E I P

A-300

Assembly Language for E-300 and N-300 controllers

Novemberr 21st 2000

CH-1667 ENNEY - Téléphone 026 / 921 27 24 – 921 27 13 - Téléfax 026 / 921 32 87

Note: The chapters 7.2 and 7.3 are not implemented in version 1.00

1. INTRODUCTION TO THE E-300 ASSEMBLY LANGUAGE

The purpose of this manual is to provide the necessary information to program E-300 controllers. The hardware aspects relevant to the programmer are also covered.
From the point of view of the programmer, the E-300 is a set of 8 identical processing units sharing common storage areas and common physical devices. The Figure 1. shows the basic organization. Figure 1. is not representative of the actual electronic circuits, but merely a functional representation.

- The program interpreter has a time-sharing mechanism, which allows simultaneous execution of up to 8 different programs or tasks. An additional task is reserved to the communication link.
- A simultaneous program has its own set of 8 register banks, one for each procedure nesting level. A register bank has 6 registers, a program counter and status flags. Each register is able to hold a full definition numerical quantity or to serve as an index.
- The E-300 is basically a "three address" machine, for example, a single instruction will add the contents of SOURCE1 with the contents of SOURCE2 and store the sum into DESTINATION.
- The E-300 operates on numerical quantities, integer or floating-point, with a 32 bit precision and also on single bits or boolean elements.
- The E-300 language supports a full set of floating-point instructions, including transcendental functions.
- An unique "parametric procedure call" system allows the user to create its own set of instructions in order to obtain the best adequation of the language to its particular needs.
- The E-300 System comes with a FLASH-resident language interpreter. The assembly and debugging tools are running on an IBM PC or compatible.

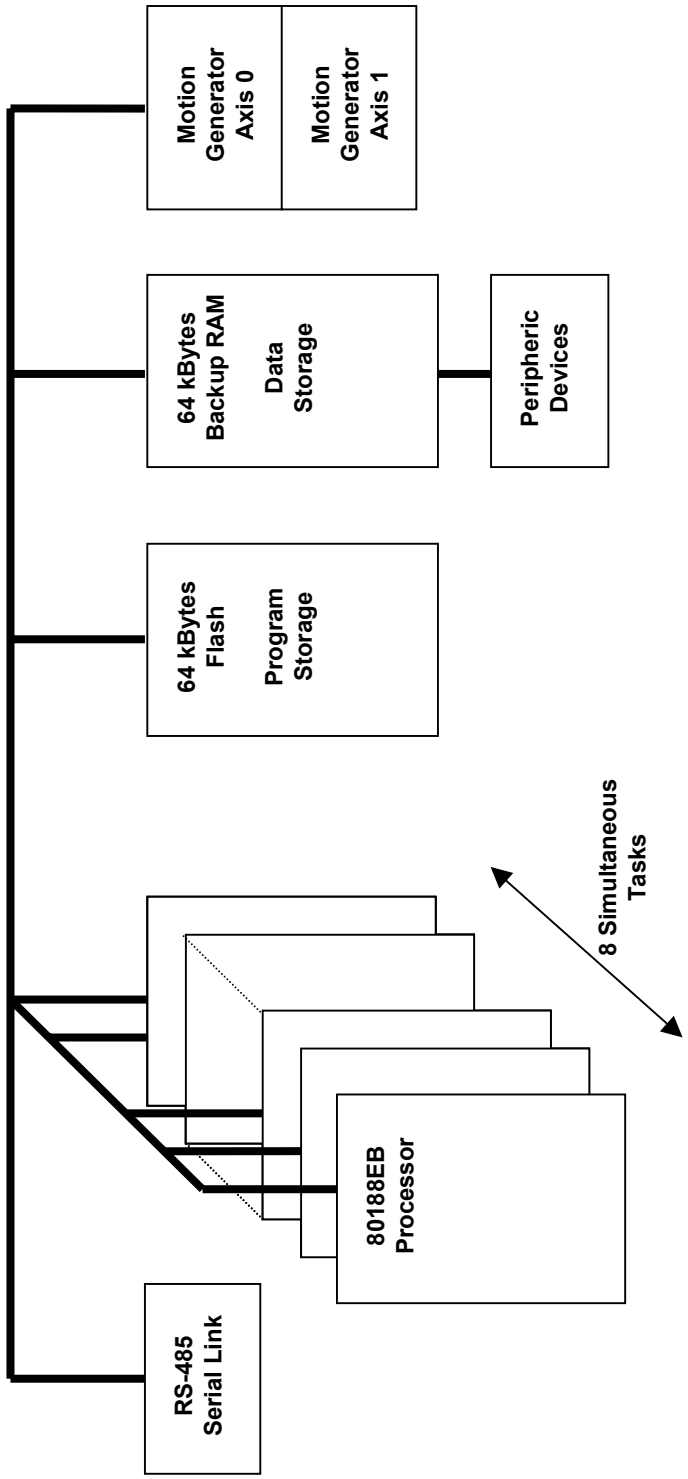

Figure 1: The E-300 Controller

2. ELEMENTS OF THE LANGUAGE

2.1. Legal Characters

Only the characters listed hereafter are legal within an E-300 program.
- Decimal Figures: 0, 1, .., 8, 9,

- Lower Case Letters: a, b, .., y, z,
- Upper Case Letters: A, B, .., Y, Z,
- Special Characters: + - . $ / % * \ ? # !

, : ; Space Carriage Return Tab EOF

"" " <> () = _ {} []

- Space and Tabs at the end of the line, Line Feed ($A), Form Feed ($C) and End Of File ($1A) characters are transparent to the assembler.

The meaning and the usage of the special characters will be described in the following sections.

2.2. Numbers

The E-300 assembler accepts decimal and hexadecimal integers and floating-point numbers given in decimal notation.
An INTEGER is a decimal number or an hexadecimal number in the range +-2^31, (-2'147'483'648 to +2'147'483'647).
An ORDINAL is a positive integer in the range 0 to (2^32)-1.
The assembler recognizes a decimal number as an integer if it is written without a decimal point. An hexadecimal number, **with the $-prefix**, must be a positive integer.

1 123 -4 -123456 $18AB
are integers.
It is also possible to give the ordinal value of one or several ASCII characters. The characters are enclosed in single or double quotation marks.
Examples:
'A' = $41
"AB" = $4142
'ABCDEF' = $43444546

Maximum capacity is 4 characters, or the last 4 of a long string.
A FLOATING-POINT NUMBER is a real decimal number with an absolute value ranging of $\pm 9.2^{E}18$. **The assembler recognizes a floating-point number if the decimal number is written with the decimal point or in scientific notation**.

3. 3.0 -3.1415 $1.294^{E}4$ $.45^{E}-5$

are floating point numbers.

Notes: a) $1.294^{E}4 = 1.294^{E}+4 = 12940.0$

b) Space characters are not allowed within a number.

c) Always use the decimal point, never the comma.

2.3. Symbols

A symbol is a name built in accordance with the following rules:

- it starts with a letter,
- the second and following characters can be letters or decimal figures,
- it contains neither special characters nor space (except ' _ ' underscore),
- the number of characters is limited to 32.

The assembler has values for the dedicated symbols (see appendix A), and assigns values to user defined symbols. The different methods of defining symbols are to be found in chapter 4.

2.4. Strings

A STRING is a sequence of ASCII characters and/or ordinals enclosed in <...>. ASCII characters must be designated by single or double quotation marks. The ordinals in a string are interpreted as the code of an ASCII character. The Carriage-Return and Line- Feed characters introduced through the keyboard are transparent in a string; if they are required, they must be represented by their numerical equivalent $D and $A.

<'abcd'> <65,"BCD",$D,$A,'EFG'> are strings.

2.5. Statements

Statements are "sentences" made from the above elements and written in such a way as to be understood by the assembler. Certain statements produce code sequences, i.e. program or data storage, other, merely direct the operation of the assembler.

A statement normally starts at the beginning of a line and terminates at the Carriage Return character. However, a statement may be broken into several lines if a comma is used at the end of each line but not at the last one.

FADD RA, RB, RC is a statement,

CALL PROC PAR1, PAR2, PAR3 can be written as

```
CALL PROC   PAR1,
            PAR2,
            PAR3
```

2.6. Comments

A COMMENT is fully transparent to the assembler. Thus, a comment can be written after a statement without affecting the syntax. All ASCII characters are allowed in comments.
A single line comment starts with a semi-colon (;) and terminates with a Carriage Return.
A bloc of comments is a text enclosed in braces {....} .

3. CODE GENERATING ELEMENTS

The purpose of this chapter is to present the language elements generating **executable code**. The reservation of data storage space with or without significant contents is dealt with in chapter 4.
Instruction mnemonics are a trivial case of a code generating items. Most instructions need one or several operands -or arguments- to specify their actions. Unlike most assembly languages, the E-300 language uses a specific prefix code for each different kind of arguments. This prefix will be called the **type** of that particular argument. For example, a two-byte integer as an immediate argument has not the same prefix as a variable defined to hold a two-byte integer. The type prefix allows the design of an instruction set where a given instruction can operate on many different items while retaining formally the same syntax. The prefix code is automatically generated by the assembler.
The E-300 assembler makes the difference between lower case and upper case letters. All the mnemonics and assembler directives must be written with upper case letters.

3.1. Instruction

The code for an instruction mnemonic is one byte in length. The most significant bit is always 1.

3.2. Immediate Arguments

An immediate argument is an operand whose value is written directly into the instruction. Immediate arguments can be numbers or strings. The table 3.2 gives all possible immediate arguments.
Comments to Table 3.2:

- The code of an immediate string has a 1-byte string length just after the prefix.
- The integers 0 and 1 can be used as immediate boolean elements in relevant instructions (except in complementing instructions). They are coded by the assembler as "short immediate integers".

Table 3.2. Immediate Arguments

TYPE	RANGE
Short Immediate Integer	0..7
1-byte Immed. Positive Integer	0..255
2-byte Immed. Positive Integer	0..65535
3-byte Immed. Positive Integer	0..16'777'215
4-byte Immed. Positive Integer	0.. 2'147'483'647
1-byte Immed. Negative Integer	-1..-256
2-byte Immed. Negative Integer	-1..-65536
3-byte Immed. Negative Integer	-1..-16'777'216
4-byte Immed. Negative Integer	-1..-2'147'483'648
Immed. Pos. Floating-Point Number	0....9.2^{E}+18 approx.
Immed. Neg. Floating-Point Number	-9.2^{E}18..-2.7^{E}-20 approx.
Immediate String	250 characters

3.3. Variables as Arguments

A variable is a fixed area within the storage space (or within the physical environment) able to hold one or several items of the same type. According to the nature of the contents of a variable, we shall speak about "numerical variables", "boolean variables" and "string variables". When a variable holds more than one item, it becomes an "**array**".
When an instruction has a variable as argument, it operates on the **contents** of that variable.
A variable is fully defined if the assembler has its **address** (the physical address of the first byte of the area) and its **type**. The programmer has to use a symbolic name for each variable and the syntax of chapter 4 to define the type and the size of the array. Whenever a variable is invoked by an instruction, its code prefix will tell the instruction how to fetch and how to store its contents.
According to its nature or according to the intended use by the programmer, a variable can be stored in different areas:

- a **FLASH-Variable** resides in the FLASH MEMORY user segment,
- a **RAM-Variable** resides in the RAM user segment,
- a **SYSTEM-Variable** resides in the RAM system segment.

(Refer to Figure 1.)
See the table 3.3 for a list of the available variables with their generic names. These names are recognized by the assembler; the assembler assigns the address 0 to these variables. With a suitable offset or index (see hereafter), these variables can access any storage location, see also section 4.4.

Table 3.3. Variables

FLASH	RAM	SYSTEM	CONTENTS	RANGE of CONTENTS
VO1F	VO1R	VO1S	1-bytes Ordinal	0..255
VO2F	VO2R	VO2S	2-bytes Ordinal	0..65535
VO3F	VO3R	VO3S	3-bytes Ordinal	0..16'777'215
VO4F	VO4R	VO4S	4-bytes Ordinal	0..2'147'483'647
VI1F	VI1R	VI1S	1-bytes Integer	-128..+127
VI2F	VI2R	VI2S	2-bytes Integer	-32'768..+32'767
VI3F	VI3R	VI3S	3-bytes Integer	-8'388'608..+8'388'607
VI4F	VI4R	VI4S	4-bytes Integer	-2'147'483'648..+2'147'483'647
VFF	VFR	VFS	Floating-Point Number (4-bytes)	approx. $-9.2^{E}18..+9.2^{E}18$, smallest number is $2.7^{E}-18$
1)	1)	VBS	Boolean Element	0 or 1
1)	1)	/VBS 2)	Inverted Boolean	1 or 0
VSTRF	VSTRR	VSTRS	String	250 characters max

1) Flash and Ram variables are dedicated variables; no generic name will work.
2) An inverted boolean variable has a slash in front of its name (/ OR \). When used as a source argument, the contents of the slashed variable is inverted before the operation; as a destination argument, the result is inverted before storage.

Tables of all dedicated variables are to be found in Appendix A. Dedicated variables have special functions within the E-300. The assembler already has definitions several dedicated variables, for other, an included definition file with equate statements must be part of the assembly.
In order to address an item within an array, an **offset** is appended to the variable name. The offset must be an integer (positive or negative) and can be regarded as the order number of that item in the array. The offset is enclosed in parentheses (..) or in brackets [..]. The offset is not applicable to strings.
In the examples which follow, the mnemonic VAR is used to designate a variable regardless of its type (except strings).

Examples:
- VAR1 points to the item nb 0 in the array VAR1
- VAR1(6) points to the item nb 6 in the array VAR1.
- VAR1(-15) points to the 15^{th} item before the array VAR1.

In some special cases, the programmer does not want an offset according to the type of the variable but rather an offset in bytes. This will allow dismantling of numbers and strings or access to peripherals not directly supported by the language. The syntax is as follows:

- VAR1(BP 3) points to the byte nb 3 in the array.
- VAR1(2+BP 3) points to the byte nb 3 of the item nb 2.

BP stands for "Byte Pointer". For single byte variables, BP can be omitted.
The code for a variable is made of the type prefix and a two-byte address. The assembler evaluates the contents of the parentheses and adds that value to the address of the variable.

3.4. Registers as Direct Arguments

The general purpose registers mentioned in chapter 1. are very often used as direct arguments in instructions. Remember that each subroutine nesting level and each simultaneous program has its own set of 6 registers, RA to RF.
Unlike other argument types, the coding of register arguments does not carry information about the nature of the contents. When an instruction loads a register with an integer shorter than 4 bytes, it

extends the number to 4 bytes with correct sign propagation. When storing a register contents into a variable with a short type, truncation eventually occurs.

3.5. The "Program Number" as a Direct Argument

The Program Number, **PNB**, is a special variable which holds the ordinal 0 to 7 according to the simultaneous program being executed. The PNB allows the programmer to direct a procedure to operate on different items in different simultaneous programs. In other words, using PNB as index promotes the writing of re-entrant routines.

3.6. Dummy Arguments

Dummy arguments are used within a procedure to reference parameters appended to the procedure call. This matter will be discussed in length in section 6.5. Like a variable array, the dummy arguments belong to an ordered set and they must be written with an offset. 256 dummy arguments are available.
Syntax:

% or %(0) %(5) %(15) 255 is max. offset.

Let us mention here the "Phantom Parameter", a matter related to the procedure call. When an asterisk is written in lieu of a parameter in a procedure call, the instructions of the procedure referencing this parameter are executed as No-Operation, see also section 6.5.

3.7. Indexing

Most variables -numerical, boolean or string variables- form ordered sets of items or arrays. The programmer will usually attempt to use the same program loop or the same procedure to operate on different items in the set. For example, a motion procedure can be used with any one of the 4 axes by merely changing the contents of an index.
All variables and the dummy argument can be indexed.
The following elements can be used as index:

- The registers RA to RF,
- The integer and ordinal type variables without offset or index, i.e. indexing can not be nested.

The registers RA to RF themself do not form an indexable set.

Syntax examples:
```
VAR1(RA)
VAR2(5+RB)
VAR3(3+5+RA+RB)
VAR4(RA-4)
%(2+RF)
%(PNB)
VAR1(VAR5)
VAR1(VAR5+2)
VAR1(VAR5+RA+1)
```

For the third example, the assembler computes the offset 3+5 = 8, modifies the address according to the type of VAR3 and then appends the codes for the index RA and for the index RB. During execution of the program, the interpreter will add the contents of RA and RB and modify once more the address according to the type of VAR3. If VAR3 is of the "3-byte integer" type, the indexing step is 3 bytes. If the variable is of the "string" type there is no fixed step expressed in bytes , but the indexing mechanism steps through the strings. The result of the expression contained in the parentheses can be negative, but the minus sign can't be written in front of a register or a variable name:

Wrong syntax:
```
VAR1(5-RA)
VAR2(5-VAR5)
VAR3(VAR5(RA))
VAR4(VAR5(3))
```

As is the case with the offset, it may be advisable to index directly in bytes. The type of the indexed variable is then immaterial.

Syntax examples:
```
VAR3(BP VAR1)
VAR4(BP RA+BP RB)
VAR5(BP 4+BP RC)
VAR6(3+BP RA+RD)
```

The last example shows that mixing the two indexing methods is legal.

3.8. Labels and Addresses

A LABEL is a symbol (max. 32 characters), whose value is an address within an executable program. Labels are the arguments of program control instructions. The assembler codes a label argument as a 2-byte immediate positive integer.

When an instruction needs the ADDRESS of a variable rather than the variable itself, the programmer has to prefix the variable name with a # . The assembled code for such a parameter is again a 2-byte immediate positive integer.

Examples:

```
MOV RA,VAR          The contents of VAR is moved to RA
MOV RA, #VAR(4)     The address of VAR(4) is moved to RA
```

The character # is legal in front of a variable with offset, but it cannot be assembled with an indexed variable.

Wrong use of #:
```
MOV RA, #VAR(RB)
```
The character # is not legal with boolean variables.

4. SYMBOL DEFINITION, VARIABLE DEFINITION AND ASSEMBLY DIRECTIVES

This chapter deals with syntax elements and statements which do not generate executable code, but provide the assembler with user symbol definitions and direct its operation.

4.1. Definition of Labels.

As already mentioned, a LABEL is a symbolic address within the executable code. Symbolic addresses referencing the data storage space are "variables" and we know from chapter 3 that their coding is different.
A symbol is defined as a label if
- it has a colon (:) appended,
- it is not followed by a "definition directive", see next section.

While generating the code, the assembler holds a location counter -or byte counter- and when a label definition is encountered, it uses the contents of the location counter to assign a value to the label.
Syntax examples:

```
LABEL:      AND  DEST, SRC
....        .....
START:      JMP  LABEL
```

If there is no instruction between two labels, they will have the same value.

4.2. Variable Definition

To define a variable, the programmer has to provide the assembler with
- the name of the variable,
- the address of the variable,
- the type,
- the size of the array,
- optionally, its initial contents.

The name and the address are given by the colon definition as in section 4.1.
The "definition directives" provide the information about what the variable is intended to hold. The table 4.2 lists all definition directives, see also the table 3.3.
Boolean variables are not defined by a directive. All the boolean variables are dedicated variables already known by the assembler. However, the programmer can rename the boolean variables by equivalence statements, section 4.4.
For syntax examples, see the next section.

Table 4.2: Variable Definition Directives

STORAGE AREA	CONTENTS of VARIABLE	DEFINITION DIRECTIVE
FLASH	1-byte Ordinal	DO1F
FLASH	2-byte Ordinal	DO2F
FLASH	3-byte Ordinal	DO3F
FLASH	4-byte Ordinal	DO4F
FLASH	1-byte Integer	DI1F
FLASH	2-byte Integer	DI2F
FLASH	3-byte Integer	DI3F
FLASH	4-byte Integer	DI4F
FLASH	Floating Point Number	DFF
FLASH	String	DSTRF
RAM	1-byte Ordinal	DO1R
RAM	2-byte Ordinal	DO2R
RAM	3-byte Ordinal	DO3R
RAM	4-byte Ordinal	DO4R
RAM	1-byte Integer	DI1R
RAM	2-byte Integer	DI2R
RAM	3-byte Integer	DI3R
RAM	4-byte Integer	DI4R
RAM	Floating Point Number	DFR
RAM	String	DSTRR

4.3. Initial Contents of Variables.

The programmer can instruct the assembler to assign contents to the variables. These contents are written into the storage area when the program is down loaded to the system. FLASH-located variables must have an initial contents specified, or they must be written by program during running time. For RAM variables, an initial contents may be useful for test purpose, but the program must have provision to fill the RAM variables with the appropriate values as part of the power-up sequence.
For RAM-located string variables, some sort of initial contents is always required to enable the assembler to compute the necessary storage allocation. A string variable is stored as a chain of ASCII characters preceded by one byte giving the length of the chain. Therefore, **the length of the chain must be set by the program at power up for RAM located string variables**.
The size of the variable array is implicitly given if initial contents are specified. To reserve an array where each position is initially filled with the same contents, the programmer can use the "Duplicate" directive,
n DUP. The DUP directive works properly for string variables if all strings in an array have the same length.
Initial contents are given by explicit numerical or string values. When a **quotation mark (? or !)** is used as an initial value, the storage locations are left unchanged while down loading the program.
Syntax Examples:

```
VAR1:  DO1R  ?          ; reserve space for one 1-byte ordinal in RAM
VAR2:  DFF   1.2, 2.34  ; array of 2 float. numbers, contents specified
VAR3:  DI4R  16 DUP ?   ; space for an array of 16 4-byte integers
```

```
TEXT1: DSTRF <"abcd">,
             <"pqrst", $A,$D,"uvwxyz">        ; 2 strings with specified text

TEXT2: DSTRF 6 DUP <"------------">           ; reserve space for 6 strings of 12
                                              ; characters in length
```

4.4. Generic Variables.

The assembler has a set of generic variables defined a address 0. They are intended to access any storage locations with a suitable offset or to define specific variables by equivalence statements. The table 4.4. lists the generic variable names.
Examples:

```
MOV   VI3R(BP $13), RA        ; deposits a 3-byte integer at RAM address 13 hex

MOV   RA, VO1F(BP 1000)       ; fetches a single byte ordinal from FLASH location 1000 ;
                              decimal

MOV   VFR(BP$8000+RB), RA; deposits a floating number at RAM address
                                    ; (8000 hex + 4.<RB>)
```

Table 4.4. The Generic Variables

NAME	TYPE	LOCATION
VO1F	1-byte ordinal	FLASH
VO2F	2-byte Ordinal	FLASH
VO3F	3-byte Ordinal	FLASH
VO4F	4-byte Ordinal	FLASH
VI1F	1-byte Integer	FLASH
VI2F	2-byte Integer	FLASH
VI3F	3-byte Integer	FLASH
VI4F	4-byte Integer	FLASH
VFF	Floating Number	FLASH
VSTRF	String Variable	FLASH
VO1R	1-byte ordinal	RAM
VO2R	2-byte Ordinal	RAM
VO3R	3-byte Ordinal	RAM
VO4R	4-byte Ordinal	RAM
VI1R	1-byte Integer	RAM
VI2R	2-byte Integer	RAM
VI3R	3-byte Integer	RAM
VI4R	4-byte Integer	RAM
VFR	Floating Number	RAM
VSTRR	String Variable	RAM
VO1S	1-byte Ordinal	SYSTEM
VO2S	2-byte Ordinal	SYSTEM
VO4S	4-byte Ordinal	SYSTEM
VI1S	1-byte Integer	SYSTEM
VI2S	2-byte Integer	SYSTEM
VI4S	4-byte Integer	SYSTEM
VFS	Floating Number	SYSTEM
VBS	Boolean Variable	SYSTEM
/VBS	Inverted Boolean Var.	SYSTEM

4.5. Equivalence Statements.

An equivalence statement has the general syntax:

SYMBOL = RM (RM = Right Member)

The assembler assigns the value of the right member to SYMBOL. This implies that:

- the RM is already defined or known by the assembler,
- SYMBOL will have the same type as RM.
- SYMBOL can be used in the program **after** this equivalence statement only.

All items of the E-300 assembly language can be the right member of an equivalence statement, numbers, strings, variables with offset, instruction mnemonics, or labels. **A right member with operational capabilities is not allowed**. RM can not be an instruction with arguments or an indexed variable.

Examples:

Statement	Type of Defined Symbol
NUM1 = 5	1-byte Integer
NUM2 = 3.1415	Floating Number
VAR2 = VAR1(5)	Same as VAR1
ADD = # VAR1(5)	2-byte Integer
START = IN(12)	Boolean
STOP = /IN(2)	Inverted boolean
TEXT = <"E-300">	String
OP = IADD	Instruction
VAR3 = VAR1(NUM1+3)	Same as VAR1, NUM1 is a number, not a variable
TEMP = VO3R(BP $1000)	3-byte Ordinal

4.6. The ORG Directive

The assembler has to know where a section of code -either a program segment or a data area- has to be physically stored into the E-300 system. The ORG (ORiGine) directive transfers the value of its argument to the location counter. The argument of ORG must be an ordinal number or an already defined symbol, whose value is an ordinal.
Example:

```
        ORG   $D00          ; set RAM location counter at $D00

VAR1:   DO4R  2 DUP ?       ; add 8 to RAM location counter
VAR2:   DFR   ?             ; add 4 to RAM location counter
                            ; now RAM loc. counter holds $D0C

        ORG   $4010         ; set ROM location counter at $4010

STRT:   MOV   RA, 10        ; add 4
        MOV   RB, $1FF      ; add 5 to ROM location counter
        ..................
        ..................

TEXT:   DSTRR <"-----">; add 6 to RAM location counter
        ..................
        ..................
```

4.7. The END Directive.

An assembly terminates properly only if the directive **END** is present in the last line of the program. A carriage return must terminate this line. The END directive can be used with an argument giving the auto-start address of the program. Without an argument, the program can only be started in the debugging mode.
Syntax: END label

4.8. The INCL and PATH Directives.

The **INCL**ude directive tells the assembler to get a source program file from the disk and assemble this file into the program just being assembled. This directive allows the partitioning of source programs for easy editing. Standard procedures may be introduced into a program by this directive.

The program being assembled and all the included files form a single program. Thus, all symbols must be different.
The included files must terminate with an END directive without argument. 64 files maximum can be included. INCL does not support nesting, i.e. the INCL directive is not allowed within an included file.
Syntax: INCL (Drive:\Path\FileName.Ext)

Note: In the parentheses, lower case letters are treated as upper case letters. (Max. 20 characters, spaces ignored).

Example: INCL (C: \UNIP \LIB \JOG.OVL)
INCL (C: \UNIP \LIB \POS.OVL)

The PATH Directive
The PATH directive causes specified directory to be searched for Included Files not found by a search of the current directory. Thus, it is not mandatory to specify the path in the INCL directive.
Syntax: PATH (Drive: \Path \ )
PATH must be written before the INCL directive.
Note: In the parentheses, maximum 20 characters are permitted. Default Path is the current directory.

Examples: PATH (C: \UNIP \LIB \ )
INCL (JOG.OVL)
INCL (POS.OVL)

4.9. The MODule Directive.

This directive is required for multi controller systems. If one controller is used, the programme must begin with the directive:

MOD 1

Important notice:
The directives ORG, END, INCL, PATH, MOD must be the sole item in a program line. A label in front of these directives is not allowed, an instruction or an other directive is not legal in the same line. Comments are allowed. A carriage- return character always terminates a directive.

4.10. The LISTON and LISTOFF directives.

Any source lines between LISTON and LISTOFF are assembled but masked on the screen and in the listing file (made with F6).
Example:

```
code or data lines a)
.
.
.
LISTOFF
.
.
code or data lines b)
.
.
.
LISTON
.
.
.
code or data lines c)
.
.
```

The lines a) and c) are visible, the lines b) are hidden.

5. E-300 MEMORY ORGANIZATION, DEBUG MODE

The INTEL 188 processor has an Harvard architecture, i.e. the program store and the data memory are physically distinct, they are selected by different strobes but they use the same address span: 0 to FFFF hex. The program memory is usually made of read-only chips and the data memory is made of read-write chips. The E-300 departs from this strict Harvard organization in order to provide a **Debugging Mode** and a **Program Bootstrap Zone**. The processor strobes and the chip selects sent to memory chips are made mode and address dependent by a combinatorial logic. The resulting organization is depicted in Figure 5.

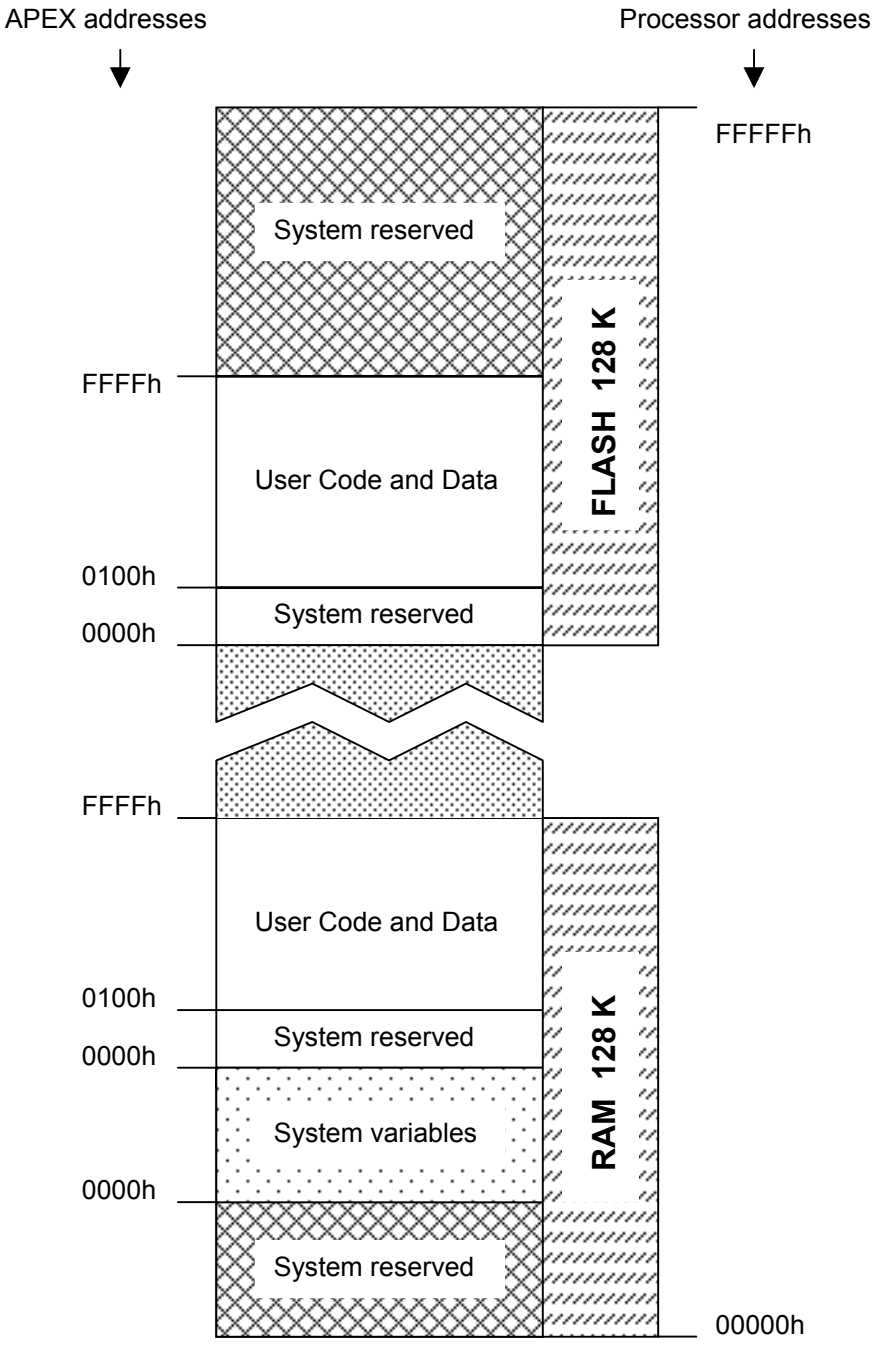

Figure 5.: E-300 Memory Organization

Comments about Figure 5.:

1. The **RUN Mode** is the normal operating mode of the E-300; the **Debug Mode** is introduced by the Power-Up Utility, see next section.
2. The program memory is made of a 128 kbytes FLASH. The upper half of the FLASH is used by the language interpreter and is normally not available to the E-300 programmer.
3. The data memory is made of a 128 kbyte CMOS RAM with battery back-up. The lower half of the RAM is used by the language interpreter and is not available for the programmer except the system variable segment.

5.1. The POWER-UP Utility

If the F10 key is depressed while switching the power on, a special hidden utility is entered. The display prompts for the selection of one of 4 functions (key F1 to F4). The screen also shows the version of the interpreter.

E-300 1.00 09.30.99
RUN DEB FPR BAUD

Note for N300: An *Initialization Key* is plugged on the J2 (input) connector during power-up. When using the Key, make sure that it has all bits are set correctly.

1) The **F1 (RUN)** key enters the **RUN Mode** menu. To start the program stored in the FLASH, depress the **F4 (GO)** key ("RUN MODE" is displayed). The start address of the E-300 programmes is 0 in the FLASH. (The ESCape key returns to the basic selection menu if depressed before F4.)
2) The **F2 (DEBUG)** key enters the **DEBugging Mode** menu.

   A normal debugging session begins with the depression of **F3 (HOLD)** and waits for data from the A-300 development tool ("DEBUG MODE HOLDING" is displayed):
   Loading a program from the Debugger, 'T' Transfer command:
   With this command, the object code can be downloaded into the RAM segment.
   Running a program from the Debugger, 'R' Run command
   A program in the RAM segment can be started with this command. The execution starts at the address offset loaded in the Run vector (located at address 0 RAM).
   If an already tested program resides in the RAM, it is possible to directly run the program again by depressing **F4 (GO)** instead of F3.
3) The **F3 (FPR)** key enters the flash programming utility. To start programming of the FLASH from the RAM depress **F4 (PROG)**.

   During programming (about 15 seconds) the message "Flashing page xxx" is displayed. The message "FLASH OK,press a key" terminates the programming.

The action of the Flash programming is illustrated in figure 5.1.

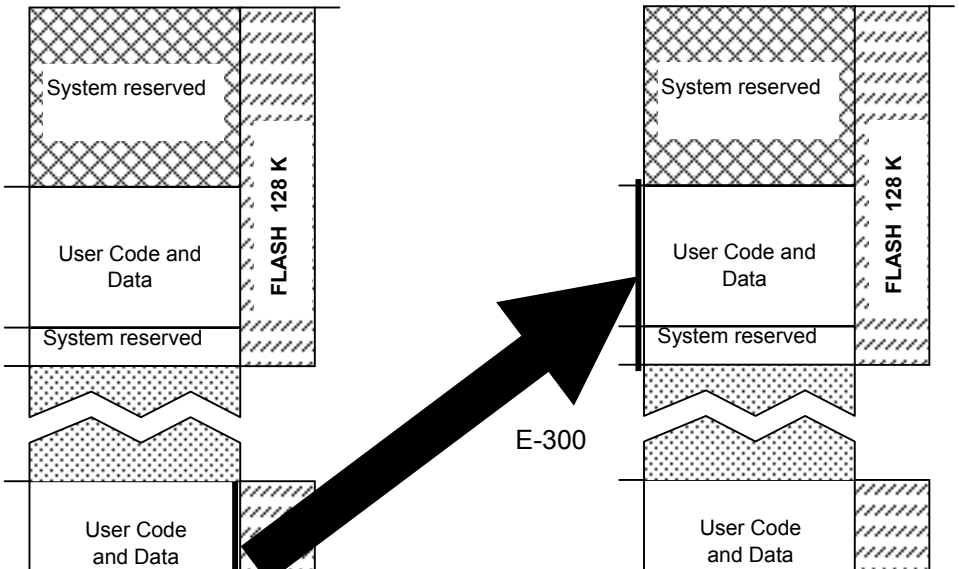

Figure 5.1. Flash Programming.

4) The **F4 (BAUD)** key enters the RS-485 baud rate selection. Press F1 to select 9600 bauds, F2 for 19200 bauds. At power-up, the default rate is 9600 bauds.

6. E-300 INSTRUCTIONS

An instruction statement is composed of the following elements:
- an optional label,
- an instruction mnemonic,
- one or several arguments separated by a comma.

The arguments can be **immediate values, variables, registers** or **dummy arguments** (See also section 6.5 for the phantom parameter utilization).
An instruction always operates on the immediate argument itself but on the **contents** of a variable or of a register. Remember that a label used as an argument is an immediate argument. The same holds for a #-prefixed variable.

6.1. Two-operand Arithmetic Instructions

This instruction class is basically a "three address" class: the operation is performed between the two source arguments and the result is sent to the destination argument. The instructions in this class can also be used with two arguments only: the operation is then performed between the destination and source arguments.
When two different instructions must be provided for an operation on integers and on floating-point numbers, the same mnemonics is used with a I- or a F- prefix.
Syntax a)

INSTR dest, src1, src2

Syntax b)

INSTR dest, src

where INSTR stands for the mnemonic of the instruction; dest, src1, src2, src are arguments.
Action of syntax a): <dest> = <src1> * <src2>

Action of syntax b): <dest> = <dest> * <src>

where * stands for the operator effected by INSTR, and <...> means "the contents of".
The instructions are listed below with reference to syntax a), going to syntax b) is a straightforward matter.
The contents of the source arguments remain unaltered.
Three flags are affected by the result of most arithmetic instructions, either of the two-operand or of the single operand class.
- The Zero Flag, **ZF**, is set if the result is zero,
- the Sign Flag, **SF**, is set if the result is negative,
- the Odd Flag, **ODF**, is set if the result is an odd number, Integer Addition:

IADD dest, src1 src2	<dest> = <src1> + <src2>

Integer Subtraction:

ISUB dest, src1 src2	<dest> = <src1> - <src2>

Integer Multiplication:

IMUL dest, src1, src2	<dest> = <src1> * <src2>

The addition, subtraction and multiplication are effected in accordance with the rules of the two's complement arithmetic. The arithmetic unit has a width of 4 bytes; the source operands are expanded according to their types; truncation of the result eventually occurs if required by the type of the destination.
Floating-Point Addition:

FADD dest, src1, src2	<dest> = <src1> + <src2>

Floating-Point Subtraction:

FSUB dest, src1, src2	<dest> = <src1> - <src2>

Floating-Point Multiplication:

FMUL dest, src1, src2	<dest> = <src1> * <src2>

Floating-Point Division:

FDIV dest, src1, src2	<dest> = <src1> / <src2>

The Overflow Flag, OVF, will be set if |<src2>| < 1.0E-5.
Integer Division:
(Invariable syntax)

IDIV dest1, dest2, src1, src2	<dest1> = int <src1> / <src2> <dest2> = <src1> modulo <src2>

The Overflow Flag, OVF, will be set if <src2>=0.
Logical Bit-by-Bit Multiplication (AND):

AND dest, src1, src2	<dest> = <src1> & <src2>

Logical Bit-by-Bit Addition (OR):

OR dest, src1, src2	<dest> = <src1> V <src2>

Logical Bit-by-Bit Modulo 2 Addition:

XOR dest, src1, src2	<dest> = <src1> + <src2>

Logic instructions make sense when operating on integer and boolean elements.

6.2. Single Operand Arithmetic Instructions

This class of instructions can be written with one or with two arguments. The syntax a) will save a subsequent MOV instruction in most situations.

Syntax a)

INSTR dest, src

Syntax b)

INSTR src

Action of syntax a): <src> is transferred to dest after alteration commanded by the instruction. The contents of src remains unaltered.

Action of syntax b): <src> is altered by the instruction and rewritten into src.

Unless indicated, the flags are set according to the result, see section 6.1.

Integer/Floating-Point Negation:

INEG dest, src **FNEG dest, src**	<dest> = - <src>

Integer/Floating-Point Absolute Value:

IABS dest, src **FABS dest, rc**	<dest> = |<src>|

Note: The Sign Flag is set according to the sign of <src>.
Logical Bit-by-Bit Complement of an Integer:

CPL dest, src	<dest> = /<src>

Integer Incrementation:

INC dest, src	<dest> = <src> + 1

Integer Decrementation:

DEC dest, src	<dest> = <src> - 1

Floating-Point Inversion:

INV dest, src	<dest> = 1/<src>

The Overflow Flag, OVF, will be set if |<src>| < 1.0E-5.
Floating-Point Square:

SQU dest, src	<dest> = $<src>^2$

Floating-Point Square Root:

SQR dest, src	<dest> = $<src>^{½}$

The Overflow Flag, OVF, will be set if <src> < 0.0.
Note:
The argument of a trigonometric function is supposed to be given in radian if the RAD flag is set. Otherwise, the argument is given in degrees. Automatic "first quadrant reduction" is provided.
Floating-Point Sine:

SIN dest, src	<dest> = sin<src>

Floating-Point Cosine:

COS dest, src	<dest> = cos<src>

Floating-Point Tangent:

TAN dest, src	<dest> = tan<src>

The Overflow Flag, OVF, will be set if <src>= 6π(2n+1)/2.
Floating-Point Arc-Tangent:

ATN dest, src	<dest> = Arctan<src>

Integer to Floating-Point Conversion:

CFLO dest, src

The integer in src is converted to a floating-point number and written to dest.
Floating-point to Integer Conversion:

CINT dest, src
CINR dest, src

The floating-point number in src is converted to an integer and written into dest. CINT effects a conversion with **truncate**, CINR uses a commercial **rounding** procedure.

6.3. Miscellaneous Instructions

Data Move:

MOV dest, src

The contents of src is copied to dest. <src> remains unaltered.
Data Exchange:

XCHG src1, src2

The contents of src1 and src2 are exchanged.
MOV and XCHG operate on integer, floating-point and boolean arguments. No arithmetic flags are affected by these instructions. Arguments may be of different type but some adjustments will be made:
Example:

```
MOV     FLG(0), 455     ; <FLG(0)> = 1, (FLG is a boolean variable)
```

Data Bloc Move:

XFER dest, src, element_nb

- element_nb = number of elements (size of the bloc), 0 <= element_nb <= 65535.

A bloc is copied from source to destination, the element type is given by the type of dest. If source and destination zones are overlapping, the source is partially destroyed and the destination contains the entire copy of the bloc. If not, the source is not altered.

Rotate an Integer:

ROT1	**dest, src, n**	n = -7..+7
ROT2	**dest, src, n**	n = -15..+15
ROT3	**dest, src, n**	n = -23..+23
ROT4	**dest, src, n**	n = -31..+31

The contents of src is rotated n bit-places and the result is written to dest. If <n> is positive, the shift is to the left, if <n> is negative, the shift is to the right. <n> can be any argument, not necessarily an immediate number.

ROT1 rotates the least significant byte only, recirculating the most significant bit into the least significant for positive n and the least significant bit into the most significant for negative n. ROT2, ROT3 and ROT4 rotate 2, 3 or 4 bytes respectively, see figure 6.3.

The flags are set according to the result. The Sign flag is set if the most significant bit after the rotation is 1.

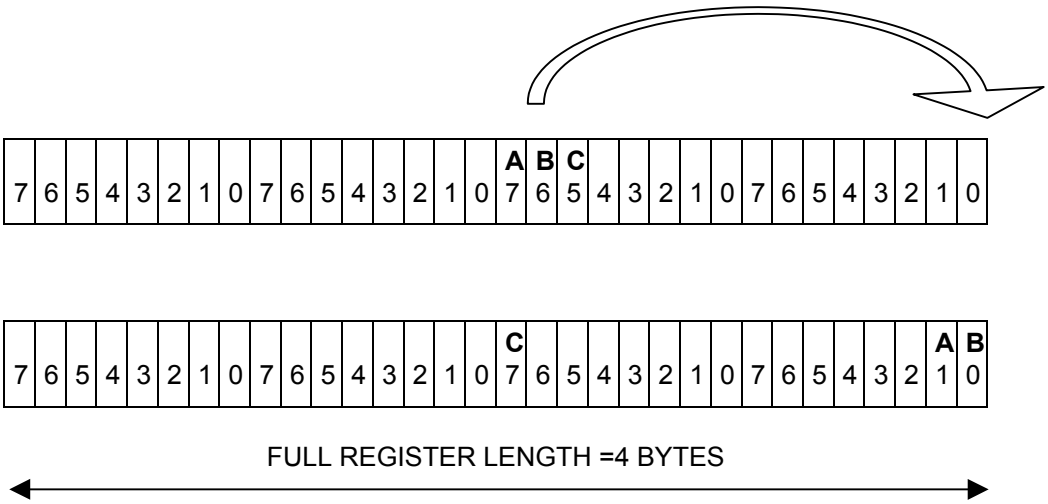

Figure 6.3. "ROT2 dest, src, 2"

A 2-argument syntax is also accepted:

ROTi	**src, n**

The rotated <src> is rewritten into src.
Number Comparison:

ICMP	**src1, src2**	For two's complement integers
FCMP	**src1, src2**	For floating-point numbers

The contents of src1 and src2 are not altered. The flags are set according to the result of the operation <src1> - <src2>:

<src1> = <src2> ZF = 1, SF = 0
<src1> < <src2> ZF = 0, SF = 1
<src1> > <src2> ZF = 0, SF = 0

The comparison instruction can also be used with three arguments, it becomes a "range comparator". ZF is set if <src1> is in the range <src2>...<src3>. The range **includes the boundaries**. See figure 6.3bis.

ICMP	**src1, src2, src3**
FCMP	**src1, src2, src3**

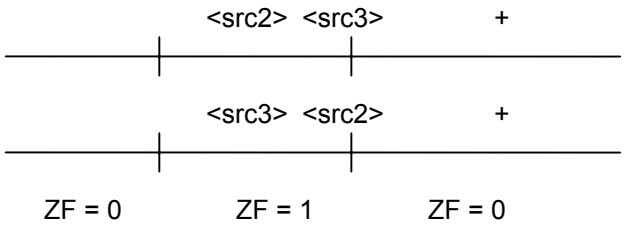

Figure 6.3bis: Range Comparison

String Comparison:

SCMP src1, src2

"src1" and "src2" are string type.

Function:

The ZF flag is set if both strings are indentical, character by character, and if the lengths are the same.

SET to One:

SET dest1, dest2, ... destn

RESET to Zero:

RES dest1, dest2, ... destn

These two instructions can have any number of destination arguments appended. The type of the variables can be mixed.

If the destinations are boolean variables, the action of SET and RES is straightforward. With integer or ordinal variables as destinations, SET sets the words to exactly 1, RES to all 0's. RES works properly with floating-point variables too. If a SET or a RES is followed by a variable declaration, an operation code (instruction) must be inserted.

Example:

```
            RES    a, b, c, SAF(PNB)
            NOP

VAR:        DO1R   ?
```

No flag is affected.

Caution:

A situation arises, where a RES instruction is not necessarily followed by an operation code: the termination of a simultaneous program by the instruction RES SAF(i). The interpreter may not be able to stop the decoding of the arguments. A NOP after the RES SAF(i) instruction will cure the problem.

Return the Address of a Variable:

ADDR dest, var

Action:

The address of "var" is returned to "dest". "var" can be any defined numerical variable, **with or without an offset, with or without an index expression**.

No Operation:

NOP	No action, can be used to trim timing loops.

6.4. Program Control Instructions

Unconditional Jump:

JMP	**label**	Direct jump to "label"
JMP	**var**	Indirect jump, i.e. jump to the contents of "var".

"label" must point to a location within the program area.
Upon execution of a JMP, the program execution is transferred to the location "label" or to the location <var>.
Example of indirect jump: The programmer wants to jump to several locations according to the contents of register RA.

```
            JMP    JMPTABLE(RA)
            NOP

JMPTABLE:   DO2F   LABEL0,
                   LABEL1,
                   LABEL2,
                   ..........,
                   LABELn
```

In this special case, it is allowed to define the variable JMPTABLE after its use by the indirect JMP instruction, despite the rule of section 4.2. This is made possible by the fact that a VO2 variable is assumed as the unknown argument of a jump.
The NOP instruction after the indirect JMP is required to allow unambiguous decoding of the argument. The decoding process assumes arguments until an op-code is encountered.
In the above example, it will be good practice to test the range of the RA contents before the jump.
Conditional Jump:
Conditional jumps allow branching upon the contents of one boolean variable or upon the combination of several variables. The instructions can be used in two different manners:

a)	**J(condition) label**	Direct Branch to "label"
	J(condition) var	Indirect Branch to the contents of "var"

b)	**J(condition) var2, label**	Direct Branch
	J(condition) var2, var1	Indirect Branch

With syntax a), the actual status of the flags (ZF, SF, ODF, OVF, NPAR, PHP) is tested. The form a) is normally used to test the result of arithmetic and logic operations using the flags left by the instruction. A conditional jump with syntax a) is required after execution of a comparison instruction.
With syntax b), the variable "var2" itself is tested. **The flags of the arithmetic unit, ZF, SF, ODF are not affected**.
The mnemonics accepted by the E-300 assembler are listed in the table 6.4. For convenience, several conditions have two names. For example, JE (Jump if Equal) is a synonym for JZ (Jump if Zero). JE will be used after a comparison instruction.
The syntax b) is not applicable to the instructions marked with an asterisk in the table 6.4.
The syntax a) is not applicable to the instructions marked with two asterisks in the table 6.4.
The conditional jump instructions, which imply a comparison -such as JE, JNE, JLT, JGT, JLE and JGE- refer to the result of instruction ICMP or FCMP, where <src1> is compared upon <src2>.
Examples:

```
a)      ICMP    RA, 100
        JLT     Smallra         ; Branch if <RA> < 100
        JGT     Largera         ; Branch if <RA> > 100
        .......                 ; <RA> = 100


b)      JZ      RF, Rfzero      ; Branch to "Rfzero" if
                                ; <RF> is Zero. The flags are not affected.
```

The instructions JT and JF are alternate mnemonics for JOD and JEV. They are used especially to test inputs, outputs and single- byte ordinal variables used as boolean variables. The test is made on the least significant bit of the variable, see Table 6.4.

Table 6.4. Conditional Jumps:

MNEMONICS	ACTION	BRANCH CONDITION
JE	Jump if Equal	ZF = 1
JNE	Jump if Not Equal	ZF = 0
JZ	Jump if Zero flag set (= JE)	ZF = 1
JNZ	Jump if No Zero flag (= JNE)	ZF = 0
JLT	Jump if Less Than	SF = 1
JM	Jump if Minus (= JLT)	SF = 1
JGT	Jump if Greater Than	/SF & /ZF = 1
JLE	Jump if Less or Equal	SF v ZF = 1
JGE	Jump if Greater or Equal	/SF v ZF = 1
JP	Jump if Positive or Zero (=JGE)	/SF v ZF = 1
JOD	Jump if ODd (var=...1)	ODF = 1
JEV	Jump if Even (var=...0)	ODF = 0
JT	Jump if True (= JOD)	ODF = 1
JF	Jump if False (= JEV)	ODF = 0
* JOV	Jump if Overflow	OVF = 1
* JNOV	Jump if No Overflow	OVF = 0
**JPAR	Jump if PARameter	NPAR = 0
**JNPAR	Jump if No PARameter	NPAR = 1
* JPH	Jump if PHantom Param.	PHP = 1
* JNPH	Jump if No PHantom Param.	PHP = 0

Conditional Execution

a) **IF(condition) instruction**

b) **IF(condition) var, instruction**

Action:
If "condition" is true, "Instruction" is executed, otherwise, "Instruction" is skipped. The syntax a) samples the arithmetic flags, the syntax b) acts on the status of var. The instruction which follows the condition **cannot be an IF**.
All the conditions of table 6.4. can be used, simply replace "J" by "IF".
Example: IFT bvar, RET ; Return if <bvar> is true.
If "instruction" is a procedure call, the mnemonic "CALL" must be used (see section 6.5).

Conditional Wait:

W(condition) var

Action:
The program does nothing as long as "condition" is true.
The Wait instruction can be considered as a short-hand notation for

Label: J(condition) var, Label.

Wait works with all conditions of table 6.4., replace "J" by "W", except for JGT, JLE, JOV, JNOV, JPAR, JNPAR, JPH and JNPH.

Simultaneous Program Activation:

ASIM i, label

Activate Simultaneous program nb i starting at label or <var>.

or **ASIM i, var**

Action:
The CPU nb i (Figure 1.) starts or restarts execution at "label" or at <var>. Level 0 is set in the register stack. Thus, ASIM i (or PNB) written in the program nb i can be used to escape from any level of procedure nesting and to branch anywhere in program i at level 0.

The concept of "simultaneous programs" is made self-explanatory by the figure 1. Simultaneous programs allow the programmer to effect several tasks at the same time without having to worry about the exact time relationship between the events occurring in the various tasks. They are also very useful to detect randomly occurring events, such as the depression of a STOP button. In effect, the concept of simultaneous programs is a very practical replacement for the interrupt system found in conventional processors. The E-300 controller is a true time-sharing system.
Operational Properties of Simultaneous Programs:
- A program can call a simultaneous program by issuing the instruction "ASIM i label".
- A program can momentary deactivate and reactivate a simultaneous program by resetting or setting the corresponding SAF(i) bit. A program can deactivate itself.
- A program can test the activity status of another program by testing the corresponding SAF(i) bit.
- A procedure can be indexed on the program number using the special variable PNB as an index.
- Number 0 is the default simultaneous program. At power-up, "ASIM 0,..." is executed.

It is not allowed to stop a simultaneous program by RES SAF while it is executing a number input instruction (see section 6.7.).
Suspension of the simultaneous program execution:

PSIM

no argument
Action:
All the simultaneous programs are paused with the exception of the program containing the PSIM instruction.

Reactivation of paused simultaneous programs:

RSIM	no argument

Action:
All the simultaneous programs paused by the instruction PSIM are reactivated. The execution of each simultaneous program resumes where it was interrupted.

6.5. Procedure Call

The E-300 language recognizes any defined label as the instruction mnemonics for a procedure call. The E-300 interpreter makes no difference between a procedure call and subroutine call. Thus, the term "procedure " is used in lieu et place of the term "subroutine". The mnemonics CALL is also recognized by the assembler.
Syntax:

(CALL) PROCNAME PAR0, PAR1, PAR2,.....

The parameters PARi are optional and their number is limited to 256.
Action: - Stack level = Stack level + 1
- The program counter of the calling level points to PAR0,
- The value of PROCNAME is put into the program counter of "Level + 1"
- The registers RA to RF of the calling level are copied into the registers of "Level + 1", but the flags are not copied.

This syntax is the basis for the generation of new instructions. Any procedure can be regarded as a new instruction and a set of well designed and documented procedures can be an application specific language.
The parameters become the arguments of the just created instruction. Within the procedure, an instruction referencing the first parameter uses the dummy argument %(0) as an argument; to reference the parameter nb i in the procedure call, %(i) must be used. The use of parametric calls and dummy arguments is shown in chapter 7.
A procedure call can have 256 parameters appended. Within the procedure, references to the parameters must be made by the dummy parameter array %(0) to %(255).
A procedure call within an "IF" statement **must** be written with the mnemonic "CALL".
The "No PARameter Flag, NPAR:
A procedure call can be made very flexible if the number of parameters can be left undefined. In order

to instruct the procedure that the parameters have been exhausted, the NPAR flag come to use. If the procedure call has parameters with ranking numbers from 0 to n, any reference to one of these parameters will left the NPAR flag cleared; reference to the parameter n+1 will set NPAR.

The Phantom Parameter:

It is sometimes useful to call a procedure (or instruction) while leaving one or several parameters unchanged, i.e. keeping for these parameters the value they were given before the procedure call. To effect this, the phantom parameter "*" is put in the call instead of a real parameter. Any instruction referencing the phantom parameter is a No-Operation except for the setting of the flags. If the phantom parameter is used as destination argument, the PHP flag is set and some other flags are affected, according to the instruction. The phantom parameter as a source argument has no practical meaning.

Example: IADD *, RA, RB ; The arithmetic flags are affected, the result is lost.

Return from a Procedure:

RET

Action:

RET decrements the level pointer and transfers the program control to the instruction just following the CALL instruction.

Return from a Procedure with Jump:

RETJ	**label**
RETJ	**var**

Action:

RETJ decrements the Level pointer, transferring the program control to the calling level at "label" or at <var>.

An indirect RETJ must be followed by an op-code, eventually a NOP must be inserted, see the note in section 6.4.

6.6. Motion Generator Instructions

Initialize Motion Generator:

IMG	**axis, displacement**

Action:

IMG initializes a motion generator for a motion. IMG sets the DIR(Axis) flag according to the sign of "displacement". This instruction does not start the motion. "axis" must be an ordinal and "displacement" must be an integer with a range of up to 4 bytes. See the contents of chapter 7 for more information about IMG.

Designate a Travel Limit Input:

The E-300 controller has 2 inputs per axis which can be designated as travel limit inputs **INA(x)** and **INB(x)**. If one of these inputs is designated as a travel limit by setting the **INAE(x)** or **INBE(x)** flag (**E**nable), the motion of the designated axis will stop immediately when an over-travel occurs. If the enable flags INAE or INBE are reset to 0, the inputs can be used as general purpose inputs.

A motion on the axis is abruptly stopped when the "INA(x) or INB(x)" input is activated. (The RUN flag is reset to zero, see chapter 7). The **INAI(x)** and **INBI(x)** flags (**I**nvert) are used to choose a normally closed or normally open switch. They must be set to 1 for a normally open switch and to 0 for a normally closed switch.

The **LSA(x)** flag is set to one if the motion has been stopped by this mechanism. The relevant LSA flags must be reset by the programmer in the initialization sequence and after the detection of an over-travel.

Example:

```
.
.
RES    LSA(1)
RES    INAI(1) ; Axis 1 normally closed
SET    INAE(1)
IMG    1, 100000
WF     ZSF(1)
SET    RUN(1)          ; Start the motion toward home switch
WT     RUN(1)          ; Wait end of motion
RES    LSA(1)          ; Switch encountered
MOV    SPLD(1), 1
IMG    1, -10000
WF     ZSF(1)
SET    RUN(1)
WT     RUN(1)
RES    INAE(1); Deactivate the detection
RES  PABS(1), LSA(1)  ; Reference made
.
.
```

6.7. Keyboard and Display Instructions

6.7.1. Keyboard Instructions

The E-300 motion controller has circuitry to read a matrix of max. 4 * 8 push-buttons (hereafter "keys") and to drive 8 LEDs.
The keys matrix is encoded as ASCII characters through a coding table provided by the programmer. The address of the coding table must be loaded into the variable KXCOD. The table has 32 bytes (complete key matrix), the cross points not implemented can be filled-in with any character, such as Space.
Coding Table used in the E-300 Controller:

```
KEYT:  DO1F  F10,   F9,    F8,    F7,     F6,     F5,     SWR,    F2,
             '7',   '4',   '1',   '-',    INS,    ESC,    SWG,    F1,
             '8',   '5',   '2',   '0',    CLR,    RTA,    UPA,    F4,
             '9',   '6',   '3',   '.',    ENT,    DNA,    LFA,    F3,
```

```
CLR   =    08H
ENT   =    0DH              ; ENTer
UPA   =    80H              ; UP Arrow
DNA   =    81H              ; DowN Arrow
F1    =    82H
F2    =    83H
F3    =    84H
F4    =    85H
F5    =    86H
F6    =    87H
F7    =    88H
F8    =    89H
F9    =    8AH
F10   =    8BH
LFA   =    8CH              ; LeFt Arrow
RTA   =    8DH              ; RighT Arrow
INST  =    8EH              ; INSerT (INS)
SWR   =    90H              ; SWitch Red   (STOP  button)
SWG   =    91H              ; SWitch Green (START button)
ESC   =    0C0H             ; ESCape
```

The key designators refer to Figure 6.7.1. and they must be defined as ASCII characters by equivalence statements:

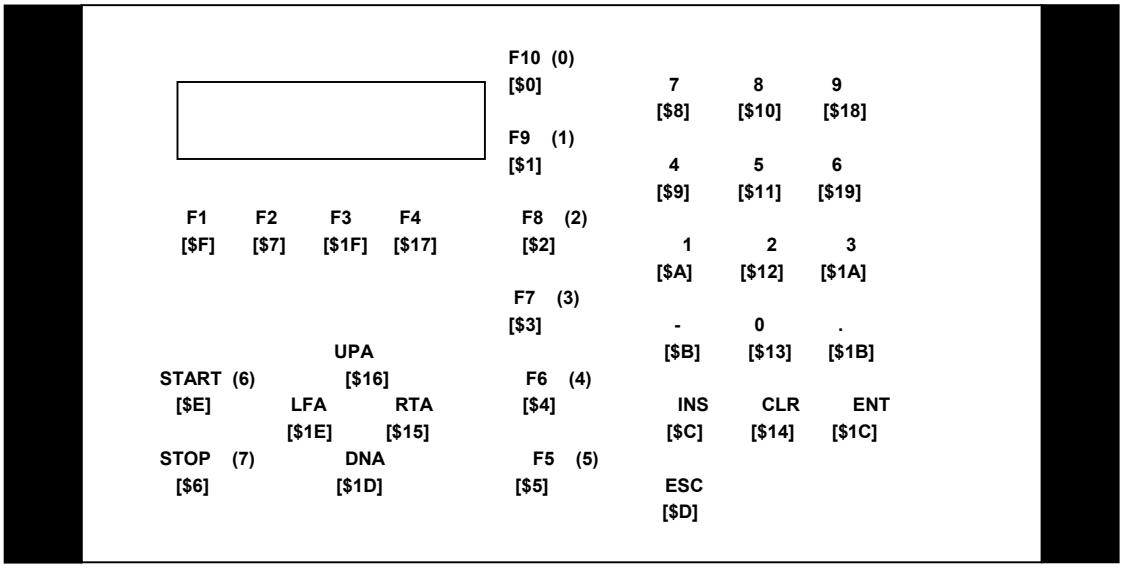

Figure 6.7.1: E-300, Key Designations (j) = LED offset assignement
[$i] = KEY offset assignement

The above coding table is the default one for the E-300.
To activate another coding table, the instruction **MOV KXCOD, #KEYT** must be executed prior to any keyboard instruction.
The code of the last depressed key is to be found in the variable KCOD(0)
The function of the keyboard input instructions strongly depends upon the **Key Available Flag, KAF**. KAF is set by a key depression. KINP instructions may operate in different simultaneous programs at the same time. Therefore, each simultaneous program has its own KAF.
An input instruction waits for KAF = 1 (the KAF belonging to its simultaneous program) and resets it after writing the input value to the destination. If KAF was set at instruction entry, the input instruction immediatly picks the character from KCOD(0) and resets KAF.
Note: The simultaneous programs cannot access the keyboard or the display at the same time.
Character Input Instruction:

KINP dest

Action:
The program waits for KAF = 1, then copies the code of the depressed key from KCOD(0) to "dest" and resets KAF (Key Available Flag).

Integer input:

IINP dest (,n)

Action:
If successful, the number typed at the keyboard is stored into dest. The optional argument "n" sets the maximum number of figures to be entered. The key depressions are echoed to the display. When the nth figure is typed, a carriage return character is sent automatically to the display and terminates the instruction. The number is stored into dest. When not specified, the maximum number of figures is 38.
The following ASCII characters are accepted:
'0' to '9' , '+' , '-' , $08 (CLR) , $0D (ENT)
The following characters are ignored:
$0 to $7F .
The following characters terminate the instruction, the content of dest is not modified:
$80 to $FF , $08 (BS) if this character has been typed as first character or if all the figures entered before have been deleted by $08, $0D if typed as first character.
Floating input:

FINP dest(,n)

Action: Similar to IINP, but the instruction accepts the decimal point and the scientific notation.

Example:

$3.456^{E}3$

n is optional. See IINP instruction.
The following characters are accepted:
'0' to '9' , '.' , '-' , '+' , 'E' , $08, $0D
The following characters are ignored:
$0 to $7F.
The following characters terminate the instruction, the content of dest is not modified:
$80 to $FF , $08 (see IINP) , $0D if typed as first character.
Note: The character "E" is not part of the standard E-300 coding table; an modified coding table may be devised to incorporate ASCII "E".

String input:

STINP dest

STINP accepts all characters. If the entered chain is longer than the length of the destination variable, truncation occurs; if the chain is shorter, a null ($0) character is appended to the chain. Maximum number of characters is 38 before automatic carriage return.
The following characters terminate the instruction, the destination will contain the characters $0:
$08 (see IINP) , $0D if typed as first character.
After an input instruction, the variable KCOD(0) contains one of the following codes:
$08 if the instruction has been terminated by $08,
$0A if only a carriage return has been typed,
$80 to $FF if the instruction has been terminated by a character from $80 to $FF,
$0D if the item has been successfully entered and stored into destination by typing carriage return or by automatic carriage return. This instruction is of very limited utility with the E-300 keyboard.
During execution of IINP, FINP, STINP, the character $08 (backspace) allows correction of the typed characters. The characters are echoed on the display. The carriage return character $0D is required to store the inputted items into the destination and to terminate the instruction.

6.7.2. The Keys as Boolean Variables

The keyboard can also be read as an array of boolean variables **KEY(i)**, where i belongs to (0..63). KEY is a dedicated name of the assembler and the offset i is the rank of the cross point in the coding matrix. For example, the key "F5" reads as KEY($5), "8" as KEY($10). Reading a key as a boolean element allows the programmer to sense the duration of the depression, a possibility not available with the instruction KINP.

6.7.3. The Keyboard LEDs

The LEDs illuminating several keys of the keyboard are controlled by the boolean array **LED(j)**, where j belongs to (0..7). The assignment for the E-300 keyboard is given in Figure 6.7.1. and in Table 6.7.3bis.

To get a blinking LED, add 8 to the offset of the LED-variable. The blink status overrides the on status according to the table 6.7.3.

Table 6.7.3. LED Status:

LED Status	LED(j)	LED(j+8)
off	0	0
blinking	0	1
on	1	0
blinking	1	1

Table 6.7.3bis.: E-300 Led Assignement

KEY	Boolean LED
F10	LED(0)
F9	LED(1)
F8	LED(2)
F7	LED(3)
F6	LED(4)
F5	LED(5)
START	LED(6)
STOP	LED(7)

Example:

```
SET LED(4)        ; LED of key F6 on
SET LED(4+8)      ; LED F6 is blinking
RES LED(4+8)      ; LED F6 on
RES LED(4)        ; LED F6 off
```

6.7.4. Display Instructions

The LC Display has 2 rows of 20 characters which are made of a 7 x 5 dots matrix :

00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39

Figure 6.7.4.: Character Positions of the Display.

The programmer writes the message to be displayed into one of two RAM areas, here termed "Windows". The display is refreshed automatically from the active window. The display status is controlled by "printing" special characters, see table 6.7.4.
The Figure 6.7.4bis illustrates the way the windows and the display operate. The switch at left designates the destination of the display instructions; the switch at right selects the data to be

effectively displayed.

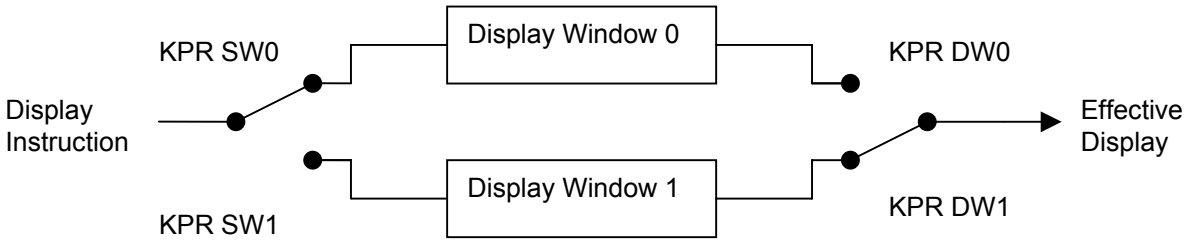

Figure 6.7.4bis: The Window Selection, Instruction writes to window 0, display from window 1.

Note: In this section, "print" means "write to the display".
Integer print:

IPR src

Action:
The contents of **PMI** direct the printing. The PMI belonging to the current simultaneous program must be used; it is good practice to index PMI with the simultaneous program number, i.e. using PMI(PNB).

<PMI>= 0 left justification of the number, any number of digit.

<PMI> n > 0 n digit saturation, right justified. The sign is not included in the digit number and is printed at the cursor position.

<PMI> n < 0 for positive numbers only. n digit saturation, right justified, the free characters before the number are zeros "0". No sign.

Examples:

<PMI> = 0

```
 - 1
 - 1 0
 1 0 0
 1 2 3 4
^ cursor position
```

<PMI> = 4

```
-         1
-       1 0
  1 0 0 0
  9 9 9 9
^ cursor position
```

Saturation ! number >= 9999

<PMI> = -4

0 0 0 1
0 0 1 0
^ cursor position

Floating print:

FPR src

Action:
The contents of **PMF0** and **PMF1** govern the printing. As exposed with PMI, the status variables PMF0 and PMF1 are usually indexed with PNB: PMF0(PNB), PMF1(PNB).

<PMF0> = 0 and <PMF1> = 0 : scientific notation with 7 significant digits.

<PMF0> = 0 and <PMF1> = n > 0 : notation with n digits with floating decimal point, too short numbers are followed by zeros "0".

<PMF0> = n and <PMF1> = m : n digits at the left of the decimal point and m digits after the decimal point.

The sign is written just before the number field. It is not included in the digit count.
Examples:
The number to be printed is $9.2^{E}18$ (max range)

<PMF0> = 0 and <PMF1> = 0 : $9.2^{E}18$

<PMF0> = 2 and <PMF1> = 3 : 99.999 saturation!

The number to be printed is 12,345 :

<PMF0> = 0 and <PMF1> = 6 : 12.3450

^ cursor position

Character print:

KPR src1,src2,..,srcn

Action:
All characters from srci are written to the selected display window, src1 first. Table 6.7.4 shows character codes for the LCD.

String Print Instruction:

STPR src

Action: src is written to the selected display window; src is either an immediate string or a string variable.
The string may be made of text and control characters.
The character 0 (hexa) marks the end of a string prior its length, Thus, we cannot have a 0 in a string:

STPR <DP, 0, 'abcdef"> is not allowed

Table 6.7.4.: Control Characters (to be used in KPR and STPR instructions):

SW0	= $0	Select Window 0 to write (Default value)
SW1	= $1	Select Window 1 to write
DW0	= $4	Select Window 0 to dislay (default value)
DW1	= $5	Select Window 1 to display
DP,n	= $10,n Display	Position at n. The cursor is set at character position n. n = 0..39
CURSON	= $13	Cursor ON
CURSOFF	= $14	Cursor OFF
CLN, n	= $1C	Clear n characters from cursor position
CLEOL	= $1D	Clear End of Line. The characters from cursor to the end of the

		line are deleted, the cursor doesn't move.
CLEOS	= $1E	Clear End Of Screen. The characters from cursor to the end of the screen are deleted, the cursor doesn't move.
CLS	= $1F	Clear Screen. The screen is entirely cleared, the cursor position is set to 0 (upper left corner), the cursor is on. At power on, the screen is clear.

The symbols of table 6.7.4 are given for convenience only; they are not recognized by the assembler. It is a good practice to include their definitions into a keypad and LED definition file. This file is then included in all E-300 programs.

Table 6.7.4bis. Display Characters

	0	1	2	3	4	5	6	7	D	E	F
0	SW0	DP	SP	0	@	P	`	p		α	
1	SW1		!	1	A	Q	a	q			
2			"	2	B	R	b	r		β	υ
3		CURSON	#	3	C	S	c	s		ε	:
4	DW0	CURSOFF	$	4	D	T	d	t		μ	ς
5	DW1		%	5	E	U	e	u		σ	ü
6			&	6	F	V	f	v		ρ	Σ
7			'	7	G	W	g	w			π
8	BS		(	8	H	X	h	x		√	
9			)	9	I	Y	i	y			
A	LF		*	:	J	Z	j	z			
B			+	;	K	[	k	{			
C		CLN	,	<	L	□	l	|		Φ	
D	CR	CLEOL	-	=	M	]	m	}			4
E		CLEOS	.	>	N	^	n	→		ñ	
F		CLS	/	?	O	_	o	←	°	ö	▮

To read the table: The rows are the lower hexadecimal digit, the columns are the upper hexadecimal digit of the ASCII character, for example "A" = 41 hex

6.8. Input, Output, I/O Bus, AD and DA Converter Instructions, Temperature Sensor

The E-300 controller has 4 inputs available at the motor connectors: **INA**(0), **INB**(0), **INA**(1), **INB**(1). These inputs are dedicated travel limit inputs, section 6.6. If not used as limit switches, they are general purpose inputs. In addition the E-300 has 8 inputs **IN**(0) to **IN**(7) and the EIP I/O bus. The I/O extension bus of the E-300 controller can address 120 inputs implemented in remote modules. The dedicated symbols for these inputs are **IN**(8) to **IN**(127).

The I/O extension bus addresses also 120 general purpose outputs: **OUT**(8) to **OUT**(127). The E-300 controller has 8 outputs available at the I/O connector: **OUT**(0) to **OUT**(7); in other words, the compact E-300 has an Output Module built-in.

INA(x), INB(x), IN(x) and OUT(x) are boolean elements and as such can be used as argument in all relevant instructions: MOV, SET, RES, JT, JF, WT, WF.

Analog-to-Digital Converter

An 8 bit analog-to-digital converter is implemented. It is tied to the cursor of the potentiometer mounted on the operator's panel or to the Analog I/O connector (selectable). The conversion range is 0 to 5V or 0 to 10V (configurable), or, in numeric format, 0 to 255. The system variable **ADC** reflects the analog input value.

As default mode, the ADC selects the potentiometer input and the conversion result is used, as explained in section 7.1.3, to control the axis velocity.

Digital-to-Analog-Converter:

Writing a single byte ordinal to the system variable **DAC** loads the converter. DAC is a VO1S variable. The output, available at the Analog I/O connector has a range of 0 to 10 V.

Temperature Sensor:

The E-300 is equipped with a temperature sensor which is located on the mother board. The state of this sensor is readable via the **INIT** boolean variable. INIT comes to 1 if the temperature goes over

68°C. The **NTC** flag sets the "manual" or "automatic" temperature watching mode:
NTC=0: The programmer must watch **INIT** to do the appropriate sequence when it comes to **1**.

NTC=1: When the temperature comes over 68°C, the simultaneous programmes are stopped, the motors are stopped and the message **"Internal temperature over 68 degrees"** is displayed. The E-300 controller must be switched off to restart the programme.

6.9. FLASH Memory Programming

The FLASH can be read as RAM but writing needs a special procedure through the instruction:

PROG dest, src, element_nb

Action: As an XFER instruction, but the execution time is about 3 ms. It is advisable to compare source and destination after a "PROG" to detect FLASH failures.
Example:

```
                ORG  $E000
EEFLASH_VAR:        DFF   ?


                .
                .
                PROG        FLASH_VAR, RA, 1
                FCMP        FLASH_VAR, RA
                JNE         ERROR
                .
                .
```

6.10. ASCII Serial Reception

Line buffer:

To simplify "ASCII row" acquisition, a line buffer is introduced. This buffer has the following specifications:

- The buffer supports one line, composed of elements of different "types". An element pointer is used and each element holds a position number, the first is zero.
- The elements are separated by a comma, or one or several "space" ($20) or "tab" ($09). These are called separators. They are stored in the buffer but ignored while reading the elements.
- The end of the line is given by the $0D character.
- The first character in the buffer is the line length (the $0D is counted).
- The character $0A (line feed) is ignored at buffer entry, the same holds for all characters sent after a semi-colon ";" (comment).
- The legal characters are from $00 to $DF. $E0 to $FF are system reserved characters

A byte, **LNF**, is associated with the line buffer. LNF is different from zero when the buffer has recieved a valid line.

Initialization of the Line Buffer:

CLINP

No argument.

Action: - The internal input buffer is cleared,
- LNF is set to one.

This instruction is usually used once at the program start.

Buffer preparation:

LINP

No argument.

Action: - LNF byte is set to zero,
- the element pointer is set to zero,
- the line length is set to zero.

"LINP" doesn't wait for data input. A stored line is available when the LNF byte becomes one.

Reading a String Type Element:

STXINP dest (,src)

Action:

This instruction reads the element pointed to by the element pointer and puts it into "dest". The pointer is incremented. The element can be directly addressed by its position number, if it is specified in "src", see the example.

Reading an Integer Type Element:

IXINP dest (,src)

Action: as STXINP.

Reading a Floating Type Element:

FXINP dest (,src)

Action: as STXINP.

If the "type" of the element does not correspond with the instruction, the conversion is automatically made. If the element pointer is larger than the element number, the OVF flag is set. If the element is an empty element or doesn't exist, the instruction becomes a No-Operation instruction (NOP) and the OVF flag is set.

Line Acquisition Example:

Line send to the module over the serial link:

n POS **tab** X 100.00 , Y - 1.23 , ,, ; comment ($0D)

The line buffer contains:

n POS **tab** X 100.00 , Y - 1.23 , , , ($0D)
0 1 2 3 4 5 6 7 <- element pointer

line length = 40

Program example:

```
START:   CLINP
         .
         .
         .
         LINP
         WF   LNF                    ; Wait for line complete
         STXINP   INST          ; $0D received, read string
         STXINP   AXE           ; AXE = "X"
         FXINP    VALUE1             ; VALUE1 = 100.0 (floating)
         STXINP   AXE(1)             ; AXE(1) = "Y"
         IXINP    VALUE2             ; VALUE2 = 1 (integer)
         IXINP    RA, 2         ; RA unchanged
         FXINP    RB, 5         ; RB unchanged
         FXINP    RC, 6         ; RC unchanged
         FXINP    RD, 7         ; RD unchanged
         FXINP    RE, 8         ; RE unchanged , OVF = 1
         .
         .
         LINP
         WF .....
```

6.11. Serial ASCII Transmission

Integer "Print":

IXPR src

"src" is the source to be "printed"; it must be an ordinal, an immediate integer, or a numerical variable holding an integer.

Action:

The contents of "src" is outputed on the serial link as a decimal number. The output format is governed by the contents of the Print Mode variable PMI(8). See section 6.7.4.

Floating Number "Print":

FXPR src

Action:

As IXPR, but "src" must be a floating number and PMF0(8) and PMF1(8) govern the output format.

String "Print":

STXPR string **STXPR var**

An immediate string or the contents of a variable of the string type is outputed on the serial link. The printing terminates if a Null character is encountered in the string.

Single Characters "Print":

KXPR src1, src2, src3, ..., srcn

One or several ASCII characters are sent on the serial link.

Example: KXPR $0D, $0A

or KXPR $0D
KXPR $0A

or STXPR <$0D, $0A>

7. THE E-300 MOTION GENERATORS

The E-300 has 2 physical motion generators to allow up to 2 simultaneous point-to-point motions. A 3rd "soft" motion generator is intended to control the motion along the path when a set of axes is involved in an interpolation. Path control will be presented in section 7.3. The motion generators associated with physical axes are named "axis 0" and "axis 1" and the path is axis 2.
The 2 axes are intended to control step-motor translators.
The motion generator computes the velocity profile and outputs a pulse frequency proportional to the current velocity.

7.1. Variables Pertaining to Motion Generation

Important Notice: The parameters and variables introduced in this chapter are declared for the axes 0 and 1; most of them also exist for the path axis. For clarity we shall write, for example, "DIV" as a generic name instead of DIV(i).

7.1.1. PABS, Absolute Position Register

A proprietary circuit generates the step frequency. It works in close relationship with a processor interrupt routine, which loads the generator with the current velocity. As this loading occurs at regular time intervals, the value of the velocity may be seen as an elementary length segment. The elementary length segments are accumulated in a register, **the Absolute Position Register, PABS.** PABS continuously keeps track of the actual position of the axis. Each axis, of course, has its own PABS. The contents of PABS is a 4-byte **integer**, thus no rounding inaccuracies can corrupt the relationship between the motor position and the contents of PABS.

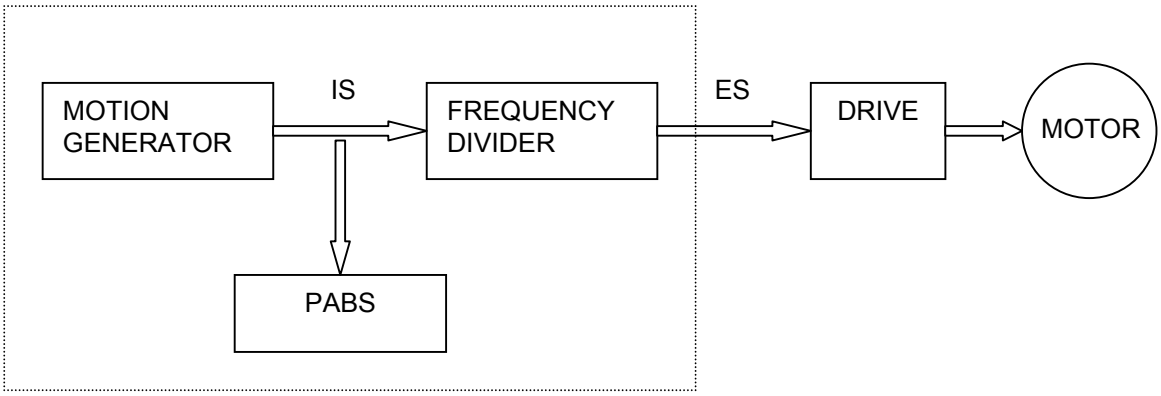

Figure 7.1.1. E-300 Motion Generator

7.1.2. DIV, Frequency Divider

PABS contains the coordinate expressed in **Internal Steps**, IS in Figure 7.1.1. In order to adapt the step frequency to various drive/motor combinations, a frequency divider is provided. After the division, the length unit is the **External Step**, ES. Thus, the value of parameter **NDIV** is defined as:

<DIV> = number of IS for 1 ES	(1)

The parameter NDIV sets the maximal obtainable external pulse frequency, fE_{max}, according to (1), fE_{max} in kHz:

fE_{max} = 11910/<NDIV> [kHz]

The width of the output pulse (for a step motor driver) is 0.021 * <NDIV> [μs].

7.1.3. The Steady State Velocity, SPLD, SPLI

During any motion, the plot of the velocity -or pulse frequency- versus time looks like the figure 7.1.3. The pulse frequency is proportional to the axis velocity for a positioning move; for a continuous path, the tangential velocity is significant.
Starting from the rest, the velocity grows, following an inverse exponential law; then it eventually reaches its programmed steady state value and finally decreases to zero with a similar law. The maximum pulse frequency, fE_{max}, is the asymptote of the acceleration and deceleration curves.
The inverse exponential acceleration law is the best choice to move an inertial load with a motor exhibiting a falling torque versus frequency characteristics, an assumption which holds roughly for most stepper motor/driver combinations.

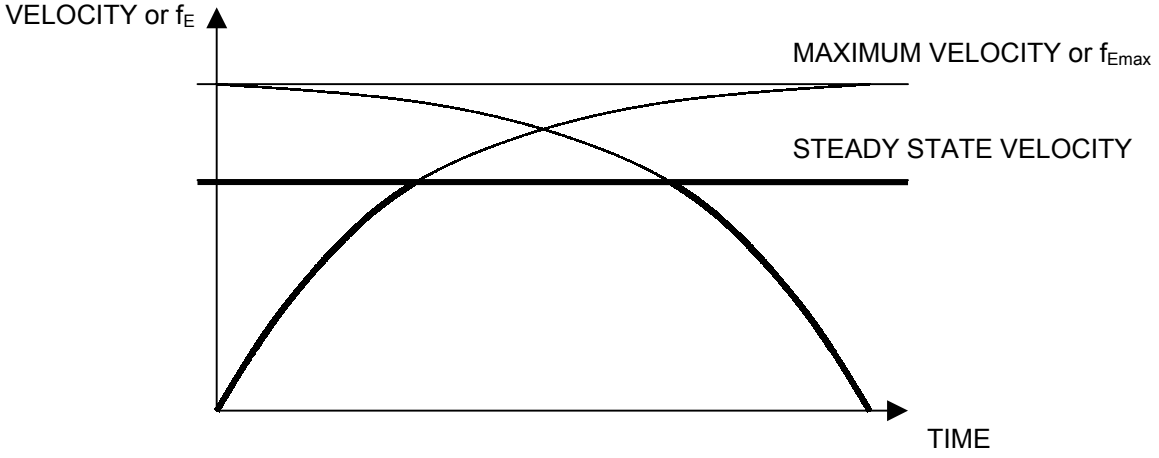

Figure 7.1.3. Velocity Profile

The steady state velocity is governed by two variables: **SPLD** (SPeed Limit Direct) and **SPLI** (SPeed Limit Indirect). SPLD has the range 0..32639 and SPLI points to a single byte ordinal number or variable. The steady state frequency is given by (2).

Steady State Frequency = 12'000 * [<SPLD> * <<SPLI>>/223]/<NDIV> [kHz] (2)**
= 1.43 * <SPLD> * <<SPLI>>/<NDIV> [Hz]

The steady state velocity may be changed at any instant, i.e. also during a motion. Any alteration of the velocity during a motion, either through SPLD or through SPLI is done according to the acceleration or deceleration laws. Thus, full range velocity steps are permitted.
The "feed overdrive" potentiometer, which allows the velocity to be set from 0 to 100% of the programmed speed is a typical example of the usage of the indirect speed control. The cursor of the potentiometer is fed to the analogue input and the variable SPLI points to the Analog-to-Digital Converter register.
The variable SPLI contains the address of a single byte variable located anywhere in the memory space. The contents of the addressed byte, regarded as a positive integer in the range 0..255, modifies the steady state velocity as per equation (2).
Special Values of SPLI:
- <SPLI> = #VMAXAD VMAXAD holds 255, the steady state velocity has its programmed value.
- <SPLI> = #VMINAD VMINAD holds 0, the motion stops with ramp.
- <SPLI> = #ADC the effective velocity is controlled by the potentiometer (through the analogue-to-digital converter).

In addition to the variables governing the steady state velocity, the **instantaneous velocity** is available to the programmer through **VACTN**. VACTN is a read-only variable, writing to VACTN can disturb the generation of the motion.

7.1.4. Acceleration and Deceleration, KUP and KDN

The initial slope in figure 7.1.3., i.e. the initial acceleration, is directly related to the contents of the variable **KUP**. The same holds for the final slope, which is proportional to **KDN**.
These two variables set the initial and final slopes according to (3).

$d(f_E)/dt$ = 67.055/<NDIV> * <KUP> [kHz/s] (acceleration) (3)
$d(f_E)/dt$ = - 67.055/<NDIV> * <KDN> [kHz/s] (deceleration)

7.1.5. Axis Status Flags: RUN, ZSF, NBOOST, BSTE, BSTI, FAULT, LSA

RUN has the value 1 during a motion, it resets at completion of the move. RUN(0) and RUN(1) refer to the individual axes, RUN(2) is the run flag of the path axis.
ZSF continuously monitors the zero status of the VACTN. This boolean variable is available for each individual axis and for the path axis. It is used whenever a motion is stopped by setting one of the variables SPLD or <SPLI> to zero. ZSF allows the program to wait for the end of the down ramp, see example below.
BST is an output associated with the physical axes 0 and 1. It is intended to alter the motor current. Its action within the motor driver may be an effective boost or a reduction of the current, depending upon the driver circuitry. The action of the BST outputs is conditioned by the status of **BSTE**, Boost Enable, and **BSTI**, Boost Invert.
If BSTE is set, the BST output is automatically asserted during a move of its axis. If BSTE is zero, BST can be used as a regular output. The voltage level of the asserted BST output at the driver connector can be inverted by setting BSTI true. BSTI is active only when BSTE is true.
Table 7.1.5: BST Logic

BSTE	BSTI	RUN	BST
0	X	X	Regular Output
1	0	0	0

1	0	1	1
1	1	0	1
1	1	1	0

The **FAULT** flag can be read by the programmer to know if the translator or the servo-amplifier is working properly. There is no automatic sampling of the FAULT input.
INAI, INAE, INBI, INBE. The function of these flags has been presented with the instruction IMG in section 6.6.

7.1.6. Example of Motion Programs

a) Simple motion of axis 0, displacement in engineering units in PHMOV. In this example, the variables NDIV, SPLD and SPLI are supposed to be already properly loaded.

```
                X =             0               ; axis name
                SCALEX =        160.34          ; scale factor, float. number
                                                ; external steps/length unit

PHMOV:          DFR             4  DUP  ?       ; variable definition

        ; first compute the internal scale factor as part of the initialization
                CFLO    RA, NDIV(X)                     ; NDIV is an ordinal
                FMUL    KMUL(X), RA, SCALEX ; KMUL is a dedicated variable
                ................

                FMUL    RA, PHMOV(X), KMUL(X)   ; displacement in IS units
                CINR    RA                      ; displacement as integer
                IMG     X, RA                   ; motion prepared
                WF      ZSF(X)                  ; wait for completion of a preceding
                                                ;       move
                SET     RUN(X)                  ; start
                WT      RUN(X)                  ; wait for end of move
                ..........
```

b) Jogging Motion in Positive Direction, control by key JOGK, Panel Potentiometer active.

```
                JOGK = KEY(21)                  ; refer to 6.7.2.

                JF      JOGK, CONTINUE          ; if key already released
                MOV     SPLD(X), 10000
                MOV     SPLI(X), #ADC           ; SPLI points to ADC
                FMUL    RC, STROKEX, KMUL(X)    ; see example a), STROKEX is
                                                ;       max. travel for axis
                CINR    RC
                IMG     X, RC
                WF      ZSF(X)                  ; wait for completion of a preceding move
                SET     RUN(X)
                WT      JOGK                    ; wait until key released
                RES     SPLD(X)                 ; clear velocity
                WF      ZSF(X)                  ; wait end of down ramp
CONTINUE:       .......
                .......
```

Note: Whenever an axis has to execute two moves in rapid sequence, it is strongly recommended to test the ZSF flag before setting the RUN flag for a new move.

7.2. Software Travel Limits

The E-300 motion generator has an over travel detection system based upon the contents of the absolute position register PABS. However, the software travel limit system makes sense only if the axis has been referenced by a homing procedure.
The travel limit in positive direction is loaded into **XLPOS**, the limit in negative direction into **XLNEG.** The contents of XLPOS and XLNEG is a signed integer and it measures the travel in internal steps (IS). The software travel limit system is activated by setting the flag **PLWF.** Whenever an axis is in over-travel, the flag **PLAF** is set and the number of the faulty axis is to be found in the variable **PLAX.** The travel limit system is effective on all 4 axes (provided PLWF is set) as long as the velocity is non-zero (ZSF = 0). If XLPOS and XLNEG hold the default values of 2^{31} - 1, resp. -2^{31} the corresponding axis is **not** protected.

XLPOS, XLNEG are defined for all physical axes, PLWF, PLAF and PLAX are common to all axes. To sum up, the programmer has to effect the following steps in order to use the software travel limit system:

- Load XLPOS and XLNEG with the limit positions for the axis.
- Set PLWF true, the system is now activated.
- Test PLAF, for example, in a dedicated simultaneous program.
- When an over travel is detected, read PLAX to identify the faulty axis.

7.3. The E-300 Interpolator

The E-300 interpolator is basically a generator of vectors. The generation of curved paths must be done by a straight segment approximation. The individual segments are computed by an E-300 program and the components of the vectors are passed to the interpolator.
The E-300 interpolator has the following properties:
- it works in an n-dimensional space, with n = 1..15,
- it generates vectors of any paraxial length, up to 2^{23} IS steps,
- it automatically chains straight segments to form a continuous path,
- it controls the speed along the path.

7.3.1. Principle of Vector Generation

The principles given hereafter apply strictly for two axes in a rectangular coordinate system.
The vector δx, δy has to be generated, δx and δy are given in engineering units. E-300 first computes the displacements in IS units:

$\Delta X = \delta x$ * <KMUL(X)>
$\Delta Y = \delta y$ * <KMUL(Y)>,

where KMUL is a dedicated variable loaded by the program:

<KMUL> is the number of internal pulses required to generate a motion of 1 unit
or
<KMUL> is (number of external pulses required to generate a motion of 1 unit) * <NDIV>

To control the velocity along the vector, the system has to know the modulus of the vector in IS units:

$\delta s = (\delta x^2 + \delta y^2)^{1/2}$
$\Delta S = \delta s$ * <KMUL(S)> KMUL(S) = KMUL(4)

During the vector generation, the velocities of the X- and Y-axes are proportional to the vector components, more precisely proportional to the value of Px and Py:

$Px/2^{24} = |\Delta X/\Delta S| = [\delta x$ * <KMUL(X)>] / [δs * <KMUL(S)>]
$Py/2^{24} = |\Delta Y/\Delta S| = [\delta y$ * <KMUL(Y)>] / [δs * <KMUL(S)>]

Px and Py are stored as 4 byte integers, $\delta x/\delta s$ and $\delta y/\delta s$ can have the value 1 (for an horizontal or a vertical vector respectively). Therefore, KMUL(S) must verify the two conditions:

<KMUL(S)> > <KMUL(X)> and <KMUL(S)> > <KMUL(Y)>.

This will allow the generation of vectors of any slope in the X-Y space. To take rounding inaccuracies in account, set KMUL(S) = largest KMUL(axis) * 1.005

7.3.2. The Interpolation Buffer

A continuous path is made of a sequence of straight segments. The instructions of the next section compute the values of ΔX, ΔY,... ΔS, Px, Py, ... for all segments and store these results into the **Interpolation Buffer**. The computation are made prior to the move in order to obviate computing speed limitations.
Table 7.4.2. The Interpolation Buffer

Axis enables	2 bytes	HEADER Stored by instruction OPENC, Up-dated by SEGMC
Number of segments	2 bytes	
Segment size	1 byte	
Path length	4 bytes	
ΔS_0	3 bytes	STORAGE FOR SEGMENT #0 Example of segment storage For 3 axes computed by SEGMC from Vector data.
ΔX_0	3 bytes	
Px_0	3 bytes	
ΔY_0	3 bytes	
Py_0	3 bytes	
ΔU_0	3 bytes	
Pu_0	3 bytes	
ΔS_1	3 bytes	STORAGE FOR SEGMENT #1
ΔX_1	3 bytes	
Px_1	3 bytes	
…….	…….	

Interpolation Buffer Entries:

Axis enables: The axes involved in the interpolation are marked with a "1" in the bit position. Example: Two-dimensional space (1 & 3) or (Y & U)

				1	0	1	0

Number of segments: Total number of segments in the buffer.

Segment size: The number of bytes required to store one segment: 15 for 2 axes, 21 for 3 axes, 27 for 3 axes or for n axes:

Number of segments = 3+n*6

Path length: This field is cleared by OPENC and up-dated by SEGMC. When a segment is stored, its $\triangle S$ is added to the Path Length. The total path length is used to generate the velocity profile along the path.

The interpolation buffer is located in the RAM storage space.

7.3.3. Interpolation Instructions

Initialize the Interpolation Buffer:

OPENC buf_addr, axis, axis, ...

Arguments: **buf_addr** is an immediate ordinal or an ordinal variable pointing to the physical address of the buffer. This address may be obtained, for example, by the #-prefix or by the ADDR instruction.

axis is the axis number (an ordinal or an ordinal variable which hold an ordinal between 0 and 14. Up to 15 "axis" are accepted. The number of arguments gives the size of the buffer. If "axis" evaluates to 15, the argument is discarded; this may prove useful in writing procedures for different spaces.

Segment Computation and Storage:

SEGMC buf_addr, mode0, value0, mode1, value1, mode2, value2, mode3, value3

Arguments: **buf_addr** refers to OPENC

modei is a boolean or an ordinal, 0 or 1. If mode = 0, value is taken as the coordinate of the vector end point, the origin of the coordinate system being at the starting point of the path. If mode = 1, value is the component of the current vector.

valuei is a floating point number in engineering units, either a coordinate within the path or the component of the current segment, according to "modei".

The argument "modei" and "valuei" must be written for all axes in increasing order, starting at 0. If axis 3 is not involved, mode3 and value3 can be omitted. If axis 0 is not used, mode0 and value0 must be present, but their contents are not relevant, provided axis 0 was not mentioned in instruction OPENC.

Examples of valid instruction pairs:

```
OPENC $5500, 0, 1
SEGMC $5500, 0, 35.2, ; axis 0
                1, 43.5 ; axis 1
```

```
OPENC #BUF, 1, 3
SEGMC #BUF,  0, 0.0,          ; axis 0, not used, values not relevant
             0, 35.2,         ; axis 1, involved
             *, *,            ; axis 2, not used, * can be used
             1, 43.4          ; axis 3, involved
```

SEGMC computes the elements of the current segment and stores them into the buffer. It adds ΔS to the PATH LENGTH in the header of the buffer and increments the segment number.
SEGMC up-dates also two variables for each involved axes:
CFPABS(i) which holds the current coordinate in technical units (floating),
CIPABS(i) which holds the current coordinate in IS units (integer).

The origin for CFPABS and CIPABS is the starting point of the path, not the system origin as given by PABS.

Partitioning of Long Straight Vectors:
If the modulus of a single vector ΔS is $\geq$ 2^23, the instruction SEGMC automatically generates two or more partial vectors. This partitioning is fully transparent to the programmer, but the size of the interpolation buffer is affected.
Next Free Storage Location:

NEXTC dest, buf_addr

Action: NEXTC returns into "dest" the first free location after the buffer starting at "buf_addr".
This instruction is used to built several path buffers in sequence.

Path Execution:

EXECC buf_addr

Action: The path stored at "buf_addr" is outputted to the motion generators. The flag **RUN(15)** is on during the motion. The buffer remains unchanged; it can be executed again and again. The variables controlling the velocity -SPLD(15), SPLI(15)- have to be set prior to executing EXECC.

Example: This example assumes that:

1) The KMUL of the implied axes have been computed from the mechanical data and <NDIV>, see example of sect. 7.1.6.,
2) SPLD(15) and SPLI(15) have been loaded.

```
BUFFER:     DO1R  1000 DUP ?

INTERPOL:   FMUL        KMUL(4), 4000.0, 1.005 ; Scale factor for path axis *)
            OPENC       #BUFFER, 0, 2          ; Buffer for axes 0, 2
            SEGMC           #BUFFER, 0, 0.2,       ; First segment, axis 0
                              0, 0.0,              ; axis 1 not implied
                              0, 0.4               ; axis 2
            SEGMC           #BUFFER, 0, 0.75,      ; Second segment
                              0, 0.0,
                              0, 0.94
            EXECC       #BUFFER                ; Execution
            WT          RUN(15)                ; Wait for path end
            NEXTC       RA, #BUFFER            ; RA contains address for the next
                                                ; buffer
            .......
```

*) 4000.0 is supposed to be the largest KMUL of the implied axes. The multiplication by 1.0005 ensures the condition of sect. 7.4.1. This multiplicator factor does not arm the geometric accuracy, merely it slightly modifies the path velocity

8. EXAMPLES OF PROCEDURES

The purpose of this chapter is to present several procedures involving the various E-300 functions. The procedures serve also as programming examples.
A program is generally written according to the following structure:

MODule directive
Equivalence Statements
Constant Definitions (FLASH-Variables)
Variable Definitions (RAM-Variables)
Procedures
Main Program(s)
"END label" Directive

The order of the items between MODule directive and END directive is not mandatory.

8.1. Dead Time Procedures

The timers of the E-300 are the dedicated variable TMR(i), i = 0 to 7, associated with the control and status bits TAF(i). Whenever TAF is set, the contents of the corresponding TMR is decremented every millisecond. TAF is automatically reset when <TMR> goes through zero. Split timing is possible by switching TAF on and off. TMR belongs to the VO4S-type.
The procedures "WAIT" and "WAITS", introduce a dead time into a process. WAIT needs an argument in millisecond for short timing; WAITS accepts a time in seconds and decimal fractions. The procedures are fully re-entrant, i.e. they can be called simultaneously from all programs as they use a different timer for each simultaneous program through PNB-indexing.

Procedure Calls:

```
        WAIT   time     ; parameter 0 = time in ms, immediate integer
                        ; or numerical variable

        WAITS  time     ; parameter 0 = time in seconds, immediate
                        ; floating-point number or variable
```

Procedure Listings:

```
WAIT:   MOV    TMR(PNB), %(0)        ; time into TMR indexed by the
                                     ; program number
        SET    TAF(PNB)              ; activation of timer
        WT     TAF(PNB)              ; wait if timer active
        RET
WAITS:  FMUL   RA, %(0), 1000.0      ; seconds  ->  milliseconds
        CINR   TMR(PNB), RA          ; time in ms, integer
        SET    TAF(PNB)
        WT     TAF(PNB)
        RET
```

8.2. Procedures for I/O Control

The procedure "Set an Wait", SW, sets an output (more generally a boolean variable) and waits for an acknowledge input.
Procedure Call:

```
        SW     var1, var2       ; parameter 0 = variable to be set
                                ; parameter 1 = acknowledge signal,
```

Procedure Listing:

```
SW:     SET     %(0)
        WF      %(1)    ; wait if var = 0
        RET
```

The procedure "Wait with Timeout", WTMT, is very useful in most control systems: many dysfunctions can be detected and signaled if an acknowledge signal does not come in time.

Procedure Call:

```
        WTMT        var,    ; parameter  0 = acknowledge signal, eventually with /
                    time,   ; parameter  1 = time in sec. floating
                    label   ; parameter 2 = return label if time-out occurs

        TIMOUT:     ….      ; Return label in case of time-out.
                            ; Appropriate action can be taken.
                            ; TIMOUT must be at the same nesting level as WTMT.
```

Procedure Listing:

```
        WTMT:       FMUL    RA, %(1), 1000.0
                    CINR    TMR(PNB), RA        ; time in ms
                    SET     TAF(PBN)            ; activate timer
        WTMT1:      IFT     %(0) RET            ; test var, normal return if variable = 1
                    JT      TAF(PNB), WTMT1     ; test timer flag, loop if set
                    RETJ    %(2)                ; time-out occurs, return and jump to
                    NOP                         ; TIMOUT
```

8.3. Positioning Procedure

The procedure accepts either a target position (absolute move) or a displacement (relative move). Up to 4 axes may be simultaneously controlled, the velocity is passed to the procedure as a parameter, exit of the procedure after starting the motions or after the completion of the moves, as specified by a parameter.
The displacement or the coordinate are given as a number of technical units, such as mm. inches, degrees or the like. The displacement must be a floating-point number. The scale factors, KMUL, are supposed to be already computed.

Procedure Call:

```
        POS     axis, mode, displ., speed, wait,
                axis, mode, displ., speed, wait
```

With:
axis = number of the axis to be activate, 0 or 1 (ordinal type)
mode = positioning mode; 0 for absolute, 1 for relative (ordinal type)
displ = displacement or coordinate (floating type)
speed = velocity to SPLD, 0..32639 (ordinal type)
wait = wait flag: 1 if the procedure must wait until the motion is executed (ordinal)

Procedure listing:

```
X       = 0             ; AXIS
Y       = 1


POS:    RES     RF              ; pointer in parameter structure
POS1:   JNPAR %(RF), POS2                       ; end of the structure ?
```

```
        MOV     RA, %(RF)       ; get axis number into RA
        MOV     SPLD(RA), %(3+RF)           ; get speed
        MOV     RC, %(2+RF)                 ; get displacement
        FMUL    RC, KMUL(RA)                ; convert from units to steps,
                                            ; all KMUL already computed !
        CINR    RC                          ; displ. in IS-units
        IFZ     %(1+RF)  ISUB RC, PABS(RA) ; absolute ?
        IMG     RA, RC                      ; initialize motion
        WF      ZSF(RA)                     ; wait end of previous move
        SET     RUN(RA)                     ; start motion
        IADD    RF, 5                       ; next axis
        JMP     POS1

        ; now, all the axes are started

POS2:   RES     RF                          ; parameter structure pointer
POS3:   MOV     RE, %(4+RF)                 ; get mode parameter
        JF      RE, POS4                    ; next axis if no wait
        MOV     RA, %(RF)                   ; get axis number into RA
        WT      RUN(RA)                     ; wait until motion stopped

POS4:   IADD    RF, 5                       ; next parameter set
        IFNPAR          %(RF), RET
        JMP     POS3
```

8.4. Jogging Procedures

A motion in the "Jogging" mode is controlled by a boolean variable, generally a push-button or a switch. The motion starts when the button is depressed and stop gently when it is released. If the procedure is called when no button is depressed, exit is immediate.

```
JOGPK =         KEY(21)                 ; Right arrow button (see sect.6.7.2)
JOGNK =         KEY(30)                 ; Left arrow button


JOGSP:          DO2F    10000,          ; jogging velocities for all
                        5000,           ; 4 axes, to be loaded into SPLD
                        5000,
                        8000
NLIM:           DI4F    0,              ; soft limits in IS-units
                        0,
                        0,
                        0
PLIM:           DI4F    100000,
                        200000,
                        75000,
                        500000

; jogging procedure call:
;       JOGGING  button, direction, axis

;       with:   button = jogging control
;               direction = 1 for negative motion, 0 for positive
;               axis = axis number 0..3

; jogging procedure:
JOGGING:        IFF     %(0), RET                       ; button ?
```

```
                MOV     RA, %(2)                ; get axis number
                MOV     SPLD(RA), JOGSP(RA)     ; set jogging speed
                MOV     SPLI(RA), #ADC          ; potentiometer active
                MOV     RB, PLIM(RA)            ; get limit
                IFNZ    %(1), MOV RB, NLIM(RA)  ; negative motion ?
                ISUB    RB, PABS(RA)
                IMG     RA, RB                  ; initialize and
                SET     RUN(RA)                 ; start motion
                WT      %(0)                    ; motion is running
                RES     SPLD(RA)                ; set speed to zero
                WF      ZSF(RA)                 ; wait until motion terminated
                RES     RUN(RA)                 ; cancel motion
                RET


; Main Program
LOOP:           .......
                JOGGING     JOGN, 1, 0          ; axis 0, neg. direction
                JOGGING     JOGP, 0, 0          ; axis 0, pos. direction
                JOGGING     JOGN, 1, 1          ; axis 1, neg. Direction
                JOGGING     JOGP, 0, 1          ; axis 1, pos. direction
                 .........
```

8.5 Axis Position Display

The actual position of the axes X and Y are displayed in a continuous manner by a simultaneous program running in the background.

```
        MOD    2

        INCL (SYSTEM.INC)
        INCL (KBD300.INC)
X =     0
Y =     1


MAINPROG:   .........
            .........
            ASIM    1, DISPSIM
            ........

; Display simultaneous program:
DISPSIM:    KPR     CLS                 ; VFD cleared
            KPR     CURSOF                      ; cursor off
            STPR    <"X       Y       ">        ; legend
            MOV     PMF0(PNB), 3        ; printing format
            MOV     PMF1(PNB), 3
DISPS1:     CFLO    RA, PABS(X)
            FDIV    RA, KMUL(X)         ; convert to unit
            KPR     DP, 1               ; display position
            FPR     RA                  ; printing
            CFLO    RA, PABS(Y)
            FDIV    RA, KMUL(Y)
            KPR     DP, 12
            FPR     RA
            MOV     TMR(PNB), 70        ; to slacken the display
            SET     TAF(PNB)
```

```
WT      TAF(PNB)
JMP     DISPS1
```

APPENDIX A: E-300 DEDICATED VARIABLES

Legend:

Type: according to the definitions of chapter 4.
"boolean" stands for non mapped boolean variables.

Array size: Size of array, not always physically fully implemented.

Range: "full" means full range according to the variable type

Init: "R": actual value given by device
"prec.": preceeding value retained into battery-backed RAM
"0": initialized to 0

Assy: "yes": the A-600 assembler has definition for the variable name,
" ": the definition must be provided by the source program (included definition file).

Variable Name	Type (address)	Function	Array	Range	Init	Assy
ADC	VO1S (BP $13C)	Analog to Digital Converter	1	Full	R	
BST	Boolean	Driver BooST Output	16		0	
BSTE	VBS (BP $050)	BooST Enable	16		0	
BSTI	VBS (BP $060)	BooST Invert	16		0	
CST0	VO1S (BP $15^{E})	ConSTant=0	1		0	
CST255	ConSTant 255	ConSTant=255	1		$FF	
DAC	VO1S (BP $13D)	Digital to Analog Converter	1	Full	0	
DIV	VO2S (BP $0DC)	Frequency DIVider	16	40..300	$0100	
DIR	VBS (BP $030)	DIRection flag	16		R	
FAULT	IN($0C0)	Axis FAULT	15			
FLG	Boolean	General Purpose FLAGs	256		Prec	yes
IN	Boolean	Inputs	128		R	yes
INA	IN($0A0)	Reference inputs	2		R	
INAE	VBS (BP $070)	INput axis A Enable	16		R	
INAI	VBS (BP $080)	INput axis A Invert	16		R	
INB	IN($0B0)	Inputs	2		R	
INBE	VBS (BP $090)	INput axis B Enable	16		R	
INBI	VBS (BP $0A0)	INput axis B Invert	16		R	
INIT	VBS (BP $029)	INIT input (temp. sensor)	1		R	
KAF	VBS (BP $010)	Key Available Flag	32		0	
KEY	Boolean	Keys as boolean	32		R	yes
KCOD	VO1S (BP $060)	Keyboard CODe	2	Full	0	
KDN	VO2S (BP $0BC)	Acceleration	16	1..65535	$03^{E}8	
KUP	VO2S (BP $09C)	Acceleration	16	1..65535	$03^{E}8	
KXCOD	VO2S (BP $062)	Keyboard matriX CODe	1	Full	$10	
LED	boolean	LEDs	8		0	yes
LSA	VBS (BP $0B0)	Limit Switch Acknowledge	16		0	

Variable Name	Type (address)	Function	Array	Range	Init	Assy
NTC	VBS (BP $028)	Temperature Watching Mode Flag	1		1	
OUT	boolean	OUTputs	128			yes
PABS	VI4S (BP $0FC)	Position ABSolute	16	Full	0	
PI	3.141592654	Constant				
PMF0	VO1S (BP $06C)	Print Mode Floating 0 (left)	8	0..10	3	
PMF1	VO1S (BP $074)	Print Mode Floating 1 (right)	8	0..10	3	
PMI	VO1S (BP $064)	Print Mode Integer	8	0..10	6	
RAD	VBS (BP $02A)	RADian flag	1		1	
RST	boolean	Driver ReSeT output	15		0	
RTE	VO1S (BP $160)	RunTime Error	1		R	
RUN	VBS (BP $018)	RUN flag	16		0	
SAF	VBS (BP $008)	Simultaneous Active Flag	8		0	
SPLD	VO2S (BP $07C)	SPeed Limit Direct	16	0..32639	$1000	
SPLI	VO2S (BP $13^{E})	SPeed Limit Indirect	16	Full	#CST255	
TAF	VBS (BP $000)	Timer Active Flag	8		0	
TMR	VO4S (BP $040)	TiMeR	8		0	
VACT	VO2S(BP $160)	ACTual Velocity	16	0..32639	R	
VER	VSTRS (BP $1C1)	Interpreter Version (20 char)	1		R	
ZSF	VBS (BP $040)	Zero Speed Flag	16		R	