ | ⌡ | ⌡ |
| record_number = 13 | 5000 | sub_type = 5 | surf_name = "STRPNT" | surf_index = 0 | x_apt = 0.000 | y_apt = 50.000 | z_apt = 100.000 |

### 2. Action specified with CL-record is executed.

**On line 13** the NC-group "N" is incremented by variable "block_incr".
Suppose N = 110 and block_incr = 10. The new value of N becomes 120. (In this example is chosen for a NC-group named "N" and an address attribute of this NC-group equal "N". Another name for the NC-group may be "block_number", having the same address attribute. The statement on line 15 then would be: "block_number := block_number + block_incr;". But the output would be exactly the same. This shows the independence of the name of the NC-group and the address of the NC-word.)

**On line 14** is evaluated whether feedrate mode is on or not. Suppose it is not. This means that the NC-group "Gcode" becomes the value 0. This NC-group has an address attribute equal "G".

**On line 15** the output of a NC-block is specified.
In this example the values of x_apt, y_apt and z_apt in the CL-record is directly stored in the NC-groups "x_apt", "y_apt" and "z_apt". These NC-groups have the address attribute "X", "Y" and "Z".
So the NC-block looks like:

N120 G0 X0. Y50. Z100.

### 3. The next record will be read from the CL-file.

# 10. Postprocessor statements.

## 10.1. General.

The type 2000 CL-records are a special type of records in the CL-file. Type 2000 CL-records are postprocessor statements or machine commands. Postprocessor statements are directions to the machine to perform a specific action. For example to load a tool, start the spindle, rotate the machine table and so on.

In CAM-postwriter the type 2000 CL-records are threated also in a special way. They are not specified in a CL-record declaration. (See previous chapter.) Instead postprocessor statements are defined directly in CAM-postwriter.

Below a few examples of postprocessor statements.

```
1   LOADTL / toolnr
2   FEDRAT / MMPR, feedmmpr
3   COOLNT / ON
4   ROTABL / CLW, angle
5   END
```

The major and minor words in this example are typed in **bold** case, parameters are typed in *italic* case. Postprocessor statements are assembled from major words and zero or more minor elements. If there are minor elements, the major word and the minor elements are separated by a "/". The major word is always the first word in the postprocessor statement. (There can only be one major word in a postprocessor statement.) A minor element consists of a minor word followed by zero or more parameters. There is one exception in CAM-postwriter to this rule. In only the first minor element after the "/" the minor word can be omitted. See above example on line 1.

A list of major and minor words will be found in appendix 1.

You can build postprocessor statements from any combination of major words, minor words and parameters. The major and minor words are predefined, you have to choose from the list in appendix 1. For the parameters you can choose any name, within the given restrictions. So the number of postprocessor statements you can build is enormous.

Exceptions to the rules above exist for the major words PARTNO, PPRINT and INSERT. These statements have a predefined format:

- PARTNO <text>
- PPRINT <text>
- INSERT <text>

Everything that follows these major words is considered as a string. The maximum length of the string is dependent on the CAM-system, while the number of characters in a data field in a CL-record is also. These dependencies however are solved by the CAM-postwriter CL-file interface.

## 10.2. Definition of postprocessor statements in CAM-postwriter.

Below an example of the definition of a postprocessor statement.

```
1    PP_STAT
2      SPINDL/ CLW;
3      SPINDL/ CCLW;
4      SPINDL/ AUTO;
5      SPINDL/ revolutions_per_min;
6      SPINDL/ SFM, surface_speed;
7      SPINDL/ SFM, surface_speed, CLW;
8      SPINDL/ SFM, surface_speed, CCLW;
9      SPINDL/ OFF;
10     SPINDL/ OFF, orientation_angle;
11
12   MNR_LMNT CLW
13     . . .
14   END;
15
16   MNR_LMNT CCLW;
17     . . .
18   END;
19
20   MNR_LMNT AUTO;
21     . . .
22   END;
23
24   MNR_LMNT OFF
25     . . .
26   END;
27
28   MNR_LMNT revolutions_per_min;
29     . . .
30   END;
31
32   MNR_LMNT SFM, surface_speed;
33     . . .
34   END;
35
36   MNR_LMNT OFF, orientation_angle;
37     . . .
38   END;
39
```

The definition of the postprocessor statement starts with the keyword "PP_STAT" on line 1. On line 2 to 10 all occurrences are listed of the SPINDL statement. Each postprocessor statement has a specific purpose, and is declared by the postprocessor writer with this purpose in mind. On line 5 an example of a minor element without a minor word. On line 12 to 38 all occurrences of minor elements are declared in the list of postprocessor statements.
Notice that the minor element "`CLW`" is used in the postprocessor statements on line 2 and 7. The minor element "`SFM, surface_speed`" is used in the postprocessor statements on line 6, 7 and 8.

**Important.**
The minor word and **the number of parameters** uniquely identify a minor element. **Not the name of the parameters** in the minor element.

In a CL-record a postprocessor statement is recognized by the 2000 record type. The minor element in this record is recognized by the integer code number of the minor word followed by a **number** of values. These values represent the values of the parameters in the minor element. So there can't be made a difference between the minor element "`SFM, surface_speed`" and the minor element "`SFM, speed`". Does the value belong to the parameter "surface_speed" or "speed"?

The postprocessor statements "`SPINDL/ SFM, surface_speed, CLW`" and "`SPINDL/ CLW`" have a common minor element: "CLW".

When an existing minor element is declared once again with different parameters an error message will be given.

**Notice:**
The postprocessor statements "`LOADTL/ SFM, surface_speed`" and "`SPINDL/ SFM, surface_speed`" have a NO common minor element. The first minor element is part of a LOADT statement and the second is part of a SPINDL statement. They, however, might use the same parameter(s) and perform the same action.

## 10.3. Execution of postprocessor statements by the postprocessor.

Below an example of the definition of a postprocessor statement.

```
1    PP_STAT
2      SPINDL/ CLW, SFM, surface_speed;
3
4      MNR_LMNT CLW;
5        VAR  rotation_direction : CHAR*4;
6        INIT rotation_direction := "";
7      ACTION
8        rotation_direction := "CLW";
9      END;
10
11     MNR_LMNT SFM, surface_speed;
12       VAR surface_speed : 0. .. 1000.;
13     ACTION
14       block_nr := block_nr + block_incr;
15       G := 96;
16       S := surface_speed;
17       if rotation_direction = "CLW" then Mcode := 3 else Mcode := 4;
18       NC_BLOCK( N, G, S, M )
19     END;
20
21   END;
```

The structure of line 1 to 21 is called a postprocessor statement block. On line 2 the postprocessor statement is declared. In this example only one, there could have been many more spindle statements.

All the used minor elements in the declared postprocessor statements have to be declared in the same postprocessor statement block. This is done on line 4 through 9 and 11 through 19.
On line 5 the parameter "rotation_direction" is declared, on the next line this parameter is given an initial value. On line 12 the parameter "surface_speed" is declared. This parameter is part of the minor element and used here for the first time. The parameter "surface_speed" is given here a specific range of possible values. When a value outside this range is given in the postprocessor statement, an error message will be given.
On line 14 to 18 the actions are defined.

**What will happen when a SPINDL postprocessor statement is read from the CL-file?**

When a postprocessor statement is read from the CL-file the first action is to see if it is declared in a postprocessor statement block. If it is an unknown postprocessor statement, the major word is not used in a postprocessor statement block, the postprocessor statement is ignored and the next record is read from the CL-file. When the postprocessor statement is recognized, it will be broken down in its components and checked against the declared postprocessor statements. If there is no match an error message is given:

```
"*** INPUT ERROR ***"
"Unknown postprocessor statement format."
"On line  . . . of part program."
```

If the postprocessor statement in the CL-file matches a postprocessor statement declaration, the parameters in the minor elements are given the corresponding values in the CL-record.

**Example:**
The postprocessor statement " SPINDL / CLW, SFM, surface_speed " is declared in the specification of the postprocessor. (See previous example.)

In the CAM/NC programming system a command is given to the machine to start the spindle in clockwise direction and the rotation speed of the spindle has to meet a surface speed of 350 m/s at the tool tip. This command is given by the postprocessor statement " SPINDL / CLW, SFM, 350 ".
The postprocessor statement in the CL-file is represented by a sequence of numbers. See the first row. The next row gives an explanation of the meaning of the numbers.

CL-record

| | | | | | | |
|---|---|---|---|---|---|---|
| | 2000 | 1031 | 60 | 115 | 350,000 | |

| | | | | | | |
|---|---|---|---|---|---|---|
| | type 2000 record | code for SPINDL | code for CLW | code for SFM | value 350 | |

When the postprocessor reads the CL-file, the postprocessor statement is recognized by the number 2000. The number 1031 indicate it is a SPINDL statement. Next the minor elements "CLW" and "SFM, surface_speed" are recognized. The first minor element has no parameters, so no update takes place. The second minor element has one parameter, "surface_speed". This parameter will be assigned the value "350".

The next step is to see if there are actions defined for major word or the minor elements. In this example there is no "major action". (It could have been defined between line 19 and 21.) There are 2

minor element actions defined for this example. The first minor element "CLW" has the action that the spindle rotation direction is set to "CLW". The second minor element performs several (sub) actions:

| line number | action |
|---|---|
| 14. | increments the block number. |
| 15. | sets the G-code for surface speed machining. |
| 16. | sets the S-word with the value of the surface speed. |
| 17. | sets the M-code for the direction of spindle rotation. |
| 18. | outputs a NC block with the specified parameters. |

# 11. NCDATA

## 11.1. General

The output of the part program by the postprocessor is called the NCDATA. The NCDATA is ready to be fed into the machine control, to manufacture the part on the machine. (The NCDATA is not the only output of the postprocessor. There may be a tooling list, a set up list etc.)

The NCDATA consists of NC blocks. A NC block is 1 line of information. (The NCDATA is a text file.) Next an example of a few lines, NC blocks, of the NCDATA file.

N0010 G0 X100 Y50 M03
N0020 Z2
N0030 G01 Z-20 F0.2

Each NC block consists of NC words. For example the G codes and M codes are NC words. A NC word consists of an address and a value. In the NC word G01, the address is "G" and the value is "01".

Standard for NCDATA.

In CAM-postwriter the concept of NC-groups is introduced. This is because specific ranges of values in NC words have a specific meaning. For example G0, G1, G2 and G3 are motion interpolation modes, while G53, G54, G55, G56 and G57 activate coordinate systems. (In fact different purposes are supported by 1 NC word.)
In CAM-postwriter, and also in some machine control programming manuals, the NC words are separated in specific NC-groups to keep this aspect in mind.

## 11.2. Nc_group_block

A NC-group is declared in a nc_group_block. The syntax of a nc_group_block is described below. Also an example and explanation is given.

```
 1  NC_GROUP X_coord : -999.9999 .. 999.9999;
 2
 3    ATTR    nc_data          = true;
 4            address          = 'X';
 5            modal            = false;
 6            format           = 9:4;
 7            plus_sign        = false;
 8            minus_sign       = true;
 9            sign_pos         = front;
10            leading_zeros    = false;
11            trailing_zeros   = false;
12            spaces           = false;
13            nclist           = true;
14            position         = 24;
15
16    INIT    X_coord := 999.9999
17
18  END;
```

The nc_group_block starts with the keyword "NC_GROUP". Next follows the identifier of the NC-group, the name of a parameter, a colon ":", a type specification and a semicolon ";". The identifier is the name of a variable. It cannot be the name of a constant. The type of the variable is integer, real or character. In this case a specific range is given. The control is not capable of processing smaller or larger values. By limiting the value of the NC-word in this way an error message will be generated when the listed values are exceeded.

On line 3 the keyword "ATTR". On the next lines follow a number of attributes of this NC-group. In this example all attributes are listed and explained. Most attributes have default values that match modern machine controls. Using these defaults the same example may be written like:

```
NC_GROUP X_coord : -999.9999 .. 999.9999;
  ATTR   address          = 'X';
         format           = 9:4;
         position         = 24;
  INIT   X_coord := 999.9999
END;
```

On line 16 the value of the NC-group X_coord is initialized. X_coord has an initial value of 999.999.

Next follows an explanation of all attributes, their meaning and default values.

| Attribute | Possible values | Default | Meaning |
|---|---|---|---|
| nc_data | true \| false | true | This attribute specifies whether this NC-group is part of the NCDATA or not.<br>If the value of this attribute is true and output of this NC-group is written, it will be written into the NCDATA.<br>If the attribute is false the output will never be written into the NCDATA. |
| address | <string> | | This attribute contains the address of the NC word. The address is a string of up to 32 characters.<br>The address and name of the NC-group doesn't have to be the same. |
| modal | true \| false | false | Only when the value of the NC word has changed it will be output in the NC block.<br>The output of a non-modal NC-group will be repeated in every NC block it is specified. |
| format | <l_field>:<l_decimal_field> \| <l_field> | | Gives the format of the output of the NC word. For NC-groups of type REAL the format has the form <l_field>:<l_decimal_field>. <l_field> specifies the total field length, included with the address of the NC-group. <l_decimal_field> specifies the decimal field length of the NC-group. <l_decimal_field> = 3 specifies 3 digits behind the "."<br>For NC-groups of type INTEGER or CHAR the format has the form <l_field>. |

| Attribute | Possible values | Default | Meaning |
|---|---|---|---|
| plus_sign | true \| false | false | If plus_sign = true then the "+"-sign will be used in positive values of the NC-group. |
| minus_sign | true \| false | true | If minus_sign = true then the "-"-sign will be used in negative values of the NC-group. |
| sign_pos | front \| rear | front | This attribute gives the position of the sign(s). |
| leading_zeros | true \| false | false | If leading_zeros = true then leading zeros will not be removed from the value of the NC-group. (X = 299.8 will be output as X 00299.8)<br>If leading_zeros = false then leading zeros will be removed from the value of the NC-group. (X = 299.8 will be output as X 299.8) |
| trailing_zeros | true \| false | false | If trailing_zeros = true then trailing zeros will not be removed from the value of the NC-group. (X = 299.8 will be output as X 299.800)<br>If trailing_zeros = false then trailing zeros will be removed from the value of the NC-group. (X = 299.8 will be output as X 299.8) |
| spaces | true \| false | true | Only significant when attribute nc_data = true. If spaces = false then all spaces will be removed from the NC-group before it is written to the NCDATA. |
| nc_list | true \| false | true | This attribute specifies whether this NC-group is written to NCLIST or not.<br>If the attribute is false the output will never be written into NCLIST. |
| position | 1 .. 132 | | Only significant when attribute nc_list = true.<br>If the value of attribute nc_list = true and output of this NC-group is written, the first character will be written into NCLIST on column specified by attribute position. |

# 12. NCLIST.

## 12.1. General

The purpose of the NC-listing, NCLIST file, is to present the information in the NCDATA in a more readable format. In the NCLIST file additional information can be presented like the geometry names, the statement numbers of the part program or the processing time of the workpiece.
Errors and warnings, automatically generated by the postprocessor, will be written into the NCLIST file.

**Note:**
NCLIST has to be reviewed for possible errors or messages. At the end of NCLIST a summary of the number of errors will be reported.

Whether or not the output of NC-blocks, in particularly NC-words, is written into the NCDATA or NCLIST files depends on the attributes "nc_data" and "nc_list" specified in the declaration of the NC-group.

The width and length of the pages in NCLIST are set in a configuration file.
The same applies for the number of empty lines at the top and at the bottom of each page.

The first 2 lines of NCLIST hold a fixed header. This fixed header can't be changed. In this 2 lines the name CAM-postwriter and version, the date and time and the page number is listed. See example on one of the next pages.

The space below the fixed header is available for a user-defined header. A HEAD or HEADLN-statement writes each user-defined header line. The string specified in the HEAD(LN) statement is appended to the user-defined header. The HEADLN( <string> ) statement appends <string> to the user defined header and positions on the next line. The parameter in the HEAD(LN) statement may be a string constant or the name of a parameter.

The maximum number of lines in the user-defined header is 10. Next an example of the definition of a user-defined header in the NC-listing. In this example a part of the specification file is given. The user-defined header is defined here by using character constants in the HEADLN statements. In one HEADLN statement the name of a variable is used, partno_string. This variable is given a value by the PARTNO postprocessor statement and gives a description of the part to be machined. The value of the variable will be printed in the user-defined header on each page. When the value changes, the new value will be printed in the next user-defined header.

The last 5 lines in the example define a mask for each line of information in the NC-listing below the fixed header and the user-defined header. The purpose is to create columns in which the values of the NC-groups can be listed. One of the attributes of a NC-group is the position on the line on which its value will be printed.

```
NC_LIST_HEAD
  HEADLN;
  HEADLN( '**********************************',
          '                                  ',
          '                                  ',
          '**********************************' );

  HEADLN( '*      Postprocessor  ERCMC1       *',
          '                                  ',
          '                                  ',
          '*   < C O M P A N Y  N A M E >    *' );

  HEADLN( '*    Mazak H400, Mazatrol M32      *',
          '                                  ',
          '                                  ',
          '*       < L O C A T I O N >       *' );

  HEADLN( '**********************************',
          '                                  ',
          '                                  ',
          '**********************************' );
  HEADLN;
  HEADLN( '                          ', partno_string:66 );

  HEADLN;
  HEADLN( '  O/N * G * G/L *H/D/P*   X/I/E  *   Y/J/K  ',
          '*   Z/B    *   R/I    *   P/J    *    Q     ',
          '*  F/S   * T * M  | Z-MACH *TIME*DRIVE SURF.' );
  HEADLN( '------*---*-----*-----*----------*----------',
          '*----------*----------*----------*----------',
          '*--------*---*----|--------*----*-----------' )
END;

NC_LIST_MASK
  MASK  ( '      *   *     *     *          *          ' );
  MASK  ( '*          *          *          *          ' );
  MASK  ( '*        *   *    |        *    *          ' )
END;
```

The pictures below show the result. (The fixed header is given in the first example here but comes before every user-defined header.) The user-defined header will be printed on each new page in the NC-listing. On the position of <partno_string> the value of this variable will be printed. The last 4 lines in the last example are masks for each line below the user-defined header. The purpose is to separate the output of the different NC-groups.

The NC-listing has a maximum width of 132 characters. You can of course use less, depending on your needs.
(The example below uses a very small character size to show the full width of the NC-listing. Normally you will use a landscape orientation to keep it readable.)

Example of fixed header.

```
                                                                                time 10.14.56     date 15/ 7/2004     page   4
CAM-Postwriter (C)    vrs.  3.03 released  17/ 4/2004
```

Example of user-defined header and user defined mask.

```
**********************************                                                  **********************************
*      Postprocessor  ERCMC1      *                                                *   < C O M P A N Y  N A M E >    *
*    Mazak H400, Mazatrol M32     *                                                *       < L O C A T I O N >       *
**********************************                                                **********************************

                                                    <partno_string>

  O/N * G * G/L *H/D/P*   X/I/E  *   Y/J/K  *   Z/B    *   R/I    *   P/J    *    Q     *  F/S   * T * M  | Z-MACH *TIME*DRIVE SURF.
------*---*-----*-----*----------*----------*----------*----------*----------*----------*--------*---*----|--------*----*-----------
      *   *     *     *          *          *          *          *          *          *        *   *    |        *    *
      *   *     *     *          *          *          *          *          *          *        *   *    |        *    *
      *   *     *     *          *          *          *          *          *          *        *   *    |        *    *
      *   *     *     *          *          *          *          *          *          *        *   *    |        *    *
```

# 13. User defined variables and constants.

## 13.1. General.

In CAM-postwriter the user can define variables and constants. The constants have a fixed value, which cannot be changed, set at the moment of declaration. The variables and constants have a name, also called "identifier", by which their values can be referenced. Variables and constants have to be declared before their values can be referenced in, for example, an equation. The CAM-postwriter specification file is read from begin to end, the declaration has to occur before the reference. A name of a variable or constant can only be used once in a declaration and the identifier has to be unique.

The name of the variables and constants consist of alphanumeric characters and has to start with a letter. The name may also contain underscore(s) "_". The complete set of characters for an identifier is {A..Z, 0..9, "_"}.
The maximum length of a name is 32 characters. A name may not be equal to a CAM-postwriter key word. Names of variable and constants are not case sensitive. The variable "ABC" is the same as "abc".

## 13.2. Predefined constants.

Predefined constants are:

| Name | Type | Value | Remarks |
|---|---|---|---|
| MAX_INTEGER | integer | +2147483648 | This value is implementation dependent. |
| MIN_INTEGER | integer | - 2147483648 | This value is implementation dependent. |
| MAX_REAL | real | +3.4E38 | This value is implementation dependent. |
| MIN_REAL | real | -3.4E38 | This value is implementation dependent. |
| TRUE | logical | true | |
| FALSE | logical | false | |

Because the name of a variable or constant has to be unique, a user cannot declare variables or constants with above listed names. These names are reserved by CAM-postwriter.

## 13.3. Types of variables and constants.

Variables and constants in CAM-postwriter are of type integer, real, character or logical.

**Integer**.
A variable of type integer is capable of representing all natural numbers like -342, -1, 0, 1, 45 or 35425255. The smallest and biggest natural numbers are represented by the predefined constants MIN_INTEGER and MAX_INTEGER.

**Real**.
Variables of type real are capable of representing real numbers like -234.99, -2.3, -1.0, 0., 1.3, 3.141569 and 999.99. MIN_REAL and MAX_REAL represent the smallest and biggest real numbers.

**Character.**
Variables of type character are capable of representing strings of characters. The maximum length of a string is 128 characters. A character is one of the set of ASCII characters.

**Logical.**
A variable of type logical is capable of representing logical values. There are only two logical values: true and false. Two constants, TRUE and FALSE, are predefined.

## 13.4. Arrays.

All types of variables can be declared as an array. The upper bound and lower bound are not limited, they have to be greater or equal MIN_INTEGER and less or equal MAX_INTEGER. However the upper and lower bound determine the number of array elements in the array. Large numbers of array elements might probably result in a storage problem in CAM-postwriter. You will get an error message in this situation. In case you encounter this problem contact CAM-postwriter to increase the storage capabilities.

The maximum dimension of an array is 4. This means that the maximum number of subscripts is 4.

Next an example of the declaration of an array and the assignment of a value to an array element.

```
VAR
  I_array : ARRAY[ 1..10, 1..10, 1..10, 1..10 ] OF INTEGER;

ACTION
  I_array[ 1, 8, 2, 9 ] := 3;
    .
```

## 13.5. Range of variables.

For variables of type integer and real a range can be specified. This range describes the legal values of the variable. In next example the declaration of a variable with the declaration of the range of the variable.

**Example:**

```
VAR
  tool_number   : 1   .. 4;
  surface_speed : 10. .. 400.;
   .
   .
```

In this example a variable **tool_number** is declared. The variable is of type integer. This is because the range itself is of type integer. The variable **surface_speed** is of type real. The lower and upper ranges have to be of the same type and the lower range has to be smaller than the upper range.

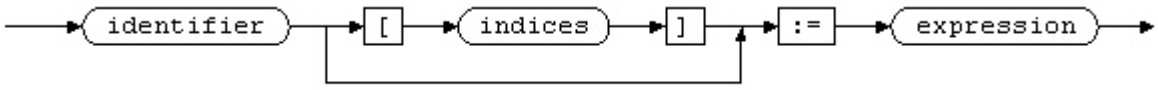
Possible values of tool_number are 1, 2, 3 and 4. Assigning any other value to tool_number will result in a error message.

**By using ranges for variables the user is capable of generating, implicit, error messages.**

**Example:**
The number of tool positions on a machine is for example 1 to 4. The variable tool_number is used in a postprocessor statement to activate a tool change on the machine.
(The statement looks like: LOADTL / tool_number.) By defining a range for tool_number, any value used outside the range will result in an error message. This without having any code written to check the tool number.

# 14. Assign statement.

The following figure describes the syntax of an assign statement.




An assign statement is used to assign a value to a variable. The variable can be a simple type or of type array. The right number of array indices has to be specified, in case of a simple type **NO** indices.

In CAM-postwriter the concept of "strong typing" is used. This means that **NO** implicit type conversion takes place. The type of the result of the expression has to be equal to the type of the variable. (Integer, real, character or logical.)

Signaled compilation errors in an assignment statement are:

```
"Unknown variable in an assign statement."
"Trying to assign a value to a constant."
"Type conflict in an assign statement."
"Incorrect number of subscripts."
```

Signaled runtime errors in an assignment statement are:

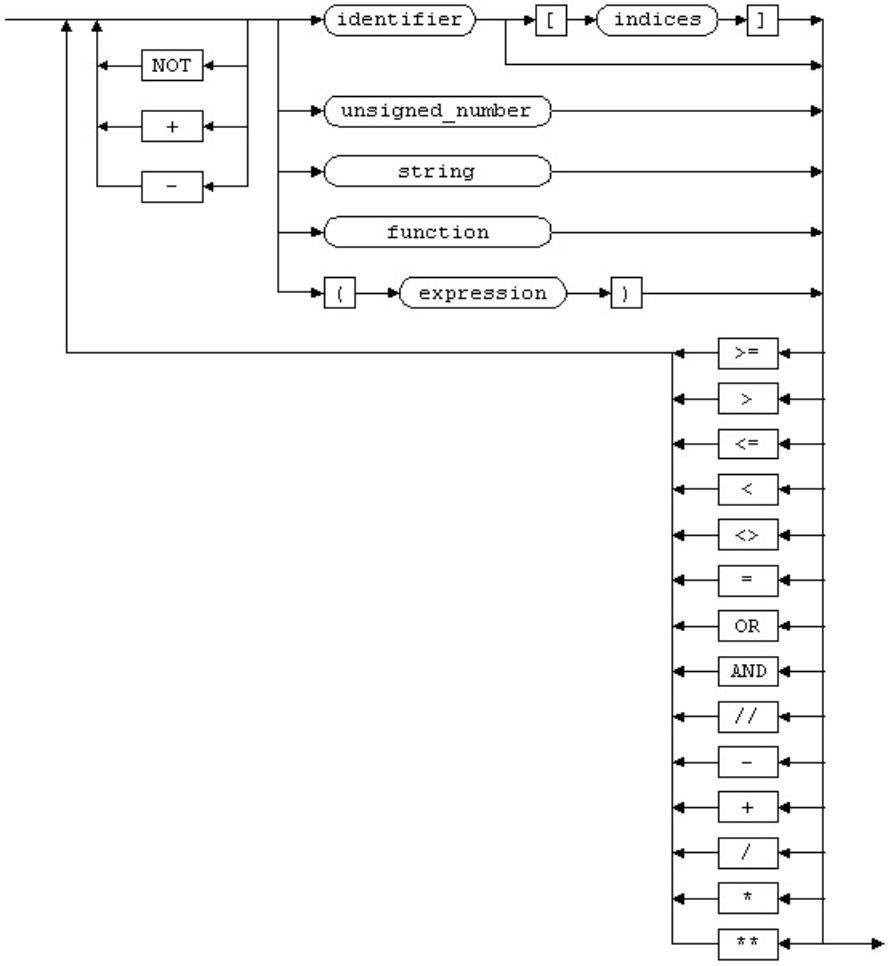
```
"Array out of bounds."
"Operand overflow"
"Operand underflow"
```

Below a few examples of this error messages. The line number referenced here is the line number of the specification file. The item is the variable that is attempted to assign a value to, which is too large or small. See also the chapter about error messages.

```
*** RUNTIME ERROR ***
Operand overflow.
On line :        99
Item    : C_ARRAY

*** RUNTIME ERROR ***
Operand underflow.
On line :       103
Item    : I2

*** FATAL RUNTIME ERROR ***
Array out of bounds.
On line :       112
Item    : C_ARRAY
```

# 15. Expression.

An expression consists of operand(s), operators, functions and possibly one or more sub expressions. An operand is a variable, constant or an array element. The syntax diagram of an expression is given in next figure.




No type conflicts are allowed in an expression. This means that the right type of operands and operators or functions has to be used. For example the addition of an integer and a real is not allowed. One has to use an explicit type conversion of one of the numbers to add the numbers.

For a complete list of operators, functions and their number and type of operands, the resulting type and priority see Appendix 2. "Operators and Functions."

**Compilation stage.**
In case of a type conflict in an expression an error message will be generated.
The number of items in an expression is limited. For the current version is the maximum number of items 128. When this number is exceeded an error message will be given.

**Execution stage.**
At the time evaluation of the value of an expression takes place, all the operands in the expression must have a value. A value can be assigned by initialization during declaration or by an earlier assignment in the execution stage. When an operand is referenced without an initial value an error message is given.

When operators are equal in priority and "(" and ")" are omitted, evaluation of the expression takes place from left to right. An exception on this rule are the operators "**" (power), NOT, + (plus sign) and - (minus sign). Here evaluation from right to left takes place.

Examples:

| expression | evaluation order | wrong evaluation order |
|---|---|---|
| 12 / 6 * 2 | ( 12 / 6 ) * 2 = 4 | 12 / ( 6 * 2 ) = 1 |
| 4 ** 3 ** 2 | 4 ** ( 3 ** 2 ) = 262144 | ( 4 ** 3 ) ** 2 = 4096 |
| not false or true | ( not false ) or true = true | not ( false or true ) = false |

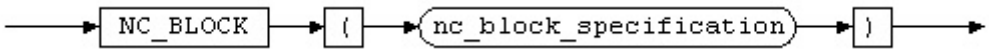
Signaled compilation errors in an expression are:

```
"Too many expression items in an expression."
"Unknown operand in an expression."
"Type conflict in an expression."
"Unknown file identifier."
```

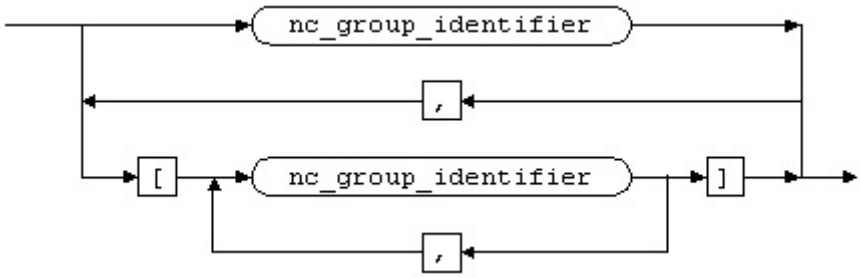
# 16. NC_BLOCK statement.

The next syntax diagrams describe the syntax of a nc_block statement.

**Nc_block statement:**




**Nc_block_specification:**




The nc_block statement allows the construction of a NC block. (A NC block is one line of information for the machine control.) Depending on the attributes of the NC_groups, information will be written to the files NCLIST and NCDATA.

Nc_groups specified between square brackets ("[" and "]") will be output optionally. The attribute MODAL of nc_groups specified for optional output must be equal TRUE. In other words: only modal nc_groups can be specified to be conditional output. When nc_groups are enclosed between square brackets they will be output only when their value has changed since their last output. Nc_groups not enclosed between square brackets will be unconditional output.

The maximum number of nc_groups in a nc_block statement is 32. When this number is exceeded a warning will be given.

Signaled compilation errors in a nc_block statement are:

```
"Unknown nc_group in an nc_block_specification."
"Optional output can only be specified for a modal nc_group."
"Too many nc_groups in een nc_block_specification."
```

**Examples:**

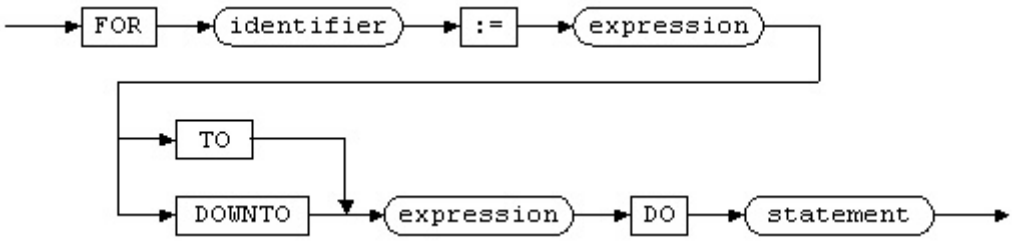
```
  .
703 *              nc_block( unknown_var );

*** SEMANTIC ERROR ***
Unknown nc_group in an nc_block_specification.
On line :       703
Item    : unknown_var
 .
 .
707 *              nc_block( [block_number] );

*** SEMANTIC ERROR ***
Optional output can only be specified for a modal nc_group.
On line :       707
Item    : block_number
```

# 17. Loop - statements.

As many programming languages CAM-postwriter provides "loops". Loops are used to perform specific actions several times.

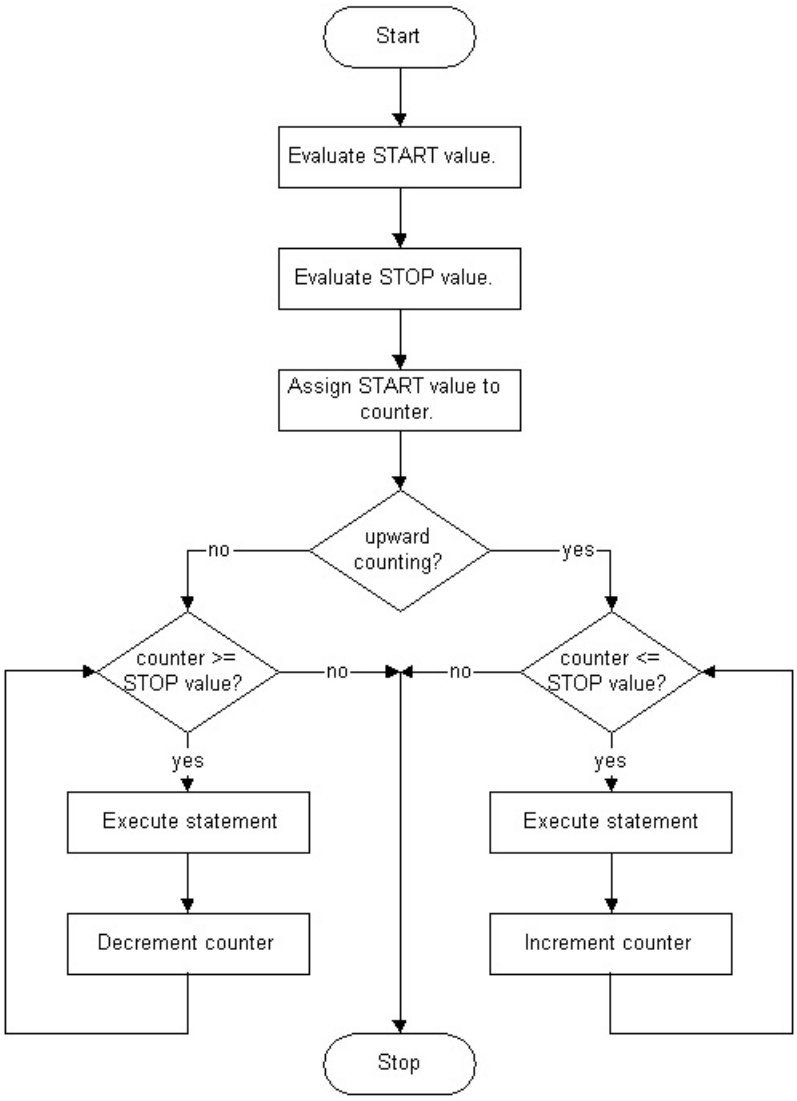
## 17.1. FOR - statement.

In next figure the syntax of the FOR-loop is given.



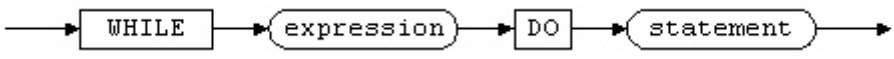

The FOR-loop is a counting loop. The counter is the identifier specified in the syntax diagram. At the start of the FOR-loop the counter is assigned the value of the (first) expression. This value is called the START value.
The STOP value of the counter is specified in the second expression in the FOR-statement. Both expressions are evaluated only once at the start of the FOR-loop. The counter and the expressions have to be of type integer.

When the counter is in the range of the START and STOP value the statement will be executed. After execution of the statement the counter will be increment or decrement by 1, depending on the counting direction "TO" or "DOWNTO". After this the counter will be checked again to be in the range of the START and STOP value. When it is, the statement will be executed once again and so on, until the counter has a value outside the range of the START and STOP value. (The value of the counter cannot be changed in the statement.) This process is showed in next flow diagram, which illustrates the FOR-loop.

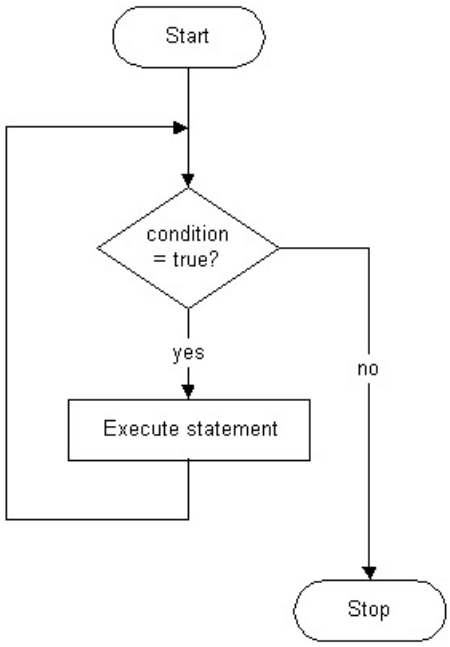



## 17.2. WHILE - statement.

In next figure the syntax of the WHILE-loop is given.



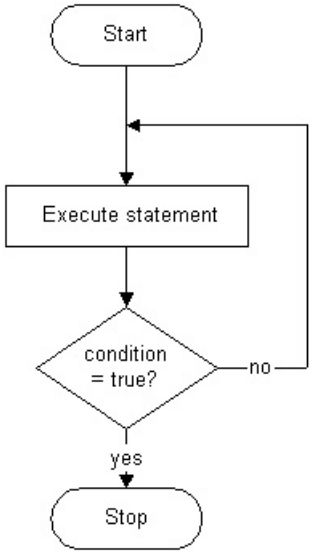

The WHILE-loop is a conditional loop. At the start of the loop the value of the expression will be evaluated. When the value is "true", the statement will be executed. After this again the expression will be evaluated. When the value of the expression is "false" execution of the WHILE-loop will stop. The expression has to be of type logical. This expression is also called the condition of the loop. Next flow diagram illustrates the WHILE-loop.




## 17.3. REPEAT - statement.

In next figure the syntax of the REPEAT-loop is given.

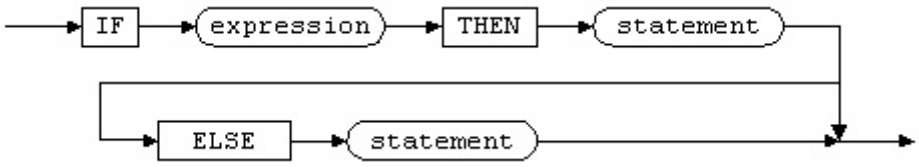

The REPEAT-loop is a conditional loop. At the start of the loop the statement will be executed. (Notice that the statement in a REPEAT-loop will be executed always at least one time.) Next step is to evaluate the expression. When the value of the expression is "true" the REPEAT-loop will stop. When the value of the expression is "false" the statement will be executed again. The expression has to be of type logical. This expression is also called the condition of the loop. Next flow diagram illustrates the REPEAT-loop.


Start
Execute statement
condition
= true?
no
yes
Stop

# 18. IF ... THEN … ( ELSE … ) statement.

The IF - THEN - statement is a conditional statement. This means that the statement only will be executed if the condition is "true". Another form is the IF - THEN - ELSE - statement. In this statement also a statement is given which will be executed if the condition is "false".

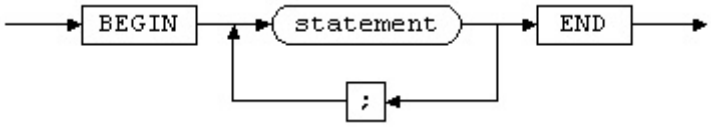
In next figure the syntax of the IF-statement is given.




The expression has to be of type logical.

# 19. Compound statement.

In next figure the syntax of the compound statement is given.




The purpose of the compound statement is to group several statements together as one.
Loop - statements and if - then - statements for example contain only **one** statement in their syntax. By using the compound statement one can group statements together and use them as one statement in a loop. See the example below.

```
ACTION
    .
    .

{ Example of FOR-statement without compound statement. }

  FOR i := 1 TO 10 DO i_array[ i ] := i;
  r_array[ i ] := float( i );

{ Example of FOR-statement with compound statement. }

  FOR i := 1 TO 10 DO
  BEGIN
    i_array[ i ] := i;
    r_array[ i ] := float( i );
    c_array[ i ] := 'Example'
  END;

    .
    .
```

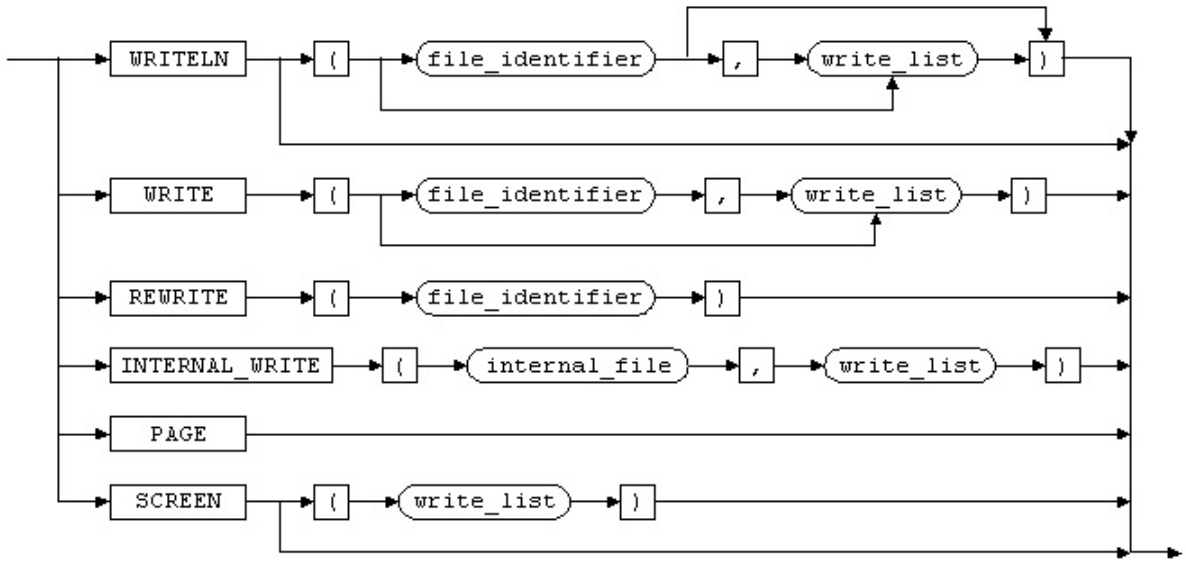
The first FOR statement is a FOR loop with just one statement. The only statement in the loop is the assignment of a value to an array element of "i_array". The assignment of a value to an array element of "r_array" is outside the loop!

In the second FOR statement a compound statement is used. A compound statement is interpreted by CAM-postwriter as one statement. The compound statement can contain an unlimited number of statements. So in the second FOR statement assignments are made to the array elements of "i_array", "r_array" and "c_array".

# 20. OUTPUT and INPUT to files.

## 20.1. General.

The write statement allows writing of information to external files (files on disk) or to internal files (internal files are just character strings). See the syntax of the write statement in next diagram.

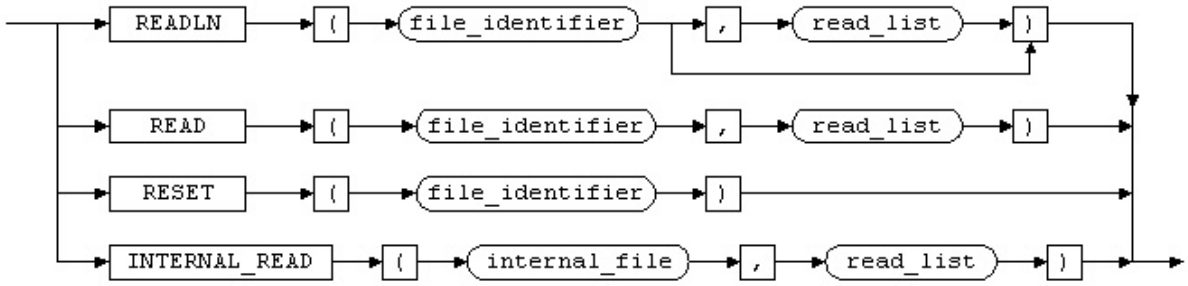



A write statement containing a file identifier writes information into an external file. A write statement without a file identifier writes information into NCLIST.

Positioning on a new line of the file is accomplished by the keyword WRITELN. When all information in the write statement is written to the file the last action will be to position on a new line (to write a newline). When the keyword WRITE is used no newline will be written at the end of the action. The next write statement will append information to the same line.

The PAGE statement positions on a new page of NCLIST. The SCREEN statement enables you to write information to the monitor of your workstation.

The read statement allows reading of information from external or internal files. See the syntax of the read statement in next diagram.




The reading of information starts form the first character in the external or internal file. The positioning on a new line is accomplished by the keyword READLN. Any characters left on the current line are ignored.

Important in the process of reading information from file are the functions EOF and EOLN. These functions inform you whether you arrived at an end of file or an end of line performing the read actions. Performing a READ action when you arrived at the end of a line while result in an error. You have to position first on the next line, if any, with the READLN action before you read the next character. When you arrived at the end of a file any new READ action from this file would result in an error.

## 20.2. Output to NCLIST.

A write statement without a file identifier writes information into NCLIST.
NCLIST is a special kind of external file. CAM-postwriter writes output to NCLIST. NC blocks and for example page headers are written by CAM-postwriter into NCLIST. CAM-postwriter also provides you with the possibility to write information into NCLIST. For example to generate a remark, warning or error message.

## 20.3. External files.

Before information can be written to an external file, the file has to be declared and has to be in write mode. You can open a file in write mode by the rewrite statement. If the file doesn't exist it will be created by the rewrite statement. If the file does exist, all information in the file will be overwritten and the file will be empty.

Before you can read information from a file it also has to be declared and the file has to be in read mode. You can open a file in read mode by the reset statement. The reset statement opens the file and positions on the first character in the file. If the file doesn't exist the reset statement will create an empty file.

**Example:**

```
51 *
52 *     VAR   text_file : TEXT;
53 *           message   : CHAR*13;
54 *
55 *     ACTION
56 *
57 *       REWRITE( text_file );
58 *         WRITE( text_file, 'Error message' );
59 *       WRITELN( text_file );
60 *
61 *       RESET( text_file );
62 *        READ( text_file, message:13 );
63 *
```

On line 52 is the declaration of the external file. The rewrite statement on line 57 creates and brings the file in write mode. (We assume here that the file didn't exist before this action.) On line 58 the characters 'Error message' are written into the file, while line 59 appends a new line to the file.

Line 61 brings the file in read mode and positions on the first character in the file. On the line 62 the variable "message" is filled with 13 characters from the file, so message contains 'Error message'.

## 20.4. Internal files.

An internal file is a variable of type character. One can write and read information to and from an internal file by the internal_write or internal_read statement. An internal_write statement is in fact an assignment. The same rules for the assign statement apply here. A value can be assigned to the internal file, the variable “string” in the internal_write statements in the example below. Or, a value can be read from the internal file and assigned to a variable. On line 83 in the example below a value is assigned to variable “i”. Notice that the string has to represent a valid integer number, otherwise an error message will be returned.

**Examples:**

```
58 *
59 *  VAR string : CHAR*128;
60 *      i      : INTEGER;
..
63 *  ACTION
..
72 *    INTERNAL_WRITE( string, 'Error message');
74 *
75 *    INTERNAL_WRITE( string, 'On linenr : ',stat_nr:4,
76 *                            ' Rapid Circle motion' ) ;
..
83 *    INTERNAL_READ ( string, i:15 );
```

# Appendix 1. List of major and minor words.

Depending on the CAM/NC programming system you are using, there might be small differences. See the documentation of the CAM/NC programming system. Major and minor words can be added or deleted from the used set and coding of the words may be changed also to meet the requirements of your CAM/NC programming system.
When your CAM/NC programming system does use a different set of major and / or minor words contact CAM-postwriter. You will be provided with a set, which meets the requirements of your CAM/NC programming system.

CAM-postwriter is initially provided with a set of major words and a set of minor words. These sets are listed below.

### List of major words and integer code numbers.

| Major word | Integer code number |
|---|---|
| AIR | 1011 |
| AUXFUN | 1022 |
| CHECK | 1023 |
| CHUCK | 1032 |
| CLAMP | 1060 |
| CLEARP | 1004 |
| CLRSRF | 1057 |
| COOLNT | 1030 |
| COPY | 1040 |
| COUPLE | 1049 |
| CUTCOM | 1007 |
| CYCLE | 1054 |
| DELAY | 1010 |
| DRAFT | 1059 |
| END | 1 |
| FEDRAT | 1009 |
| HEAD | 1002 |
| INDEX | 1039 |
| INTCOD | 1020 |
| INSERT | 1046 |
| LEADER | 1013 |
| LETTER | 1043 |
| LOADTL | 1055 |
| MACHIN | 1015 |
| MCHTOL | 1016 |
| MODE | 1003 |
| OPSKIP | 1012 |
| OPSTOP | 3 |
| ORIGIN | 1027 |

| Major word | Integer code number |
|---|---|
| PARTNO | 1045 |
| PENDWN | 12 |
| PENUP | 11 |
| PICKUP | 9 |
| PITCH | 1050 |
| PLOT | 1041 |
| PPLOT | 1014 |
| PPRINT | 1044 |
| PREFUN | 1048 |
| RAPID | 5 |
| RESET | 15 |
| RETRCT | 7 |
| REVERS | 1008 |
| REWIND | 1006 |
| ROTABL | 1026 |
| ROTHED | 1035 |
| SAFETY | 1028 |
| SELCTL | 1056 |
| SEQNO | 1019 |
| SPINDL | 1031 |
| STOP | 2 |
| SWITCH | 6 |
| THREAD | 1036 |
| TMARK | 1005 |
| TOOLNO | 1025 |
| TRACUT | 1038 |
| TRANS | 1037 |
| TURRET | 1033 |
| UNLOAD | 10 |

**List of minor words and integer code numbers.**

| Minor word | Integer code number |
|---|---|
| ALL | 51 |
| ARC | 182 |
| AT | 189 |
| ATANGL | 1 |
| AUTO | 88 |
| AVOID | 187 |
| BEVEL | 201 |
| BORE | 82 |
| BOTH | 83 |
| CCLW | 59 |
| CENTER | 2 |
| CIRCUL | 75 |
| CLW | 60 |
| DEEP | 153 |
| DRILL | 163 |
| FACE | 81 |
| FLOOD | 89 |
| GRID | 67 |
| HIGH | 62 |
| IN | 48 |
| INTOF | 5 |
| INCR | 66 |
| INVERS | 6 |
| IPM | 73 |
| IPR | 74 |
| LARGE | 7 |
| LEFT | 8 |
| LINEAR | 76 |
| LOW | 63 |
| MAIN | 93 |
| MANUAL | 158 |
| MAXIPM | 96 |
| MAXRPM | 79 |
| MEDIUM | 61 |
| MILL | 151 |
| MIRROR | 56 |
| MIST | 90 |
| MMPM | 73 |
| MMPR | 74 |
| MODIFY | 55 |
| NOMORE | 53 |
| OFF | 72 |
| OMIT | 186 |

| Minor word | Integer code number |
|---|---|
| ON | 71 |
| OUT | 49 |
| PARAB | 77 |
| PARLEL | 17 |
| PAST | 70 |
| PERPTO | 18 |
| RADIUS | 23 |
| RANDOM | 183 |
| RANGE | 145 |
| REAM | 167 |
| RETAIN | 184 |
| RIGHT | 24 |
| ROTREF | 68 |
| ROUND | 202 |
| RPM | 78 |
| RTHETA | 106 |
| SAME | 54 |
| SCALE | 25 |
| SFM | 115 |
| SMALL | 26 |
| START | 57 |
| STEP | 92 |
| TANTO | 27 |
| TAP | 168 |
| THRU | 152 |
| TIMES | 28 |
| TO | 69 |
| TRANSL | 29 |
| TURN | 80 |
| XAXIS | 84 |
| XCOORD | 116 |
| XLARGE | 31 |
| XSMALL | 32 |
| XYPLAN | 33 |
| XYROT | 34 |
| YAXIS | 85 |
| YCOORD | 117 |
| YLARGE | 35 |
| YSMALL | 36 |
| YZPLAN | 37 |
| ZCOORD | 118 |
| ZXPLAN | 41 |
| | |

# Appendix 2. Operators and Functions

| Operator | Priority 1) | Operand 1 (type) | Operand 2 (type) | Result (type) | Description |
|---|---|---|---|---|---|
| OR | 11 | logical | logical | logical | Logical or |
| AND | 10 | logical | logical | logical | Logical and |
| NOT | 9 | logical | - | logical | Logical not |
| >= | 7 | integer | integer | logical | Greater or equal |
| | | real | real | logical | |
| | | char | char | logical | |
| <= | 7 | integer | Integer | logical | Less or equal |
| | | real | real | logical | |
| | | char | char | logical | |
| <> | 7 | integer | Integer | logical | Not equal |
| | | real | real | logical | |
| | | char | char | logical | |
| > | 7 | integer | Integer | logical | Greater than |
| | | real | real | logical | |
| | | char | char | logical | |
| < | 7 | integer | Integer | logical | Less than |
| | | real | real | logical | |
| | | char | char | logical | |
| = | 7 | integer | Integer | logical | Equal |
| | | real | real | logical | |
| | | char | char | logical | |
| | | logical | logical | logical | |
| // | 6 | char | char | char | Concatenate |
| + | 4 | integer | integer | integer | Addition |
| | | real | real | real | |
| - | 4 | integer | integer | integer | Subtraction |
| | | real | real | real | |
| * | 3 | integer | integer | integer | Multiply |
| | | real | real | real | |
| / | 3 | integer | integer | integer | Divide |
| | | real | real | real | |
| ** | 2 | integer | integer | integer | Power |
| | | real | real | real | |
| + | 1 | integer | - | integer | Positive sign |
| | | real | - | real | |
| - | 1 | integer | - | integer | Negative sign |
| | | real | - | real | |

1) The lowest priority number means the highest priority in execution.

| Function | Priority | Operand 1 (type) | Operand 2 (type) | Result (type) | Description |
|---|---|---|---|---|---|
| CHAR | 5 | integer | - | char*1 | Returns the character by giving the ASCII character code. |
| ICHAR | 1 | char*1 | - | integer | Returns the ASCII character code by giving the character. |
| SQRT | 1 | real | - | real | Returns square root of a number. |
| INT | 1 | real | - | integer | Returns an integer value of a real number. |
| FLOAT | 1 | integer | - | real | Returns a real value of an integer number. |
| ATAN | 1 | real | - | real | Returns the arc tangent. (Angle in radians) |
| COS | 1 | real | - | real | Returns co-sinus of an angle. |
| SIN | 1 | real | - | real | Returns sinus of an angle. |
| IABS | 1 | integer | - | integer | Return absolute value of an integer number. |
| ABS | 1 | real | - | real | Return absolute value of a real number. |
| EOF | 8 | file identifier | - | logical | End of file.<br>Returns 'true' if all characters have been read from file else 'false'. |
| EOLN | 8 | file identifier | - | logical | End of line.<br>Returns 'true' if all characters have been read from current line else 'false'. |
| TIME | 5 | - | - | char | Returns current time. |
| DATE | 5 | - | - | char | Returns current date. |
| MNR_LMNT_ACT | 8 | minor element | - | logical | Returns 'true' if minor element is part of current postprocessor statement else 'false'. |
| ONLY_ MNR_LMNT_ACT | 8 | minor element | - | logical | Returns 'true' if minor element is the only minor element in current postprocessor statement else 'false'. |
| ONLY_MJR_ACT | 8 | - | | logical | Returns 'true' if there are no minor elements in current postprocessor statement else 'false'. |

# Appendix 3. List of Compile error messages.

## Compile errors

### Syntax errors

```
'Unexpected item in input file.'
'Illegal identifier.'
'Illegal character in unsigned_integer_number.'
'Illegal character in unsigned_real_number.'
'Source_file exhausted.'
'Illegal character in input.'
'Illegal post_processor identifier.'
```

### Semantic errors

```
'Unknown item in a cl_record_declaration.'
'Illegal key in a cl_record_declaration.'
'Too many items in a cl_record_declaration.'
'Too many items in a variable_declaration.'
'Integer overflow.'
'Real overflow.'
'Operand already declared.'
'Lower_range_bound >= upper_range_bound.'
'Length specification out of range.'
'Variable doesn''t exist.'
'Variable already initialized.'
'Variable initialized beyond its range.'
'Variable and initial value of different type.'
'Trying to assign a value to a constant.'
'Unknown variable in an assign_statement.'
'Type conflict in an assign_statement.'
'A for_loop_counter must be of type integer.'
'Unknown for_loop_counter in a for_statement.'
'A for_loop_bound must be an integer_expression.'
'A condition must be a logical_expression.'
'Too many write_parameters in a write_statement.'
'Unknown operand in a write_statement.'
'Decimal_field_length >= field_length.'
'No decimal_field_length allowed for others than type real.'
'Constant of an illegal type.'
'Operand must be a constant.'
'Constant doesn''t exist.'
'Lower- and upper_range_bound not of the same type.'
'Too many expression_items in an expression.'
'Unknown operand in an expression.'
'Type conflict in an expression.'
'Field length must be greater than 0 and less or equal 128.'
'Multiple declaration of an action_block.'
'Multiple use of a for_loop_counter in nested for_loops.'
'Too many nested for_loops.'
'A for_loop_counter cannot be modified.'
'Too many nested call_statements.'
'Unknown action_identifier in call_statement.'
'Existing minor_element, here specified with different identifier.'
'Number of different minor_elements in a pp_statement_block <= 32.'
'Only one major_word allowed in a pp_statement_block.'
'Number of identifiers in a minor_element must be less or equal 16.'
'Operands in an minor_element must be of type variable.'
```

```
'Operands in an minor_element of type logical not allowed.'
'Unknown identifier in an minor_element.'
'Unknown minor_element in this context.'
'Undefined minor_element.'
'Multiple declaration of an minor_element.'
'Multiple declaration of a pp_statement_block.'
'First three cl_record_items must be of type integer.'
'Multiple declaration of a cl_record.'
'Unknown operand in this nc_group_declaration.'
'List_position of nc_group out of range.'
'No constants allowed in a cl_record_declaration except for the second
and third item.'
'Unknown nc_group in an nc_block_specification.'
'Too many nc_groups in an nc_block_specification.'
'Optional output can only be specified for a modal nc_group.'
'Too many read_parameters in a read_statement.'
'Unknown operand in a read_statement.'
'Unknown file_identifier.'
'Unknown operand in a parameter_list.'
'Parameter of wrong type.'
'Incorrect number of parameters.'
'No decimal_field_length specified for type real.'
'Internal_file must be of type character.'
'Unknown operand.'
'Number of bounds must be less or equal 4.'
'Lower_bound must be less than upper_bound.'
'Array out of bounds.'
'Incorrect number of subscripts.'
'A nc_group cannot be an array.'
'Subscript must be of type integer.'
'Lower_bound must be less or equal upper_bound.'
```

### System errors

```
'Full operand_file.'
'Too many action_blocks in action_file.'
'Full minor_element_list.'
'Full cl_record_list.'
'Full nc_group_file.'
'Full register_file.'
'Full ext_file.'
'Full parameter_list.'
'Full action_file.'
```

# Appendix 4. List of Runtime error messages.

## Runtime errors

### Input errors

```
'Overflow of operand in cl_record.'
'Underflow of operand in cl_record.'
'Unknown cl_record in input.'
'Unknown postprocessor statement format.'
'Overflow of operand in postprocessor statement.'
'Underflow of operand in postprocessor statement.'
```

### Non-fatal runtime errors

```
'Operand overflow.'
'Operand underflow.'
'Reference of a not initialized operand.'
'Information printed in NCLIST beyond position 132.'
'Information printed into file longer than 128 characters.'
```

### Fatal runtime errors

```
'String too long.'
'Function argument out of range.'
'Integer overflow.'
'Integer underflow.'
'Real overflow.'
'Real underflow.'
'Square root of a negative number.'
'Division by zero.'
'File is not in write mode.'
'File is not in read mode.'
'Trying to read past end of line.'
'Trying to read past end of file.'
'Syntax error.'
'Array out of bounds.'
```