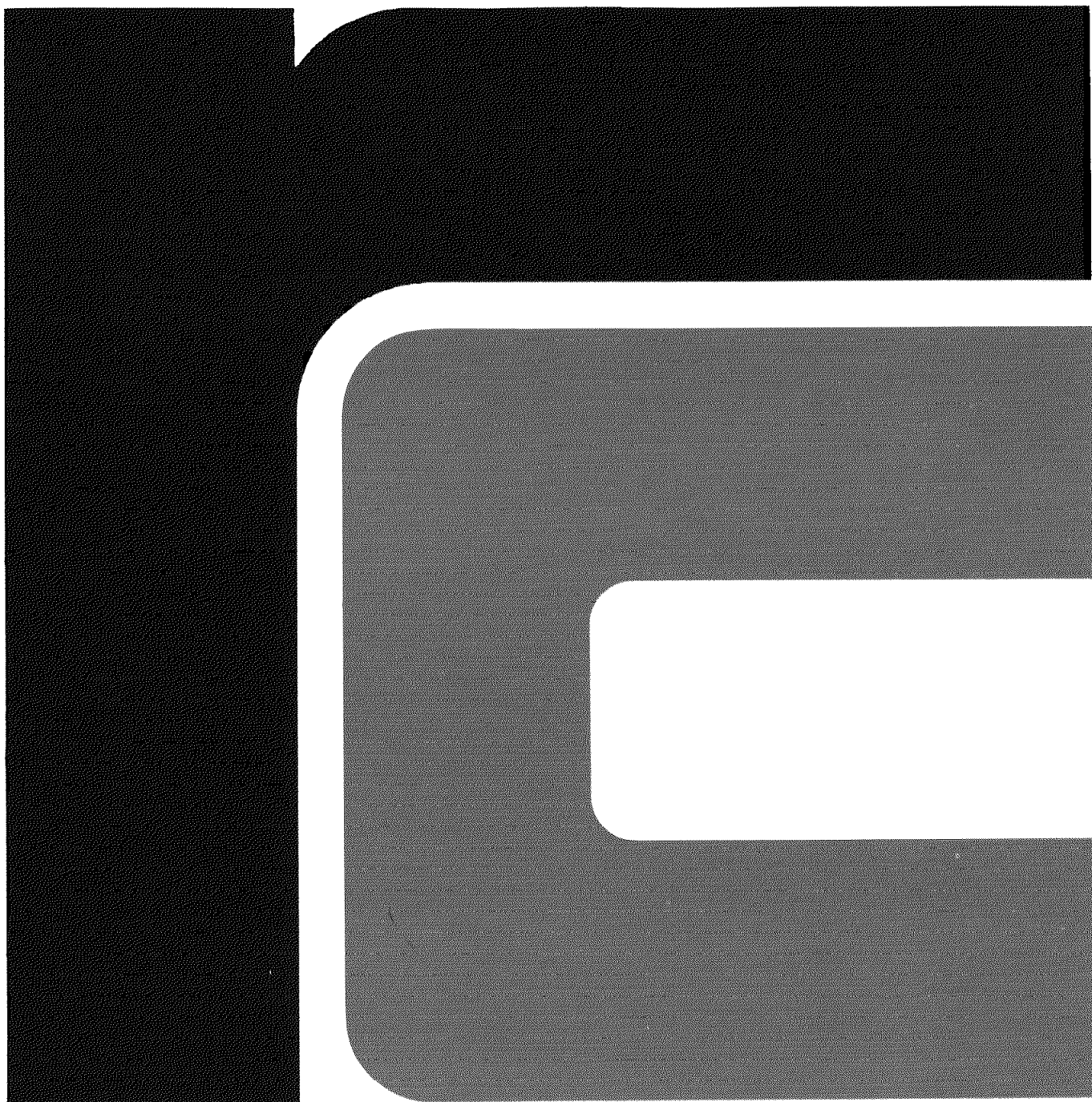


**DOMAC**  
**Domus Macro Assembler**  
**User's Guide**



**36000**

**DOMAC**  
**Domus Macro Assembler**  
**User's Guide**

A/S REGNECENTRALEN  
Information Department

First Edition  
July 1978  
42-i 0833

Author: Jens Lovmand Hvid

KEY WORDS: RC 3600, Macro Assembler, User's Guide.

ABSTRACT: This paper describes the RC 3600 Macro Assembler language and operation of the DOMAC Macro Assembler.

Users of this manual are cautioned that the specifications contained herein are subject to change by RC at any time without prior notice. RC is not responsible for typographical or arithmetic errors which may appear in this manual and shall not be responsible for any damages caused by reliance on any of the materials presented.

Copyright © A/S Regnecentralen, 1978.  
Printed by A/S Regnecentralen, Copenhagen.

# Table of Contents

PREFACE	page 7
1 INTRODUCTION	9
1.1 Macro Assembly	9
1.1.1 Machine Language	9
1.1.2 Assembly Language	9
1.1.3 Translation of Assembly Language	10
1.1.4 The Macro Facility	11
1.2 Assembler Input/Output	12
1.2.1 Assembler Input	12
1.2.2 Scanning Modes	13
1.2.3 Assembler Output	15
1.2.3.1 Error messages	15
1.2.3.2 Program listing	15
1.2.3.3 Relocatable binary object program	18
1.2.4 Linkage editing	19
2 SYNTACTIC ELEMENTS	21
2.1 Terminals	21
2.1.1 Operators	21
2.1.2 Break Characters	22
2.2 Numbers	23
2.2.1 Integer numbers, single precision	23
2.2.1.1 Source representation	24
2.2.2 Integer numbers, double precision	27
2.2.2.1 Source representation	28
2.2.3 Floating point numbers, single precision	29
2.2.3.1 Source representation	31
2.3 Symbols	32
2.4 Special elements	32
2.4.1 The character @	32
2.4.2 The character #	33
2.4.3 The characters **	33
2.4.4 The characters ++	33
2.4.5 The character ←	33
3 SYNTAX	34
3.1 Symbols	34
3.1.1 Permanent symbols	35
3.1.2 Semi-permanent symbols	35
3.1.3 User Symbols	36
3.1.4 Symbol table listing	36
3.2 Expressions	36
3.2.1 Operator hierarchy	37
3.2.2 Relocation Characteristics	41

3.3	Instructions	page 43
3.3.1	Arithmetic and Logical Instructions	43
3.3.2	Program Flow Control Instructions	47
3.3.2.1	Addressing	48
3.3.3	Data Transfer Instructions	50
3.3.4	Input/Output Instructions	51
3.3.4.1	Input/output instructions with accumulator	51
3.3.4.2	Input/output instructions without accumulator	53
3.3.4.3	Input/output instructions without device codes	54
4	PERMANENT SYMBOLS	57
4.1	Source interpretation	57
4.2	Program control	58
4.3	Semi-permanent symbol definition	58
4.4	External reference	59
4.5	Macro definition	59
4.6	Alphabetic list of permanent symbols	60
	.	60
	.ARGCT	61
	.BLK	63
	.DALC	64
	.DIAC	66
	.DICD	67
	.DIO	68
	.DIOA	69
	.DISC	70
	.DMR	71
	.DMRA	73
	.DO	75
	.DUSR	76
	.DXOP	77
	.EJEC	78
	.END	79
	.ENDC	80
	.ENT	81
	.EOT	82
	.EXTA	83
	.EXTD	85
	.EXTN	86
	.EXTU	87
	.GOTO	88
	.IF	89

.LIST	page 91
.LOC	92
.MACRO	93
.MCALL	94
.MSG	95
.NOCON	96
.NOLOC	97
.NOMAC	98
.NREL	99
.PASS	100
.POP	101
.PUSH	102
.RDX	103
.RDXO	104
.TITL	105
.TOP	106
.TXT	107
.TXTM	109
.TXTN	110
.XPNG	112
.ZREL	114
5 MACRO PROGRAMMING	115
5.1 Macro definition	115
5.1.1 Interpretation	117
5.1.1.1 The character %	117
5.1.1.2 The character †	117
5.1.1.3 The character ←	117
5.1.2 Nesting of Macros	118
5.2 Macro call	119
5.2.1 Syntax of Macro call	119
5.2.2 Substitution of arguments	120
5.3 Repetitive and conditional operations in Macros	121
5.4 Listing of Macro expansions	122
5.5 Examples of Macros	123
5.5.1 Macro ZONE	123
5.5.2 Macro NDEF	130
6 EXTENDED CAPABILITIES	136
6.1 Generation of numbers and symbols	136
6.2 Generation of labels	139
7 ASSEMBLER OPERATION	141
7.1 External requirements	141
7.2 Call of DOMAC assembler	141

7.3 Assembly execution	page 144
7.4 DOMAC error messages	147
7.4.1 Source input errors	147
A Addressing error	148
B Bad character	148
C Macro error	148
D Radix error	148
E Equivalence error	149
F Format error	149
G Global error	149
I Input parity error	149
K Conditional error	149
L Location counter error	149
M Multiple definition error	150
N Number error	150
O Overflow error	150
P Phase error	150
Q Questionable line	151
R Relocation error	151
U Undefined symbol error	151
V Assembler label error	151
X Text error	151
7.4.2 Run time errors	151
APPENDIX A	153
APPENDIX B	157
APPENDIX C	159
APPENDIX D	168
APPENDIX E	169

# Preface

## Notation Conventions

This manual is intended to describe the syntax of the DOMUS Macro Assembler Language. To do so with sufficient clarity, it will be necessary to supplement the ordinary text with a number of special symbols - together with some rules regarding the application of those symbols - which are themselves not part of the Macro Assembler language. These symbols and their rules of application are as follows:

- ↓ represents a carriage return (new line).
- ↓↓ represents a form feed (new page).
  
- CAP capital letters are used whenever some part of the format described in the text is a literal part of the symbolic language and which therefore must appear in context exactly as indicated.
  
- low lower case letters combined with underscoring are used to denote those parts of the format which are variable and where consequently the programmer will have to decide on whichever string of symbols or characters should be used. If ample space is at hand, the variable descriptor will be written out fully, but in other cases it may be abbreviated. If more than one word is needed to describe the variable, the individual words will be joined by means of a hyphen.
  
- < > parts of a format appearing in between angle brackets indicates, that those specific parts of the format are optional. For example:  
           .END<Δexpression>
  
- (():()) ordinary parentheses separated by colon is used to indicate a choice of possible formats of which one only is to be used. Such alternate choices may themselves be optional, in which case the outer set of parentheses will be supplanted by angle brackets. Examples of the above are:  
           .IF((E):(G):(L):(N))  
           .TXT<(E):(F):(O)>Δ delimiter



- Δ represents any number and combination of break characters, i.e. the characters comma, space or tabulation.
- Ä represents one single break character (comma, space or tabulation) - a restriction which must be observed in certain formats.
- .... represents an omission of one or more words of a format in cases where the consequent space-saving effect does not compromise the necessary clarity.

In this manual octal representation of numbers will be used extensively alongside normal decimal notation. To differentiate between these two representations octal numbers in the text itself will always be distinguished by the suffix  $g$ .

In examples octal representation will be the rule and will not normally be specifically indicated; representation to other bases must be inferred from the actual context.

# 1 Introduction

## 1.1 Macro Assembly

The function of any language is to ease the transfer of information from its point of origin to the recipient. In this context it is of no real importance what medium is used for actual transmission - the most well-known of course being air, which carries our normal spoken conversation.

What is important is that both originator and recipient are able to understand the language.

### 1.1.1 Machine Language

In the RC 3600 series computers - as in all similar electronic computers - information is transferred through the use of small electric currents, and the language itself takes the form of sequences of either "current" or "no current". In this way the actual configuration of the sequence, which is usually called the code, will determine the meaning involved. To visualize the code to human beings the digit 1 is used to indicate "current" and the digit 0 is used to indicate "no current"

The computer will perform a series of operations according to those instructions which are given to it in the program - but as outlined above these instructions must be given to the computer in the language which it is able to understand, i.e. as a basically simple but rather long sequence of ones and zeroes. Thus if the programmer wishes to load a value, which is at present stored in the memory cell numbered 6, into the accumulator numbered 0, the corresponding code in machine language would be:

```
001000000000110
```

### 1.1.2 Assembly Language

The machine language is however not really convenient for human beings - mainly due to its inherent monotony which easily leads to use of erroneous codes and thus to unpredictable results. To avoid this another type of language is created, in which the machine language codes are replaced by symbols. These symbols are then given a form, which makes the resulting language more meaningful to the programmer and consequently easier for him to learn and remember. The machine language instruction mentioned in the preceding section is thus in the assembly language replaced by the following instruction in symbolic form:

```
LDA 0 6
```

The sequence of characters "LDA" is now a symbol - called an "instruction mnemonic" - which means: load accumulator. The accumulator to be loaded is indicated by the digit 0, and the memory address containing the number (or other type of information) to be loaded is given as the digit 6.

But the assembly language is designed in a way, which offers further advantages: instead of indicating the address of the memory cell containing the information by the digit 6 in the above example, the programmer may use a symbolic representation. In the program the programmer can define the address in question through the symbol ASST, for example, and the instruction would then take the form:

```
LDA 0 ASST
```

### 1.1.3 Translation of Assembly Language

Use of the assembly language leads to easier and more convenient programming, but the computer is unable to interpret this language directly. It must consequently be translated into machine language, a task, which is easily performed by the computer itself, provided a program is available which contains the necessary instructions to perform the translation. This special program is called an assembler program and the process of translation is called assembly.

The assembler program reads the symbolic language and converts the instructions and other information contained therein to the appropriate numeric codes of the machine language. In this way is obtained a direct correspondence between the symbolic language and the machine language, i.e. one line of assembly language is converted into one line of machine language. (It is quite obvious that the actual design of the assembly language is an extremely demanding task, but a detailed description of these problems is outside the scope of this work.)

What has been described above is in fact only the preparations, which must be carried out in order to use the computer for the actual job of processing some information in a specified way. The first step - writing the program in assembly language - results in the source program. The next step - translating the source program - results in the object program, and only then can the computer tackle the real job: execution of the object program to provide the desired processing of data.

#### 1.1.4 The Macro Facility

It was mentioned in the preceding section that provision of the symbolic assembly language was intended to simplify the design of actual object programs. This process of simplification is carried a step further by implementation of the Macro Facility.

The idea underlying this concept can be stated as follows: usually the writing of a program will involve several repetitions of the same set of symbolic instructions; to avoid this tedium - and thus enable the programmer to concentrate on the fundamental structure of his program - macro assembly makes it possible to define such a set (or sets) of instructions only once in the program and to refer to the complete set as a whole wherever it must appear in the program.

Use of the macro facility is implemented in the following way:

The appropriate set (or sets) of instructions is written by the programmer as a macro definition, which is then given a name by the programmer.

Wherever this particular set of instructions should appear in the program, the programmer writes a macro call (which must as a minimum contain the name of the macro definition.)

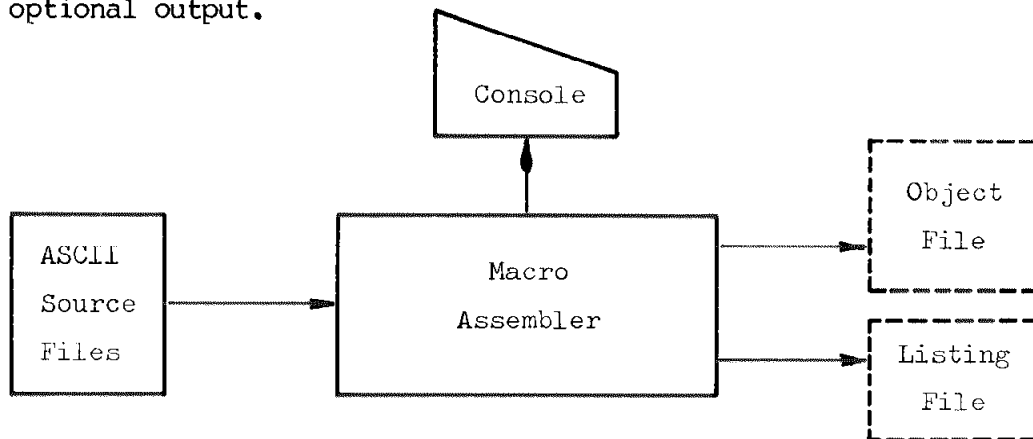
During translation the assembler will then - through the use of the so called macro processor - substitute the set of instructions contained in the macro definition for the macro call. This effect is called macro expansion.

Another feature of the macro facility, which enhances its versatility still more, is its capability to accept dummy arguments. This means that if the program contains sets of instructions which are identical except for accumulators and memory addresses, it is possible to define these sets in one macro definition only, in which the necessary arguments are indicated by dummy symbols; when the actual macro call is employed in the program it will contain the actual arguments, and these will then be substituted for the dummy arguments during the macro expansion. In this way the macro definition is normally only a framework of the actual instruction set embodied in the resulting program.

## 1.2 Assembler Input/Output

Input to the macro assembler consists of one or more files of source program written in accordance with the syntactic rules outlined in the following chapters of this manual.

Output from the macro assembler can be more or less extensive. As a minimum it will consist of a list of source program errors, which will be output on the console. It may however be extended to include a program listing for reference purposes and a file containing the object program, which may thereafter be loaded and executed. The object program file is output in a form called "relocatable binary", the details of which will be outlined in a following section. The input/output structure is shown in the diagram below, where the boxes drawn in broken line represent optional output.



### 1.2.1 Assembler Input

Input to the assembler is one or more source program file(s), which the assembler will read line-by-line. One line of source program consists of all characters read until a carriage return character (CR) or a form feed character (FF) is encountered, and is subdivided into a maximum of four fields:

<label-field><operation-field><operator-field><comment-field>

some of which fields may be empty in any specific line depending on assembler syntax and/or the context in which the line appears. The label-field is used to assign a symbolic name (a label) to a memory location and contains a user-symbol followed by a colon. The operation-field is used to specify the actual contents of the (possibly) labelled memory location. The operation-field may contain either a value - that is: a constant or an expression -

or a symbol, which may be either a permanent symbol or a semi-permanent symbol.

The operator-field contains the specific operators associated with the contents of the preceding operation-field. Its actual contents will therefore be highly dependent on the operation-field, but will usually be one or more expressions.

The comment-field contains a string of characters, which will be read and listed by the assembler, but which will otherwise be ignored. This field is thus solely intended to aid human interpretation of the finished source program.

All characters appearing in the source input will be read by the assembler, provided that they are standard ASCII characters. However, three characters will without exception be ignored:

NUL - the null character (code:000<sub>8</sub>)  
 NL - the line feed character (code:012<sub>8</sub>)  
 DEL - the rubout character (code:177<sub>8</sub>)

The character SUB (code:032<sub>8</sub>) will - if it occurs in the input - be printed in the output listing as the character \ , but it will otherwise be ignored, except that the line containing this character will be flagged with parity error.

The character EM (code: 031<sub>8</sub>) represents the physical end-of-file and the source file will not be scanned beyond this character.

### 1.2.2 Scanning Modes

The string of characters input to the assembler will be scanned in one of two alternative modes: string mode or normal mode. In string mode all ASCII characters will be accepted as input, and no interpretation of the string will take place. String mode is applied for either of the following three purposes:

#### a) Comments

A comment is initiated by a semicolon and is terminated by a carriage return character or a form feed character.

For example:

```
.....; ABSOLUTE VALUE OF VARIABLE ↓
```

#### b) Macro Definition Strings

A macro definition string is initiated by the directive .MACRO and is terminated by the character %.

For example:

```

.MACRO  SUBST
LDA     0   6
MOVE#   3   3  SZC

```

⌘

### c) Text Strings

A text string is initiated by the directive `.TXT` (or one of the optional equivalents to this) followed by one or more break characters. After this follows a delimiter and the text string proper, which is ultimately terminated by reappearance of the same delimiter. The delimiter may be any character, but it should obviously not be one of those occurring in the text string itself, as this would result in premature termination of the text string.

For example:

```

.TXT / RANGE: 230-270/

```

All input not in string mode will be accepted by the assembler in normal mode. In this mode only a definite subset of the full ASCII character set will be accepted as legal input, and the input character string must be divided into lines. Line termination is - as previously mentioned - indicated by either CR or FF characters. In normal mode a definite interpretation of input will take place; primarily certain characters and/or groups of characters will be interpreted as specific syntactic elements, further discussion of which is however deferred to the next chapter. Further rules of interpretation concerns lower case alphabetic characters, which are without exception interpreted as their upper case equivalents. If the input source file contains a character, which is not an element of the subset accepted in normal mode, such a character will be unconditionally ignored in respect to assembler syntax, but it will appear with a B (bad character) flag in the error listing.

The subset of ASCII characters accepted in normal mode is listed on the following page.

### 1.2.3 Assembler Output

The assembler program may produce three distinctly different types of output:

- Error messages to the console
- Program listing
- Relocatable binary object program

1.2.3.1 Error messages are given in the form of single letter codes and will be output to the console. During the assembler program's first pass over the source file, a fairly small class of errors will be detected if they occur. The corresponding error messages will be output at that time.

During the second pass all errors detected will be output, either as part of the program listing or - if the optional program listing has not been requested - as a separate error listing output to the console.

A more detailed survey of error messages and codes appears in section 7.4.

1.2.3.2 Program listing is intended to supply the programmer with information about the program as it has been read and accepted by the assembler, thereby to ease comparison with his original program. In addition the program listing may contain error indications as mentioned in the preceding section. The program listing is output in the form of lines corresponding to the lines of source file input. Each line will contain the following information:

Columns 1 -3: If no errors are detected in the input, these columns will contain a two-digit line number followed by one space. If errors are detected, a maximum of three single-letter error codes will be output in these columns (cf. section 7.4)

Columns 4 - 8: These columns will contain the value of the program counter whenever this applies. Otherwise they will be left empty.

Column 9: This column contains the relocation flag relevant to the program counter listed in columns 4 - 8.

Columns 10-15: These columns will form the data field whenever this applies. Otherwise they will be left empty.

Column 16: This column contains the relocation flag relevant to the data field in column 10- 15.



7-bit Octal Code	Character	7-bit Octal Code	Character	7-bit Octal Code	Character
011	HT	074	<	134	\
014	FF	075	=	135	]
015	GR	076	>	136	†
031	EM	077	?	137	←
032	SUB	100	@	141	a
040	SP	101	A	142	b
041	!	102	B	143	c
042	"	103	C	144	d
043	#	104	D	145	e
045	%	105	E	146	f
046	&	106	F	147	g
047	'	107	G	150	h
050	(	110	H	151	i
051	)	111	I	152	j
052	*	112	J	153	k
053	+	113	K	154	l
054	,	114	L	155	m
055	-	115	M	156	n
056	o	116	N	157	o
057	/	117	O	160	p
060	0	120	P	161	q
061	1	121	Q	162	r
062	2	122	R	163	s
063	3	123	S	164	t
064	4	124	T	165	u
065	5	125	U	166	v
066	6	126	V	167	w
067	7	127	W	170	x
070	8	130	X	171	y
071	9	131	Y	172	z
072	:	132	Z		
073	;	133	[		

Columns 17-: The remainder of the available columns will contain the line of source input as read by the assembler. If macro expansion has occurred during assembly this will be incorporated in the listing.

As part of the program listing the assembler may produce a cross-reference listing of the symbol table. This table may be restricted to the user-defined symbols appearing in the program, or it may be extended to include semi-permanent symbols as well, depending on the programmer's choice. An example of a cross-reference listing is given below:

EX123	000002'		2/22						
EX124	000003'	EN	2/05	2/25					
EX125	000004'		2/28						
EX126	000005'		2/31						
EX127	000006'		2/34						
EX130	000007'		2/37						
EX131	000010'		2/40						
EX132	000011'		2/43						
EX888	177777	XN	2/06						
EX889	000001\$	XD	2/07						
EX999	000002\$	XD	2/07						
EXMAC	000000	MC	2/10	2/14	2/16				
I	000133		2/20	2/22	2/23	2/25	2/26	2/28	2/29
			2/31	2/32	2/34	2/35	2/37	2/38	2/40
			2/41	2/43	2/44				
JMP	000000	PS	2/48						
LDA	020000	PS	2/47						
SENDM	006004	PS	2/49						

The relocatability symbols appearing in the example will be detailed in the following section; the symbols appearing in the "type of symbol" column in the example are given in the list below:

ΔΔ	user-defined symbol
EN	entry (.ENT)
XD	external displacement (.EXTD)
XN	external normal (.EXTN)
PS	semi-permanent symbol (.DUSR)
MC	macro name

As previously mentioned, the output listing is an optional feature and it may be omitted according to the programmer's choice. Similarly the programmer can choose to suppress certain lines of the list if he so wishes.

1.2.3.3 The relocatable binary object program is a line-by-line translation of the source program into machine code, which is however eventually output in a specially blocked binary code (cf. appendix C). The majority of lines will after translation appear as one 16-bit binary number, which is the basic unit of information handled by the computer and is called a word. The words of the final object program will on execution be placed in CPU memory by the loading program, to do which it must be able to associate a memory address with the individual words.

When the assembler assigns an address to each word of the object program it will not necessarily be the absolute memory address into which the word will be located by the loader; this may be the case, but the effective address may on the other hand be changed during the loading operation, thereby effecting relocation of the object program. Consequently the assembler must produce an object file, which in addition to the contents of each address also contains the necessary information to the loader regarding this relocation in memory.

Information about the intended relocation is provided by the assembler, which maintains three program counters of which one only will be current at any given time. Initially the three program counters will be set to zero; as each successive word is generated during assembly, the current program counter will be incremented by one. The three program counters correspond to absolute, normal and page zero relocation respectively, whereby absolute relocation means, that the relative memory location is not changed during load; normal relocation means, that a constant value is added to the relative memory location value during load, and page zero relocation means, that a constant similarly is added to the relative memory location value during load.

Byte relocation - whether normal or page zero - means, that the appropriate relocation constant is added twice during load. The constants associated with normal and page zero relocation will usually be different.

The programmer may select which among the three program counters should be current at any specific stage of assembly. This is done by including in the source text the appropriate permanent symbol (.LOC, .NREL and .ZREL respectively); these symbols may likewise be used to reference the respective program counter values in the program (cf. section 4.6).

Although the preceding passage has concentrated on the relocation of word addresses, it should be noted, that also the contents of the storage word may undergo relocation during load, i.e. a constant value may be added to the word as well as to its address during the loading operation.

NOTE: The loading programs used by the MUS/DOMUS operating systems will only accept binary object program having normal relocatable addresses. (They will in fact also accept absolute relocatable addresses, but such addresses are destined for use solely by the operating system itself).

As previously mentioned the program counter value generated during assembly will be output in the (optional) program listing followed by the one-character relocation flag. This flag refers to the various possible relocation characteristics and may take the following forms:

Flag character	Relocation characteristic
space	absolute
-	page zero relocatable
=	page zero byte relocatable
'	normal relocatable
"	normal byte relocatable

Similarly the relocation characteristic relevant to the data field as it appears in the program listing will be indicated by the following flags:

Flag character	Relocation characteristic
space	absolute
-	page zero relocatable
=	page zero byte relocatable
'	normal relocatable
"	normal byte relocatable
\$	externally defined data field.

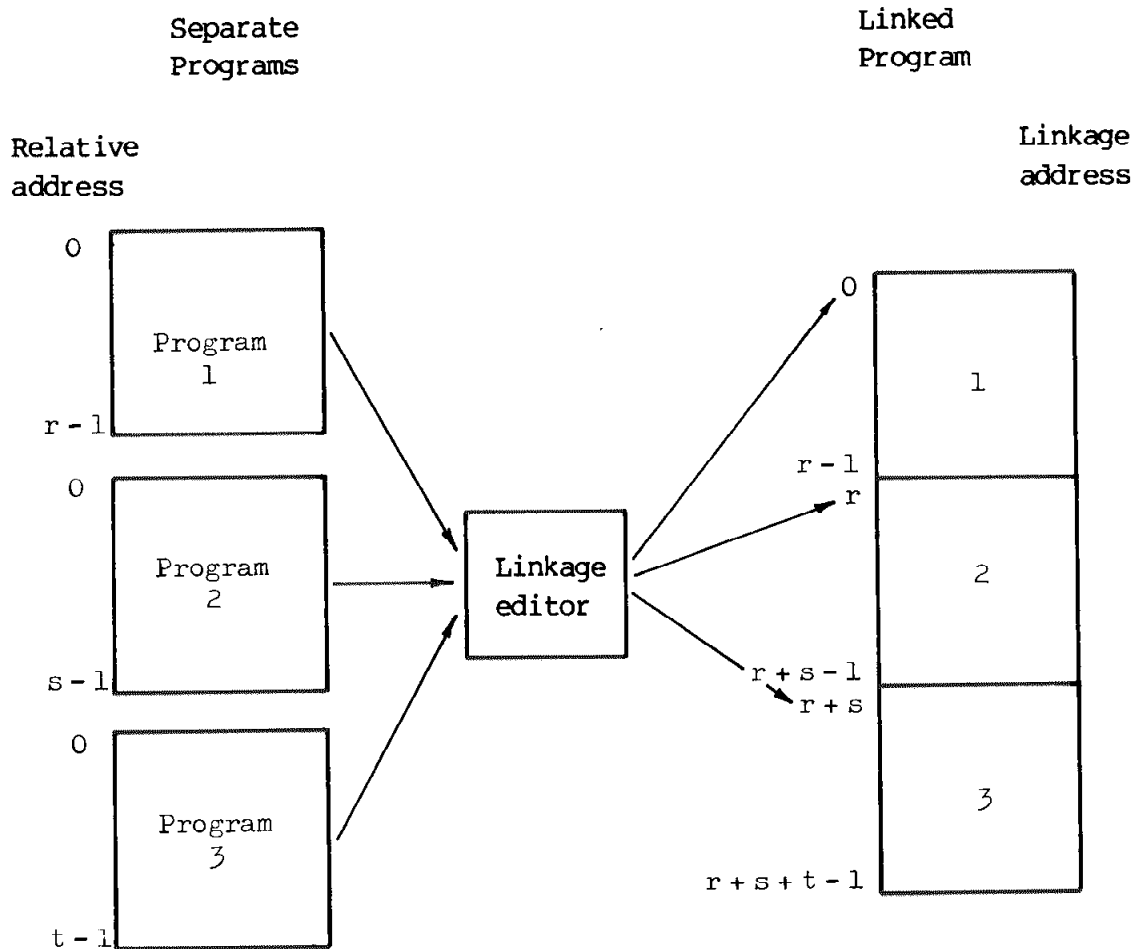
#### 1.2.4 Linkage editing

If the option is taken up, the assembly process will yield a binary relocatable object program, which in its turn may be loaded into the computer for actual execution.

When the object program is about to be loaded, it must however appear to the loading program as one complete program conforming to the blocking structure previously mentioned. Consequently all individual segments of some specific source program must be assembled together and cannot be separately assembled. This situation is inconvenient, as different programs often will employ identical program segments, and the restriction mentioned above means that a considerable amount of assembly runs in reality would be superfluous.

To by-pass this difficulty the concept of linkage editing has been introduced. Linkage editing makes it possible to join several, separately assembled programs into one complete - and usually more elaborate - program.

The program performing this duty - the linkage editor - will also ensure that the individual programs to be joined together are given properly sequenced addresses. The principle is shown in the diagram below:



## 2 Syntactic Elements

The assembly process depends on the ability of the assembler to recognize the individual syntactic elements of the language as well as constructions of a more elaborate structure involving these basic elements. Furthermore the assembler must translate these elements and constructions into the proper machine code and as a by-product recognize and keep track of syntactic errors in the original text.

This makes it imperative to employ very concise definitions and a completely un-ambiguous syntax. This chapter and the following ones are intended to provide an explanation of these formal rules.

The basic syntactic elements of the assembly language are divided into four specific classes, viz.:

- Terminals
- Numbers
- Symbols
- Special elements

### 2.1 Terminals

The class of terminals contains those elements whose function is to separate numbers and symbols from other numbers and symbols, and it is further divided into two subclasses: operators and break characters.

#### 2.1.1 Operators

The subclass "operators" contains those elements that are used to separate numbers and symbols - jointly classified by the term "operands" - from each other while at the same time indicating, that some type of operation involving the operands is to be performed. The sequence of numbers, symbols and operators then becomes an "expression". Operators may be of type arithmetic, logical or relational; the characters used to indicate the different types of operation are as follows:

## Arithmetic operators:

B Bit alignment (shift) operator  
 + Addition  
 - Subtraction  
  
 \* Multiplication  
 / Division

## Logical operators:

& And  
 ! Or

## Relational operators:

< Less than  
 <= Less than or equal to  
 == Equal to  
 >= Greater than or equal to  
 > Greater than  
 <> Not equal to

The shift operator B is identical to the normal alphabetic character, which might appear in a symbol on which the bit shift should be performed. To avoid confusion between these two possible uses of that character, the following rule apply: B is taken to be the shift operator if the syntactic element immediately preceding it is a single precision integer number or if the character immediately preceding it is a right parenthesis.

## 2.1.2

Break Characters

The subclass break characters contains those elements that are used exclusively to indicate separation between other syntactic elements and consists of the following:

- △ This character represents the subclass space i.e. a space, a comma, a tabulation or any number and combination of these individual elements.
- : Colon is used to indicate, that the symbol immediately preceding the colon is a user-symbol, i.e. user-symbol:
- = The equal sign is similiary used to indicate, that the symbol immediately preceding it is a user-symbol, i.e. user-symbol=

The actual user symbol and the colon or equal sign may of course be separated by the class of spaces,  $\Delta$ , without inferring that  $\Delta$  is then taken to be the user-symbol.

- ( ) Parentheses are used to enclose a symbol or an expression and thereby change the order of precedence of operators as more fully explained in chapter 3.
- [ ] Square brackets are used to enclose the actual arguments of a macro call.
- ; Semicolon indicates the beginning of a comment and thus leads to input being read in the string mode.
- ↓ Carriage return indicates the end of a line
- ⇓ Form feed similarly indicates the end of a page.

## 2.2 Numbers

The DOMAC Macro Assembler accepts three types of numbers:

Integer numbers, single precision.  
 Integer numbers, double precision.  
 Floating point numbers, single precision.

Although the RC 3600 series Central Processing Unit will only operate on single precision integers, the acceptance of all three types of numbers mentioned above is accomplished by packing double precision integers and floating point numbers into two words. This packing - and the corresponding programming measures necessary to handle such numbers - is implemented by the DOMAC assembler and thus makes it possible for the programmer to employ the full range of numbers in his programs. Certain limitations do however apply as will be explained in the following sections.

### 2.2.1 Integer number, single precision

An integer number is represented as an ordinary binary number consisting of the digits 0 and 1. When the integer number is characterized as being given in single precision it means, that one word only is used for storage of the number. As one computer word consists of 16 bits, an un-signed integer in single precision must lie in the range from 0 to  $2^{16}-1$ , which in ordinary



decimal notation will be expressed as 0 to 65535. Numbers are however often represented in octal notation, in which case the number always (in this manual) will be given the suffix <sub>8</sub> to avoid confusion with numbers in ordinary decimal notation. The range mentioned above will in octal notation be expressed as 0 to 177777<sub>8</sub>.

To represent signed integer numbers in single precision the first bit (bit 0) of the word is used as sign bit - a positive sign being indicated by 0, a negative one by 1 - while the remaining 15 bits of the word is used to represent the value in two's-complement notation. In this case then, the range of values will cover the interval from -100000<sub>8</sub> to +77777<sub>8</sub> (-32768 to +32767).

- 2.2.1.1 Source representation of single precision integer numbers may be given a variety of formats. First of all the programmer may choose any radix (number base) between 2 (i.e. binary notation) and 20 (i.e. duodecimal notation) through the use of the directive. RDX (cf. chapter 4); radices less than or equal to 10 will only require the usual arabic numerals for number representation, while radices greater than that require supplementing of the ordinary numerals with the first letters of the alphabet according to the convention illustrated in the table below:

Digit Representation	Digit Value	Radix	Digit representation	Digit Value	Radix
0	0	-	A	10	>=11
1	1	>= 2	B	11	>=12
2	2	>= 3	C	12	>=13
3	3	>= 4	D	13	>=14
4	4	>= 5	E	14	>=15
5	5	>= 6	F	15	>=16
6	6	>= 7	G	16	>=17
7	7	>= 8	H	17	>=18
8	8	>= 9	I	18	>=19
9	9	>=10	J	19	=20

Integer numbers (in single precision) must be presented for input to the assembler in a definite format as follows:

<+>d<d.....d>.break

In this description of the format the d's indicate the individual

digits of the number, which must of course correspond to the representation and radix as given in the table. The current input radix - as this has been previously determined by the programmer's use of .RDX - will be used for interpretation of the number except if the optional decimal point shown in the format above is included. In this case the number will be interpreted as given in radix 10 - regardless of current input radix.

The first digit in the number must always be a numeral in the range 0 to 9. To comply with this rule, those numbers - presupposing a radix greater than 10 - which could otherwise have an alphabetic character as its first digit, must be written with an initial 0; as an example of this rule consider the following numbers, which are written in base 10 notation first followed by the same value written in base 16 notation and finally as written in source representation to base 16:

51351	C897	0C897
58835	E5D3	0E5D3
41629	A29D	0A29D
48707	BE43	0BE43

break in the above format terminates the number and can be any character except a digit inside the range of the current radix or a decimal point. Normally break will be a character from the set of terminals, i.e. either an operator or one of the break characters Δ, ), † or ;. Note that a space will be interpreted as break. The shift operator B and the digit B occurring in numbers to base 12 and higher might also be confused. Whenever B is intended to be interpreted as the shift operator under such circumstances it must be immediately preceded by the character ←. This character will act as a break character to the number format but is otherwise ignored and correct interpretation will thus be ensured. As an example of this convention consider the following two character strings (where a current input radix of 16 has been assumed):

+0C3B9	here B will be interpreted as digit 11
+0C3 ← B9	here B will be interpreted as shift operator.

In addition to the normal integer format described above two special formats exist, which may be used to pack specific bit configurations into a word. The first of these special formats is the following:

"c

where c represents an arbitrary, single ASCII character with the exception of the three previously mentioned blind characters: NL (code 012<sub>8</sub>), DEL (177<sub>8</sub>) and NUL (000<sub>8</sub>).

When this format is applied, the character immediately following the quotation mark will be converted to its seven-bit octal value (cfr. with ASCII subset in section 1.2.2), and this value will then be stored in the rightmost byte of the word i.e. in bits 8 to 15 with bit 8 always being 0.

For example:

"↑

used as input will be converted to its octal value 136<sub>8</sub> and stored in the word as follows:

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

It should be noted, that this format can be used as an ordinary operand within an expression as in the following examples:

"?+529. yielding: 0 000 001 001 010 000 (001120<sub>8</sub>)  
 "z/2 yielding: 0 000 000 000 111 101 (000075<sub>8</sub>)  
 "&"G yielding: 0 000 101 010 001 010 (005212<sub>8</sub>)

Note however, that although "↑ will correctly be stored as that character's octal value, it will simultaneously terminate the line of source input.

The second of the special formats utilizes two single apostrophes enclosing a string of characters:

'string' <↓>

The string may contain any number of ASCII characters, but of these only the first two characters in the string will be converted to their respective octal values and stored in the word. When this format is used, the full 16 bits of the word will be used for packing; the first character being stored in the leftmost byte of the word while the second character will be stored in the rightmost byte as shown in the following examples:

```
'(number)' stored as: 0 0 1 0 1 0 0 0 0 1 1 0 1 1 1 0
'7'         stored as: 0 0 1 1 0 1 1 1 0 0 0 0 0 0 0 0
```

Note that if the digit zero is stored in this format, the word will not contain absolute zero:

```
'00'        stored as: 0 0 1 1 0 0 0 0 0 0 1 1 0 0 0 0
```

To generate a word containing zeroes in all bit positions the apostrophes must be written without any character in between:

```
"          stored as: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

This second special format may of course be used under exactly the same circumstances as the first, i.e. as an ordinary operand within an expression or wherever an integer number is allowed. Note that the string will be terminated if a carriage return occurs before the second apostrophe; in analogy with the first special format the value of the carriage return character will be stored in the word provided that this character appears as the first or second character following the initial apostrophe.

The following examples illustrate the various effects possible with these formats:

```
"?      yields: 0000778
'r      -   : 0001628
":      -   : 0000728
""      -   : 0000428
'?      -   : 0374008
' &'    -   : 0200468
'"?'    -   : 0210778
'r'     -   : 0710008
'+|'    -   : 0254158
'?-' :r' -   : 0022168
```

### 2.2.2 Integer numbers, double precision

To increase the range of numbers, which may be handled by the computer, integers may be represented in double precision. This means, that the number will be stored in two consecutive words and that these words will be accepted as one 32-bit binary number. The range will thus be extended to encompass the numbers

from 0 to 4294967295 (0 to  $3777777777_8$ ). When representing signed integer numbers, the first bit of the first word (bit 0 of the high-order word) is used as sign bit in exactly the same way as for single precision representation, giving a range from -2147483648 to +2147483647. As for single precision integers, negative numbers are represented in two's-complement notation.

In contrast to integers in single precision, those in double precision are limited to appear in data statements only.

- 2.2.2.1 Source representation of double precision integer numbers is in most respects analogous to that of single precision integers, although the limited field of application warrants some slight changes. Radices of integer numbers in double precision may lie in the range from 2 to 20 and the representation of numerals follows the convention given for single precision (section 2.2.1.1).

The input format of double precision integers is:

`<+>d<dd....d><.>D break`

As before the d's in this format indicate the individual digits of the number, and the optional decimal point results in interpretation of the number as given in radix 10.

The first digit in the number must - as before - be a numeral in the range 0 to 9; thus an initial zero must be used according to circumstances when writing numbers to a radix greater than 10.

break is the terminal character and will normally be one of the characters

Δ ; ↓

Note that as double precision integers may only occur in data statements, the class of operators is without meaning in context with such numbers and consequently cannot be used for break. (If it should happen a format error will be indicated during assembly). The character D immediately preceding the break character is a special feature of this format and indicates that the number is an integer in double precision format. The use of this letter may lead to conflict with its use as a digit where numbers to base 14 or above are concerned. This conflict is avoided by adopting the previously mentioned convention as shown in the examples immediately below (where a current input radix of 16 has

been assumed):

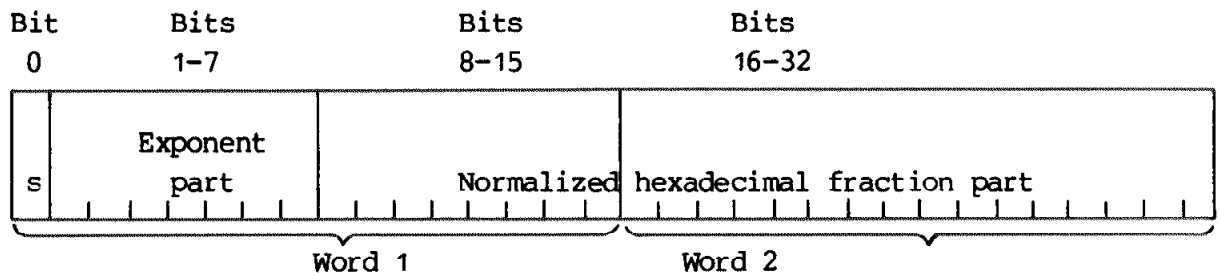
+148D ;D will be interpreted as the digit "13"  
 +148-D ;D will be interpreted as signifying a double  
 precision format.

Some further examples of data statements involving double precision integers are listed below - together with the assembled values of the numbers given in octal representation - assuming current input radix = 8:

+4112767D	000020 <sub>8</sub>	112767 <sub>8</sub>
-4112767D	177757 <sub>8</sub>	065011 <sub>8</sub>
+4112767.D	000076 <sub>8</sub>	140577 <sub>8</sub>
-4112767.D	177701 <sub>8</sub>	037201 <sub>8</sub>

### 2.2.3 Floating point numbers, single precision

A floating point number is represented in exponent form, that is: it is considered as the product of a proper fraction and the number 16 raised to a integer power. Floating point numbers utilize two consecutive words in storage; such a double word - consisting of 32 bits - is subdivided into three parts in the following manner:



bit 0 is the sign bit, 0 for positive, 1 for negative sign

bits 1 to 7 hold the exponent part of the number

bits 8 to 31 hold the fractional part of the number

The seven-bit exponent part is an ordinary binary number in the range from 0 to 177<sub>8</sub>, but to cope with negative exponents it is assumed, that the actual number appearing in bits 1 to 7 must be added to 100<sub>8</sub> to yield the correct value of the exponent. By using two's-complement notation with an implied sign bit in bit 1 the actual contents of the exponent part will lead to the interpretation shown in the following list:

Exponent=000 <sub>8</sub>	Actual exp.=100 <sub>8</sub>	(decimal: -64)
"- =026 <sub>8</sub>	"- - =126 <sub>8</sub>	("- : -42)
"- =100 <sub>8</sub>	"- - =000 <sub>8</sub>	("- : 0)
"- =126 <sub>8</sub>	"- - =026 <sub>8</sub>	("- : +22)
"- =177 <sub>8</sub>	"- - =077 <sub>8</sub>	("- : +63)

On account of the fact, that 16 has been chosen as the base of the exponent part of the number, the fractional part of the number must be a hexadecimal fraction; consequently the 24 available bits are viewed as six hexadecimal digits of four bits each, which means, that the fractional part would normally be in the range:

$$.000001 < \text{fraction} < .FFFFFF$$

However, most routines handling floating point numbers are designed to work on the assumption, that all floating point operands (except zero) are normalized, that is: by suitable multiplication of the fractional part and consequent adjustment of the magnitude of the exponent part the number format is rearranged so that the first digit of the fraction is a non-zero digit. Consequently the fractional part will actually be inside the range:

$$.100000 < \text{fraction} < .FFFFFF$$

All floating point conversions by the DOMUS Macro Assembler are normalized.

Negative or positive numbers are indicated by the sign bit exclusively; the exponent part and the fractional part are un-altered by the change of sign. Zero is represented as true zero, i.e. a double word containing 0 in all bit positions. The floating point representation as given above will lead to a numerical range from

$$.100000 \cdot 16^{-64} \text{ to } .FFFFFF \cdot 16^{+63}$$

which in decimal notation is approximately:

$$5.40 \cdot 10^{-79} \text{ to } 7.24 \cdot 10^{+75}$$

As with double precision integers, floating point numbers are restricted for use in data statements only.

2.2.3.1 Source representation of floating point numbers exhibit two basic formats, which can be described in general terms by the following:

$$\begin{aligned} &\langle + \rangle \underline{d} \underline{dd} \dots \underline{d} \rangle . \underline{d} \underline{dd} \dots \underline{d} \rangle \langle E \langle + \rangle \underline{d} \underline{d} \rangle \rangle \text{break} \\ &\langle + \rangle \underline{d} \underline{dd} \dots \underline{d} \rangle E \langle + \rangle \underline{d} \underline{d} \rangle \text{break} \end{aligned}$$

Note that in the first of these formats the E is optional owing to the fact, that his format employs a decimal point, whereas in the second format the E is obligatory (if it is omitted there will be no indication, that a floating point number is intended). The d's in the above formats are numerals in the range from 0 to 9. Implied here - as in the use of the decimal point - is the fact that floating point numbers will always be interpreted as given to radix 10.

As before, break is a terminal character and will usually be one of the terminals

Δ ; ↓

whereas operators are not allowed as terminal characters for floating point numbers.

Some examples of floating point numbers are given below - including examples of the alternative use of the two formats:

-1.3	-13E-1
4.121	4121E-3
+0.96	+96E-2
65200.0	652E+2
-10000.0	-1E4
374.1E+8	+3741E+7
+3.11E-18	311E-20

Although floating point numbers are always interpreted as being in decimal notation irrespective of current radix, the appearance of an E - as for inst. in the number -1E4 above - may still lead to confusion, if current input radix is 15 or above. As before this difficulty is avoided by the use of the ← convention:

- 1←E4



## 2.3 Symbols

One of the primary functions of the DOMUS Macro Assembler is the identification and interpretation of symbols - including of course those basic symbolic elements already mentioned such as numbers etc. As this function however is closely tied in with the syntactical structure of the language, a more detailed discussion of this topic will be deferred to the chapter dealing explicitly with syntax (chapter 3).

## 2.4 Special Elements

Five special elements are ignored by the assembler while the line-by-line reading of the source program is taking place and will only have any effect after the scan of the whole line has been completed. The five elements are the following:

- @ used to indicate indirect addressing
- # used to cause the no-load bit to be set
- \*\* used to cause suppression of listing of a whole line
- ++ used to cause suppression of listing of a part line
- ← used to alter interpretation of succeeding character

The effect of each of these five special elements will be:

### 2.4.1 The character @

This character is meaningful in connection with Program Flow Control instructions, Data Transfer instructions or in lines containing expressions. If the character @ - or any number of such characters - appears anywhere in a line containing a Program Flow Control or a Data Transfer instruction, it will cause a digit 1 to be stored in bit 5 of the instruction when assembly of the instruction itself has been effected. When the object program is executed, the instruction thus modified will be taken to contain an indirect address.

If the character @ - or any number of such characters - appears anywhere in a line before an expression, it will cause bit 0 of that data word, which holds the result of the calculation, to be set to 1. In data words bit 0 has exactly the same significance as bit 5 of Program Flow Control instructions or Data Transfer instructions, i.e. addressing will take place in indirect mode when that particular word is referenced during execution of the object program.

#### 2.2.4 The character #

This character is meaningful in connection with Arithmetic and Logical Instructions. If the # character - or any number of such characters - appears anywhere in a source line containing an Arithmetic and Logical Instruction, it will - when the rest of the instruction has been assembled - cause a 1 to be stored in bit 12 of the resulting word. This bit is the no-load bit, and a 1 in this bit will mean, that the result of the operation is not loaded into an accumulator.

#### 2.4.3 The characters \*\*

Two or more consecutive asterisks appearing anywhere in a source line will cause suppression of listing of that specific line. Note however, that within macro definitions there will be no such effect.

#### 2.4.4 The characters ++

Two or more consecutive plus signs appearing anywhere in a source line will cause suppression of listing of that part of the line which follows this special element. Note however, that within macro definitions there will be no such effect.

#### 2.4.5 The character ←

The reverse arrow is itself blind, that is: this character will be unconditionally ignored by the assembler, but its appearance anywhere in a source line will cause the character immediately following the arrow to be interpreted in string mode (q.v.).

## 3 Syntax

The concept of syntax covers the structural relationship, that must - in any language - exist between the individual components of the language in order to ensure correct conveying of information. In computer languages the demands on syntax are even more severe than those posed by our everyday vocal languages due to the increased necessity of avoiding any ambiguity. This makes the proper understanding of the syntactical rules of the DOMUS Macro Assembler language, as explained in this chapter, mandatory for correct application to problem solving.

The preceding chapter covered the rules pertaining to the basic elements of the language; this chapter extends this to more complex elements and their combination into "sentences", and treats the three concepts of: symbols, expressions and instructions.

### 3.1 Symbols

Symbols are used for two purposes, viz. to ensure some specific action on the part of the assembler and to represent a - possibly variable - numerical value. Symbols are classified into three distinct groups: Permanent, Semi-permanent and User symbols; all three groups however use the same format in source representation as follows:

a<bb.....b>break

In this description of the format a is one of the characters of the alphabet A to Z, the full stop . or the interrogation sign ?. Similarly b is any one of the same characters A to Z, . or ?, but includes also the numerals 0 to 9.

break is the terminating character and may be any character with the exception of the previously mentioned characters.

Note that there is no set length of the symbol string according to the format, but the assembler will only recognize the first five characters of the format; all characters in excess of five will be ignored.

### 3.1.1 Permanent symbols

Permanent symbols are defined inside the assembler itself, i.e. the sequence of characters in such symbols have an inherent meaning to the assembler and cannot in any way be changed or used to indicate anything but that particular symbol.

Permanent symbols are however used for the two purposes previously mentioned, viz. to direct the assembly process or to represent numerical values.

Those permanent symbols used to direct the assembly process are called "directives" and perform a multiplicity of roles, such as setting the input radix, initiating text strings etc. The complete set of available directives is listed and described in detail in the following chapter.

Some permanent symbols may in addition to their function as directives also represent numerical values of internal assembler variables; others may only have the last-mentioned function. If a permanent symbol can be used in both ways, the assembler will determine the intended use through the context in which the symbol appears. In doing so the following rules apply:

If the first syntactic element of a line is a directive it will be used to direct the assembler process.

If a directive occurs in any position of the line but the first, the symbol value will be used.

Examples of the use of these rules will be included in the detailed description contained in chapter 4.

### 3.1.2 Semi-permanent symbols

The class of semi-permanent symbols is of extreme importance as is implied by its alternative designation: operation codes. Semi-permanent symbols can be defined by the use of appropriate directives following which they may be saved and used during later runs of the assembler without any need of renewed definition. As supplied by RC the assembler contains a set of semi-permanent symbols, a list of which is included in appendix A. These semi-permanent symbols are defined in such a way, that they correspond to the RC instruction set, but if he so wishes, the user may eliminate this set in part or as a whole and define his own set of semi-permanent symbols, or he may retain the set and extend it with his own symbols.

### 3.1.3 User symbols

All symbols - defined by the programmer in the source program - which do not correspond to a permanent or semi-permanent symbol, will be classified as user symbols. Such symbols are used extensively in programming and for a variety of purposes: to assign a name to a memory address or a numeric value to a variable parameter, to name external values etc. User symbols are retained throughout the assembly process in the symbol table, so that once defined they can be referred to anywhere in the program, but they cannot be saved, and their possible re-use in other programs is thus dependent on renewed definition of the symbol. User symbols may be defined as either local or global; this has a bearing on the actual value of the symbol at different stages of the process. The value of a local symbol is only known to the assembler during the assembly process; the value of a global symbol is known also at linkage edit time, which makes it possible to use such globally defined symbols for communication between different program modules.

### 3.1.4 Symbol table listing

As previously mentioned (cf. section 1.2.3) the program listing optionally output by the assembler may include a cross-reference listing of the symbol table mentioned above. This cross-reference listing will include all user symbols defined during assembly. Semi-permanent symbols will not normally be included in the cross-reference listing unless this has been explicitly requested by specifying MODE.A or MODE.R in the DOMAC load command (cf. Operating Procedures, chapter 7). Permanent symbols are never included in the cross-reference listing.

## 3.2 Expressions

Expressions are combinations of basic elements whose purpose is to indicate a sequence of operations to be performed on the elements appearing in the expression. The format of an expression is:

$$\langle \underline{\text{operand}}_1 \rangle \quad \underline{\text{operator}} \quad \underline{\text{operand}}_2$$

In this format,  $\underline{\text{operand}}_1$  and  $\underline{\text{operand}}_2$  may take either of three possible forms, viz.:

an integer number in single precision

a symbol

an expression (evaluating to a single precision integer)

Note that expressions are defined recursively by inclusion of expressions as operands.

Note also, that an operand normally must precede the operator. Only the unary operators + and - may precede an expression (or follow another operator without any intervening operand) and thus operand<sub>1</sub> is optional only this case.

The assembler operators have been listed already, but the list is repeated here for convenience:

```

B   Bit alignment (shift) operator
+   Addition (plus)
-   Subtraction (minus)
*   Multiplication
/   Division
&   And
!   Or
<   Less than
<=  Less than or equal to
==  Equal to
>=  Greater than or equal to
>   Greater than
<>  Not equal to

```

### 3.2.1 Operator hierarchy

The recursive definition of the concept of expression means, that actual expressions may contain several operands and operators in sequence. The operators appearing in any one expression may be of more than one type (i.e. arithmetical, logical or relational), and to ensure that evaluation of expressions proceeds without ambiguity the following rules apply:

Operators are ranged in three levels of precedence, which are numbered 1, 2 and 3, and of which level 1 takes precedence over level 2, which will again take precedence over level 3.

Expressions involving operators in more than one level of precedence will be evaluated in the following way: all operations indicated by an operator in level 1 will

be performed first; thereafter all operations indicated by operators in level 2 will be performed and finally all operations involving level 3 operators conclude the evaluation.

Expressions or parts of expressions involving operators of only one level of precedence will be evaluated from left to right.

The order of evaluation implied in the levels of precedence of the operators may be changed by enclosing an expression or parts of an expression in parentheses; those expressions or parts of expressions enclosed in parentheses are evaluated prior to any remaining parts of the complete expression.

Parts of an expression enclosed in parentheses are themselves evaluated subject to the above rules. (This means that nesting of parentheses is allowed.)

The levels of precedence of the individual operators are as follows:

```
Level 1 (highest): B
Level 2:           + - * / & !
Level 3 (lowest): < <= = > > <
```

There is no check for overflow during evaluation of an expression. Expressions (or parts of expressions) involving relational operators will as the result of the evaluation yield either absolute zero (false) or absolute one (true). This is shown in the following examples where the values  $X=7$  and  $Y=5$  have been assumed:

$X==Y$	yields	000000
$X>Y$	-	000001
$X-Y+3< >Y$	-	000000
$X&Y==2$	-	000000
$(X==Y)!(X-Y>=0)$	-	000001

The last two examples above are also examples of the use of the logical operators  $\&$  and  $!$ . While the last example is fairly straightforward in the sense, that the two relational expressions enclosed in parentheses each yield a logical value and thus in reality become logical operands for the  $!$  - operator, the last example but one is a little more unusual. The situation becomes

clear, however, when it is taken into account, that the logical operations & and ! are subject to the following rules:

The logical operations are effected as a bit-by-bit comparison of the two sixteen-bit operands.

The logical AND (&) will yield the result 1 in a specific bit position if and only if both operands have a digit 1 in that bit position.

The logical OR (!) will yield the result 1 in a specific bit position if either of the operands or both have a digit 1 in that bit position.

These rules can be illustrated by the following examples:

$054625_8 \& 063571_8$  yields  $040421_8$

To realize this, consider the bit configurations:

$$\begin{array}{r} 054625_8 = \quad \quad \quad 0\ 101\ 100\ 110\ 010\ 101 \\ 063571_8 = \quad \quad \quad 0\ 110\ 011\ 101\ 111\ 001 \\ \hline 054625_8 \& 063571_8 = 0\ 100\ 000\ 100\ 010\ 001 = 040421_8 \end{array}$$

$054625_8 ! 063571_8$  yields  $077775_8$

Similarly:

$$\begin{array}{r} 054625_8 = \quad \quad \quad 0\ 101\ 100\ 110\ 010\ 101 \\ 063571_8 = \quad \quad \quad 0\ 110\ 011\ 101\ 111\ 001 \\ \hline 054625_8 ! 063571_8 = 0\ 111\ 111\ 111\ 111\ 101 = 077775_8 \end{array}$$

The rules governing the *modus operandi* of the bit alignment (shift) operator B also merits a more detailed explanation: This operator will shift the entire bit configuration of operand<sub>1</sub> a number of positions to the left; in doing this the high-order bits of the original operand<sub>1</sub> will be lost as they "overflow" the leftmost bit position, while the low-order bits of the original operand<sub>1</sub> will be replaced by zeroes in the rightmost bit positions. The number of places, which the bit configuration will be shifted, is indicated by operand<sub>2</sub>, the value of which indicates the bit position to which the original bit 15 has been shifted. To clarify the operation, consider the number:  $013043_8$ , which will have the following bit configuration:



0 1 3 0 4 3 :

0 001 011 000 100 011

then:

0 1 3 0 4 3 B 9 yields:

1 000 100 011 000 000

In this example it has been assumed, that a current radix 8 is in force, but it should be noted, that even then only operand<sub>1</sub> will be interpreted to this radix, while operand<sub>2</sub> regardless of this will be interpreted to radix 10. If it is desired to have operand<sub>2</sub> interpreted in current radix it must be enclosed in parentheses; thus the same result as above will occur if the expression is given the form:

0 1 3 0 4 3 B (1 1)

Due to the fact, that to shift a binary number one bit position to the left corresponds exactly to a multiplication by 2, the result of the shift operation can also be given as:

$$\text{operand}_1 * 2^{(15 - \text{operand}_2)}$$

Where operand<sub>2</sub> (as mentioned before) is assumed to be taken to radix 10, and where \*\* denotes exponentiation.

Examples of the shift operation are given below. In these examples it has been assumed, that current radix = 8, and they show the result of the operation in binary as well as in octal representation:

13043B15	0 001 011 000 100 011	013043 <sub>8</sub>
13043B12	1 011 000 100 011 000	130430 <sub>8</sub>
13043B10	1 100 010 001 100 000	142140 <sub>8</sub>
13043B6	0 100 011 000 000 000	043000 <sub>8</sub>
13043B3	0 011 000 000 000 000	030000 <sub>8</sub>
13043B0	1 000 000 000 000 000	100000 <sub>8</sub>

The two operands on either side of the B operator may of course themselves be expressions, as in the examples below, where the following values have been assumed:  $X = 13_8, Y = 15_8$

(X+25)B(Y*3-37)	010000 <sub>8</sub>
(X+25BY)*3-37	000376 <sub>8</sub>
(X+25)BY*3-37	000541 <sub>8</sub>
(X+25)BY*(3-37)	171000 <sub>8</sub>

It has been mentioned before, that the shift operator may be mistaken for a digit; similarly it may be mistaken for a (part of a)

symbol appearing as operand<sub>1</sub>. In such cases the ← convention may be used, or the number or symbol preceding the shift operator may be enclosed in parentheses, as in the following example:

$$(X)B(Y * 3 - 37).$$

Further examples of evaluation of expressions are given below (where the values of X and Y are the same as above):

$X*(Y-10)/Y$	000006 <sub>8</sub>
$X*Y-Y/10$	000216 <sub>8</sub>
$(15+X)*(Y-10)+27/X$	000172 <sub>8</sub>
$((15+X)+Y)/3+Y)*7$	000257 <sub>8</sub>
$15+X+Y/3+Y*7$	000167 <sub>8</sub>

### 3.2.2 Relocation Characteristics

It has previously been mentioned, that in addition to a numerical value symbols (and addresses) have associated to them a relocation characteristic, which will ultimately come into effect at the time when the object program is loaded into core memory prior to execution.

During assembly the relocation characteristic will determine which of the three available program counters should be utilized for addressing purposes at any given instant. This obviously indicates, that relocation characteristics of a specific expression cannot be completely free from restrictions, and consequently the following rules must be observed:

An expression cannot simultaneously contain both normal relocatable and page zero relocatable operands.

An expression can simultaneously contain absolute relocatable and either normal relocatable or page zero relocatable operands (but not both) subject to the restrictions implied in the following example, in which are also listed the resulting relocation characteristic of the expression as well as the numeric fields and the flags given in the corresponding output listing:

```

000012      .LOC      10.      ;
000007      ABS=      7        ; ABSOLUTE RELOCATABILITY
           .NREL
000003'     REL=      .+3      ; RELOCATABLE
00000'000016  ABS+ABS      ; RESULT ABSOLUTE
00001'000012' ABS+REL      ; RESULT RELOCATABLE
00002'000006" REL+REL      ; RESULT BYTE RELOCATABLE
00003'000000  ABS-ABS      ; RESULT ABSOLUTE
00004'177774' REL-ABS      ; RESULT RELOCATABLE
00005'000000  REL-REL      ; RESULT ABSOLUTE
00006'000061  ABS*ABS      ; RESULT ABSOLUTE
00007'000001  ABS/ABS      ; RESULT ABSOLUTE
00010'000007  ABS&ABS      ; RESULT ABSOLUTE
00011'000007  ABS!ABS      ; RESULT ABSOLUTE
00012'003400  (ABS)B(ABS)   ; RESULT ABSOLUTE

00013'000000  REL==ABS      ; RESULT FALSE
00014'000001  REL<>ABS     ; RESULT TRUE

```

All operations involving the relational operators will always have a result of absolute relocatability. If the operands are not of the same relocation characteristic, the result of the evaluation will always be "false", except if the operator is <> in which case the result will be given as "true". Note that the actual values of the operands will in this situation be of no consequence.

In addition to those rules of application shown in the example above, one other possibility exists. This is a non-standard extension of the concept of relocatability and is embodied through the medium of the .EXTA directive; further details of this must be sought in the description of the said directive in section 4.6 of this manual.

All other possible combinations of relocation characteristics of operands than those already mentioned are unconditionally illegal.

Note: The present versions of the MUS/DOMUS operating systems for the RC 3600 series computers do not include implementation of page zero relocation characteristics.

### 3.3 Instructions

Basically any program consists of a sequence of instructions, whose function is to convey information to the computer about the operations to be performed. Evidently the information must be made available to the computer in a form compatible with the capabilities of the computer, that is: as a sequence of binary digits.

Instructions are formatted into a number of fields, which in the course of the assembly process are translated into one (or two) word(s) of 16 bits. The first field of the instruction format is a semi-permanent symbol - called the instruction mnemonic - after which follows a number of fields separated by space, comma or tabulation characters. The fields following the instruction mnemonic must of course correspond in number and type to the requirements of the format specified for the actual instruction category.

The RC 3600 series computer will recognize a series of basic instructions, which will fall into a number of categories. To each category a corresponding directive exists, enabling definition of semi-permanent symbols within the various categories. These are:

Instruction category	Defining directive
Arithmetic and Logical (AL)	.DALC
Extended AL (2 accumulators, no skip)	.DISD
Program Flow Control	.DMR
Data Transfer	.DMRA
Count and Destination	.DICD
Input/Output (with accumulator)	.DIOA
Input/Output (without accumulator)	.DIO
Central Processor Functions (1st type)	.DIAC
Central Processor Functions (2nd type)	.DUSR
Extended Operation	.DXOP

#### 3.3.1 Arithmetic and Logical Instructions

The source format of an arithmetic and logical instruction is:

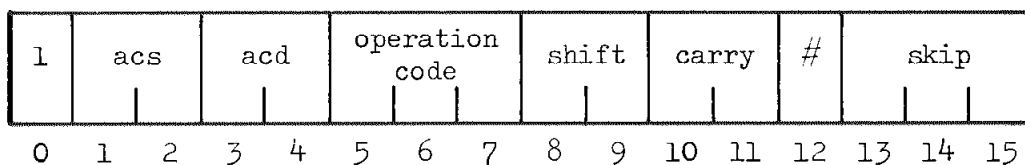
mnemonic<carry><shift><acs><acd><skip>

In this format the individual fields characterized are:

- mnemonic one of the following semi-permanent symbols:  
 ADD SUB NEG ADC MOV INC COM AND
- carry an optional mnemonic specifying the state of the carry bit
- shift an optional mnemonic indicating shift action
- acs one of the digits 0, 1, 2 or 3 indicating which accumulator is to be used to provide data for the operation
- acd one of the digits 0, 1, 2 or 3 indicating which accumulator is to be used to receive data from the operation
- skip an optional mnemonic indicating possible skip action

Not included in the above description of format, the character # may be added anywhere in the source line. If this is done, the assembly process will place the digit 1 in bit position 12 of the assembled instruction word instead of the digit 0. Bit 12 is the no-load bit, and a digit 1 in this position will mean, that operation output (from the shifter of the arithmetic unit) will not be loaded into the destination accumulator.

The word containing the assembled instruction will be structured in the following way:



The operation code and effect of the eight AL instructions are:

- ADD 110 Adds the contents of source accumulator and destination accumulator and places the result in destination accumulator
- SUB 101 Subtracts the contents of source accumulator from those of destination accumulator

- NEG 001 Forms the negative (two's complement) of the contents of source accumulator and places it in destination accumulator.
- ADC 100 Adds the logical complement of the contents of source accumulator to the contents of destination accumulator.
- MOV 010 Performs the shift operation indicated in the instruction on the contents of source accumulator and places the result in the destination accumulator.
- INC 011 Increments the contents of source accumulator by one and places the result in destination accumulator.
- COM 000 Forms the logical complement of the contents of the source accumulator and places the result in destination accumulator.
- AND 111 Forms the logical "and" of the contents of source and destination accumulators and places the result in destination accumulator.

The optional mnemonics controlling the "carry", "shift" and "skip" actions, the corresponding bit configurations and consequent effects are given in the following list:

Carry:

- |   |    |   |
|---|----|---|
|   | 00 | Leaves carry bit unaltered                            |
| Z | 01 | Sets carry bit to zero                                |
| O | 10 | Sets carry bit to one                                 |
| C | 11 | Sets carry bit to the complement of its present value |

Shift:

	00	No shift operation performed
L	01	Shifts the result of the operation one bit to the left before placing it in the destination accumulator
R	10	Shifts the result of the operation one bit to the right before placing it in the destination accumulator
S	11	Exchanges the two 8-bit halves of the result of the operation before placing it in the destination accumulator

Skip:

	000	No skip - proceed with the next sequential instruction
SKP	001	Always skip next sequential instruction
SZC	011	Skip next sequential instruction if carry bit equals zero
SNC	011	Skip next sequential instruction if carry bit equals one
SZR	100	Skip next sequential instruction if result equals zero
SNR	101	Skip next sequential instruction if result unequal to zero
SEZ	110	Skip next sequential instruction if either carry or operation result (or both) equals zero
SBN	111	Skip next sequential instruction if both carry and operation result are unequal to zero

NOTE:

More detailed information on the organization of arithmetic and logical operations and the associated instructions can be found in: RC 3603 Programmer's Reference Manual

### 3.3.2 Program Flow Control Instructions

The source format of program flow control instructions may be given either of the following two forms:

mnemonic Δ displacement Δ mode  
mnemonic Δ address

In these formats the individual fields characterized are:

mnemonic one of the following semi-permanent symbols:

JMP JSR ISZ DSZ

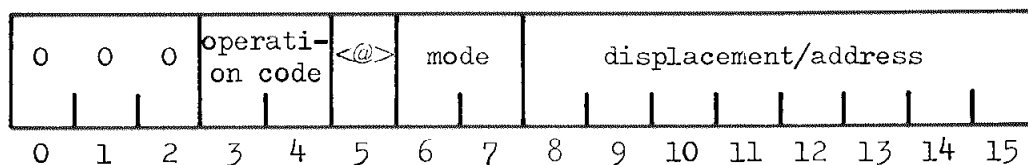
displacement any legal expression which will produce an eight-bit integer between  $-200_8$  and  $+177_8$

mode one of the digits 0, 1, 2 or 3 indicating which mode of effective address calculation is to be applied

address any legal expression which will produce an eight-bit integer in either of the following two intervals: 0 to  $+377_8$  or  $-200_8$  to  $+177_8$

Not included in the above description of format, the character @ may be used as a break character anywhere in the source line. If this is done, the assembly process will result in the digit 1 being placed in bit position 5 of the assembled word instead of the digit 0. Bit 5 is the indirect addressing bit, and the digit 1 in this position will mean, that the address given in the instruction contains in itself the address to be used during execution of the instruction. (This address may also be an indirect address, thus continuing the addressing chain.)

The word containing the assembled instruction will be structured in the following way:





The operation code and effect of the four program flow control instructions are:

- JMP 00      Loads the effective address into the program counter, and thus executes an unconditional jump to this address.
- JSR 01      Increments current value of program counter by one and loads this into accumulator 3; subsequently loads the effective address into the program counter and thus executes a jump to this address.
- ISZ 10      Increments contents of effective address by one. If the result of this incrementation is zero, the next sequential instruction is skipped.
- DSZ 11      Decrements contents of effective address by one. If the result of the decrementation is zero, the next sequential instruction is skipped.

3.3.2.1 Addressing The action of program flow control instructions depends in all cases on the result of a calculation of effective address according to the information contained in the instruction. Calculation of effective address may be carried out in slightly different ways as outlined in the following section.

If the format of the source instruction corresponds to the first of the two forms given on the preceding page, the addressing mode will be directly included in the instruction, and will have the following effect:

mode = 0: Page zero addressing. Index bits will be 00. In this addressing mode the number given in displacement is taken directly to be the effective address. Thus the effective address will lie in the range from  $0_8$  to  $377_8$ . This first block of 256 ( $400_8$ ) words in the CPU memory is known as page zero.

mode = 1: Relative addressing. Index bits will be 01. In this addressing mode the effective address is found by adding the number given in displacement to the contents of the program counter. As the program counter contains the address of the instruction currently being executed, the effective address calculated will lie in a block of  $400_8$  words distributed evenly on either side of the current instruction.

mode = 2: Index register addressing. Index bits will be 10. In this addressing mode the effective address is found by adding the value of displacement to the contents of accumulator 2. As the contents of accumulators are not subject to any limitation, the effective address may be any address inside 64K words of core memory.

mode = 3: Index register addressing. Index bits will be 11. In this addressing mode the effective address is found by adding the value of displacement to the contents of accumulator 3.

If the format of the source instruction corresponds to the second of the two forms given previously, the addressing mode will depend on the value of address and will be determined according to this by the assembler.

If the value of address lies in the interval (program counter  $-200_8$ )  $<$  address  $<$  (program counter  $+177_8$ ), the assembler will set index to 01 and proceed as outlined for addressing mode = 1, that is: the displacement field of the assembled instruction word will be given the value (address - value of program counter). If address is not in the interval as stated above, but is a value between 0 and  $377_8$ , the assembler will set the index bits to 00 and determine the displacement field of the assembler instruction according to the following rules:

If address is absolute, the displacement field is set to address

If address is assembled with the directive .ZREL, and thus page zero relocatable, the displacement field is set to address with page zero relocation. The line will be flagged with a minus sign in column 16 of the program listing

If address is assembled with the directive .EXTD, that is: externally defined, the displacement will be set to the externally specified value. The line will be flagged with a Dollar sign in column 16 of the program listing.

All addressing mentioned above tacitly assumes, that indirection is not in force. If it is, the information will still apply with the modifications consequent of indirect addressing.

If the expressions yielding address or displacement of the source

instruction does not produce a value within the ranges mentioned above, the assembly will produce an A code (addressing error) in the error listing.

NOTE: Further information about addressing can be found in:  
RC 3603 Programmer's Reference Manual.

### 3.3.3 Data Transfer Instructions

The source format of data transfer instructions may be given either of the following two forms:

mnemonic Δ accumulator Δ displacement Δ mode  
mnemonic Δ accumulator Δ address

In these formats the individual fields characterized are:

mnemonic one of the following semi-permanent symbols:

LDA STA

accumulator one of the digits 0, 1, 2 or 3 indicating the accumulator to receive or provide the data to be transferred

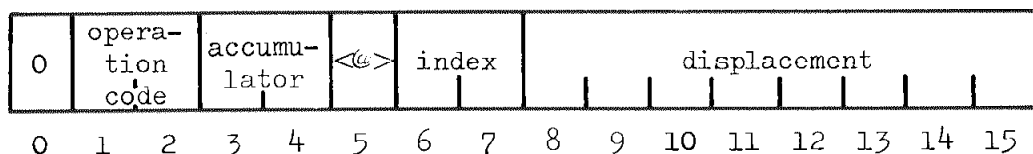
displacement same as for program flow control instructions

mode same as for program flow control instructions

address same as for program flow control instructions

As in the description of the program flow control instructions, the character @ may be used as a break character anywhere in the source line of data transfer instructions and with the same effect.

The word containing the assembled instruction will be structured in the following way:



The operation code and effect of the two data transfer instructions are:

- LDA 01 Loads the contents of the memory location identified by the effective address calculation into the accumulator specified in the instruction
- STA 10 Stores the contents of the accumulator specified in the instruction in the memory location identified by the effective address calculation.

As previously mentioned, addressing follows the same rules as those given for program flow control instructions.

### 3.3.4 Input/Output Instructions

Input/output instructions come in four different forms, viz.:

- Input/output instructions with accumulator
- Input/output instructions without accumulator
- Input/output instructions without device code
- Input/output instructions without arguments

The various forms of I/O instructions will be discussed in the following section.

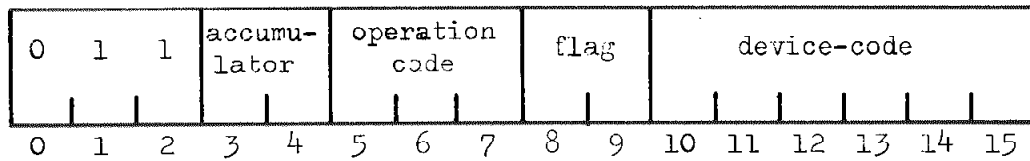
#### 3.3.4.1 Input/output instructions with accumulator. The source format of instructions of this type is:

mnemonic<flag> Δ accumulator Δ device-code

In this format the individual fields characterized are:

- mnemonic one of the following semi-permanent symbols:  
DIA DIB DIC DOA DOB DOC
- flag an optional mnemonic indicating the device status
- accumulator one of the digits 0, 1, 2 or 3 indicating the accumulator to receive or provide data for input/output
- device-code any legal expression which will produce a six-bit integer specifying the input/output device

The word containing the assembled instruction will be structured in the following way:



The operation code and effect of the six input/output instructions of this type are:

- DIA 001 Places the contents of the "A" input buffer on the selected device in the accumulator specified in the instruction
- DIB 011 Places the contents of the "B" input buffer on the selected device in the accumulator specified in the instruction
- DIC 101 Places the contents of the "C" input buffer on the selected device in the accumulator specified in the instruction
- DOA 010 Places the contents of the accumulator specified in the instruction in the "A" output buffer of the selected device
- DOB 100 Places the contents of the accumulator specified in the instruction in the "B" output buffer of the selected device.
- DOC 110 Places the contents of the accumulator specified in the instruction in the "C" output buffer of the selected device

The optional mnemonics controlling the device status flag, the bit configuration and the consequent effects are given in the following list:

	00	Does not affect the device status
S	01	Starts the device
C	10	Idles the device
P	11	Pulses the special in-out bus control line. The actual effect will depend on the device selected

The device codes for various peripheral devices are listed in appendix B.

NOTE: If input/output instructions are used with the device code  $77_8$ , the instructions will lead to some special functions being performed, among other things involving the state of the interrupt system. For more detailed information of this refer to the RC 3603 Programmer's Reference Manual.

3.3.4.2 Input/output instructions without accumulator. The source format of instructions of this type is:

mnemonic<flag>  $\Delta$  device-code

In this format the individual fields characterized are:

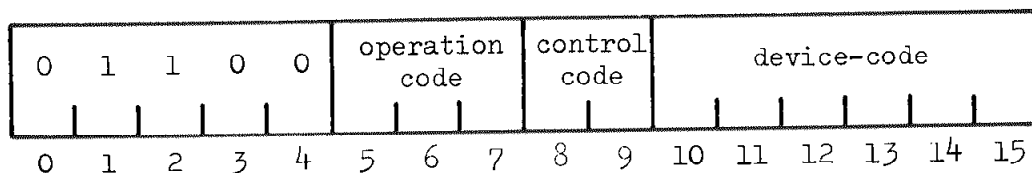
mnemonic one of the following semi-permanent symbols:

SKPBN SKPBZ SKPDN SKPDZ NIO

flag an optional mnemonic controlling device status (only applicable to NIO symbol)

device-code any legal expression which will produce a six-bit integer specifying the input/output device.

The word containing the assembled instruction will be structured in the following way:



The operation code, control bits and effect of the five

instructions of this type are as follows:

- SKPBN 111 00 Tests the state of the Busy flag of the peripheral device indicated in the instruction. If the device is currently in operation (Busy = 1) the next sequential instruction will be skipped
- SKPBZ 111 01 Tests the state of the Busy flag of the peripheral device indicated in the instruction. If the device is currently not in operation (Busy = 0) the next sequential instruction will be skipped
- SKPDN 111 10 Tests the state of the Done flag of the peripheral device indicated in the instruction. If the Done flag equals one the next sequential instruction will be skipped
- SKPDZ 111 11 Tests the state of the Done flag of the peripheral, device indicated in the instruction. If the Done flag equals zero the next sequential instruction will be skipped
- NIO 000 xx Sets the Busy and Done flags according to the optional mnemonic and corresponding control code.

The optional mnemonics controlling the device status flag and which can be used in conjunction with the NIO instruction, the corresponding control code and resultant effects are as follows:

- 00 Does not affect the device status
- S 01 Starts the device by setting Busy = 1 and Done = 0
- C 10 Idles the device by setting Busy = 0 and Done = 0
- P 11 Pulses the special in-out bus control line. The actual effect will depend on the type of device selected.

- 3.3.4.3 Input/output instructions without device codes. Among the device codes  $77_8$  refers to the Central Processing Unit (CPU) and will - when used with some of the previously mentioned instructions - lead to certain special functions being performed. Some of these special functions do occur with such regularity, that equivalent instructions dispensing with the need for the device code have

been defined, i.e. these instructions will always be assembled with digits 1 in position 10 to 15 of the instruction word.

Of these instructions three uses an accumulator during execution, which means, that the accumulator must be specified in the source instruction, while four do not require any arguments at all. The source formats of instructions of these different types are accordingly:

for the first type:

mnemonic  $\Delta$  accumulator

In this format the individual fields characterized are:

mnemonic one of the following semi-permanent symbols:

READS INTA MSKO

accumulator one of the digits 0, 1, 2 or 3 indicating the accumulator used to receive or provide data during execution

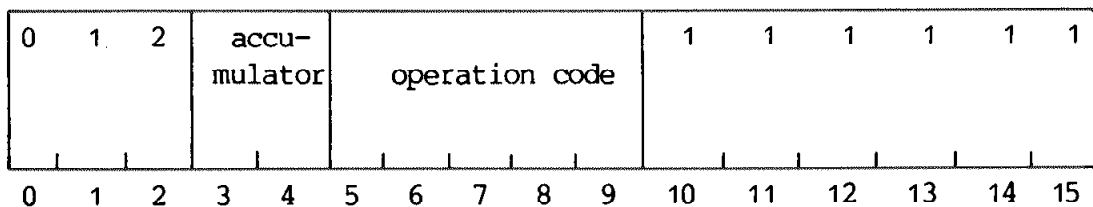
and for the second type:

mnemonic

This field must contain one of the following semi-permanent symbols:

INTEN INTDS IORST HALT

The word containing the assembled instruction will be structured in the following way:



The operation codes and associated effects of these seven instructions are as follows:

READS 00100 Places the current setting of the data switches situated on the front frame of the CPU board in the



accumulator specified in the instruction

- INTA 01100 Places the six bits of the device code of that interrupting peripheral device, which is physically closest to the CPU on the I/O bus, in position 10 to 15 of the accumulator specified in the instruction. The remaining bits in the accumulator are set to 0.
- MSKO 10000 Transfers the mask bits in the accumulator specified to the peripheral devices in accordance with their priority level, thereby allowing program control of the interrupt system response
- INTEN 00001 Sets the Interrupt flag digit = 1, thus permitting interrupts to take place
- INTDS 00010 Sets the interrupt flag digit = 0, thus preventing interrupts from taking place
- IORST 10110 Resets all connected input/output controllers to an idle state.
- HALT 11000 Suspends all processing in the CPU

In the case of the last four instructions the accumulator field will contain the bits 00. The special instructions referred to above are equivalent to the general I/O instructions, although as issued with specific values of the various fields. The equivalents are given below:

READS	equivalent to:	DIA <u>accumulator</u> , CPU
INTA	-	:- DIB <u>accumulator</u> , CPU
MSKO	-	:- DOB <u>accumulator</u> , CPU
INTEN	-	:- NIOS, CPU
INTDS	-	:- NIOC, CPU
IORST	-	:- DICC 0, CPU
HALT	-	:- DOC 0, CPU

## 4 Permanent Symbols

Permanent symbols are as previously mentioned an integral part of the assembler and they cannot be altered in any way. The majority of permanent symbols are directives, that is: their purpose is to direct the assembly process in some predetermined way. Those permanent symbols, that are not directives, are used to represent numerical values of internal assembler variables, but it should be noted, that a large proportion of directives may also perform this function in addition to their primary duty.

This chapter describes the individual permanent symbols and the syntactic rules pertaining to their application in the program, as well as the consequent effects of this.

The chapter falls in two parts: a general outline of the various classes of permanent symbols and a listing in alphabetical order of the individual permanent symbols. If a directive may also be assigned a value, this will be included in the description of the directives.

The general outline section of the chapter dealing with the various classes of permanent symbols is subdivided according to the function of the symbol as follows:

- Source interpretation
- Program structure
- Semi-permanent symbol definition
- External reference
- Macro definition

### 4.1 Source Interpretation

Permanent symbols of this class make it possible for the programmer to ensure, that the assembler interprets the source program file in a predetermined way - and especially so where it is desired, that this interpretation should deviate from the inherent assembler interpretation.

In addition to this, permanent symbols of this class are used to determine the source input structure and to influence the various possible options available for assembler output.

The permanent symbols of this class are the following:

.TITL	.RDX	.RDXO
.TXT	.TXTM	.TXTN
.END	.EOT	
.LIST	.EJEC	.MSG
.NOCON	.NOLOC	.NOMAC

## 4.2 Program Control

Permanent symbols of this class make it possible for the programmer to exert control over the final logical structure of the program as assembled, for instance by allowing him to define requirements in terms of the internal CPU configuration and to utilize features of the internal CPU structure.

Furthermore permanent symbols of this class will enable repetitive and conditional assembly to be prescribed, and finally they will make it possible to call into effect variations of the relocation characteristics.

Permanent symbols of this class are the following:

.BLK	.LOC	.
.NREL	.ZREL	
.DO	.IF	.GOTO
.ENDC		
.PUSH	.POP	.TOP
.PASS		

## 4.3 Semi-permanent Symbol Definition

Permanent symbols of this class make it possible for the programmer to define his own semi-permanent symbols - including the possibility of re-defining or deleting existing semi-permanent symbols, both his own and those pre-defined by RC.

All permanent symbols of this class (except .DUSR) correspond to a definite type of semi-permanent symbol; consequently those semi-permanent symbols defined by application of permanent symbols belonging to this class must be used with the appropriate formatting of the type in question. Otherwise a format error will be indicated during assembly.

As this formatting involves specific fields in the machine word being produced during assembly, care must also be taken to ensure, that the values assigned to the fields can be accommodated in those fields. Otherwise overflow will similarly be indicated during assembly.

If a specific user-symbol is defined as a semi-permanent symbol more than once in the same source program, the symbol table (see section 3.1.3) will contain the semi-permanent symbol in the form defined the latest.

In the alphabetic section of this chapter the description of permanent symbols of this class will include a description of the inherent format.

Permanent symbols of this class are the following:

.DALC	.DIO	.DIOA
.DIAC	.DICD	.DISD
.DMR	.DMRA	
.DXOP	.DUSR	
.XPNG		

## 4.4 External Reference

Permanent symbols of this class make it possible for the programmer to define symbols that can be referenced from several, separately assembled, programs thus establishing a means of communication between individual programs.

Permanent symbols of this class are the following:

.ENT	.EXTA	
.EXTD	.EXTN	.EXTU

## 4.5 Macro Definition

Permanent symbols of this class are used to control the structuring of macro definitions inside the assembled program and contains the following permanent symbols:

.MACRO	.ARGCT	.MCALL
--------	--------	--------

## 4.6 Alphabetic List of Permanent Symbols

Symbol: .

Type: Symbol

Class: Program control

Syntax: .

Effect: -

Value: The numeric value assigned to the symbol . will be equal to that of the program counter. Similarly the relocation characteristic of . will be equivalent to that of the program counter in its current state.

Default: -

Example:

```
00000 000000      .           ; VALUE OF CURRENT PC
00001 000004      JMP      .+3   ; JUMP WITHOUT LABEL
```

Symbol: .ARGCT

Type: Symbol

Class: Macro definition

Syntax: .ARGCT

Effect: -

Value: This symbol will have a value equal to the number of arguments actually employed in the most recent macro call.

If the .ARGCT symbol is used in situations, where it has not been specified in a macro definition, the value of the symbol will be -1

Default: -

Example:

```

00000 177777      (.ARGCT)      ; OUTSIDE MACRO

                .MACRO LIST      ; DEF. MACRO
                I= 1              ; **
                .DO .ARGCT        ; ** FOR EACH ARG. DO
                ↑I                ; ** STORE ARG.
                I= I+1            ; **
                .ENDC             ; **
                %                  ; END OF MACRO DEF.

                LIST 1,2          ; CALL LIST WITH 2 PARMS.
                I= 1              ; **
                .DO .ARGCT        ; ** FOR EACH ARG. DO
                1                  ; ** STORE ARG.
                I= I+1            ; **
                .ENDC             ; **
                2                  ; ** STORE ARG.
                I= I+1            ; **
                .ENDC             ; **
                LIST 1,2,3,4,5    ; CALL LIST WITH 5 PARMS.
                I= 1              ; **
                .DO .ARGCT        ; ** FOR EACH ARG. DO
                1                  ; ** STORE ARG.
                I= I+1            ; **
                .ENDC             ; **
                2                  ; ** STORE ARG.
                I= I+1            ; **
                .ENDC             ; **

```

```
00005 000003      3      ;** STORE ARG.
          000004      I=    I+1 ;**
          .ENDC      ;**
00006 000004      4      ;** STORE ARG.
          000005      I=    I+1 ;**
          .ENDC      ;**
00007 000005      5      ;** STORE ARG.
          000006      I=    I+1 ;**
          .ENDC      ;**

          LIST      ; CALL LIST WITH NO PARMS.
          000001      I=    1      ;**
          000000      .DO    .ARGCT ;** FOR EACH ARG. DO
          ;** STORE ARG.
          I=    I+1 ;**
          .ENDC      ;**
```

Symbol: .BLK

Type: Directive

Class: Program control

Syntax: .BLK  $\Delta$  expression

Effect: This directive will ensure reservation of a block of consecutive words in core memory. The value of expression indicates the actual number of words in the block; this value will also be used for incrementation of the program counter, thus ensuring that the continuation of assembly does not lead to interference with the reserved block.

Value: None

Default: -

Example:

```

                                .NREL
00000'010402      ISZ      CNTRS      ;  COUNTERS(0):= COUNTERS(0)+1;
00001'024403      LDA      1  CNTRS+2  ;  AC1:= COUNTERS(2);
00002'000007 CNTRS: .BLK      7      ;  INTEGER ARRAY: COUNTERS(7);

```



Symbol: `.DALC`

Type: Directive

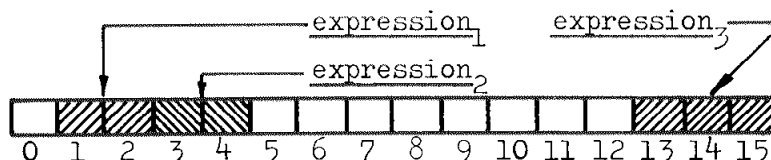
Class: Semi-permanent symbol definition

Syntax: `.DALC Δ user-symbol =((instruction):(expression))`

Effect: This directive will define user-symbol as a semi-permanent symbol to which is assigned the value of either instruction or expression. Furthermore use of the semi-permanent symbol will imply the use of a format corresponding to that of an arithmetic and logical instructions:

`semi-perm-symbol Δ expr1 Δ expr2 <Δexpr3>`

which will be assembled in the following way:



NOTE: The character # can be included as a break character when using the semi-permanent symbol. If so, the assembly process will place a digit 1 in bit 12 of the word, thus inhibiting load into the destination accumulator.

NOTE: A user-symbol defined once may be re-defined in a subsequent application of the `.DALC` directive. The latest definition will always be the one applying to user-symbol.

NOTE: If the symbol defined as semi-permanent by the use of this directive consists of three characters these may be followed by a fourth character as an optional mnemonic corresponding exactly to the rules pertaining to arithmetic and logical instructions as outlined in section 3.3.1 and having the identical consequences with regard to the state of the "carry" and "shift" bits.

Value: None

Default: -

Example:

```
101220 .DALC HALF= MOVZR 0,0 ; DEFINE 1/2 OPERATION
00000 125220 HALF 1,1 ; AC1:= AC1 / 2;
00001 135225 HALF 1,3 SNR ; AC3:= AC1 / 2;
00002 063077 HALT ; IF AC3=0 THEN STOP EXECUTION;
```

Symbol: `.DIAC`

Type: Directive

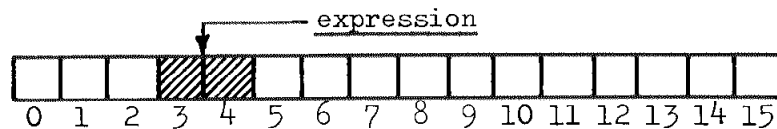
Class: Semi-permanent symbol definition

Syntax: `.DIAC Δ user-symbol = ((instruction):(expression))`

Effect: This directive will define user-symbol as a semi-permanent symbol to which is assigned the value of either instruction or expression. Furthermore use of the semi-permanent symbol will imply the use of a format of a Central Processor Function (1st type) instruction, i.e. requiring one accumulator:

`semi-permanent-symbol Δ expression`

which will be assembled in the following way:



Value: None

Default: -

Example:

```

020040 .DIAC  LDCUR= LDA    0  CUR ; DEFINE LOAD CUR;
00000 030040      LDCUR  2          ; LOAD AC2 WITH CUR;

```

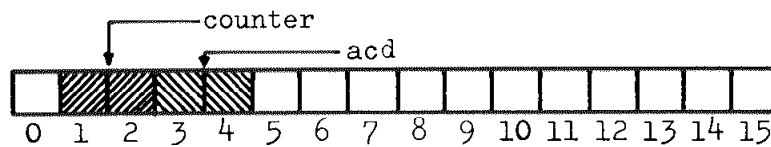
Symbol: .DICD

Type: Directive

Class: Semi-permanent symbol definition

Syntax: `.DICD Δ user-symbol = ((instruction):(expression))`

Effect: This directive will define user-symbol as a semi-permanent to which is assigned the value of either instruction expression. Furthermore the use of the semi-permanent symbol will imply the use of a format embodying two fields of which one will be that corresponding to the destination accumulator, while the other is a numeric counter:



Value: None

Default: -

Example:

```

102414 .DICD SKPEQ= SUB # 0,0 SZR ; DEFINE "SKIP IF EQUAL";
00000 112414 SKPEQ 1,2 ; IF AC1=AC2
; THEN THIS
; ELSE THAT;

```

Symbol: .DIO

Type: Directive

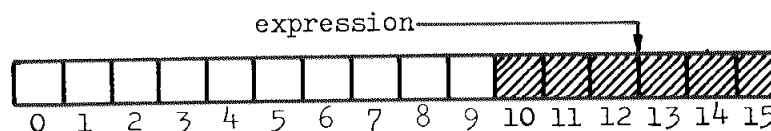
Class: Semi-permanent symbol definition

Syntax: `.DIO Δ user-symbol = ((instruction):(expression))`

Effect: This directive will define user-symbol as a semi-permanent symbol to which is assigned the value of either instruction or expression. Furthermore, use of the semi-permanent symbol will imply the use of a format corresponding to that of an Input/Output instruction without accumulator field:

`semi-permanent-symbol Δ expression`

which will be assembled in the following way:



NOTE: If the symbol defined as semi-permanent by the use of this directive consists of three characters these may be followed by a fourth character as an optional mnemonic corresponding exactly to the rules pertaining to Input/Output instructions as outlined in section 3.3.4.2 and having the identical consequences with regard to the state of the "busy" and "done" bits.

Value: None

Default: -

Example:

```

060100 .DIO   GOI.0= NIOS   0       ; DEFINE "GO INPUT/OUTPUT"
000076      MYDEV= 62.     ; DEFINE MYDEF DEVICE CODE

00000 060176 GOI.0   MYDEV      ; START(MYDEV);

```

Symbol: .DIOA

Type: Directive

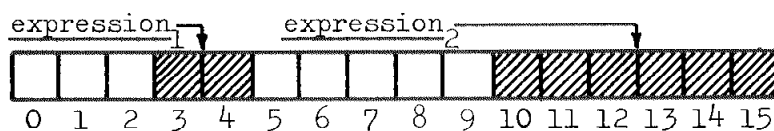
Class: Semi-permanent symbol definition

Syntax: `.DIOA Δ user-symbol = ((instruction):(expression))`

Effect: This directive will define user-symbol as a semi-permanent symbol to which is assigned the value of either instruction or expression. Furthermore use of the semi-permanent symbol will imply the use of a format corresponding to that of an Input/Output instruction with an accumulator field:

`semi-permanent-symbol Δ expression1 Δ expression2`

which will be assembled in the following way:



Note: If the symbol defined as semi-permanent by the use of this directive consists of three characters these may be followed by a fourth character as an optional mnemonic corresponding exactly to the rules pertaining to Input/Output instructions as outlined in section 3.3.4.1 and having the identical consequence with regard to the state of the "busy" and "done" bits.

Value: None

Default: -

Example:

```

062000 .DIOA MYOUT= DOB 0 0 ; DEFINE "MY OUTPUT"
000076 MYDEV= 62. ; DEFINE "MY DEVICE"

00000 066076 MYOUT 1 MYDEV ; OUTPUT(ACL,MYDEV);

```



Symbol: .DMR

Type: Directive

Class: Semi-permanent symbol definition

Syntax: `.DMR Δ user-symbol = ((instruction):(expression))`

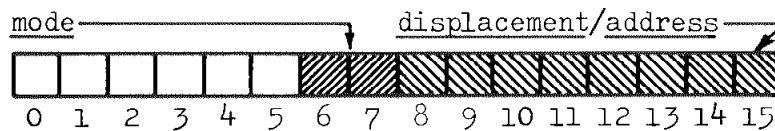
Effect: This directive will define user-symbol as a semi-permanent symbol to which is assigned the value of either instruction or expression. Furthermore, use of the semi-permanent symbol will imply the use of a format corresponding to that of a Program Flow Control instruction:

`semi-permanent-symbol Δ displacement Δ mode`

or:

`semi-permanent-symbol Δ address`

which will be assembled in the following way:



When the semi-permanent symbol thus defined is used the fields indicating displacement, mode and address will conform to the addressing rules as outlined in section 3.3.2.1.

NOTE: The character @ can be included as a break character when using the semi-permanent symbol. If so, the assembly process will place a digit 1 in bit 5 of the word indicating indirect addressing.

Value: None

Default: -



## Example:

```

010000 .DMR      COUNT=  ISZ      0      ; DEFINE "COUNT"
00000 010002    COUNT      TOTAL    ; TOTAL:=TOTAL+1;
00001 013003    COUNT @    VECTO,2   ; VECTO(AC2):= VECTO(AC2)+1;

00002 000000    TOTAL:  0          ; INTEGER: TOTAL;
00003 000000    VECTO:  0          ; INTEGER ARRAY: VECTO (3);
00004 000000    0          ;
00005 000000    0          ;

```

Symbol: `.DMRA`

Type: Directive

Class: Semi-permanent symbol definition

Syntax: `.DMRA Δ user-symbol = ((instruction):(expression))`

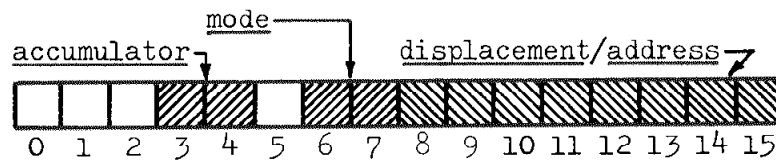
Effect: This directive will define user-symbol as a semi-permanent symbol to which is assigned the value of either instruction or expression. Furthermore, use of the semi-permanent symbol will imply the use of a format corresponding to that of a Data Transfer instruction:

semi-permanent-symbolΔaccumulatorΔdisplacementΔmode

or:

semi-permanent-symbolΔaccumulatorΔaddress

which will be assembled in the following way:



When the semi-permanent symbol thus defined is used, the fields indicating accumulator, displacement, mode and address will conform to the rules outlined in section 3.3.3 (including the addressing rules in section 3.3.2.1).

**NOTE:** The character @ can be included as a break character when using the semi-permanent symbol. If so, the assembly process will place a digit 1 in bit 5 of the word, thus indicating indirect addressing.

Value: None

Default: -

## Example:

```
020000 .DMRA  LOAD=  LDA    0  0  ; DEFINE "LOAD"
00000 024111  LOAD   1  .128  ; AC1:= 128;
00001 023407  LOAD  @ 0  EVENT,3 ; AC0:= CUR.NEXT EVENT;
```

Symbol: .DO

Type: Directive

Class: Program Control

Syntax: .DO  $\Delta$  expression

Effect: This directive will cause repetition of specific source program lines. Repetition will affect those lines following the .DO directive and until the terminating directive .ENDC occurs. The lines in question will be repeated the number of times given by the value of expression.

NOTE: Repetition can be nested to any depth, in which case the innermost .DO directive will correspond to the innermost .ENDC directive and so forth.

Value: None

Default: -

Example:

```

;          INPUT SOURCE TO THE FOLLOWING REPETITION
;          IS:
;          .DO      3
;          I
;          I=      I+1      ;
;          .ENDC      ; END OF REPETITION
;
000000    I=      0      ; CREATE A TABLE
000003    .DO      3      ; OF NO'S 0 TO 2.
00000    I
000001    I=      I+1      ;
          .ENDC      ; END OF REPETITION
00001 000001    I
000002    I=      I+1      ;
          .ENDC      ; END OF REPETITION
00002 000002    I
000003    I=      I+1      ;
          .ENDC      ; END OF REPETITION

000000    .DO      I==0    ; .DO USED AS CONDITIONAL
          2
          .ENDC

```

Symbol: .DUSR

Type: Directive

Class: Semi-permanent symbol definition

Syntax: `.DUSR Δ user-symbol = ((instruction):(expression))`

Effect: This directive will define user-symbol as a semi-permanent symbol to which is assigned the value of either instruction or expression. In contrast to other semi-permanent symbols defined by directives of this class, the use of a semi-permanent symbol defined by the .DUSR directive does not imply any specific format. The semi-permanent symbol thus defined may be used as an ordinary operand in single precision.

Value: None

Default: -

Example:

```

000016 .DUSR  DISP=  14.           ; DEFINE "DISPLACEMENT"
00000 031416      LDA    2  DISP,3   ; AC2:= DISP(AC3);
00001 000022      DISP+4           ; CONST: DISP+4;

```

Symbol: .DXOP

Type: Directive

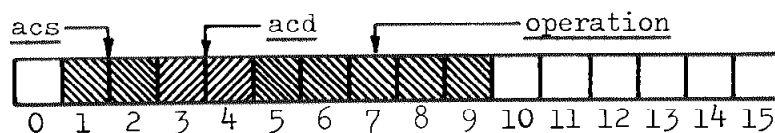
Class: Semi-permanent symbol definition

Syntax: .DXOP  $\Delta$  user-symbol = ((instruction):(expression))

Effect: This directive will define user-symbol as a semi-permanent symbol to which is assigned the value of either instruction or expression. Furthermore, use of the semi-permanent symbol will imply the use of a format containing three fields, of which the first two are accumulator fields, while the last is an operation field:

semi-permanent-symbol  $\Delta$  acs  $\Delta$  acd  $\Delta$  operation

which will be assembled in the following way:



Value: None

Default: -

Example: -

Symbol: .EJEC

Type: Directive

Class: Source interpretation

Syntax: .EJEC

Effect: This directive will cause ejection of the remainder of the current page of printer output conforming to the start of a new page of listing output.

Value: None

Default: -

Example: -

Symbol: .END

Type: Directive

Class: Source interpretation

Syntax: .END <Δexpression>

Effect: This directive will indicate termination of the source program input file simultaneously providing an end of program indication to the loading program. Expression is an optional argument, the value of which can be used to specify a starting address for the object program or a descriptor address for a MUS/DOMUS process.

Value: None

Default: -

Example:

```
00000 000000 END: 0
00001 000001      1
                .END  END      ; END WITH START-ADDRESS
```



Symbol: .ENDC

Type: Directive

Class: Program control

Syntax: .ENDC <Δuser-symbol>

Effect: The effect of this directive will depend on the actual syntax used.

If the syntax used is: .ENDC, then the directive will terminate repetitive assembly of the lines following a .DO directive or it will terminate conditional assembly of the lines following an .IF directive.

If the syntax used is: .ENDC Δ user-symbol, then the directive will have the same effect as outlined above, but it will have the additional effect of suppressing assembly of the lines following the .ENDC directive. Suppression of assembly will remain in force until user-symbol - enclosed in square brackets - is re-encountered in a source input line. If user-symbol does not reappear before end of current source file, suppression will be terminated at that point.

Value: None

Default: -

Example:

```

000000      .DO      2<>2      ; FALSE
              1              ; NOT GENERATED
              .ENDC  ELSE      ;
00000 000002      2              ; THEREFORE GENERATED
              [ELSE]          ;

000001      .DO      2==2      ; TRUE
00001 000001  1              ; GENERATED
              .ENDC  ELSE      ; SO NOT GENERATED
              [ELSE]          ;

```

Symbol: .ENT

Type: Directive

Class: External reference

Syntax: .ENT $\Delta$ user-symbol<sub>1</sub>< $\Delta$ user-symbol<sub>2</sub>... $\Delta$ user-symbol<sub>n</sub>>

Effect: This directive will cause a user-symbol to be accepted as globally defined, which means, that a user-symbol defined within one program can be referenced by other, separately assembled programs.

Each of the user-symbols appearing in the .ENT directive must be defined within the program containing this directive. Additionally it must be unique in relation to other user-symbols defined in programs referencing the globally defined user-symbol. If this is not the case, the loader will produce an error message indicating illegal multiplicity of definitions. Referencing globally defined user-symbols in other programs is accomplished by use of the directives .EXTA, .EXTD, .EXTN or .EXTU

Value: None

Default: -

Example:

```

                                .ENT   ENTR1   ; DECLARE ENTRIES
                                .ENT   ENTR2   ;
                                .NREL          ; NORMAL REL.
00000'000012  ENTR1: 10.          ; THESE TWO LOCATIONS MAY
00001'000000  ENTR2: 0            ; BE ADDRESSED FROM OTHER
                                ; PROGRAM MODULES

```

Symbol: .EOT

Type: Directive

Class: Source interpretation

Syntax: .EOT

Effect: This directive will indicate the end of a file, but termination of the complete source program input file will not be implied.

Value: None

Default: Physical end-of-file or the EM character implies the .EOT directive, if other source files follow.

Example: -

Symbol: .EXTA

Type: Directive

Class: External reference

Syntax: .EXTA  $\Delta$  user-symbol  $\Delta$  expression

Effect: This directive will enable a program to reference a user-symbol which has been defined in another program, while it will simultaneously generate a storage word containing the sum of user-symbol and expression.

The user symbol must be unique in relation to other user-symbols defined in the program (and in other programs as well), whereas expression may have any legal value. Negative values of expression will of course mean, that the value of user-symbol and expression are actually subtracted.

Following linkage editing of the programs referencing each other, the memory location involved will contain the following value:

$$(A+R_1) + (\text{sign}(B) * (\text{abs}(B)+R_2))$$

where

A is the value of the external user-symbol defined as an entry in another program

$R_1$  and  $R_2$  are the relocation constants used during linkage editing

B is the value of expression

The relocation characteristic of the value as given above will be in accordance with the rules listed in the following table:

Relocation quality of:

<u>user-symbol</u>	<u>expression</u>	<u>result</u>
absolute	any	any
any	absolute	any
positive .NREL	positive .NREL	byte .NREL
positive .NREL	negative .NREL	absolute
positive .ZREL	positive .ZREL	byte .ZREL
positive .ZREL	negative .ZREL	absolute

All other combinations are illegal.

Note, that by adding a negative .NREL value to an external positive .NREL value the result will be an absolute value, i.e. a size.

Default: -

Example: -

Symbol: .EXTD

Type: Directive

Class: External reference

Syntax: .EXTD user-symbol<sub>1</sub> < user-symbol<sub>2</sub> ... user-symbol<sub>n</sub> >

Effect: This directive will enable a program to reference one or more user-symbols which have been defined in other programs.

The user-symbol must have been declared by use of the .ENT directive in the programs where they are defined; they must be unique in relation to other user-symbols appearing in the different programs.

Globally defined symbols may be used as an address or a displacement of Program Flow Control instructions or Data Transfer instructions; they may also be used to specify the contents of an ordinary 16-bit memory word. If the symbol is used as a page zero address or as a displacement care must naturally be exercised to ensure that its value corresponds to the restrictions pertaining to this particular use (cf. sections 3.3.2 and 3.3.3).

Value: None

Default: -

Example:

```

                                .EXTD  DISP    ; DECLARE EXTERNAL
                                .EXTD  ADR     ; DISPLACEMENTS
                                .NREL
00000'000002$.ADR:  ADR                ; EXTERNAL ADDRESS
00001'032777      LDA  @ 2, .ADR      ; EXTD USED AS EXTN
00002'041001$    STA  0, DISP, 2; EXTD USED AS DISPLACEMENT

```

Symbol: .EXTN

Type: Directive

Class: External reference

Syntax: .EXTN user-symbol<sub>1</sub> < user-symbol<sub>2</sub> ... user-symbol<sub>n</sub> >

Effect: This directive will enable a program to reference one or more user-symbols which have been defined in other programs.

The user-symbols must have been declared by use of the .ENT directive in the program where they are defined; they must be unique in relation to other user-symbols appearing in the different programs.

Symbols referenced through the .EXTN directive can be used solely to specify the contents of an ordinary 16-bit memory word; the value of such symbols must therefore at load time lie in the range from 0 to 177777<sub>8</sub>.

Value: None

Default: -

Example:

```

        .EXTN  PROC      ; DECLARE EXTERNAL
        .EXTN  VALUE     ; NORMALS

        .NREL

00000'177777 .PROC:  PROC      ; POINTERS FOR
00001'177777 .VALU:  VALUE     ; INDIRECT ADDRESSING

00002'026777     LDA @ 1, .VALU ; LOAD VALUE
00003'006775     JSR @  .PROC  ; CALL PROC

```

Symbol: .EXTU

Type: Directive

Class: External reference

Syntax: .EXTU

Effect: This directive will affect all symbols which are still undefined after pass one of the assembly process; such symbols will - by inclusion of this directive - be treated by the assembler as if they were listed in a .EXTD directive.

NOTE: Great care must be exercised when use of this directive is contemplated, as its inclusion in the program will have the consequence of making the error indication "undefined" (error code: U) irrelevant. Thus undefined symbols may actually be present in the program without any outward indication thereof.

Value: None

Default: -

Example:

```

                                .EXTU           ; SET EXTU MODE
00000 000001$                 A               ; IMPLICIT .EXTD A
00001 024002$                 LDA           1 B   ; IMPLICIT .EXTD B

```



Symbol: .GOTO

Type: Directive

Class: Program control

Syntax: .GOTO  $\Delta$  user-symbol

Effect: This directive will unconditionally suppress assembly of the lines following the directive. Suppression of assembly will remain in force until user-symbol - enclosed in square brackets - is re-encountered in a source input line. If user-symbol does not reappear before the end of current source file, suppression will be terminated at that point.

Value: None

Default: -

Example:

```
00000 000001          1          ; NORMAL GENERATION
                   .GOTO  UNCON  ;
                   2          ; SKIPPED
                   3          ; SKIPPED
00001 000004 [UNCON] 4          ; NORMAL GENERATION
00002 000005          5          ; AGAIN
```

Symbol: .IF((E):(G):(L):(N))

Type: Directive

Class: Program control

Syntax: .IF((E):(G):(L):(N))  $\Delta$  expression

Effect: These four alternative directives will cause suppression of assembly of the lines following the .IF directive if the condition specified in the directive is not satisfied.

The condition specified in the directive will depend on the choice of alternative form as indicated below:

.IFE Assemble if expression is equal to zero  
.IFG Assemble if expression is greater than zero  
.IFL Assemble if expression is less than zero  
.IFN Assemble if expression is unequal to zero

(The data field of the program listing (columns 10 to 15) will be 1 if the condition is true and 0 if the condition is false).

Conditional assembly will be terminated by the appearance of the .ENDC directive in a subsequent source input line.

NOTE: Conditional assembly can be nested to any depth, in which case the innermost .IF directive will correspond to the innermost .ENDC directive and so forth.

Value: None

Default: -

## Example:

```
000007      I=      7      ;
000000      .IFE     I      ; FALSE
            JMP     .+2     ;
            .ENDC          ;
000001      .IFN     I      ; TRUE
00000 000002  JMP     .+2     ;
            .ENDC          ;
000001      .IFG     I      ; TRUE
00001 000003  JMP     .+2     ;
            .ENDC          ;
000000      .IFL     I      ; FALSE
            JMP     .+2     ;
            .ENDC          ;
```

Symbol: `.LIST`

Type: Directive

Class: Source interpretation

Syntax: `.LIST  $\Delta$  expression`

Effect: This directive will control the extent of source listing by the assembler depending on the value of expression. If the value of expression equals zero listing of all subsequent lines of source file will be suppressed. This state will be maintained until an end-of-file marker appears in the input source file, or until a new `.LIST` directive with a non-zero value of expression appears.

If the value of expression does not equal zero the input source file will be listed by the assembler.

NOTE: The `.LIST` directive will be overridden by a `/N` switch in the source specification of the DOMAC command. Both the `.LIST` directive and the `/N` switch will be overridden by specifying `MODE,0` in the DOMAC command. (cf. Operating Procedures, chapter 7).

Value: None

Default: Listing of input source file will be produced.

Example: -

Symbol: .LOC

Type: Directive/symbol

Class: Program control

Syntax: .LOC  $\Delta$  expression

Effect: This directive will change the current value of the program counter to that given by expression. Simultaneously the relocation characteristic of the program counter will be made equivalent to that of expression.

Value: When used as an ordinary symbol, the numeric value and the relocation characteristic assigned to (.LOC) will be equal to the current value and relocation characteristic of the program counter.

NOTE: An exception to this exists, namely if .LOC is used in conjunction with the .PUSH directive, thus placing the program counter in the assembler variable stack (cf. .PUSH) and later being restored.

In this case the value of (.LOC) will be ignored and only the relocation characteristic will be affected. The reason for this must be seen in conjunction with the macro facility; the relative value of the program counter may have been altered within the macro definition and should not be changed when the program counter relocation characteristic is restored after exit from the macro.

Default: Zero, absolute

Example:

```

001000      .LOC    1000    ; SELECT ABS PC:= 1000;
01000 000001 1          ;
001004      .LOC    .LOC+3 ; INCREMENT PC WITH 3;
```

Symbol: .MACRO

Type: Directive

Class: Macro definition

Syntax: .MACRO  $\Delta$  user-symbol

Effect: This directive will define user-symbol as the name of a macro definition.

The macro definition must be contained in the lines immediately following the .MACRO directive.

The macro definition is terminated by the appearance of the character %.

The user-symbol thus used to name the macro definition can then be used to call the macro in subsequent lines of the program.

Value: None

Default: -

Example:

```

                                .MACRO MAC      ; MACRO DEFINITION
                                ↑1              ; MAC ↑1 ↑2 ↑3
                                ↑2              ;
                                ↑3              ;
                                %                ; END OF MACRO DEF.

                                MAC      1,2,3    ; MACRO CALL
00000 000001                    1              ; MAC 1 2 3
00001 000002                    2              ;
00002 000003                    3              ;

                                MAC      1+2,.PASS,3*4 ; MACRO CALL
00003 000003                    1+2            ; MAC 1+2 .PASS 3*4
00004 000001                    .PASS          ;
00005 000014                    3*4            ;

```

Symbol: .MCALL

Type: Symbol

Class: Macro definition

Syntax: .MCALL

Effect: -

Value: This symbol will have a value which depends on whether or not that macro definition, in which the symbol appears, has been called during the current pass of the assembler. On the first call of the macro definition, the value of .MCALL will be 0, while on all subsequent calls its value will be 1.

If the .MCALL symbol is used in situations, where it has not been specified in a macro definition, the value of the symbol will be -1.

Default: -

Example:

```

00000 177777      (.MCALL)      ; OUTSIDE MACRO
                .MACRO MC      ; DEFINE MACRO
                (.MCALL)      ; DISPLAY .MCALL VALUE
                %
00001 000000      MC           ; FIRST CALL
                (.MCALL)      ; DISPLAY .MCALL VALUE
00002 000001      MC           ; SECOND CALL
                (.MCALL)      ; DISPLAY .MCALL VALUE
00003 000001      MC           ; THIRD CALL
                (.MCALL)      ; DISPLAY .MCALL VALUE

```

Symbol: .MSG

Type: Directive

Class: Source interpretation

Syntax: .MSG  $\Delta$  string

Effect: This directive will cause a message to be displayed on the console during assembly.

string is any sequence of characters (except null, line feed or rubout), which the assembler will treat as if it is an ordinary comment with the added feature of this being displayed on the console when the carriage return character is encountered.

The message will be displayed during both passes of the assembly unless precautions are taken to avoid this.

Value: None

Default: No messages displayed

Example:

```
000001      .DO      .PASS      ; PASS 2 ONLY
             .MSG     PASS 2    NOW
             .ENDC                    ; END PASS 2 ONLY

000000      .DO      .PASS==0; PASS 1 ONLY
             .MSG     PASS 1    NOW
             .ENDC                    ; END PASS 1 ONLY
```



Symbol: .NOCON

Type: Directive/symbol

Class: Source interpretation

Syntax: .NOCON  $\Delta$  expression

Effect: This directive will determine the extent of listing of input source file containing conditional parts. If such conditional parts of a program do not meet the conditions permitting assembly, they may be listed or not depending on the value of expression.

If the value of expression equals zero listing will be included. If the value of expression does not equal zero listing will be suppressed.

Those conditional parts of a program, that do meet the conditions permitting assembly will not be affected by the .NOCON directive.

Value: When used as an ordinary symbol, the numeric value assigned to (.NOCON) will be equal to the value of the expression occurring in the latest use of the .NOCON directive.

Default: Listing of conditional parts included.

Example: -

Symbol: .NOLOC

Type: Directive/symbol

Class: Source interpretation

Syntax: .NOLOC  $\Delta$  expression

Effect: This directive will determine the extent of listing of input source file containing lines without a location field. Such lines may be listed or not depending on the value of expression.

If the value of the expression equals zero such lines are included in the listing. If the value of expression does not equal zero listing of such lines will be suppressed.

Value: When used as an ordinary symbol, the numeric value assigned to (.NOLOC) will be equal to the value of the expression occurring in the latest use of the .NOLOC directive.

Default: All lines will be included in the listing.

Example: -

Symbol: .NOMAC

Type: Directive/symbol

Class: Source interpretation

Syntax: .NOMAC  $\Delta$  expression

Effect: This directive will determine the extent of listing of macro expansions depending on the value of expression. If the value of expression equals zero macro expansions will be included in the listing. If the value of expression does not equal zero listing of macro expansions will be suppressed.

Value: When used as an ordinary symbol, the numeric value assigned to (.NOMAC) will be equal to the value of the expression occurring in the latest use of the .NOMAC directive.

Default: Macro expansions are included in the listing.

Example: -

Symbol: .NREL

Type: Directive/symbol

Class: Program control

Syntax: .NREL

Effect: This directive will cause assembly of all subsequent source program statements to take place in accordance with normal relocation characteristics and consequently cause utilization of the program counter associated herewith.

Value: When used as an ordinary symbol the numeric value assigned to (.NREL) will be equal to the current value of the normal relocation counter.

Default: -

Example:

```

                                .NREL          ; SELECT NREL PC
00000'020100          LDA      0   TWO ; LOAD ABS. ADR.
00001'101520          INCZL   0,0   ;
                   000100          .LOC   100   ; SELECT ABS PC:= 100
00100 000002'TWO:    (.NREL)      ; VALUE OF NREL PC
                   .NREL          ; RESUME NREL
00002'040100          SPA      0   TWO ;

```

Symbol: .PASS

Type: Symbol

Class: Source interpretation

Syntax: .PASS

Effect: -

Value: The value of this symbol will depend on the assembly stage currently in effect.

During pass 1 of the assembler the symbol .PASS will have the value zero, while during pass 2 of the assembler it will have the value one.

Default: -

Example:

```

000000      .DO      .PASS==0; I WILL BE DEFINED IN
              I=      1      ; PASS 1 AND SKIPPED IN
              .ENDC    ; PASS 2

00000 000001      I      ; I IS DEFINED

```

Symbol: .POP

Type: Symbol

Class: Program control

Syntax: .POP

Effect: Although not a directive, the use of the symbol .POP will be associated with the effect of removing the latest element to be stored in the variable stack from the stack (and hereby "exposing" the next element in the stack).

Value: The value and relocation characteristic of this symbol is equal to the value and relocation characteristic of the latest expression stored in the internal variable stack by application of the .PUSH directive.

If no values have been stored in the stack, the symbol .POP will have zero absolute value. In that case a field overflow will furthermore be indicated in the listing.

Default: -

Example:

```

000001      .PUSH  1      ; SAVE 1'ST ELEMENT
000002      .PUSH  2      ; SAVE 2'ND ELEMENT
000003      .PUSH  3      ; SAVE 3'RD ELEMENT
00000 000003 .TOP        ; SHOW 3'RD ELEMENT
00001 000003 .TOP        ; AND AGAIN
00002 000003 .POP        ; REMOVE 3'RD ELEMENT
00003 000002 .TOP        ; SHOW 2'ND ELEMENT
00004 000002 .POP        ; REMOVE 2'ND ELEMENT
00004 000001 .POP        ; REMOVE 1'ST ELEMENT

```

Symbol: .PUSH

Type: Directive

Class: Program control

Syntax: .PUSH  $\Delta$  expression

Effect: This directive will save the value and relocation characteristic of a legal expression by storing it in an internal variable stack.

Repetitive stacking of expressions is permissible as long as the variable stack is not exhausted. A maximum of 20 consecutive entries to the stack will be accepted.

Restoration of an expression value stored in the stack is accomplished by means of the .POP symbol.

The stack is operated on the "last in - first out" principle.

Value: None

Default: -

Example:

```

000000      .PUSH  .TXTM  ; SAVE TEXT-MODE
000000      .PUSH  .TXIN  ; SAVE TEXT-NODE
000001      .TXTM  1      ; SET LOCAL TEXT-MODE
000001      .TXIN  1      ; AND LOCAL TEXT-NODE
00000 040502 .TXT   "ABCD" ;
041504
000000      .TXIN  .POP   ; RESTORE GLOBAL
000000      .TXTM  .POP   ; VALUES (LAST FIRST)

```

Symbol: .RDX

Type: Directive/symbol

Class: Source interpretation

Syntax: .RDX  $\Delta$  expression

Effect: This directive will define the radix of representation which will be applied for conversion of all numeric input in the source file subsequent to the appearance of the directive.

expression will be evaluated to radix 10, and must yield a number in the interval:  $2 \leq \underline{\text{expression}} \leq 20$

Value: When used as an ordinary symbol, the numeric value assigned to (.RDX) will be equal to the current input radix.

Default: Conversion of input will by default take place to radix 8.

Example:

```

000010      .RDX   8      ; INPUT RADIX NOW 8
00000 000011  11      ; = 9 (DECIMAL)
000012      .RDX  10     ; INPUT RADIX NOW 10
00001 000013  11      ; = 11 (DECIMAL)
000002      .RDX   2     ; INPUT RADIX NOW 2
00002 000003  11      ; = 3 (DECIMAL)
00003 000002  (.RDX)  ; VALUE = CURRENT INPUT RADIX

```



Symbol: .RDXO

Type: Directive/symbol

Class: Source interpretation

Syntax: .RDXO  $\Delta$  expression

Effect: This directive will define the radix of representation which will be applied for conversion of all numeric output in the listing subsequent to the appearance of the directive.

expression will be evaluated to radix 10, and must yield a number in the interval:  $8 \leq \text{expression} \leq 20$

Value: When used as an ordinary symbol, the numeric value assigned to (.RDXO) will be equal to the current output radix.

Default: Conversion of output will be default take place to radix 8.

Example:

```

00000 000012      .RDX  10      ; INPUT RADIX 10
00000 000012      10          ; OUTPUT RADIX 8
00001 00010      .RDXO  10      ; OUTPUT RADIX 10
00001 00010      10          ;
00002 0010      .RDXO  16      ; OUTPUT RADIX 16
00002 000A      10          ;
00003 001E      30          ;
00004 0010      (.RDXO)      ; VALUE = CURRENT OUTPUT RADIX

```

Symbol: .TITL

Type: Directive

Class: Source interpretation

Syntax: .TITL  $\Delta$  user-symbol

Effect: This directive will assign the name "user-symbol" to the program being assembled.

The name does not necessarily have to differ from other symbols defined by the program.

The name is used for identification of the relocatable binary output produced by the assembler; for inst. the loading program or the library files will refer to the program by its name.

Value: None

Default: .MAIN

Example:

```
.TITL  MYPGM
```

Symbol: .TOP

Type: Symbol

Class: Program control

Syntax: .TOP

Effect: -

Value: The value and relocation characteristic of this symbol is equal to the value and relocation characteristic of the latest expression stored in the internal variable stack by application of the .PUSH directive.

If no values have been stored in the stack, the symbol .TOP will have zero absolute value.

NOTE: The symbol .TOP differs from the symbol .POP in that the use of .TOP will not remove the latest element of the stack from the stack.

Default: -

Example: see .POP

Symbol: .TXT  
 Type: Directive  
 Class: Source interpretation  
 Syntax: :TXT<p>Δ/string/

p is an optional mnemonic indicating the status of the parity bit (leftmost bit) of each byte containing the individual character code. The optional mnemonics and their effects are as follows:

	Sets parity bit to zero unconditionally
F	Sets parity bit to one unconditionally
E	sets parity bit for even parity
O	Sets parity bit for odd parity

/ is a delimiting character, but it is not itself part of the string. Any character (except null, line feed or rubout) may be used as delimiter, but to avoid misinterpretations it should not be any of the characters appearing in the string proper.

Effect: This directive will make the assembler scan the input following the delimiter in string mode until the first re-occurrence of the delimiter.

Carriage return and form feed characters appearing inside the string will fulfil their normal purpose of continuing the string from line to line or from page to page, but these two characters will not themselves be stored as part of the text string.

Storage of one character requires seven bits and consequently takes up one 8-bit byte; the eighth bit is used as parity bit in accordance with the optional mnemonic specified in the directive. Two consecutive characters will thus occupy one word of storage.

Arithmetic expressions can be included in the string by using angle brackets to enclose the expression. The expression will be evaluated, masked to seven bits and filled up to eight bits in accordance with the optional mnemonic specified. Note however, that logical operators

are not allowed in expressions inside a string. If it is desired to incorporate in the text string a carriage return or a form feed character, this must be done by means of an expression within the string, as these characters are otherwise ignored. Thus to store for inst. a form feed character the following directive is issued:

```
.TXT +PAGE 1<14>+
```

Value: None

Default: Bytes will be packed right/left, and a terminating byte containing zeroes exclusively will be added.

Example:

```

000001      .TXTM  1
00000 030061  .TXT   "0123" ; ZERO PARITY BIT
          031063
          000000
000003 030261  .TXTE  "0123" ; EVEN PARITY BIT
          131063
          000000
000006 130261  .TXTF  "0123" ; FORCED PARITY BIT
          131263
          000000
000011 130061  .TXTO  "0123" ; ODD PARITY BIT
          031263
          000000

```

Symbol: .TXIM

Type: Directive/symbol

Class: Source interpretation

Syntax: .TXIM  $\Delta$  expression

Effect: This directive will change the packing of bytes as generated through the use of the .TXT directive (or its optional forms). Packing will depend on the value of expression as follows:

Value of expression equal to zero: bytes packed right/left

Value of expression not equal to zero: bytes packed left/right

Value: When used as an ordinary symbol, the numeric value assigned to (.TXIM) will be equal to the value of the expression occurring in the latest use of the .TXIM directive.

Default: Bytes will be packed right/left

Example:

```

000000      .TXIM  0      ; PACK RIGHT/LEFT
00000 030460  .TXT   "012" ;
000062
000007      .TXIM  7      ; PACK LEFT/RIGHT
00002 030061  .TXT   "012" ;
031000
00004 000007  (.TXIM)    ; VALUE LAST SET

```

- Symbol: `.TXIN`
- Type: Directive
- Class: Source interpretation
- Syntax: `.TXIN  $\Delta$  expression`
- Effect: This directive will determine how a character string input by the `.TXT` directive (or its optional forms) will be terminated. The termination of the string will depend on the value of expression according to the following rules:
- If the value of expression equals zero the last storage word used will contain at least one zero byte (i.e. a byte where all eight bits are equal to zero) according to the number of characters in the string. If the string contains an even number of characters, the last storage word will contain two zero bytes; if the string contains an odd number of characters, the last storage word will contain one zero byte.
- If the value of expression is not equal to zero the last storage word used will contain at most one zero byte according to the number of characters in the string. If the string contains an even number of characters, the last storage word will contain no zero bytes (i.e. this word will contain the last two characters of the string); if the string contains an odd number of characters, the last storage word will contain one zero byte.
- Value: When used as an ordinary symbol, the numeric value assigned to `(.TXIN)` will be equal to the value of the expression occurring in the latest use of the `.TXIN` directive.
- Default: Termination with at least one zero byte. (expression = 0.)

## Example:

```

00000 000000      .TXIN  0      ; FOLLOW BY AT LEAST
00000 030460      .TXT   "0123" ; ONE ZERO BYTE.
00000 031462
00000 000000

00003 001750      .TXIN  1000. ; DO NOT DO.
00003 030460      .TXT   "0123" ;
00003 031462

00005 001750      (.TXIN)      ; LAST VALUE SET.

```



Symbol: .XPNG

Type: Directive

Class: Semi-permanent symbol definition

Syntax: .XPNG

Effect: This directive fulfils a special duty among directives of this class as it will not in reality cause any definition to take place. On the contrary, this directive will erase all semi-permanent symbol definitions (and all macro definitions) from the assembler program's table of symbols. It will thus clear the symbol table of all previously defined symbols (except of course the permanent symbols) and thus create a background for uncomplicated re-definition.

Use of the .XPNG directive normally involves the following steps:

- a) A program is written, which contains the .XPNG directive followed by statements intended to re-define any semi-permanent symbols desired.
- b) This program is assembled in the normal way, except that the DOMAC command is given in a form specifying MODE,S (cf. section 7.2). In doing this the assembler will create a new symbol table containing the re-defined symbols during pass 1, after which it will terminate assembly.
- c) The assembler will now be ready for use in assembling programs involving those semi-permanent symbols re-defined in stage a).

Value: None

Default: -

## Example:

```

.TITL MYDEF
.XPNG ; REMOVE ALL SYMBOLS
020000 .DMRA LOAD= 020000 ; DEFINE "LOAD"
040000 .DMRA STORE= 040000 ; DEFINE "STORE"
;
; TO CREATE A NEW SYMBOL TABLE FILE CONTAINING
; ABOVE TWO SYMBOLS, USE THE FOLLOWING LOAD-
; COMMAND TO DOMUS (SEE SECTION 7.2):
;
; DOMAC MODE.S MYDEF PERM.NIHIL SYMB.MYPS MACRO.MYPM
;

```

Symbol: `.ZREL`

Type: Directive/symbol

Class: Program control

Syntax: `.ZREL`

Effect: This directive will cause assembly of all subsequent source program statements to take place in accordance with page zero relocation characteristics and consequently cause utilization of the program counter associated herewith.

Value: When used as an ordinary symbol the numeric value assigned to `(.ZREL)` will be equal to the current value of the page zero relocation counter.

Default: -

Example: -

## 5 Macro Programming

It has previously been mentioned, that the DOMUS Assembly Language incorporates a feature - known as Macro Programming - which may considerably simplify the programming procedure by replacing repetitive sequences of the program with one single formal subsection, which may then be referenced at the appropriate points in the program. Thus employment of the macro feature involves two distinct stages in the source program: defining of the macro and subsequently referencing the macro.

### 5.1 Macro Definition

The macro definition consists of two or more program lines initiated by the directive `.MACRO` and terminated by the character `%`. The definition must comply with the following format:

```
.MACRO user-symbol ↓
macro-definition-string%
```

In this format user-symbol must conform with the rules pertaining to symbols in general as outlined in section 3.1. The actual user-symbol applied in conjunction with the `.MACRO` directive will be the name by which the macro definition will be referenced at those points in the program where it is desired to call it into use.

The carriage return character following user-symbol in the definition serves to separate the string of characters, which forms the macro name, from that string, which forms the proper contents of the macro definition.

The macro-definition-string contains the actual program subsection to be taken into account whenever the macro name is referenced in the program. It consists of a string of characters from among the full ASCII character set and it must conform to the actual program structure in the program subsection implied by the string; i.e. if the intended subsection consists of several source program lines, the macro definition string must incorporate the necessary carriage return characters to reproduce this subdivision into lines.

Macro-definition strings appearing in the source input to the assembler will be read in string mode (cf. section 1.2.2) and

will consequently not be subject to any interpretation at this stage except for three specific characters, which will each have its definite meaning whenever it may appear in the macro definition string. (The actual effect of these three characters will be discussed below). Although the macro definition string will be read by the assembler in string mode, the actual contents of the string will eventually be interpreted at a later stage in the assembly process; a fact, which will be dealt with in due course.

The character % appearing after the macro definition string is a terminating character and indicates the end of the string to the assembler. It is not itself a part of the macro definition. The % character must appear in either of two positions relative to the macro definition string depending on the actual program structure implied in the macro-definition-string according to the following rules:-

If the macro definition string contains a program section, which is not in itself a complete program line (for inst. if it concerns a single expression to be substituted for an operand), then the macro definition string cannot contain a carriage return or form feed character and consequently the terminating character (%) must be the last character of the line forming the macro definition string. (A further consequence of this is, that the macro definition string in this case cannot exceed the assembler's maximum line length of 132 characters.)

If the macro definition string contains a program section, which consists of one or more complete program lines, then the terminating character must be the first character of the final line of the macro definition as such.

Examples of this are:

```
.MACRO ARG
(( (15+X)+Y)/3+Y)*7%
```

and

```
.MACRO DIV2
MOVL# 2,2,SZC
MOVOR 2,2,SKP
MOVZR 2,2,SKP
MOVOR 2,2,SKP
MOVOR 2,2,
%
```

### 5.1.1 Interpretation

The three characters which will - as previously mentioned - be interpreted even as the macro definition is being input to the assembler are these:-

- 5.1.1.1 The character %, which as already explained terminates the macro-definition-string and which therefore obviously cannot appear inside the string without interpretation.
- 5.1.1.2 The character †, to which has been assigned a special function inside macro definitions and which must consequently be interpreted accordingly.

The special function allocated to this character is that of indicating to the assembler, that the character immediately following the † must be interpreted as a dummy argument of the definition. This implies, that whenever the macro definition is invoked in the program the actual value of the dummy argument will be specified by the macro call, whereby the usefulness of the macro feature is greatly enhanced.

Dummy arguments appearing in the macro definition string must take one of the following three forms:

- d where d may be any of the digits 1 to 9
- c where c may be any letter of the alphabet A to Z
- ?c where c may be either the character ? or any alpha-numeric character A to Z or 0 to 9

The effect of the different forms of dummy arguments will be explained in section 5.2 (Macro Call).

- 5.1.1.3 The character ←, which is in itself ignored, but whose appearance in the macro definition string will cause interpretation of the character immediately following the ← to be suspended. This means, that this character must be used, if the programmer wishes to include either of the characters %, † or ← in the macro definition string as characters in their own right. As an example of this consider, that the programmer wishes to incorporate the symbol ERR% as an element of the macro definition string; to avoid premature termination of the string the symbol must be written in the source text as ERR←%. The character ← may precede any character, but in the case of ordinary characters no special significance attaches to this; thus the assembler will regard SYMB and SY←MB as identical symbols.

### 5.1.2 Nesting of Macros

Macros may be nested, i.e. one macro definition string may include one or more macro definitions as elements of the string. Furthermore the "inner" macros may be temporarily terminated and subsequently continued, whereby for inst. program flow control instructions contained in the "outer" macro may be used to distinguish between different options in different calls of the "outer" macro.

This particular feature depends on the general syntactical rule for macro definitions, that if a macro definition has the same name as the one last encountered, the later macro definition will be considered to constitute a direct continuation of the previous one and will be appended to it accordingly.

It should be noted, that whenever macro definitions are nested use of the ← character becomes imperative; for inst. termination of an "inner" macro will require use of the % character, which character must however be preceded by the ← character, as failure to do so would prematurely terminate the "outer" macro - thus rendering its definition string incomplete.

An example of nested macros is given in the second example of section 5.5.

An additional capability of the macro feature - which is connected with the nesting of macros - is the acceptance of the assembler of recursive macro calls. This can be illustrated by the macro CHE, which computes the value of the n-th order Chebyshev polynomial in accordance with the recursive formula:

$$\text{CHE}(n,x) = 2*x*\text{CHE}(n-1,x) - \text{CHE}(n-2,x); \text{CHE}(0,x) = 1; \text{CHE}(1,x) = x.$$

The macro CHE is called with four arguments, the first of which is the order, n; the second is the variable x and the third and fourth are the values of the n-th and (n-1)-th Chebyshev polynomials respectively.

NOTE: The macro CHE as shown in the following example will not accept orders larger than n = 20. This limitation is a consequence of the fact that the macro utilizes the variable stack via the .PUSH directive. The limited stack length must be taken into account - otherwise the number of recursive calls may exhaust the stack.

```

        .MACRO CHE
        .DO    ↑1==0
↑3=      1
        .ENDC
        .DO    ↑1==1
↑3=      ↑2
↑4=      1
        .ENDC
        .DO    ↑1>1
        CHE    ↑1-1,↑2,↑3,↑4
        .PUSH  ↑3
        ↑3 = 2*(↑2)*↑3-↑4
        ↑4 = .POP
        .ENDC
%
```

## 5.2 Macro Call

The macro definition described in the preceding section of this chapter is a formal, generalized program subsection, which is then subsequently referenced at appropriate points in the program by means of the macro call. A specific macro definition may be referenced any number of times within a given program, but the actual macro call must conform to the structure inherent in the macro definition which the call references, i.e. the macro call must specify whichever actual arguments are intended to replace possible dummy arguments in the macro definition string.

While the macro definition is basically accepted by the assembler without any interpretation, the macro call initiates the implementation of those specific instructions etc. which appear in the macro definition string. This effect - termed macro expansion - results in the incorporation into the object program of the coded equivalents of the statements involved - simultaneously incorporating into these statements the actual arguments as specified by the macro call. At this stage compliance with the syntactical rules of the assembler language becomes imperative in respect of the contents of the macro definition string (cf. section 5.1).

### 5.2.1 Syntax of Macro Call

The macro call may take either of the following three forms depending on the structure of the corresponding macro definition:



```

user-symbol ↓
user-symbol Δ string1 <Δstring2Δ . . . Δstringn> ↓
user-symbol<Δ> [string1Δ . . . Δstringn] ↓

```

In these formats user-symbol is the name included in the macro definition and by which the definition is referenced. Similarly each string is the actual argument which will replace the corresponding formal one during expansion of the macro.

Of the macro call formats specified above, the first one is obviously intended for use in those situations where no formal arguments appear in the macro definition, while the other two formats are intended for use in situations where formal arguments do appear in the definition and where actual arguments consequently must appear in the call. The two types of call with arguments furthermore reflect a difference in the application of macros. If the second format - in which the list of actual arguments is not enclosed in square brackets - is used, a carriage return character will be inserted before the first byte of the expanded macro, while this will not be the case if the third format is used.

Thus use of the second format implies that the expanded macro forms one or more separate individual lines of object program and therefore should be used in instances where this is compatible with the structure of the macro definition.

Correspondingly use of the third format implies, that the expanded macro forms an integral part of a program line and consequently it should be used in instances where this is relevant; an example of this could be a case where a macro call is used to specify the contents of an instruction field.

The strings appearing in the argument list of the macro call may contain any ASCII character, but attention should be paid to the fact, that the individual strings of the argument list must be separated by one single break character; the first appearance of a comma, space or tabulation character will terminate the individual string, in consequence of which any succeeding commas, spaces or tabulation characters will be regarded by the assembler as leading characters of the next string in the list.

### 5.2.2. Substitution of Arguments

The list of actual arguments contained in a macro call (employing the relevant formats given above) will always be accepted by the assembler in strict numerical sequence as implied by the indices of the format. The position in the sequence of the actual argu-

ment is the key, which will determine what actual argument should replace a specified dummy argument of the definition.

If the dummy argument appearing in the definition employs the format  $\underline{d}$  - and where the  $\underline{d}$  consequently is one of the digits 1 to 9 (cf. section 5.1.1.2) - the digit in question directly refers to the position of the actual argument in the argument list of the call; thus if the dummy argument  $\uparrow 4$  appears in the definition, it will during expansion of the macro everywhere be replaced by the actual argument string<sub>4</sub>. If the dummy argument appearing in the definition employs either the format  $\uparrow \underline{c}$  or the format  $\uparrow ?\underline{c}$  (cf. section 5.1.1.2), the  $\underline{c}$  or  $?\underline{c}$  will be a symbol; during expansion of the macro the value of this symbol will be established by the assembler and this value then will refer to the position of the actual argument in the argument list of the call.

The assembler will not accept more than 63 arguments in a macro call argument list. As a further consequence of this limitation it should be noted, that a dummy argument of the form  $\uparrow \underline{c}$  or  $\uparrow ?\underline{c}$  must yield a value in the range:  $1 \leq \text{value} \leq 63$ .

If the number of arguments in the argument list exceeds the number of dummy arguments required by the definition the superfluous arguments will be ignored. If however no dummy arguments at all were specified by the definition, the appropriate format of the call should be used.

### 5.3 Repetitive and conditional operations in Macros

A macro definition may include statements specifying repetitive and/or conditional operations, i.e. statements employing the program control directives `.DO` and/or `.IF` (this latter in its various forms). Such program control directives must be terminated by the directive `.ENDC` inside the macro where the corresponding `.DO` or `.IF` directive first appear.

If the terminating `.ENDC` directive does not appear before the character `%` - which will of course terminate the macro definition - the repetitive or conditional operation aimed at will be disregarded. An apparent exception to this rule is shown in the following example:-

```

.MACRO FIN
.ENDC
%
.MACRO REP
.DO 3
. . . .
. . . .
FIN
%
REP
. . . .

```

It should be noted here, that the macro FIN is called inside the macro REP thus providing the necessary terminating directive for the .DO directive contained in the macro REP before that macro is actually terminated by the % character. If the macro FIN is not called inside REP, the .DO directive will be disregarded (and incidentally the .ENDC directive in the macro FIN will be listed with an error flag, as there will then be no corresponding .DO or .IF directive for it to terminate).

## 5.4 Listing of Macro Expansions

In general the program listing produced during assembly will include macro definitions, macro calls and the expanded macros in the program lines.

In contrast to this the binary object program file will only contain the equivalent binary code of the actual expanded macro including the appropriate actual arguments.

It should be borne in mind that output of a program listing or of an object file is an optional feature of the assembler, i.e. either or both can be suppressed if the programmer so wishes. In addition to this the programmer may selectively suppress listing of macro expansions by application of the .NOMAC directive; if this is done the program listing will solely contain the actual source program line, in which the macro is referenced, in its original form as written in the source text. The object file will of course still include the complete expansion of the macro.

The special character \*\* has no effect on listing if it appears inside a macro definition (cf. section 2.4.3). However, a kind of exception to this rule does exist, namely that if the first line of a macro definition contains this special character and if a

macro call employing the third format (i.e. arguments enclosed in square brackets) involves an argument list reaching beyond one line, then the listing will only contain the first line of arguments.

Note also, that if a macro call is followed by a comment, then the listing of the comment will not appear till after the expanded text of the macro.

## 5.5 Examples of Macros

This section contains two examples of macro usage and is mainly concentrated around actual output listings produced by the assembler. The first of these examples concerns a relatively small macro definition, which has been designed for the purpose of generating an input/output zone compatible with the MUS/DOMUS operating systems implemented on the RC 3600 series computers.

The second example shows the nesting of macros, while it simultaneously shows how this feature can be utilized to provide increased programming flexibility.

### 5.5.1 Macro ZONE

A zone is in principle a collection of information and associated storage areas necessary for the handling of data sets and consists of three parts: a zone descriptor, a number of share descriptors and a buffer area.

The zone descriptor contains information about the data set and the peripheral device, on which it is physically present; the share descriptors hold information about the current activities in the various sections of the buffer area; the buffer area is that part of CPU memory, which physically contains the descriptors and associated buffers. (cf. MUS System Introduction, Part I).

Referencing a zone is effected by allocating a name to it, which name then appears as an argument where input/output operations are concerned. The zone is not explicitly connected to any specific peripheral device, but even so the selection of zone and share size will usually have to be tuned to the requirements of the device actually being used as well as to the structure of the data being transferred in the operation. This is conveniently

accomplished by a zone generating procedure, where the desired structure can be reflected in variable quantities used as parameters during the actual generation. Thus the macro feature becomes an obvious instrument for this purpose.

When a zone is generated, the shares must be linked together in cyclical fashion; whereas the shares occupy consecutive positions inside the zone, the linking of the first and the second share (and so further on) is easily accomplished by incrementing a pointer, but to complete the cycle, the pointer must be reset to the original value - indicating the starting address of the first share - when it has reached the last share.

The macro in this first example is given the name ZONE and is called as follows:-

```
ZONE [ name, kind, bufs, size, form, reclgt, giveup, mask, conv ]
```

where:-

<u>name</u>	is the zone name; it must consist of 5 characters
<u>kind</u>	is an integer specifying the type of peripheral device
<u>bufs</u>	is the number of shares
<u>size</u>	is the size of the share (in words)
<u>form</u>	is an integer specifying the record format
<u>reclgt</u>	is the record length (in bytes)
<u>giveup</u>	is the name of the giveup action procedure, which is called if error conditions arise during transfer
<u>mask</u>	is the giveup mask
<u>conv</u>	is the name of the conversion table (by default: zero)

The macro definition of ZONE is as follows:

```

.MACRO ZONE
.TXT "↑1" ; NAME
Z+((↑3)*((↑4)+SSIZE)) ; ZONE SIZE
0 ; MODE
↑2 ; KIND
↑8 ; GIVEUP MASK
↑7 ; GIVEUP PROCEDURE
1 ; FILE
1 ; BLOCK
(↑9)B14 ; CONVERSION TABLE BYTE ADR.
.+Z-ZBUFF ; BUFFER
((↑4)+SSIZE) ; BUFFER SIZE
↑5 ; FORMAT CODE
↑6 ; RECORD LENGTH
(.+Z+SSIZE-ZFIRST)B14 ; FIRST BYTE
(.+Z+SSIZE-ZTOP)B14 ; TOP BYTE
.+Z-ZUSED ; USED SHARE
(↑4)B14 ; SHARE BYTE LENGTH
0 ; REMAINING BYTES
.BLK ZAUX ; AUXILLIARY WORDS
?NO= 1 ; SHARE COUNTER
?FSH= . ; LOCATION OF FIRST SHARE
.DO ↑3 ; GENERATE SHARES
.BLK 4 ; SHARE MESSAGE
.DO ?NO==↑3 ;
?FSH ; LAST POINTS TO FIRST
.ENDC NOTLAST ;
.+SSIZE+(↑4)-SNEXT ; NEXT SHARE
NOTLAST ;
0 ; SHARE STATE
(.+1)B14 ; FIRST SHARE BYTE
.BLK ↑4 ; SHARE BUFFER
?NO= ?NO+1 ; NEXT SHARE NUMBER
.ENDC ; END GENERATE SHARES
% ; END ZONE MACRO

```

The definition of the macro ZONE falls in two parts, the first of which contains the non-variable part of the zone description such as the establishing of correlation between the dummy arguments of the definition and the position of the specific parameters in the call. The second part of the definition contains the necessary instructions for the actual generating of the shares, which is performed by manipulation of the share descriptors in the nested .DO loops. In this second part of the definition two variables - ?NO and ?FSH - appear, which variables are used to ensure, that the proper linking of the shares, as previously described, is obtained. The variable ?NO is a count of the number of shares that have been generated, while the variable ?FSH contains the address of the first share thereby providing the necessary link.

When the macro ZONE is called, the resulting expansion will be

listed as shown below. The extent of the listing may be controlled by application of the `.NOLOC` directive; this feature is incorporated in the listing shown by including in the example two listings corresponding to two separate calls of `ZONE`. The first of these two calls - concerning the zone named "PTR<0><0>" - has been preceded by the directive `.NOLOC 0`, whereby a full listing of the expansion will be output. The second of the two calls - concerning the zone named "INPUT" - has been preceded by the directive `.NOLOC 1`, whereby a compressed listing of the expansion will be output.

In addition to the two examples - and following these - is included the cross-reference listing produced during assembly.

## Example 1

```

ZONE1:  ZONE  PTR<0><0>,1,3,80.,0,0,PTRGU,-2,0
00000 052120 .TXT "PTR<0><0>" ; NAME
        000122
        000000
00003 000437 Z+((3)*((80.)+SSIZE)) ; ZONE SIZE
00004 000000 0 ; MODE
00005 000001 1 ; KIND
00006 177776 -2 ; GIVEUP MASK
00007 001507 PTRGU ; GIVEUP PROCEDURE
00010 000001 1 ; FILE
00011 000001 1 ; BLOCK
00012 000000 (0)B14 ; CONVERSION TABLE BYTE ADR.
00013 000032 .+Z-ZBUFF ; BUFFER
00014 000127 ((80.)+SSIZE) ; BUFFER SIZE
00015 000000 0 ; FORMAT CODE
00016 000000 0 ; RECORD LENGTH
00017 000102 (.+Z+SSIZE-ZFIRST)B14 ; FIRST BYTE
00020 000102 (.+Z+SSIZE-ZTOP)B14 ; TOP BYTE
00021 000032 .+Z-ZUSED ; USED SHARE
00022 000240 (80.)B14 ; SHARE BYTE LENGTH
00023 000000 0 ; REMAINING BYTES
00024 000006 .BLK ZAUX ; AUXILLIARY WORDS
        000001 ?NO= 1 ; SHARE COUNTER
        000032 ?FSH= . ; LOCATION OF FIRST SHARE
        000003 .DO 3 ; GENERATE SHARES
00032 000004 .BLK 4 ; SHARE MESSAGE
        000000 .DO ?NO==3 ;
        ?FSH ; LAST POINTS TO FIRST
        .ENDC NOTLAST ;
00036 000161 .+SSIZE+(80.)-SNEXT ; NEXT SHARE
        [NOTLAST] ;
00037 000000 0 ; SHARE STATE
00040 000102 (.+1)B14 ; FIRST SHARE BYTE
00041 000120 .BLK 80. ; SHARE BUFFER
        000002 ?NO= ?NO+1 ; NEXT SHARE NUMBER
        .ENDC ; END GENERATE SHARES
00161 000004 .BLK 4 ; SHARE MESSAGE
        000000 .DO ?NO==3 ;
        ?FSH ; LAST POINTS TO FIRST
        .ENDC NOTLAST ;
00165 000310 .+SSIZE+(80.)-SNEXT ; NEXT SHARE
        [NOTLAST] ;
00166 000000 0 ; SHARE STATE
00167 000360 (.+1)B14 ; FIRST SHARE BYTE
00170 000120 .BLK 80. ; SHARE BUFFER
        000003 ?NO= ?NO+1 ; NEXT SHARE NUMBER
        .ENDC ; END GENERATE SHARE
00310 000004 .BLK 4 ; SHARE MESSAGE
        000001 .DO ?NO==3 ;
00314 000032 ?FSH ; LAST POINTS TO FIRST
        .ENDC NOTLAST ;
        .+SSIZE+(80.)-SNEXT ; NEXT SHARE
        [NOTLAST] ;
00315 000000 0 ; SHARE STATE
00316 000636 (.+1)B14 ; FIRST SHARE BYTE
00317 000120 .BLK 80. ; SHARE BUFFER
        000004 ?NO= ?NO+1 ; NEXT SHARE NUMBER
        .ENDC ; END GENERATE SHARES

```



## Example 2

```

ZONE2:  ZONE    INPUT,32.,2,256.,3,32.,INERR,-2,0
00437  047111  .TXT    "INPUT" ; NAME
00442  001050  Z+((2)*((256.)+SSIZE)) ; ZONE SIZE
00443  000000  0 ; MODE
00444  000040  32. ; KIND
00445  177776  -2 ; GIVEUP MASK
00446  001507  INERR ; GIVEUP PROCEDURE
00447  000001  1 ; FILE
00450  000001  1 ; BLOCK
00451  000000  (0)B14 ; CONVERSION TABLE BYTE ADR.
00452  000471  .+Z-ZBUFF ; BUFFER
00453  000407  ((256.)+SSIZE) ; BUFFER SIZE
00454  000003  3 ; FORMAT CODE
00455  000040  32. ; RECORD LENGTH
00456  001200  (.+Z+SSIZE-ZFIRST)B14 ; FIRST BYTE
00457  001200  (.+Z+SSIZE-ZTOP)B14 ; TOP BYTE
00460  000471  .+Z-ZUSED ; USED SHARE
00461  001000  (256.)B14 ; SHARE BYTE LENGTH
00462  000000  0 ; REMAINING BYTES
00463  000006  .BLK    ZAUX ; AUXILLIARY WORDS
00471  000004  .BLK    4 ; SHARE MESSAGE
00475  001100  .+SSIZE+(256.)-SNEXT ; NEXT SHARE
00476  000000  0 ; SHARE STATE
00477  001200  (.+1)B14 ; FIRST SHARE BYTE
00500  000400  .BLK    256. ; SHARE BUFFER
01100  000004  .BLK    4 ; SHARE MESSAGE
01104  000471  ?FSH ; LAST POINTS TO FIRST
01105  000000  0 ; SHARE STATE
01106  002216  (.+1)B14 ; FIRST SHARE BYTE
01107  000400  .BLK    256. ; SHARE BUFFER

```

## Cross-reference listing

INERR	001507		4/08	5/01					
PTRGU	001507		3/10	5/01					
ZONE	000000	MC	2/26	3/02	4/02				
ZONE1	000000		3/02						
ZONE2	000437		4/02						
?FSH	000471		3/25	3/29	3/40	3/51	4/22	4/23	4/28
?NO	000003		3/24	3/28	3/36	3/39	3/47	3/50	3/58
			4/22	4/23	4/27	4/28	4/32		

The macro ZONE used in this example has for reasons of clarity been restricted somewhat. It exhibits all necessary primary features, but in case of actual practical application it should be supplemented in such a way, that control of the number of parameters specified as well as control of the value of each parameter can be exercised. In further consequence of this - should the number or value of parameters exceed limits - provision must be made for descriptive error messages to be produced and output to the console.

### 5.5.2

#### Macro NDEF

The assembler language is often used for programming large data processing systems intended for the management of such tasks of a general nature, that - although they will be performed on behalf of several independent customers - they can conveniently be carried out by application of the same basic program. It will clearly be recognized, that such a situation obtains in very many instances concerning ordinary business administration - accounting etc.

In such systems the source program will usually consist of one or more general parts common to all users combined with subsections of a variable nature, which will allow each individual user to adapt the complete program to the circumstances peculiar to his own situation. An obvious example will be that of providing the option for the user of referring to his own, previously defined, filenames inside the program.

The example of the macro NDEF illustrates how such flexibility may be obtained by means of the macro feature - or even more specifically; the macro continuation feature.

The source program of the example is considered to consist of two fixed (general) parts, between which an optional (variable) part appears. The first fixed part contains a macro definition of the macro NDEF. Nested inside the definition of NDEF is a further macro definition of the macro NAMES.

The optional part contains exactly one call of the macro NDEF with a list of arguments in the shape of text strings. These text strings then are the actual filenames which it is desired to employ in the program proper.

During expansion following the call of NDEF the macro NAMES will

then be defined - including in the definition the actual filenames as they appeared as parameters of the call of NDEF.

It is now possible to call NAMES in the second fixed part of the program, in which call a single digit is used as argument - this single digit indicating the position of the filename wanted in which it occurred in the argument list of the macro NDEF. For convenience this digit should be identical with the channel number of that specific device, where the file with the corresponding name is to be found; if so, calling NAMES with the appropriate channel number will during expansion automatically create correspondence between the program, the specific peripheral device and that particular file, which is supposed to be available on that device.

The example as given below shows the definition of the macro NDEF, the expansion due to the call of NDEF in the optional part of the program and the expansion due to three successive calls of the macro NAMES in the second fixed part of the program. It should be noted, that for reasons of clarity all features of the complete program but the abovementioned have been omitted.

As with the previous example the listing below is given in two forms. The first listing is a full listing which has been obtained by calling the assembler with the command MODE.0 (cf. chapter 7).

The second listing is a compressed, normal listing, in which case the majority of the listing has been suppressed by the inclusion of the special character \*\*.

Example 1

```

;*****
;
;          FIXED PART 1
;
;*****
;
;          .MACRO NDEF
;          **.PUSH .NOMA
;          **.NOMA 1
X=          1
;
;          .MACRO NAMES
;          **.PUSH .NOCO
;          **.NOCO 1
;          **.Y= 1
←%
;          .DO .ARGCT
;          .MACRO NAMES
;          **.DO ←↑1==Y
;          .TXT "↑X"
;          **.ENDC
;          **.Y= Y+1
←%
;          X= X+1
;          .ENDC
;          .MACRO NAMES
;          **.NOCO .POP
←%
;          **.NOMA .POP
%
;
;*****
;
;          OPTION PART
;
;*****
;
;          NDEF ABC DEF LONGNAME
000000     **.PUSH .NOMA
000001     **.NOMA 1
000001 X=   1
;
;          .MACRO NAMES
;          **.PUSH .NOCO
;          **.NOCO 1
;          **.Y= 1
;
;          %
000003     .DO .ARGCT
;          .MACRO NAMES
;          **.DO ↑1==Y
;          .TXT "ABC"
;          **.ENDC
;          **.Y= Y+1
;
;          %
000002     X= X+1
;          .ENDC
;          .MACRO NAMES
;          **.DO ↑1==Y
;          .TXT "DEF"
;          **.ENDC
;          **.Y= Y+1

```

```

000003 %      X=      X+1
              .ENDC

              .MACRO  NAMES
              **.DO   ↑1==Y
              .TXT    "LONGNAME"
              **.ENDC
              **Y=    Y+1

000004 %      X=      X+1
              .ENDC

              .MACRO  NAMES
              **.NOCO .POP

000000 %      **.NOMA .POP
              ;
              ;*****
              ;
              ;      FIXED PART 2
              ;
              ;*****
              ;

CALL1:  NAMES  1
000000          **.PUSH .NOCO
000001          **.NOCO 1
000001          **Y=   1
000001          **.DO   1==Y
000001          .TXT   "ABC"
00000 041101
000103          **.ENDC
              **Y=    Y+1
000002          **.DO   1==Y
000000          .TXT   "DEF"
              **.ENDC
              **Y=    Y+1
000003          **.DO   1==Y
000000          .TXT   "LONGNAME"
              **.ENDC
              **Y=    Y+1
000004          **.NOCO .POP
000000

CALL2:  NAMES  2
000000          **.PUSH .NOCO
000001          **.NOCO 1
000001          **Y=   1
000001          **.DO   2==Y
000000          .TXT   "ABC"
              **.ENDC
              **Y=    Y+1
000002          **.DO   2==Y
000001          .TXT   "DEF"
00002 042504
000106          **.ENDC
              **Y=    Y+1
000003          **.DO   2==Y
000000          .TXT   "LONGNAME"
              **.ENDC
              **Y=    Y+1
000004          **.NOCO .POP
000000

```

```
CALL3: NAMES 3
000000 ** .PUSH .NOCO
000001 ** .NOCO 1
000001 ** Y= 1
000000 ** .DO 3==Y
          .TXT "ABC"
          ** .ENDC
000002 ** Y= Y+1
000000 ** .DO 3==Y
          .TXT "DEF"
          ** .ENDC
000003 ** Y= Y+1
000001 ** .DO 3==Y
00004 047514 .TXT "LONGNAME"
043516
040516
042515
000000
          ** .ENDC
000004 ** Y= Y+1
000000 ** .NOCO .POP
```

Example 2

```

;*****
;
;      FIXED PART 1
;
;*****
;
;      .MACRO  NDEF
;      **.PUSH .NOMA
;      **.NOMA 1
X=      1
;
;      .MACRO  NAMES
;      **.PUSH .NOCO
;      **.NOCO 1
;      **.Y=   1
;
;-----%
;      .DO      .ARGCT
;
;      .MACRO  NAMES
;      **.DO   ←↑1==Y
;      .TXT    "↑X"
;      **.ENDC
;      **.Y=   Y+1
;
;-----%
;      X=      X+1
;      .ENDC
;
;      .MACRO  NAMES
;      **.NOCO .POP
;
;-----%
;      **.NOMA .POP
;
;%
;
;*****
;
;      OPTION PART
;
;*****
;      NDEF   ABC DEF LONGNAME
;
;*****
;
;      FIXED PART 2
;
;*****
CALL1:  NAMES   1      .TXT   "ABC"
00000  041101
        000103
CALL2:  NAMES   2      .TXT   "DEF"
00002  042504
        000106
CALL3:  NAMES   3      .TXT   "LONGNAME"
00004  047514
        043516
        040516
        042515
        000000

```



## 6 Extended Capabilities

### 6.1 Generation of Numbers and Symbols

The DOMAC Assembler includes the possibility of generating numbers or symbols during the assembly process. This capability is especially convenient when handling tables or other similar arrays of information.

Bringing this particular capability into operation is accomplished by including the format:

\symbol

in the source program file. During assembly the string of characters \symbol will - wherever they appear - be replaced by a three-digit number. The actual number used will be equal to the current numerical value of symbol - expressed in the current input radix; if this value should exceed the limitations imposed by the available three digits, the number will be truncated so that only the three rightmost digits will be utilized.

The format \symbol may appear anywhere in the source line - subject to the context in which it is employed - and may either appear on its own or may immediately follow one or more numeric or alphabetic characters. If \symbol appears on its own it will after assembly constitute an integer number; if it immediately follows a sequence of numeric characters these two groupings will together constitute a number - integer or floating point; if the format immediately follows a sequence of alphabetic characters these two groupings will together constitute a symbol. The sequence of characters immediately preceding the format may contain any number of characters.

As an illustration of the rules given above consider the following examples:-

X =	37351	
TAB \ X:	. . . .	(Assembled as TAB351: . . . . )
Y =	4. \ X	(Assembled as Y = 4.351 )
Z =	Y+ \ X	(Assembled as Z = Y+351 )

When the format \symbol is used according to the rules given above the listing produced by the assembler will include the \,

the actual characters which constitute the symbol and the numerical value assigned to symbol. The object file produced by the assembler will however only contain the actual numerical value of symbol; consequently, if the format has been used in conjunction with one or more preceding alphabetic characters - thus assembling into a user-symbol - only the actual combination of letters and digits forming the user-symbol will appear in the optional cross-reference listing of the symbol table output at the end of the program listing. As an example of this consider the following source program:-

```

        .RDX 8
**      INDEX=0
        .DO 16.
        AX\INDEX=0
**      INDEX=INDEX+1
        .ENDC

```

which will produce the following program listing and - shown on the right of the listing - the corresponding entries of the cross-reference listing:-

```

        .RDX 8
        .DO 16.
        AX\INDEX000=0          AX000
        AX\INDEX001=0          AX001
        AX\INDEX002=0          AX002
        . . . . .
        . . . . .
        AX\INDEX017=0          AX017
        AX\INDEX020=0          AX020
        .ENDC

```

Note in the above example the use of the decimal point in the .DO directive to obtain interpretation of the numerical value in radix 10 instead of current input radix.

A further example follows showing the use of the symbol generation facility; this example - which includes columns 4 to 16 of the program listing - shows the storing of symbol values which will be referenced by the differing, generated, symbol names.

## Example:

```

                                .ENT   EX124
                                .EXTN   EX888
                                .EXTD   EX889,EX999
                                .NREL

                                .MACRO  EXMAC
                                1
                                %
                                EXMAC
00000'000001                   1
                                EXMAC
00001'000001                   1

                                000123   I=     123
                                000010   .DO    8.
00002'001230  EX\I123:         (I)B12
                                000124   I=     I+1
                                .ENDC
00003'001240  EX\I124:         (I)B12
                                000125   I=     I+1
                                .ENDC
00004'001250  EX\I125:         (I)B12
                                000126   I=     I+1
                                .ENDC
00005'001260  EX\I126:         (I)B12
                                000127   I=     I+1
                                .ENDC
00006'001270  EX\I127:         (I)B12
                                000130   I=     I+1
                                .ENDC
00007'001300  EX I130:         (I)B12
                                000131   I=     I+1
                                .ENDC
00010'001310  EX\I131:         (I)B12
                                000132   I=     I+1
                                .ENDC
00011'001320  EX\I132:         (I)B12
                                000133   I=     I+1
                                .ENDC

00012'021401   LDA      0   +1,3   ;
00013'000777   JMP      .-1       ;
00014'006004   SENDMESSAGE ;

```

## 6.2 Generation of Labels

Another extended capability of the DOMAC assembler, which closely resembles the generation of numbers previously mentioned, is the possibility of generating individual labels during the assembly process. The labels generated will be associated with the actual number of macro calls in the program, and this facility is consequently of particular usefulness where labels inside macros are concerned.

This particular facility is brought into operation by including the character \$ in the source program file. Under the provision, that this character is not read by the assembler in string mode, the assembler will on occurrence of the \$ replace it with three characters from the alphanumeric set: 0 to 9, A to Z.

The assembler maintains a count of the number of macro calls made in the program; this number is converted to radix 36 representation - which representation requires 36 digits to express the number corresponding exactly with the 36 characters of the above-mentioned alphanumeric set - and thus are determined the actual three characters, which will be used to replace the \$.

If macros are nested, the replacement characters for \$ in the outer macro will be saved and subsequently restored upon completion of expansion of the inner macro.

**NOTE:** The character \$ should not be used as the initial character of a label as replacement might result in the occurrence of a numeric character in the first character position of the label. This would mean that the label would be illegal.

As an example of the use of this capability consider the following example of a macro definition:

```
.MACRO BREAK
      DSZ   AXX1
L$=   .
```

%

If and when BREAK is called - and presupposing a sixfold repetition of the call - the assembler would generate the following:

L\$001=	.	
	DSZ	AXX1
L\$002=	.	
	DSZ	AXX1
L\$003=	.	
	DSZ	AXX1
L\$004=	.	
	DSZ	AXX1
L\$005=	.	
	DSZ	AXX1
L\$006=	.	
	DSZ	AXX1

The above shows the listing produced by the assembler. As in the case of generation of numbers and symbols, the object file will not contain the character \$ in the actual labels.

## 7 Assembler Operation

### 7.1 External Requirements

In order to operate the DOMAC assembler program it is necessary to have access to a computer installation comprising as a minimum the following elements:

- An RC3600 series CPU with 32K words core store
- A console device
- A direct access storage medium
- A magnetic tape unit or a paper tape reader
- The DOMUS operating system

### 7.2 Call of DOMAC Assembler

The DOMAC assembler program is called into operation by the DOMUS operation system S, to which is issued a utility program load command specifying DOMAC.

The load command must conform to the following format:

```
DOMAC<ΔMODE.modespec><ΔLIST.listfile><ΔBIN.binfile>;<ΔLINES.lines>
      <ΔPERM.permfile><ΔSYMB.symbfile><ΔMACRO.macrofile>
      <ΔXREF.xreffile>Δsourcespec
```

The optional parameters to the call must - if included in the call - contain user-defined specifications as indicated below:

`modespec` is a name containing a maximum of five characters. These characters and their resultant effect have been defined by RC as follows:

- A causes all semi-permanent symbols to be added to the cross reference listing
- O directs the assembler to ignore all source-program indications of listing suppression, i.e. to list all source lines unconditionally
- R causes those semi-permanent symbols, which are referred to in the source program, to be added to the cross-reference listing
- S directs the assembler to skip the second pass of the assembly and to create instead a new permanent symbol

- table with filename symbfile and macro definition filename macrofile
- X directs the assembler to omit the cross-reference listing
- listfile is the name of the file (or entry) to which the program listing should be output by the assembler. The listing itself is optional, and consequently no listing will be produced if listfile is omitted.
- binfile is the name of the file (or entry) to which the binary object program should be output by the assembler. The object program itself is an option, and consequently none will be produced if binfile is omitted.
- lines is an integer number indicating the number of lines to each page of output. The maximum number of lines per page is 63; if a larger number is specified, this maximum will actually be used. If lines is not specified, output will be produced with 60 lines per page.
- permfile is the name of that file from which the assembler should read the semi-permanent symbols. If permfile is omitted, the assembler will read semi-permanent symbols from the file named DOMPS.
- symbfile is the name of that file which should be used as the symbol table file. If symbfile is omitted, the assembler will use the file named DOMST for this purpose.
- macrofile is the name of that file onto which the assembler should write macro definition strings. If macrofile is omitted, the assembler will use the file named DOMMC for this purpose.
- xreffile is the name of that file onto which the assembler should write entries for the cross-reference listing. If xreffile is omitted, the assembler will use the file named DOMXF for this purpose.

The three files indicated by symbfile, macrofile and xreffile are normally intended as internal workfiles only. When the assembly has been carried out these files will be deleted.

NOTE: The filenames specified in the parameter list of the call must not coincide with the letter strings of the assembler commands

If the DOMAC assembler is called without any parameter list (except sourcespec) assembly will proceed as if the call had been:

```
DOMAC LINES.60 PERM.DOMPS SYMB.DOMST MACRO.DOMMC XREF.DOMXF
```

The final parameter in the call of DOMAC - sourcespec - is not optional, as it supplies information to the assembler about those source programs which are to be assembled.

The parameter sourcespec consists of a list of source program filenames, which must comply with the following format:

```
filename1 <cc>Δfilename2 <cc>Δ. . . filenamen <cc>
```

In this format cc indicates a choice of optional command characters to be specified by the operator. The options available are:

- /S in which case the assembler will skip the file indicated on pass 2.
- /N in which case the assembler will suppress the program listing of the file indicated.

If the option is not taken up, the assembler will proceed according to the specifications of the DOMAC call and the contents of the source file itself.

The sourcespec parameter list may contain any number of filenames; if it contains more than one filename, the source files will be assembled in sequence from left to right.

Below are given three examples of DOMAC calls:

- 1) The following call will assemble the source program named TEXT and output the resulting binary object program to the file named RLBIN. It will produce a program listing - including a cross-reference list - on lineprinter. (The lineprinter driver is described by its entry name on the direct access storage medium (disc)). The call will be:

```
DOMAC BIN.RLBIN LIST.$LPT TEXT
```



- 2) The following call will assemble the source programs named PARM and PROG and output the resulting binary object programs to the file named PROGR. It will produce a program listing - including a cross-reference list - which will be output to the file named PROGL, but this listing will only refer to the program PROG. The object program PARM will only contain parameters but no corestore locations due to the skip command appended to the filename. The call will be:

```
DOMAC BIN.PROGR LIST.PROGL PARM/S PROG
```

- 3) The following call will not perform any assembly as such, but it will create a permanent symbol table file named DOMPS and a macro definition file named DOMPM; these newly created files will contain the basic instruction definitions and the MUS system definitions which will be transferred from the magnetic tape files BIPAR and MUPAR respectively. The call must furthermore specify a non-existing file (NIHIL) as a parameter to the PERM. command; otherwise the call would endeavour to read semi-permanent symbols from DOMPS. The call will be:

```
DOMAC MODE,S SYMB.DOMPS MACRO.DOMPM PERM.NIHIL BIPAR MUPAR
```

## 7.3 Assembly Execution

To utilize fully all capabilities of the DOMAC assembler program it is advantageous to obtain some understanding of the internal structure of the assembler and hence also of its mode of operation. Such an understanding will furthermore lead to less difficulties in comprehending the underlying reasons for the various error messages output by the assembler and consequently make the correcting of errors more easily performed.

The DOMAC assembler program - which operates under the overall control of the DOMUS operating system S - is basically a program designed for the handling of a number of files. These files can be grouped into various categories, namely:

### External files:

This group of files consists of the user-defined source program files, listing files and binary object program files.

Permanent symbol files:

This group of files consists of the RC-defined files:

DOMPS	a file containing the semi-permanent symbols,
DOMPM	a file containing the semi-permanent macro definition strings.

User symbol files:

This group of files consists of the RC-defined files:

DOMST	a file to which is added the semi-permanent symbols and the user-defined symbols,
DOMMC	a file to which is added the semi-permanent macro definition strings and the user-defined macro definition strings,
DOMXF	a file to which is added the cross references for all user symbols.

The DOMAC assembler is brought into operation by issuing a call command to the operating system, which will then load the assembler into core memory simultaneously arranging for the reservation of resources for the assembler run.

When the assembler has been loaded, it will first of all carry out a check of the parameters with which it has been called. The source, list and binary (object) files are then opened, that is: the assembler gains access to these files by employing an algorithm similar to that used by the CONNECTFILE procedure (cf. DOMUS User's Guide, Part II). While the source files must exist before the assembler requests the opening of the files, the list and binary files may or may not exist at this stage. If they do not exist, they will be created in accordance with the parameters of the DOMAC call; if they do exist prior to the loading of the assembler, they should be catalogue entries to the direct access storage area.

After opening the external files, the assembler will create the user symbol files DOMST, DOMMC and - if the parameter specifying cross-referencing has been included in the call of DOMAC - DOMXF. The three user symbol files should not normally exist prior to the loading of the assembler; if they do, they will at this stage be deleted and subsequently re-created. It should be noticed, that the DOMAC call may specify other filenames for these files; those used here will be employed by default.

The assembler now copies the permanent symbol files to the appropriate user symbol files: DOMPS is copied onto DOMST and DOMPM onto DOMMC. The file DOMPS, which contains the semi-permanent symbols, may not exist. If this should be the case, the assembler will execute an implied .XPNG assembler directive (q.v.) thereby creating an empty symbol file in readiness for re-definition of semi-permanent symbols.

Now the assembler is ready to perform the actual assembly, which is executed in two passes of the source file.

During pass 1 the source file is scanned for user-defined semi-permanent symbols and ordinary user symbols, adding in the process such symbols to the user symbol file (DOMST). Similarly a scan of macro definitions is carried out and the appropriate file (DOMMC) receiving the necessary additions. Simultaneously the assembler will during this first pass perform a check of source file syntax

During pass 2 of the source file the actual assembly into 16-bit binary words is performed, while simultaneously the syntax error messages, the program listing and the relocatable binary object program is output to the appropriate peripheral devices.

When the assembly is completed, DOMAC will output the line:

```
nnnn SOURCE LINES IN ERROR
```

where nnnn is four digit number. This message will be output on the console device and - optionally - on the device used for program listing.

The assembler then concludes the run in the following manner: If the mode indicated in the DOMAC call is S - or if a cross-reference listing is to be produced - the core-resident parts of the files DOMST, DOMMC and DOMXF are written back onto the user symbol files. Furthermore, if a cross-reference listing is to be produced, DOMAC sends an internal request to that process under which it operates. The request takes the form of the command: DOMXR sybfile and causes the operating system to output the cross-reference listing on the appropriate peripheral device.

If a cross-reference listing is not wanted, a similar internal request is sent by DOMAC, but this request will take the form of an empty command: FINIS, which will cause the final termination of the assembler run.

## 7.4 DOMAC Error Messages

Operation of the DOMAC Assembler is of course subject to the condition that no faults are present during the run. Two types of faults may occur, namely errors in the source file being supplied as input to the assembler program and errors associated with the actual execution of the assembly. Of these two error types the second will usually be fatal and necessarily cause termination of processing, whereas source input errors do not affect the assembly as such, although they may of course prevent the ultimate production of a meaningful object program.

Corresponding to the two types of errors the DOMAC assembler will produce two types of error messages. The error messages, which will be output either to the console device or to the (optional) listing device, provide information for the programmer about the source of the faults causing the error condition, thereby making it much more easy to correct the faults. This applies particularly to source input errors.

The various error messages and their implications are listed in the following section.

### 7.4.1 Source Input Errors

Errors in the source file may concern either the syntax or the semantics of the language. Errors in syntax will be discovered by the assembler and the corresponding error messages output on the appropriate peripheral device. This is possible because syntax errors are related to the formal structure of the language; discovering deviations from this formal structure can therefore fairly easily be incorporated into the assembly process.

Semantic errors on the contrary need not necessarily be accompanied by any deviation from correct syntax; consequently no set routine for detecting errors of this type can be maintained and they will often be discovered only when the assembled object program is actually run. It does of course happen that semantic errors and syntax errors are interrelated, in which case correcting the syntax will simultaneously cause correction of possible semantic errors although this is by no means a certainty. In this connection it should be realized, that quite often several syntax errors likewise may be interrelated, so that correcting one will automatically cancel others.

Syntax errors are indicated by a one-letter code in the first three character positions of that line of the program listing which corresponds to the source line in which the error occurs. Consequently a maximum of three errors in any one source line may be indicated. The syntax error indication will appear as a part of the program listing as stated above. This program listing is however optional - if it has been omitted, the error indications will be output on the console device instead.

All syntax error codes are listed below together with a short explanation of the probable cause of the error as well as an example of source text containing that error.

- |   |                  |   |
|---|------------------|---|
| A | Addressing Error | <p>Indicates that a Program Flow Control instruction or a Data Transfer instruction references an address outside the possible addressing range.</p> <p>Example:</p> <pre> ISZ TEMP ; addressing error .LOC .+256.; TEMP:    0 </pre>   |
| B | Bad Character    | <p>Indicates that an illegal character appears in the line.</p> <p>Example:</p> <pre> A&amp;B:    1           ;% is illegal in a                         label </pre>   |
| C | Macro Error      | <p>Indicates either, that more than 64 arguments have been specified in a macro call, or that nested macro continuation has been attempted.</p>   |
| D | Radix Error      | <p>Indicates either, that an attempt to specify a current radix outside the legal range (2 - 20) has been made, or that an attempt to input a digit outside current radix has been made.</p> <p>Example:</p> <pre> .RDX 40 ;radix error .RDX 2  ; 23      ;radix error </pre> |

- |   |                        |   |
|---|------------------------|---|
| E | Equivalence Error      | Indicates that an undefined symbol appears to the right of an equals sign.<br>Example:<br><pre style="margin-left: 40px;">A=B          ;If B is undefined                 this will lead to                 an equivalence                 error.</pre> |
| F | Format Error           | Indicates the application of an illegal format for a given type of operation, for inst. too many or too few operands.<br>Example:<br><pre style="margin-left: 40px;">MOV 1,2,3,4 ; format error -                 too many operands</pre>               |
| G | Global Error           | Indicates an undefined external or entry symbol.<br>Example:<br><pre style="margin-left: 40px;">.ENT NIHIL ; NIHIL undefined .EXTN LOCAL ; LOCAL defined LOCAL:    0 ; .END      ;</pre>  |
| I | Input Parity Error     | Indicates that the character SUB (ASCII code 032 <sub>g</sub> ) has been found in source input. The character SUB will in the listing appear as back slash: \.<br>Example:<br><pre style="margin-left: 40px;">LDA 1 TMP ; parity error</pre>            |
| K | Conditional Error      | Indicates that an .ENDC directive occurs without any corresponding .DO directive.<br>Example:<br><pre style="margin-left: 40px;">.DO 14      ; . . . .    ; .ENDC       ; .ENDC       ; conditional error</pre>   |
| L | Location Counter Error | Indicates that a .LOC or a .BLK directive has been issued in which the <u>expression</u> is undefined or outside the legal range. Example:<br><pre style="margin-left: 40px;">.LOC -14    ; .LOC 30000. ; .BLK 10000. ; outside range</pre>             |

- M            Multiple Definition Error            Indicates that a symbol has been defined more than once in the program  
 Example:  
                       TWIN:    0 ;  
                       . . . . .  
                       TWIN:    0 ; repetition
- N            Number Error                    Indicates that a number has been specified in excess of the legal maximum (a single precision integer must be less than 65536).  
 Example:  
                       .RDX 10        ;  
                       100000.       ; above maximum
- O            Overflow Error                    Indicates either, that an instruction operation exceeds its legal maximum, or that use of the directives .PUSH, .POP and .TOP leads to over- or underflow.  
 Example:  
                       LDA 5,TEMP    ; operand too large
- P            Phase Error                        Indicates that an irregularity has been detected in between pass 1 and pass 2 of the assembly; typically such an irregularity could be a symbol with different values in either pass.  
 Example:  
 Consider the following two source files:
- |         |          |
|---------|----------|
| .TITL A | .TITL B  |
| .NREL   | .NREL    |
| 0       | PHASE: 1 |
| .EOT    | .END     |
- If those two source files are assembled by the call:  
                       DOMAC LIST,\$LPT A/S B  
 whereby the first file will be skipped on pass 2, then the label PHASE will cause phase error.

Q	Questionable Line	<p>Indicates either, that improper use of # or @ has been made, or that a page zero relocatable value has been used where an absolute relocatable value was expected.</p> <p>Example:</p> <pre> .ZREL      ; A:         0      ;            .NREL   ;            LDA 0 A,3 ; Improper displacement            MOV # 2,3 ; Improper use of # </pre>
R	Relocation Error	<p>Indicates either, that an expression will not yield a legal relocation characteristic, or that the expression contains symbols with page zero as well as normal relocation characteristics.</p> <p>Example:</p> <pre> .ZREL      ; Z:         0      ;            .NREL   ; N:         1      ;            N+N+N   ; Triple relocation            N+Z     ; Improper mixing </pre>
U	Undefined Symbol Error	<p>Indicates that an un-defined symbol has been used.</p> <p>Example:</p> <pre> .TITL UNDEF ; JMP  HOME  ; label undefined .END </pre>
V	Assembler Label Error	<p>Indicates erroneous usage of the directive .GOTO.</p>
X	Text Error	<p>Indicates an error in connection with an expression inside the test string.</p> <p>Example:</p> <pre> .TXT/TEXT ERROR: &lt;2+3+&gt;/; </pre>

#### 7.4.2 Run Time Errors

The second type of errors are those - usually fatal ones - which may occur during assembly and which are associated with the actual execution of the assembler program.



The majority of such errors will refer to faulty operation of input/output devices, but in addition the following errors may occur:

0270 \*\*\* INTERNAL ERROR: nnnnn

This message indicates, that an internal DOMAC error condition has arisen. This situation is extremely unlikely, but should it occur RC must be contacted.

0271 \*\*\* DOMAC BREAK, NO: nn

0272 \*\*\* INSUFFICIENT CORE

0273 \*\*\* PARAMETER ERROR

This message will usually indicate, that the DOMAC pre-assembly check of all specified source files has shown a discrepancy between the specification and the actual files present. It may also indicate a syntax error in the DOMUS load command.

0274 \*\*\* VIRTUAL CORE ERROR

This message indicates that an erroneous disc transfer operation has been detected. This particular transfer is a consequence of the fact, that the DOMAC assembler places symbols used during assembly on a disc file which is in reality functioning as a virtual core memory (thereby saving space in the real core memory). Transfers to this special file are checked (as are all other transfers to peripheral devices), and if that check indicates an un-recoverable error, assembly will be terminated and the above message output.

# Appendix A

## PREDEFINED SYMBOLS

A number of symbols used by the DOMAC assembler have been pre-defined by RC. Two sets of predefined symbols exist: permanent and semi-permanent symbols.

Permanent symbols - the majority of which are assembler directives - cannot be redefined by the user.

The list of semi-permanent symbols include the value of the symbol (in octal); all have absolute relocation characteristic.

### Permanent symbols

.	.EOT	.NREL
.ARGCT	.EXTA	.PASS
.BLK	.EXTD	.POP
.DALC	.EXTN	.PUSH
.DIAC	.EXTU	.RDX
.DICD	.GOTO	.RD XO
.DIO	.IFE	.TITL
.DIOA	.IFG	.TOP
.DISD	.IFL	.TXT
.DMR	.IFN	.TXTE
.DMRA	.LIST	.TXTF
.DO	.LOC	.TXIM
.DUSR	.MACRO	.TXIN
.DXOP	.MCALL	.TXIO
.EJEC	.MSG	.XPNG
.END	.NOCON	.ZREL
.ENDC	.NOLOC	
.ENT	.NOMAC	

Semi-permanent symbols

AC0	000017	CONBY	006173	GETAD	006357
AC1	000020	CONVT	000031	GETBY	006174
AC2	000021	CORE	000361	GETPO	006360
AC3	000022	CORES	000070	GETRE	006200
ADC	102000	COROU	000017	GOS	006000
ADD	103000	COUNT	000027	GOT	002000
ADDRE	000026	CPASS	006345	GOTO	006356
AFIRS	000065	CPOSI	000113	HACTI	000043
AND	103400	CPRIN	006341	HALT	063077
AREAP	000064	CPU	000077	HANSW	000044
BINDE	006232	CREAT	006346	HDELA	000045
BIT	000101	CRESE	000116	IEQ	102415
BREAD	000016	CRETU	000003	IGR	102433
BREAK	006012	CSEND	006364	ILS	102032
BSIZE	000012	CTERM	000114	INBLO	006205
BUF	000025	CTEST	006340	INC	101400
BUFFE	000011	CTOP	006367	INCHA	006207
CAC1S	000004	CTOUT	006342	INE	102414
CALL	006355	CUDEX	000053	ING	102432
CBUFF	000054	CUR	000040	INITC	006352
CCONV	000115	CUR2	000112	INL	102033
CCORO	000041	CWANS	006337	INNAM	006223
CDELA	006334	DECBI	006233	INTA	061477
CDEVI	000050	DELAY	000061	INTBR	000230
CDISC	000112	DEVTA	000370	INTDS	060277
CDUMP	000077	DIA	060400	INTEN	060177
CERAS	000111	DIB	061400	INTGI	000226
CEXIT	000001	DIC	062400	INTPR	006225
CHAIN	000002	DIV	073101	IORST	062677
CHANG	006350	DIVID	006177	ISZ	010000
CIDEN	177777	DOA	061000	JMP	000000
CLATO	000002	DOB	062000	JSR	004000
CLEAN	006011	DOC	063000	LATIM	000042
CLEAR	100166	DSZ	014000	LDA	020000
CLINT	000032	EFIRS	000057	LOOKU	006347
CLOSE	006220	EVENT	000007	M	000040
COM	100000	EXIT	000056	MASK	000067
COMLI	000362	FFIRS	000060	MCORO	000052
COMNO	000363	FREES	006210	MESS0	000006
COMON	006354	FREQU	000066	MESS1	000007

MESS2	000010	RECEI	000005	SIZE	000003
MESS3	000011	RECHA	006015	SKP	000001
MONIT	000054	REMOV	006351	SKPBN	063400
MOV	101000	RESER	000030	SKPBZ	063500
MOVE	006224	RETUR	006165	SKPDN	063600
MSEM	000051	RTIME	000074	SKPDZ	063700
MSKO	062077	RUNNI	000054	SLS	102033
MUL	073301	SADDR	000002	SNC	000003
MULTI	006176	SAVE	000024	SNE	102415
MZSTA	000234	SAVE1	000025	SNEXT	000004
NAME	000004	SAVE2	000026	SNG	102433
NEG	100400	SAVE3	000027	SNL	102032
NEXT	000000	SAVE4	000030	SNR	000005
NEXTO	006164	SAVE5	000031	SOFFL	000102
NIO	060000	SBLOC	000111	SOPER	000000
O	000025	SBN	000007	SPARI	000113
OP	000034	SBUSY	000103	SSIZE	000007
OPEN	006221	SCOUN	000001	SSPEC	000003
OPMAS	177776	SDATA	000112	SSTAT	000005
OUTCH	006212	SDEV1	000104	STA	040000
OUTEN	006214	SDEV2	000105	START	006014
OUTNL	006213	SDEV3	000106	STATE	000013
OUTOC	006216	SDISC	000101	STIME	000117
OUTSP	006211	SEARC	006010	STOPP	006013
OUTTE	006215	SEM	000114	SUB	102400
PC	000033	SENDA	006007	SZC	000002
PCWSI	000006	SENDE	000004	SZR	000004
PFIRS	000052	SENDM	006004	TABLE	000045
POWIN	000076	SEOF	000110	TIMER	000014
PREV	000001	SEQ	102414	TLENG	000036
PRIOR	000015	SETCO	006172	TOPDE	000464
PROCE	000054	SETEN	006353	TOPTA	000046
PROG	000012	SETIN	006170	TRANS	006204
PROGR	000071	SETPO	006217	TRECO	000047
PSPEC	000000	SETRE	006171	TREIU	000046
PSTAR	000001	SEZ	000006	WAIT	006002
PSW	000023	SFIRS	000006	WAITA	006005
PUTBY	006175	SGR	102432	WAITC	006336
PUTRE	006201	SIGCH	006344	WAITE	006006
PWSIZ	000014	SIGGE	006365	WAITG	006366
R	000032	SIGNA	006343	WAITI	006003
READS	060477	SILLE	000107	WAITO	006167

WAITS	006335	.1638	000102	.9	000125
WAITT	006202	.1B0	000101	.CLEA	002166
WAITZ	006222	.1B1	000102	.CLOS	002220
Z	000032	.1B10	000113	.CONB	002173
Z0	000024	.1B11	000114	.CR	000130
Z1	000025	.1B12	000115	.DIVI	002177
Z2	000026	.1B13	000116	.EDOC	000124
Z3	000027	.1B14	000117	.EVEN	000124
Z4	000030	.1B15	000120	.FF	000127
Z5	000031	.1B2	000103	.FREE	002210
ZAUX	000006	.1B3	000104	.GETB	002174
ZBLOC	000011	.1B4	000105	.GETR	002200
ZBUFF	000013	.1B5	000106	.INBL	002205
ZCONV	000012	.1B6	000107	.INCH	002207
ZFILE	000010	.1B7	000110	.LF	000126
ZFIRS	000017	.1B8	000111	.M16	000146
ZFORM	000015	.1B9	000112	.M256	000147
ZGIVE	000007	.2	000117	.M3	000114
ZKIND	000005	.2048	000105	.M4	000145
ZLENG	000016	.24	000132	.MESS	000123
ZMASK	000006	.25	000133	.MULT	002176
ZMODE	000004	.255	000143	.NAME	000116
ZN	000041	.256	000110	.NEXT	002164
ZNAME	000000	.3	000121	.NL	000126
ZREM	000023	.32	000113	.OPEN	002221
ZSHAR	000022	.3276	000101	.OPER	000035
ZSIZE	000014	.4	000116	.OUTB	002206
ZTOP	000020	.40	000134	.OUTC	002212
ZUSED	000021	.4096	000104	.OUTE	002214
.0	000055	.48	000135	.OUTN	002213
.1	000120	.5	000122	.OUTO	002216
.10	000126	.512	000107	.OUTS	002211
.1024	000106	.56	000136	.OUTT	002215
.12	000127	.6	000123	.PUTB	002175
.PUTR	002201	.SETC	002172	.SSIZ	000124
.REPE	002203	.SETI	002170	.TRAN	002204
.REIU	002165	.SETP	002217	.WAIT	002202
.RTC	00127	.SETR	002171	.Z	000150

## Appendix B

### I/O DEVICE AND MNEMONICS

Decimal code	Octal code	Mnemonic	Maskbit	Device
01	01			Extended Memory
02	02			
03	03			
04	04			
05	05	ASL		Automatic System Load
06	06			
07	07			
08	10	TTI	14	Teletype Input
09	11	TTO	15	Teletype Output
10	12	PTR	11	Paper Tape Reader
11	13	PTP	13	Paper Tape Punch
12	14	RTC	13	Real Time Clock
13	15	PLT	12	Incremental Plotter
		SPC2	9	Third Standard Parallel Controller
14	16	CDR	10	Card Reader
15	17	LPT	12	Line Printer
16	20	DSC	4	Disc Storage Channel
17	21	SPC	9	Standard Parallel Controller
18	22	SPC1	9	Second Standard Parallel Controller
19	23	PTR1	11	Second Paper Tape Reader
20	24	AMX3	2	Fourth 8 Channel Asynchronous Multiplexor
		TMX10	0	{ Second 64 Channel
21	25	TMX11	1	{ Asynchronous Multiplexor
22	26	TMX0	0	{ 64 Channel Asynchronous
23	27	TMX1	1	{ Multiplexor
24	30	MT	5	Magnetic Tape
25	31	PTP1	13	Second Paper Tape Punch
26	32	TTI2	14	Third Teletype Input OCP-Function Button Out
27	33	TTO2	15	Third Teletype Output OCP-Function Button In
28	34	TTI3	14	Fourth Teletype Input OCP-Numeric Keyboard In
29	35	TTO3	15	Fourth Teletype Output OCP-Display
		DISP	7	
30	36			OCP-Autoload
31	37	LPS	12	Serial Printer
32	40	REC	8	BSC Controller
33	41	XMT	8	

Decimal code	Octal code	Mnemonic	Maskbit	Device
34	42	REC1	8	Second BSC Controller
35	43	XMT1	8	
36	44	MT1	5	Second Magnetic Tape
37	45	CLP	12	Charaband Printer
38	46	FPAR	3	Inter Processor Channel Receiver
39	47	FPAX	3	Inter Processor Channel Transmitter
40	50	TTI1	14	Second Teletype Input
41	51	TTO1	15	Second Teletype Output
42	52	AMX	2	8 Channel Asynchronous Multiplexor
43	53	AMX1	2	Second 8 Ch. Asynchronous Mpx.
44	54	HLC	8	HDLC Controller
		FPAR2	3	Third Inter Processor Channel Receiver
45	55	HLC1	8	Second HDLC Controller
		FPAX2	3	Third Inter Processor Channel Transmitter
46	56	CDR1	10	Second Card Reader
47	57	LPT1	12	Second Line Printer
		LPS2	12	Third Serial Printer
48	60	SMX		Synchronous Multiplexor
49	61	FDD	7	Flexible Disc Drive
50	62	CRP	10	Card Reader Punch
51	63	CLP1	12	Second Charaband Printer
52	64	FDD1	7	Second Flexible Disc Drive
53	65	LPS3	12	Fourth Serial Printer
54	66	DTC	9	Digital Cartridge Controller
		LPS4	12	Fifth Serial Printer
55	67	LPS1	12	Second Serial Printer
56	70	DST		Digital Sense
57	71	DOT		Digital Output
58	72	CNT		Digital Counter
				Dial-up Controller
59	73	DKP	7	Moving Head Disc Channel
60	74	FPAR1	3	Second Inter Processor Channel Receiver
61	75	FPAX1	3	Second Inter Processor Channel Transmitter
62	76	AMX2	2	Third 8 Channel Asynchronous Multiplexor
63	77	CPU		Central Processor

NOTE: RC reserves the right to change codes and mnemonics without prior notification.

## Appendix C

### Relocatable binary block types

The relocatable binary object program optionally output by the DOMAC assembler must adhere to a specific format. Firstly, the overall structure must conform to a segmentation into a number of blocks, and secondly each block must conform to a format depending on the type of block in question.

The block structure must be as follows:

Title Block
Entry Blocks
Ext. Displ. Blocks
Relocatable Data & Ext. Addition Blocks
Normal ext. Blocks
Start Block

The actual number of these blocks present in a specific binary object program will depend on the purpose, which the program is intended to fulfil and which of course also is reflected in the contents of the source program file. The binary object program can however never consist of less than two blocks: the Title block and the Start block.

Relocatable Data Blocks and External Addition Blocks are merged and do not comply with any pre-described structure. The reason for this is, that the value produced by application of the .EXTA directive and held in the External Addition Block must be stored in memory in the appropriate location corresponding to the position of the .EXTA directive in the source program file. Consequently it may be necessary to terminate one Data block to make place for the External Addition block and then to continue the object file with further Data blocks. This procedure may of course repeat itself several times in any one object file depending on the number of times the .EXTA directive is used.



The format of the blocks themselves depends on the type of block in question and will for each type be described in the following. Some features of the block format are however common to all types and is thus conveniently described at first.

Each block consists of a number of 16-bit words; inside each block the words are referred to by a consecutive numbering of the words starting from 0. The total number of words is variable in most blocks except in the Title, External Addition and Start blocks.

In all blocks word 0 contains a number in the interval from  $000002_8$  to  $000010_8$  which indicates the type of block in question; the block type numbers will be apparent in the description of the individual blocks at the end of the appendix.

In all blocks word 1 contains a negative number (i.e. a number in two's-complement notation), which indicates the total number of words in the block. In the Title, External Addition and Start blocks the word count will be  $-3$  ( $177775_8$ ),  $-4$  ( $177774_8$ ) and  $-1$  ( $177777_8$ ) respectively as indicated in the description of the individual block formats; in Data blocks the word count will never exceed  $-15$  ( $177761_8$ ) while in all other block types the word count will never exceed  $-45$  ( $177723_8$ ).

In all blocks words 2 to 4 contain information about relocation characteristics of addresses, data and symbols wherever applicable. This information is contained in groups of three bits each, where the different relocation characteristics are represented by the bit configurations shown in the table below:

Relocation characteristic	Bit configuration
Illegal	000
Absolute	001
Normal relocatable	010
Normal byte relocatable	011
Page zero relocatable	100
Page zero byte relocatable	101
Externally defined displacement	110
Illegal	111

Although strictly speaking not common to all block types, it is nevertheless convenient here to detail the utilization of the relocation flag words of the different block types:

In the Title block all bits of words 2 to 4 are zero.

In Entry blocks bits 0 to 2 of word 2 indicate the relocation characteristic of the equivalence held in word 8; bits 3 to 5 indicate that of the equivalence held in word 11 and so forth. The block will be able to contain a maximum of 15 equivalences corresponding to a maximum of 45 bits necessitating the use of three words. In all three words the superfluous bit 15 will be set to zero; the same applies to those relocation flag bits not utilized if the block length is less than its maximum.

In Relocatable Data blocks bits 0 to 2 of word 2 indicate the relocation characteristic of the address held in word 6; bits 3 to 5 indicate that of the data held in word 7 and so forth. Data blocks may as a maximum utilize 45 bits to indicate relocation characteristics; superfluous bits will be set to zero.

In Normal External blocks bits 0 to 2 of word 2 indicate the relocation characteristic of the address held in word 8 and so forth - the situation being completely analogous to that of entry blocks.

In External Addition blocks bits 0 to 2 of word 2 indicate the relocation characteristic of the load address held in word 6; bits 3 to 5 indicate that of the value held in word 9 and all remaining bits of words 2 to 4 are set to zero.

In the Title, Entry, External Displacement, Normal External and External Addition blocks symbol entries are arranged in groups of three words. The two words appearing at the beginning of each group contain the name of the symbol and a flag indicating the type of symbol. The symbol name is written into bits 0 to 15 of the first word and bits 0 to 10 of the second word; condensing the symbol name thus into 27 bits is accomplished by using numerical representation to base 40 (cf. Appendix D). The remaining five bits of the second word hold the type flag; the bit configurations representing the different types are given in the table on the following page:

Type of symbol	Bit configuration
Entry symbol (.ENT)	00000
Normal External symbol (.EXTN)	00001
External Addition symbol (.EXTA)	00010
External Displacement symbol (.EXTD)	00011
Title symbol (.TITL)	00100

### Title Block (.TITL)

The format of the Title block is as follows:

Word no.:	0	000007 <sub>8</sub>
	1	word count = -3
	2	000000 <sub>8</sub>
	3	000000 <sub>8</sub>
	4	000000 <sub>8</sub>
	5	checksum
	6	title in radix 40
	7	representation   flags
	8	000000 <sub>8</sub>

Entry Block (.ENT)

The format of Entry blocks is as follows:

Word no.:	0	000003 <sub>8</sub>	
	1	word count	
	2	relocation flags 1	
	3	relocation flags 2	
	4	relocation flags 3	
	5	checksum	
	6	symbol in radix 40	
	7	representation	flags
	8	equivalence	
	9	. . . . .	
	.	. . . . .	
	.	. . . . .	
3 - word count		symbol in radix 40	
4 - word count		representation	flags
5 - word count		equivalence	

External Displacement Block (.EXTD)

The format of External Displacement blocks is as follows:

Word no.:	0	000004 <sub>8</sub>	
	1	word count	
	2	relocation flags 1	
	3	relocation flags 2	
	4	relocation flags 3	
	5	checksum	
	6	symbol in radix 40	
	7	representation	flags
	8	077777 <sub>8</sub>	
	9	. . . . .	
	.	. . . . .	
	.	. . . . .	
	3 - word count	symbol in radix 40	
	4 - word count	representation	flags
	5 - word count	077777 <sub>8</sub>	

In External Displacement Blocks the relocation flag bit configuration will invariably be 110 as previously indicated in the table.

Relocatable Data Block

The format of Relocatable Data blocks is as follows:

Word no.:	0	000002 <sub>8</sub>
	1	word count
	2	relocation flags 1
	3	relocation flags 2
	4	relocation flags 3
	5	checksum
	6	address
	7	data
	.	. . . . .
	5 - word count	data

Normal External Block (.EXTN)

The format of Normal External blocks is as follows:

Word no.:	0	000005 <sub>8</sub>	
	1	word count	
	2	relocation flags 1	
	3	relocation flags 2	
	4	relocation flags 3	
	5	checksum	
	6	symbol in radix 40	
	7	representation	flags
	8	address of last reference	
	9	. . . . .	
	.	. . . . .	
	.	. . . . .	
	3 - word count	symbol in radix 40	
	4 - word count	representation	flags
	5 - word count	address of last reference.	

External Addition Block (.EXTA)

The format of External Addition blocks is as follows:

Word no.:	0	000010 <sub>8</sub>	
	1	word count = -4	
	2	relocation flags 1	
	3	000000 <sub>8</sub>	
	4	000000 <sub>8</sub>	
	5	checksum	
	6	load address	
	7	symbol in radix 40	
	8	representation	flags
	9	value	

Start Block

The format of the Start block is as follows:

Word no.:	0	000006 <sub>8</sub>	
	1	word count = -1	
	2	relocation flags 1	
	3	000000 <sub>8</sub>	
	4	000000 <sub>8</sub>	
	5	checksum	
	6	address	



## Appendix D

### Condensed representation of symbols

Symbols recognized by the DOMAC assembler must contain a maximum of five characters chosen from the set: 0 to 9, A to Z, . and ? (cf. section 3.1). If symbols are less than five characters in length, the assembler will make up for this by adding the necessary number of null characters.

Ordinarily representation of one character will require one byte of 8 bits, and consequently the five characters of the symbol requires five bytes, that is: two-and-a-half words.

It is however possible to reduce this demand on space by adopting transcription of symbols into a base 40 notation, i.e. each of the 37 characters of the set mentioned above - plus the null character as the 38th character - are regarded as individual digits of a number to base 40. (Actually the base need only be 39, but this does not yield any further advantage).

In this context then, the characters and the numerical values which they represent as digits are as follows:

null	is the digit representing	0
0 to 9	are the digits representing	1 to 10 respectively
A to Z	are the digits representing	11 to 36 respectively
.	is the digit representing	37
?	is the digit representing	38

When the five-character symbol is interpreted as a five-digit number to base 40, the maximum value is:

$$\begin{aligned}
 ????? &= 38 \times 40^4 + 38 \times 40^3 + 38 \times 40^2 + 38 \times 40^1 + 38 \times 40^0 \\
 &= 97\,280\,000 + 2432\,000 + 60\,800 + 1\,520 + 38 \\
 &= 99\,774\,358
 \end{aligned}$$

This number is less than  $2^{27} = 1\,342\,178\,728$ , but it is larger than  $2^{26}$ , which means, that it can be represented by precisely 27 bits in memory, which is comfortably within two words of storage space. Using this representation, the first three characters of the symbol can then be held in 16 bits = one word (as  $??? = 62\,358 < 2^{16}$ ), while the remaining two characters can be held in 11 bits of the second word (as  $?? = 1558 < 2^{11}$ ). This representation is used in the binary object program blocks as shown in appendix C.

## Appendix E

### References

- (1) RC 3603 Programmer's Reference Manual, Copenhagen, 1977.
- (2) MUS System Introduction, Part I, Copenhagen, 1976.
- (3) DOMUS Linkage Editor, Copenhagen, 1977.
- (4) DOMUS User's Guide, Part II, Copenhagen, 1977.

**RETURN LETTER**

Title: *DOMAC, Domus Macro Assembler, User's Guide*

RCSL No.: *42-i 0833*

A/S Regnecentralen maintains a continual effort to improve the quality and usefulness of its publications. To do this effectively we need user feedback, your critical evaluation of this manual.

Please comment on this manual's completeness, accuracy, organization, usability, and readability:

---

---

---

---

Do you find errors in this manual? If so, specify by page.

---

---

---

---

How can this manual be improved?

---

---

---

---

Other comments?

---

---

---

---

---

Name: \_\_\_\_\_ Title: \_\_\_\_\_

Company: \_\_\_\_\_

Address: \_\_\_\_\_


Date: \_\_\_\_\_

Thank you

..... **Fold here** .....

..... **Do not tear - Fold here and staple** .....

**Affix  
postage  
here**

 **REGNECENTRALEN**  
**Information Department**  
**Falkoner Alle 1**  
**DK-2000 Copenhagen F.**  
**Denmark**

## **INTERNATIONAL**

### **EASTERN EUROPE**

A/S REGNECENTRALEN  
Glostrup, Denmark

## **SUBSIDIARIES**

### **AUSTRIA**

RC - SCANIPS COMPUTER  
HANDELSGESELLSCHAFT mbH  
Vienna

### **FINLAND**

OY RC - SCANIPS AB  
Espoo

### **FRANCE**

RC - COMPUTER S.a.r.l.  
Paris

### **HOLLAND**

REGNECENTRALEN BEEHER B.V.  
Rotterdam

### **NORWAY**

A/S RC - SCANIPS  
Oslo

### **SWEDEN**

RC - SCANIPS AB  
Stockholm

### **SWITZERLAND**

RC - SCANIPS (SCHWEIZ) AG  
Basel

### **UNITED KINGDOM**

REGNECENTRALEN LTD.  
London

### **WEST GERMANY**

RC - COMPUTER G.m.b.H.  
Hannover

## **REPRESENTATIVES**

### **HUNGARY**

HUNGAGENT AG  
Budapest

### **KUWAIT**

KUWAITI DANISH COMPUTER CO. S.A.K.  
Safat

## **TECHNICAL ADVISORY REPRESENTATIVES**

### **POLAND**

ZETO  
Wroclaw

### **RUMANIA**

I.I.R.U.C.  
Bucharest

### **HUNGARY**

NOTO-OSZV  
Budapest

### **CZECHOSLOVAKIA**

KSNP KANCELARSKE STROJE N.P.  
Praha

**RC** REGNECENTRALEN  
**Scanips**  
**COMPUTER**

**HEADQUARTERS: FALKONER ALLE 1; DK-2000 COPENHAGEN F · DENMARK**  
**Phone: (01) 10 53 66 · Telex: 162 62 rc hq dk · Cables: regnecentralen**