flat assembler

Documentation and tutorials.

flat assembler g

User Manual

This document describes the syntax of flat assembler g language, with basic examples. It was written with an assumption that it would be read sequentially and at any moment it uses only the concepts and constructions that have been introduced earlier. However it should be possible to jump right to the section that interests the reader, and then go back to earlier parts only when it is needed in order to better understand the later ones.

Table of contents

0. Executing the assembler

To start assembly from the command line it is necessary to provide at least one parameter, the name of a source file, and optionally a second one - name of the destination file. If the assembly is successful, the generated output is written into the destination and a short summary is displayed, otherwise an information about errors is shown. The maximum number of presented errors can be controlled with an additional "-e" switch (by default no more than one error is presented). The "-p" switch controls the maximum number of passes the assembler is going to attempt. This limit is by default set to 100. The "-r" switch allows to set up the limit of the recursion stack, that is the maximum allowed depth of entering macroinstructions and including additional source files. The "-v" switch can enable showing all the lines from this stack when reporting an error (by default the assembler tries to select only the lines that are likely the most informative, but this simple heuristic may not always be correct). If "-v" switch is used with value 2, it in addition makes all the messages displayed by commands from the source text to be shown in real time (in every consecutive pass). The "-i" switch allows to insert any command at the beginning of processed source.

1. Fundamental syntax rules

Every command in the assembly language occupies a single line of text. If a line contains the semicolon character, everything from that character up to the end of the line is treated as a comment and ignored by the assembler. The main part of a line (i.e. excluding the comment) may end with the backslash character and in such case the next line from the source text is going to be appended to this one. This allows to split any command across multiple lines, when needed. From now on we will refer to a source line as an entity obtained by stripping comments and joining the lines of text connected with backslash characters.

The text of source line is divided into syntactical units called tokens. There is a number of special characters that become separate tokens all by themselves. Any of the characters listed below is such a syntactical unit:

```
+-/*=<>()[]{}:?!,.|&~#`\
```

Any contiguous (i.e. not broken by whitespace) sequence of characters other than the above ones becomes a single token, which can be a name or a number. The exception to this rule is when a sequence starts with the single or the double quote character. This defines a quoted string and it may contain any of the special characters, whitespace and even semicolons, as it ends only when the same character that was used to start it is encountered. The quotes that are used to enclose the string do not become a part of the string themselves. If it is needed to define a string containing the same character that is used to enclose it, the character needs to be doubled inside the string - only one copy of the character will become a part of the string, and the sequence will continue.

Numbers are distinguished from names by the fact that they either begin with a decimal digit, or with the "$" character followed by any hexadecimal digit. This means that a token can be considered numeric even when it is not a valid number. To be a correct one it must be one of the following: a decimal number (optionally with the letter "d" attached at the end), a binary number followed by the letter "b", an octal number followed by the letter "o" or "q", or a hexadecimal number either prepended with "$" or "0x", or followed by the character "h". Because the first digit of a hexadecimal number can be a letter, it may be needed to prepend it with the digit zero in order to make it recognizable as a number. For example, "0Ah" is a valid number, while "Ah" is just a name.

2. Symbol identifiers

Any name can become a defined symbol by having some meaning (a value) assigned to it. One of the simplest methods of creating a symbol with a given value is to use the "=" command:

```
a = 1
```

The ":" command defines a label, that is a symbol with a value equal to the current address in the generated output. At the beginning of the source text this address is always zero, so when the following two commands are the first ones in the source file, they define symbols that have identical values:

```
first:
second = 0
```

Labels defined with ":" command are special constructs in assembly language, since they allow any other command (including another label definition) to follow in the same line. This is the only kind of command that allows this.

What comes before the ":" or "=" character in such definition is a symbol identifier. It can be a simple name, like in the above samples, but it may also contain some additional modifiers, described below.

When a name in a symbol definition has the "?" character appended to it (with no whitespace between them), the symbol is case-insensitive (otherwise it would be defined as case-sensitive). This means that the value of such symbol may be referred to (as in an expression to the right of the "=" character) by the name being any variant of the original name that differs only in the case of letters. Only the cases of the 26 letters of the English alphabet are allowed to differ, though.

It is possible to define a case-sensitive symbol that clashes with a case-insensitive one. Then the case-sensitive symbol takes precedence and the more general one is used only when corresponding case-sensitive symbol is not defined. This can be remedied by using the "?" modifier, since it always means that the name followed by it refers to the case-insensitive symbol.

```
tester? = 0
tester = 1
TESTER = 2
x = tester       ; x = 1
y = Tester       ; y = 0
z = TESTER       ; z = 2
t = tester?      ; t = 0
```

Every symbol has its own namespace of descendants, called child namespace. When two names are connected with a dot (with no whitespace in between), such identifier refers to an entity named by the second one in the namespace of descendants to the symbol specified by the first one. This operation can be repeated many times within a single identifier, allowing to refer to descendants of descendants in a chain of any length.

```
space:
space.x = 1
space.y = 2
space.color:
space.color.r = 0
```

```
space.color.g = 0
space.color.b = 0
```

Any of the names in such chain may optionally be followed by the "?" character to mark that it refers to a case-insensitive symbol. If "?" is inserted in the middle of the name (effectively splitting it into separate tokens) such identifier is considered a syntactical error.

When an identifier starts with a dot (in other words: when the name of the parent symbol is empty), it refers to the symbol in the namespace of the most recent regular label defined before current line. This allows to rewrite the above sample like this:

```
space:
.x = 1
.y = 2
.color:
.color.r = 0
.color.g = 0
.color.b = 0
```

After the "space" label is defined, it becomes the most recently defined normal label, so the following ".x" refers to the "space.x" symbol and then the ".color" refers to the "space.color".

The "namespace" command followed by a symbol identifier changes the base namespace for a section of source text. It must be paired with the "end namespace" command later in the source to mark the end of such block. This can be used to again rewrite the above sample in a different way:

```
space:
namespace space
        x = 1
        y = 2
        color:
        .r = 0
        .g = 0
        .b = 0
end namespace
```

When a name is not preceded by a dot, and as such it does not have explicitly specified in what namespace the symbol resides, the assembler looks for defined symbol in the current namespace, and if none is found, in the consecutive namespaces of parent symbols, starting from the namespace containing the parent symbol of current namespace. If no defined symbol with such name is found, it is assumed that the name refers to the symbol in the current namespace (and unless there is "?" character after such name, it is assumed that the symbol is case-sensitive). A definition that does not specify the namespace where the new symbol should be created, always makes a new symbol in the current base namespace.

```
global = 0
regional = 1
namespace regional
        regional = 2            ; regional.regional = 2
        x = global              ; regional.x = 0
        regional.x = regional   ; regional.regional.x = 2
        global.x = global       ; global.x = 0
end namespace
```

The comments in the above sample show equivalent definitions with respect to the original base namespace. Note that when a name is used to specify the namespace, the assembler looks for a defined symbol with such name to lookup in its namespace, but when it is a name of a symbol to be defined, it is always created within the current base namespace.

When the final dot of an identifier is not followed by any name, it refers to the parent symbol of the namespace that would be searched for a symbol if there was a name after this dot. Adding such dot at the end of an identifier may appear redundant, but it can be used to alter the way the definition of a symbol works, because it forces the assembler to look for an already existing symbol that it can alter instead of squarely creating a new one in the current namespace. For instance, if in the fourth line of the previous example "regional." was put in place of "regional", it would rewrite a value of the original "regional" symbol instead of making a new symbol in the child namespace. Similarly, a definition formed this way may assign a new value to a symbol regardless of whether it was previously defined as case-insensitive or not.

If an identifier is just a single dot, by the above rules it refers to the most recent label that did not start with a dot. This can be applied to rewrite the earlier example in yet another way:

```
space:
namespace .
        x = 1
        y = 2
        color:
        namespace .
                r = 0
                g = 0
                b = 0
        end namespace
end namespace
```

It also demonstrates how namespace sections can be nested one within another.

The "#" may be inserted anywhere inside an identifier without changing its meaning. When "#" is the only character separating two name tokens, it causes them to be interpreted as a single name formed by concatenating the tokens.

```
variable = 1
varia#ble = var#iable + 2       ; variable = 3
```

This can also be applied to numbers.

Inside a block defined with "namespace" there is initially no label that would be considered base for identifiers starting with dot (even if there was a label that served this purpose outside of the block, it loses this status and is brought back to use only after the block is closed with "end namespace"). A similar thing also happens in the beginning of the source text, before any label has been defined. This is connected to additional rules concerning dots in identifiers.

When an identifier starts with a dot, but there is no label that would be a parent for it, the identifier refers to the descendant of a special symbol that resides in the current namespace but has no name. If an identifier starts with a sequence of two or more dots, the identifier refers to the descedant of a similar unnamed symbol, but it is a distinct one for any given number of dots. While the namespace accessed with a single starting dot changes every time a new regular label is defined, the special namespace accessed with two or more dots in the beginning of an identifier remains the same:

```
first:
        .child = 1
        ..other = 0
second:
        .child = 2
        ..another = ..other
```

In this example the meaning of the ".child" identifier changes from place to place, but the "..other" identifier means the same everywhere.

When two names inside an identifier are connected with a sequence of two or more dots, the identifier refers to the descendant of such special unnamed symbol in the namespace specified by the partial identifier before that sequence of dots. The unnamed child namespace is chosen depending on a number of dots and in this case the number of required dots is increased by one. The following example demonstrates the two methods of identifying such symbol:

```
namespace base
        ..other = 1
end namespace

result = base.#..other
```

The "#" character has been inserted into the last identifier for a better readability, but the plain sequence of three dots would do the same.

The unnamed symbol that hosts a special namespace can itself be accessed when an identifier ends with a sequence of two or more dots - thanks to the rule that an identifier which ends in a dot refers to the parent symbol of the namespace that would be accessed if there was a name after this dot. So in the context of the previous example the "base..." (or "base.#..") would refer to the unnamed parent of the namespace where the "other" symbol resides, and it would be the same symbol as identified by simple ".." inside the namespace of the "base" symbol.

Any identifier can be prepended with a "?" character and such modifier has an effect when it is used in a context where identifier could mean something different than a label or variable to be defined. This modifier then suppresses any other interpretation. For example, identifier starting with "?" is not going to be treated as an instruction, even if it is the first symbol on the line. This can be used to define a variable that shares a name with an existing command:

```
?namespace = 0
```

If such modified identifier is used in a place where it is evaluated and not defined, it still refers to the same symbol it would refer to in a definition. Therefore, unless identifier also uses a dot, it always refers to a symbol in the current namespace.

A number can be used in a role of a name inside an identifier, but not when it is placed at the beginning, because then it is considered a literal value. This restriction also may be bypassed by prepending an identifier with "?".

3. Basic symbol definitions

When a symbol is defined as a label, it must be the only definition of this symbol in the entire source. A value that is assigned to the symbol this way can be accesed from every place in the source, even before the label is actually defined. When a symbol is used before it is defined (this is often called forward-referencing) the assembler tries to correctly predict the value of the symbol by doing multiple passes over the source text. Only when all predictions prove to be correct, the assembler generates the final output.

This kind of symbol, which can only be defined once and thus have a universal value that can always be forward-referenced, is called a constant. All labels are constants.

When a symbol is defined with a "=" command, it may have multiple definitions of this kind. Such symbol is called variable and when it is used, the value from its latest definition is accessed. A symbol defined with such command may also be forward-referenced, but only when it is defined exactly once in the entire source and as such has a single unambiguous value.

```
a = 1           ; a = 1
a = a + 1       ; a = 2
a = b + 1       ; a = 3
b = 2
```

A special case of forward-referencing is self-referencing, when the value of a symbol is used in its own definition. The assembly of such construct is successful only when the assembler is able to find a value that is stable under such evaluation, effectively solving an equation. But due to the simplicity of the resolving algorithm based on predictions a solution may not be found even when it exists.

```
x = (x-1)*(x+2)/2-2*(x+1)       ; x = 6 or x = -1
```

The ":=" defines a constant value. It may be used instead of "=" to ensure that the given symbol is defined exactly once and that it can be forward-referenced.

The "=:" defines a variable symbol like "=", but it differs in how it treats the previous value (when such exists). While "=" discards the previous value, "=:" preserves it so it can later be brought back with the "restore" command:

```
a = 1
a =: 2          ; preserves a = 1
a = 3           ; discards a = 2 and replaces it with a = 3
restore a       ; brings back a = 1
```

A "restore" may be followed by multiple symbol identifiers separated with commas, and it discards the latest definition of every one of them. It is not considered an error to use "restore" with a symbol that has no active definition (either because it was never defined or because all of its definitions were already discarded earlier). If a symbol is treated with the "restore" command, it becomes a variable and can never be forward-referenced. For this reason "restore" cannot be applied to constants.

The "label" keyword followed by a symbol identifier is an alternative way of defining a label. In this basic form it is equivalent to a definition made with ":", but it occupies an entire line. However with this command it is possible to provide more settings for the defined label. The identifier may be optionally followed by the ":" token and then an additional value to be associated with this label (usually denoting the size of the labeled entity). The assembler has a number of built-in constants defining various sizes for this purpose, but this value can also be provided as a plain number.

```
label character:byte
label char:1
```

The ":" character may be omitted in favor of a plain whitespace, but it is recommended for clarity. After an identifier and an optional size, the "at" keyword may follow and then a value that should be assigned to the label instead of the current address.

```
label wchar:word at char
```

The built-in size constants are equivalent to the following set of definitions:

```
byte? = 1       ; 8 bits
word? = 2       ; 16 bits
dword? = 4      ; 32 bits
fword? = 6      ; 48 bits
pword? = 6      ; 48 bits
qword? = 8      ; 64 bits
tbyte? = 10     ; 80 bits
tword? = 10     ; 80 bits
dqword? = 16    ; 128 bits
xword? = 16     ; 128 bits
qqword? = 32    ; 256 bits
yword? = 32     ; 256 bits
dqqword? = 64   ; 512 bits
zword? = 64     ; 512 bits
```

The "element" keyword followed by a symbol identifier defines a special constant that has no fixed value and can be used as a variable in the linear polynomials. The identifier may be optionally followed by the ":" token and then a value to be associated with this symbol, called metadata of the element.

```
element A
element B:1
```

The metadata assigned to a symbol can be extracted with a special operator, defined in the next section.

4. Expression values

In every construction described so far where a value of some kind was provided, like after the "=" command or after the "at" keyword, it could be a literal value (a number or a quoted string) or a symbol identifier. A value can also be specified through an expression containing built-in operators.

The "+", "-" and "*" perform standard arithmetic operations on integers ("+" and "-" can also be used in a unary form - with only one argument). "/" and "mod" perform division with remainder, giving a quotient or a remainder respectively. Of these arithmetic operators "mod" has the highest precedence (it is calculated first), "*" and "/" come next, while "+" and "-" are evaluated last (even in their unary variants). Operators with the same precedence are evaluated from left to right. Parentheses can be used to enclose sub-expressions when a different order of operations is required.

The "xor", "and" and "or" perform bitwise operations on numbers. "xor" is addition of bits (exclusive or), "and" is multiplication of bits, and "or" is inclusive or (logical disjunction). These operators have higher precedence than any arithmetic operators.

The "shl" and "shr" perform bit-shifting of the first argument by the amount of bits specified by the second one. "shl" shifts bits left (towards the higher powers of two), while "shr" shifts bits right (towards zero), dropping bits that fall into the fractional range. These operators have higher precedence than other binary bitwise operations.

The "not", "bsf" and "bsr" are unary operators with even higher precedence. "not" inverts all the bits of a number, while "bsf" and "bsr" search for the lowest or highest set bit respectively, and give the index of that bit as a result.

All the operations on numbers are performed as if they were done on the infinite 2-adic representations of those numbers. For example the "bsr" with a negative number as an argument gives no valid result, since such number has an infinite chain of set bits extending towards infinity and as such contains no highest set bit (this is signaled as an error).

The "bswap" operator allows to create a string of bytes containing the representation of a number in a reverse byte order (big endian). The second argument to this operator should be the length in bytes of the required string. This operator has the same precedence as the "shl" and "shr" operators.

When a string value is used as an argument to any of the operations on numbers, it is treated as a sequence of bits and automatically converted into a positive number (extended with zero bits towards the infinity). The consecutive characters of a string correspond to the higher and higher bits of a number.

To convert a number back to a string, the "string" unary operator may be used. This operator has the lowest possible precedence, so when it precedes an expression, all of it is evaluated prior to the conversion. When conversion in the opposite direction is needed, simple unary "+" is enough to make a string become a number.

The length of a string may be obtained with the "lengthof" unary operator, one of the operators with the highest precedence.

The "bappend" operator appends a sequence of bytes of a string given by the second argument to the sequence of bytes given by the first one. If either of the arguments is a number, it becomes implicitly converted into a string. This operator has the same precedence as binary bitwise operations.

When a symbol defined with the "element" command is used in an expression the result may be a linear polynomial in a variable represented by the symbol. Only simple arithmetic operations are allowed on the terms of a polynomial, and it must stay linear - so, for example, it is only allowed to multiply a polynomial by a number, but not by another polynomial.

There are a few operators with high precedence that allow to extract the information about the terms of linear polynomial. The polynomial should come as the first argument, and the index of the term as the second one. The "element" operator extracts the variable of a polynomial term (with the coefficient of one), the "scale" operator extracts the coefficient (a number by which the variable is multiplied) and "metadata" operator gives back the metadata associated with the variable.

When the second argument is an index higher than the index of the last term of the polynomial, all three operators return zero. When the second argument is zero, "element" and "scale" give information about the constant term - "element" returns numeric 1 and "scale" returns the value of the constant term.

```
element A
linpoly = A + A + 3
vterm = linpoly scale 1 * linpoly element 1     ; vterm = 2 * A
cterm = linpoly scale 0 * linpoly element 0     ; cterm = 3 * 1
```

The "metadata" operator with an index of zero returns the size that is associated with the first argument. This value is definite only when the first argument is a symbol that has a size associated with it (or an arithmetic expression that contains such symbol), otherwise it is zero. There exists an additional unary operator "sizeof", which gives the same value as "metadata 0".

```
label table : 256
length = sizeof table   ; length = 256
```

The "elementof", "scaleof" and "metadataof" are variants of "element", "scale" and "metadata" operators with the opposite order of arguments. Therefore when "sizeof" is used in an expression it is equivalent to writing "0 metadataof" in its place. These operators have even higher precendence than their counterparts and are right-associative.

The order of the terms of the linear polynomial depends on the way in which the value was constructed. Every arithmetic operation preserves the order of the terms in the first argument, and the terms that were not present in the first argument are attached at the end in the same order in which they occurred in the second argument. This order only matters when extracting terms with appropriate operators.

The "elementsof" is another unary operator of the highest precedence, it counts the number of variable terms of a linear polynomial.

An expression may also contain a literal value that defines a floating-point number. Such number must be in decimal notation, it may contain "." character as a decimal mark and may be followed by the "e" character and then a decimal value of the exponent (optionally preceded by "+" or "-" to mark the sign of exponent). When "." or "e" is present, it must be followed by at least one digit. The "f" character can be appended at the end of such literal value. If a number contains neither "." nor "e", the final "f" is the only way to ensure that it is treated as floating-point and not as a simple decimal integer.

The floating-point numbers are handled by the assembler in the binary form. Their range and precision are at least as high as they are in the longest floating-point format that the assembler is able to produce in the output.

Basic arithmetic operations are allowed to have a floating-point number as any of the arguments, but none of the arguments may contain a non-scalar (linear polynomial) terms then. The result of such operation is always a floating-point number.

The unary "float" operator may be used to convert an integer value to floating-point. This operator has the highest precedence.

The "trunc" is another unary operator with the highest precedence and it can be applied to floating-point numbers. It extracts the integer part of a number (it is a truncation toward zero) and the result is always a plain integer, not a floating-point number. If the argument was already a plain integer, this operation leaves it unchanged.

The "bsr" operator can be applied to floating-point numbers and it returns the exponent of such number, which is the exponent of the largest power of two that is not larger than the given number. The sign of the floating-point value does not affect the result of this operation.

It is also allowed to use a floating-point number as the first argument to the "shl" and "shr" operators. The number is then multiplied or divided by the power of two specified by the second argument.

5. Symbol classes

There are three distinct classes of symbols, determining the position in source line at which the symbol may be recognized. A symbol belonging to the instruction class is recognized only when it is the first identifier of the command, while a symbol from the expression class is recognized only when used to provide a value of arguments to some command.

All the types of definitions that were described in the earlier sections create the expression class symbols. The "label" and "restore" are the examples of built-in symbols belonging to the instruction class.

In any namespace it is allowed for symbols of different classes to share the same name, for example it is possible to define the instruction named "shl", while there is also an operator with the same name - but an operator belongs to the expression class.

It is even possible for a single line to contain the same identifier meaning different things depending on its position:

```
?restore = 1
restore restore ; remove the value of the expression-class symbol
```

The third class of symbols are the labeled instructions. A symbol belonging to this class may be recognized only when the first identifier of the command is not an instruction - in such case the first identifier becomes a label to the instruction defined by the second one. If we treat "=" as a special kind of identifer, it may serve as an example of labeled instruction.

The assembler contains built-in symbols of all classes. Their names are always case-insensitive and they may be redefined, but it is not possible to remove them. When all the values of such symbol are removed with a command like "restore", the built-in value persists.

The rules concerning namespace apply equally to the symbols of all classes, for example symbol of instruction class belonging to the child namespace of latest label can be executed by preceding its name with dot. It should be noted, however, that when a namespace is specified through its parent symbol, it is always a symbol belonging to the expression class. It is not possible to refer to a child namespace of an instruction, only to the namespace belonging to the expression class symbol with the same name.

```
xor?.mask? := 10101010b
a = XOR.MASK    ; symbol in the namespace of built-in case-insensitive "XOR"

label?.test? := 0
a = LABEL.TEST  ; undefined unless "label?" is defined
```

Here the namespace containing "test" belongs to an expression-class symbol, not to the existing instruction "label". When there is no expression-class symbol that would fit the "LABEL" specifier, the namespace chosen is the one that would belong to the case-sensitive symbol of such name. The "test" is therefore not found, because it has been defined in another namespace - the one of case-insensitive "label".

6. Generating data

The "db" instruction allows to generate bytes of data and put them into the output. It should be followed by one or more values, separated with commas. When the value is numeric, it defines a single byte. When the value is a string, it puts the string of bytes into output.

```
db 'Hello',13,10        ; generate 7 bytes
```

The "dup" keyword may be used to generate the same value multiple times. The "dup" should be preceded by numeric expression defining the number of repetitions, and the value to be repeated should follow. A sequence of values may also be duplicated this way, in such case "dup" should be followed by the entire sequence enclosed in parentheses (with values separated with commas).

```
db 4 dup 90h            ; generate 4 bytes
db 2 dup ('abc',10)     ; generate 8 bytes
```

When a special identifier consisting of a lone "?" character is used as a value in the arguments to "db", it reserves a single byte. This advances the address in the output where the next data are going to be put, but the reserved bytes are not generated themselves unless they are followed by some

other data. Therefore if the bytes are reserved at the end of output, they do not increase the size of generated file. This kind of data is called uninitialized, while all the regular data are said to be initialized.

The "rb" instruction reserves a number of bytes specified by its argument.

```
        db ?                    ; reserve 1 byte
        rb 7                    ; reserve 7 bytes
```

Every built-in instruction that generates data (traditionally called a data directive) is paired with a labeled instruction of the same name. Such command in addition to generating data defines a label at address of generated data, with associated size equal to the size of data unit used by this instruction. In case of "db" and "rb" this size is 1.

```
        some db sizeof some     ; generate a byte with value 1
```

The "dw", "dd", "dp", "dq", "dt", "ddq", "dqq" and "ddqq" are instructions analogous to "db" with a different sizes of data unit. The order of bytes within a single generated unit is always little-endian. When a string of bytes is provided as the value to any of these instructions, the generated data is extended with zero bytes to the length which is the multiple of data unit. The "rw", "rd", "rp", "rq", "rt", "rdq", "rqq" and "rdqq" are the instructions that reserve a specified number of data units. The unit sizes associated with all these instructions are listed in table 1.

The "dw", "dd", "dq", "dt" and "ddq" instructions allow floating-point numbers as data units. Any such number is then converted into floating-point format appropriate for a given size.

The "emit" (with a synonym "dbx") is a data directive that uses the size of unit specified by its first argument to generate data defined by the remaining ones. The size may be separated from the next argument with a colon instead of a comma, for better readability. When the unit size is such that it has a dedicated data directive, the definition made with "emit" has the same effect as if these values were passed to the instruction tailored for this size.

```
        emit 2: 0,1000,2000      ; generate three 16-bit values
```

The "file" instruction reads the data from an external file and writes it into output. The argument must be a string containing the path to the file, it may optionally be followed by ":" and the numeric value specifying an offset within the file, next it may be followed by comma and the numeric value specifying how many bytes to copy.

```
        file 'data.bin'                 ; insert entire file
        excerpt file 'data.bin':10h,4   ; insert selected four bytes
```

Table 1 Data directives

Unit (bytes)	Generate data	Reserve data
1	`db` `file`	`rb`
2	`dw`	`rw`
4	`dd`	`rd`
6	`dp`	`rp`
8	`dq`	`rq`
10	`dt`	`rt`
16	`ddq`	`rdq`
32	`dqq`	`rqq`
64	`ddqq`	`rdqq`
*	`emit`	

7. Conditional assembly

The "if" instruction causes a block of source text to be assembled only under certain condition, specified by a logical expression that is an argument to this instruction. The "else if" command in the following lines ends the previous conditionally assembled block and opens a new one, assembled only when the previous conditions were not met and the new condition (an argument to "else if") is true. The "else" command ends the previous conditionally assembled

block and begins a block that is assembled only when none of the previous conditions was true. The "end if" command should be used to end the entire construction. There may be many or none "else if" commands inside and no more than one "else".

A logical expression is a distinct syntactical entity from the basic expressions that were described earlier. A logical expression consists of logical values connected with logical operators. The logical operators are: unary "~" for negation, "&" for conjunction and "|" for alternative. The negation is evaluated first, while "&" and "|" are simply evaluated from left to right, with no precedence over each other.

A logical value in its simplest form may be a basic expression, it then corresponds to true condition if and only if its value is not constant zero. Another way to create a logical value is to compare the values of two basic expressions with one of the following operators: "=" (equal), "<" (less than), ">" (greater than), "<=" (less or equal), ">=" (greater or equal), "<>" (not equal).

```
count = 2
if count > 1
        db '0'
        db count-1 dup ',0'
else if count = 1
        db '0'
end if
```

When linear polynomials are compared this way, the logical value is valid only when they are comparable, which is whey they differ in constant term only. Otherwise the condition like equality is neither universally true nor universally false, since it depends on the values substituted for variables, and assembler signals this as an error.

The "relativeto" operator creates a logical value that is true only when the difference of compared values does not contain any variable terms. Therefore it can be used to check whether two linear polynomials are comparable - the "relativeto" condition is true only when both compared polynomials have the same variable terms.

Because logical expressions are lazily evaluated, it is possible to create a single condition that will not cause an error when the polynomials are not comparable, but will compare them if they are:

```
if a relativeto b & a > b
        db a - b
end if
```

The "eqtype" operator can also be used to compare two basic expressions, it makes a logical value which is true when the values of the expressions are of the same type - either both are algebraic, both are strings or both are floating-point numbers. An algebraic type covers the linear polynomials and it includes the integer values.

The "eq" operator compares two basic expressions and creates a logical value which is true only when their values are of the same type and equal. This operator can be used to check whether a value is a certain string, a certain floating-point number or a certain linear polynomial. It can compare values that are not comparable with "=" operator.

The "defined" operator creates a logical value combined with a basic expression that follows it. This condition is true when the expression does not contain symbols that have no accessible definition. The expression is only tested for the availability of its components, it does not need to have a computable value. This can be used to check whether a symbol of expression class has been defined, but since the symbol can be accessible through forward-referencing, this condition may be true even when the symbol is defined later in source. If this is undesirable, the "definite" operator should be used instead, as it checks whether all symbols within a basic expression that follows have been defined earlier.

The basic expression that follows "defined" is also allowed to be empty and the condition is then trivially satisfied. This does not apply to "definite".

The "used" operator forms a logical value if it is followed by a single identifier. This condition is true when the value of specified symbol has been used anywhere in the source.

The "assert" is an instruction that signalizes an error when a condition specified by its argument is not met.

```
assert a < 65536
```

8. Macroinstructions

The "macro" command allows to define a new instruction, in form of a macroinstruction. The block of source text between the "macro" and "end macro" command becomes the text of macroinstruction and this sequence of lines is assembled in place of the original command that starts with identifier of

instruction defined this way.

```
macro null
        db 0
end macro

null            ; "db 0" is assembled here
```

The macroinstruction is allowed to have arguments only when the definition contains them. After the "macro" and the identifier of defined symbol optionally may come a list of simple names separated with commas, these names define the parameters of macroinstruction. When this instruction is then used, it may be followed by at most the same number of arguments separated with commas, and their values are assigned to the consecutive parameters. Before any line of text inside the macroinstruction is interpreted, the name tokens that correspond to any of the parameters are replaced with their assigned values.

```
macro lower name,value
        name = value and 0FFh
end macro

lower a,123h    ; a = 23h
```

The value of a parameter can be any text, not necessarily a correct expression. If a line calling the macroinstruction contains fewer arguments than the number of defined parameters, the excess parameters receive the empty values.

When a name of parameter is defined, it may be followed by "?" character to denote that it is case-insensitive, analogously to a name in a symbol identifier. There must be no whitespace between the name and "?". A definition of a parameter may also be followed by "*" to denote that it requires a value that is not empty, or alternatively by ":" character followed by a default value, which is assigned to the parameter instead of an empty one when no other value is provided.

```
macro prepare name*,value:0
        name = value
end macro

prepare x       ; x = 0
prepare y,1     ; y = 1
```

If an argument to macroinstruction needs to contain a comma character, the entire argument must be enclosed between the "<" and ">" characters (they do not become a part of the value). If another "<" character is encountered inside such value, it must be balanced with corresponding ">" character inside the same value.

```
macro data name,value
        name:
        .data db value
        .end:
end macro

data example, <'abc',10>
```

The last defined parameter may be followed by "&" character to denote that this parameter should be assigned a value containing the entire remaining part of line, even if it normally would define multiple arguments. Therefore when macroinstruction has just one parameter followed by "&", the value of this parameter is the entire text of arguments following the instruction.

```
macro id first,rest&
        dw first
        db rest
end macro

id 2, 7,1,8
```

When the name of a parameter is to be replaced with its value and it is preceded by "`" character (without any whitespace inbetween), the text of the value is embedded into a quoted string and this string replaces both the "`" character and the name of parameter.

```
macro text line&
        db `line
end macro

text x+1        ; db 'x+1'
```

The "local" is a command that may only be used inside a macroinstruction. It should be followed by one or more names separated with commas, and it declares that the names from this list should in the context of current macroinstruction be interpreted as belonging to a special namespace associated with this macroinstruction instead of current base namespace. This allows to create unique symbols every time the macroinstruction is called. Such declaration defines additional parameters with the specified names and therefore only affects the uses of those names that follow within the same macroinstruction. Declaring the same name as local multiple times within the same macroinstruction gives no additional effect.

```
macro measured name,string
        local top
        name db string
        top: name.length = top - name
end macro

measured hello, 'Hello!'        ; hello.length = 6
```

A parameter created with "local" becomes replaced with a text that contains the same name as the name of parameter, but has added context information that causes it to be identified as belonging to the unique local namespace associated with the instance of macroinstruction. This kind of context information is going to be discussed further in the section about symbolic variables.

A symbol that is local to a macroinstruction is never considered the most recent label that is base for symbols starting with dot. Moreover, its descendant namespace is disconnected from the main tree of symbols, so if "namespace" command was used with a local symbol as the argument, symbols from the main tree would no longer be visible (including all the named instructions of the assembler, even commands like "end namespace").

Just like an expression symbol may be redefined and refer to its previous value in the definition of the new one, the macroinstructions can also be redefined, and use the previous value of this instruction symbol in its text:

```
macro zero
        db 0
end macro

macro zero name
        label name:byte
        zero
end macro

zero x
```

And just like other symbols, a macroinstruction may be forward-referenced when it is defined exactly once in the entire source.

The "purge" command discards the definition of a symbol just like "restore", but it does so for the symbol of instruction class. It behaves in the same way as "restore" in all the other aspects. A macroinstruction can remove its own definition with "purge".

It is possible for a macroinstruction to use its own value in a recursive way, but to avoid inadvertent infinite recursion this feature is only available when the macroinstruction is marked as such by following its identifier with ":" character.

```
macro factorial: n
        if n
                factorial n-1
                result = result * (n)
        else
                result = 1
        end if
end macro
```

In addition to allowing recursion, such macroinstruction behaves like a constant. It cannot be redefined and "purge" cannot be applied to it.

A macroinstruction may in turn define another macroinstruction or a number of them. The blocks designated by "macro" and "end macro" must be properly nested one within the other for such definition to be accepted by the assembler.

```
macro enum enclosing
        counter = 0
        macro item name
                name := counter
                counter = counter + 1
        end macro
        macro enclosing
                purge item,enclosing
        end macro
end macro

enum x
        item a
        item b
        item c
x
```

When it is required that macroinstruction generates unpaired "macro" or "end macro" command, it can be done with special "esc" instruction. Its argument becomes a part of macroinstruction, but is not being taken into account when counting the nested "macro" and "end macro" pairs.

```
macro xmacro name
        esc macro name x&
end macro

xmacro text
        db `x
end macro
```

If "esc" is placed inside a nested definition, it is not processed out until the innermost macroinstruction becomes defined. This allows a definition containing "esc" to be placed inside another macroinstruction without having to repeat "esc" for every nesting level.

When an identifer of macroinstruction in its definition is followed by "!" character, it defines an unconditional macroinstruction. This is a special kind of instruction class symbol, which is evaluated even in places where the assembly is suspended - like inside a conditional block whose condition is false, or inside a definition of another macroinstruction. This allows to define instructions that can be used where otherwise a directly stated "end if" or "end macro" would be required, as in the following example:

```
macro proc name
        name:
        if used name
end macro

macro endp!
        end if
        .end:
end macro

proc tester
        db ?
endp
```

If the macroinstruction "endp" in the above sample was not defined as an unconditional one and the block started with "if" was being skipped, the macroinstruction would not get evaluated, and this would lead to an error because "end if" would be missing.

It should be noted that "end" command executes an instruction identified by its argument in the child namespace of case-insensitive "end" symbol. Therefore command like "end if" could be alternatively invoked with an "end.if" identifier, and it is possible to override any such instruction by redefining a symbol in the "end?" namespace. Moreover, any instruction defined within the "end?" namespace can then be called with the "end" command. This slighly modified variant of the above sample puts these facts to use:

```
macro proc name
        name:
        if used name
end macro

macro end?.proc!
        end if
        .end:
end macro

proc tester
        db ?
end proc
```

A similar rule applies to the "else" command and the instructions in the "else?" namespace.

When an identifier consisting of a lone "?" character is used as an instruction symbol in the definition of macroinstruction, it defines a special instruction that is then called every time a line to be assembled does not contain an unconditional instruction, and the complete text of line becomes the arguments to this macroinstruction. This special symbol can also be defined as an unconditional instruction, and then it is called for every following line with no exception. This allows to completely override the assembly process on portions of the text. The following sample defines a macroinstruction which allows to define a block of comments by skiping all the lines of text until it encounters a line with content equal to the argument given to "comment".

```
macro comment? ender
        macro ?! line&
                if `line = `ender
                        purge ?
                end if
        end macro
end macro

comment ~
         Any text may follow here.
~
```

An identifier consisting of two question marks can be used to define a special instruction that is called only as last resort, on lines that contain no recognizable instruction. This allows to intercept lines that would otherwise be rejected with "illegal instruction" message due to unknown syntax.

The "mvmacro" is an instruction that takes two arguments, both identifying an instruction-class symbols. The definition of a macroinstruction specified by the second argument is moved to the symbol identified by the first one. For the second symbol the effect of this command is the same as of "purge". This allows to effectively rename a macroinstruction, or temporarily disable it only to bring it back later. The symbols affected by this operation become variables and cannot be forward-referenced.

9. Labeled macroinstructions

The "struc" command allows to define a labeled instruction, in form of a macroinstruction. Except for the fact that such definition must be closed with "end struc" instead of "end macro", these macroinstructions are defined in the same way as with "macro" command. A labeled instruction is evaluated when the first identifier of a command is not an instruction and the second identifier is of the labeled instruction class:

```
struc some
        db 1
end struc

get some        ; "db 1" is assembled here
```

Inside a labeled macroinstruction identifiers starting with dot no longer refer to the namespace of a previously defined regular label. Instead they refer to the namespace of label with which the instruction was labeled.

```
struc POINT
        label . : qword
```

```
        .x dd ?
        .y dd ?
end struc

my POINT        ; defines my.x and my.y
```

Note that the parent symbol, which can be refered by "." identifier, is not defined unless an appropriate definition is generated by the macroinstruction. Furthermore, this symbol is not considered the most recent label in the surrounding namespace unless it gets defined as an actual label in the macroinstruction it labeled.

For an easier use of this feature, other syntaxes may be defined with macroinstructions, like in this sample:

```
macro struct? definition&
        esc struc definition
                label . : .%top - .
                namespace .
end macro

macro ends?!
                        %top:
                end namespace
        esc end struc
end macro

struct POINT vx:?,vy:?
        x dd vx
        y dd vy
ends

my POINT 10,20
```

The "restruc" command is analogous to "purge", but it operates on symbols from the class of labeled instructions. Similarly, the "mvstruc" command is the same as "mvmacro" but for labeled instructions.

As with "macro", it is possible to use an identifier consisting of a lone "?" character with "struc". It defines a special labeled macroinstruction that is called every time the first symbol of a line is not recognized as an instruction. Everything that follows that first identifier becomes the arguments to labeled macroinstruction. The following sample uses this feature to catch any orphaned labels (the ones that are not followed by any character) and treat them as regular ones instead of causing an error. It achieves it by making ":" the default value for "def" parameter:

```
struc ? def::&
        . def
end struc

orphan
regular:
assert orphan = regular
```

Similarly to "macro" this special variant does not override unconditional labeled instructions unless it is unconditional itself.

While "." provides an efficient method of accessing the label symbol, sometimes it may be needed to process the actual text of the label. A special parameter can be defined for this purpose and its name should be inserted enclosed in parentheses before the name of labeled macroinstruction:

```
struc (name) SYMBOL
        . db `name,0
end struc

test SYMBOL
```

10. Symbolic variables and recognition context

The "equ" is a built-in labeled instruction that defines symbol of expression class with a symbolic value. Such value contains a snippet of source text consisting of any number of tokens (even zero, allowing for an empty value) and when it is used in an expression it is equivalent to inserting the text of its value in place of its identifier, with an effect similar to evaluation of a parameter of macroinstruction (except that a parameter is always identified by a single name, while a symbolic value may be hidden behind a complex identifier).

This can lead to an unexpected outcome compared to the use of standard variables defined with "=", as the following example demonstrates:

```
numeric = 2 + 2
symbolic equ 2 + 2
x = numeric*3           ; x = 4*3
y = symbolic*3          ; y = 2 + 2*3
```

While "x" is assigned the value of 12, the value of "y" is 8. This shows that the use of such symbols can lead to unintended interactions and therefore definitions of this type should be avoided unless really necessary.

The "equ" allows redefinitions, and it preserves the previous value of symbol analogously to the "=:" command, so the earlier value can be brought back with "restore" instruction. To replace the symbolic value (analogously to how "=" overwrites the regular value) the "reequ" command should be used instead of "equ".

A symbolic value, in addition to retaining the exact text it was defined with, preserves the context in which the symbols contained in this text are to be interpreted. Therefore it can effectively become a reliable link to value of some other symbol, lasting even when it is used in a different context (this includes change of the base namespace or a symbol referred by a starting dot):

```
first:
        .x = 1
        link equ .x
        .x = 2
second:
        .x = 3
        db link         ; db 2
```

It should be noted that the same process is applied to the arguments of any macroinstruction when they become preprocessed parameters. If during the execution of a macroinstruction the context changes, the identifiers within the text of parameters still refer to the same symbols as in the line that called the instruction:

```
x = 1
namespace x
        x = 2
end namespace
macro prodx value
        namespace x
                db value*x
        end namespace
end macro
prodx x         ; db 1*2
```

Furthermore, parameters defined with "local" command use the same mechanism to alter the context in which given name is interpreted, without altering the text of the name. However, such modified context is not relevant if the value of parameter is inserted in a middle or at the end of a complex identifier, because it is the structure of an identifier that dictates how its later parts are interpreted and only the context for an initial part matters. For example, prepending a name of a parameter with "#" character is going to cause the identifier to use current context instead of context carried by the text of that parameter, because initial context for the identifier is then the context associated with text "#".

If the text following "equ" contains identifiers of known symbolic variables, each of them is replaced with its contents and it is such processed text that gets assigned to the newly defined symbol.

The "define" is a regular instruction that also creates a symbolic value, but as opposed to "equ" it does not evaluate symbolic variables in the assigned text. It should be followed by an identifier of symbol to be defined and then by the text of the value.

The difference between "equ" and "define" is often not noticeable, because when used in final expression the symbolic variables are nestedly evaluated until only the usable constituents of expressions are left. A possible use of "define" is to create a link to another symbolic variable, like the following

example demonstrates:

```
a equ 0*
x equ -a
define y -a
a equ 1*
db x 2          ; db -0*2
db y 2          ; db -1*2
```

The other uses of "define" will arise in the later sections, with the introduction of other instructions that operate on symbolic values.

The "define", like "equ", preserves the previous value of symbol. The "redefine" is a variant of this instruction that discards the earlier value, analogously to "reequ".

Note that while symbolic variables belong to the expression class of symbols, their state cannot be determined with operators like "defined", "definite", or "used", because a logical expression is evaluated as if every symbolic variable was replaced with the text of corresponding value. Therefore operator followed by an identifer of symbolic variable is going to be applied to the content of this variable, whatever it is. For example if a symbolic variable is made which is a link to a regular symbol, then any operator like "defined" followed by the identifier of said symbolic variable is going to determine the status of a linked symbol, not a linking variable.

Unlike the value of a symbolic variable, the body of a macroinstruction by itself carries no context (although it may contain snippets of text that came from replaced parameters and because of that have some context associated with them). Also, if a macroinstruction becomes unrolled at the time when another one is being defined (this can only happen when called macroinstruction is unconditional), no context information is added to the arguments, to aid in preservation of this context-lessness.

It also also possible to force a macro argument to add no context information to its text. The name of such argument should be preceded by "&" character. This allows to have an argument whose text is reinterpreted in the new context during the evaluation of a macro.

```
char = 'A'
other.char = 'W'

macro both a, &b
        namespace other
                db a, b
        end namespace
end macro

both char+1, char+1 ; db 'B', 'X'
```

11. Repeating blocks of instructions

The "repeat" instruction allows to assemble a block of instructions multiple times, with the number of repetitions specified by the value of its argument. The block of instructions should be ended with "end repeat" command. A synonym "rept" can be used instead of "repeat".

```
a = 2
repeat a + 3
        a = a + 1
end repeat
assert a = 7
```

The "while" instruction causes the block of instructions to be assembled repeatedly as long as the condition specified by its argument is true. Its argument should be a logical expression, like an argument for "if" or "assert". The block should be closed with "end while" command.

```
a = 7
while a > 4
        a = a - 2
end while
assert a = 3
```

The "%" is a special parameter which is preprocessed inside the repeated block of instructions and is replaced with a decimal number being the number of current repetition (starting with 1). It works in a similar way to a parameter of macroinstruction, so it is replaced with its value before the actual

command is processed and so it can be used to create symbol identifiers containing the number as a part of name:

```
repeat 16
        f#% = 1 shl %
end repeat
```

The above example defines symbols "f1" to "f16" with values being the consecutive powers of two.

The "repeat" instruction can have additional arguments, separated with commas, each containing a name of supplementary parameters specific to this block. Each of the names can be followed by ":" character and the expression specifying the base value from which the parameter is going to start counting the repetitions. This allows to easily change the previous sample to define the range of symbols from "f0" to "f15":

```
repeat 16, i:0
        f#i = 1 shl i
end repeat
```

The "%%" is another special parameter that has a value equal to the total number of repetitions planned. This parameter is undefined inside the "while" block. The following example uses it to create the sequence of bytes with values descending from 255 to 0:

```
repeat 256
        db %%-%
end repeat
```

The "break" instruction allows to stop the repeating prematurely. When it is encountered, it causes the rest of repeated block to be skipped and no further repetitions to be executed. It can be used to stop the repeating if a certain condition is met:

```
s = x/2
repeat 100
        if x/s = s
                break
        end if
        s = (s+x/s)/2
end repeat
```

The above sample tries to find the square root of the value of symbol "x", which is assumed defined elsewhere. It can easily be rewritten to perform the same task with "while" instead of "repeat":

```
s = x/2
while x/s <> s
        s = (s+x/s)/2
        if % = 100
                break
        end if
end while
```

The "iterate" instruction (with a synonym "irp") repeats the block of instructions while iterating through the list of values separated with commas. The first argument to "iterate" should be a name of parameter, folowed by the comma and then a list of values. During each iteration the parameter receives one of the values from the list.

```
iterate value, 1,2,3
        db value
end iterate
```

Like it is in the case of an argument to macroinstruction, the value of parameter that contains commas needs to be enclosed with "<" and ">" characters. It is also possible to enclose the first argument to "iterate" with "<" and ">", in order to define multiple parameters. The list of values is then divided into section containing as many values as there are parameters, and each iteration operates on one such section, assigning to each parameter a corresponding value:

```
iterate <name,value>, a,1, b,2, c,3
        name = value
end iterate
```

The name of a parameter can also, like in the case of macroinstructions, be followed by "*" to require that the parameter has a value that is not empty, or ":" and a default value. If an "iterate" statement ends with a comma not followed by anything else, it is not interpreted as an additional empty value, to put a blank value at the end of list an empty enclosing "<>" needs to be used.

The "break" instruction plus both the "%" and "%%" parameters can be used inside the "iterate" block with the same effects as in case of "repeat".

The "indx" is an instruction that can be only be used inside an iterated block and it changes the values of all the iterated parameters to the ones corresponding to iteration with number specified by the argument to "indx" (but when the next iteration is started, the values of parameters are again assigned the normal way). This allows to process the iterated values in a different order. In the following example the values are processed from the last to the first:

```
iterate value, 1,2,3
        indx 1+%%-%
        db value
end iterate
```

With "indx" it is even possible to move the view of iterated values many times during the single repetition. In the following example the entire processing is done during the first repetition of iterated block and then the "break" instruction is used to prevent further iterations:

```
iterate str, 'alpha','beta','gamma'
        repeat %%
                dw offset#%
        end repeat
        repeat %%
                indx %
                offset#% db str
        end repeat
        break
end iterate
```

The parameters defined by "iterate" do not attach the context to iterated values, but neither do they remove the original context if such is already attached to the text of arguments. So if the values given to "iterate" were themselves created from another parameter that preserved the original context for the symbol identifiers (like the parameter of macroinstruction), then this context is preserved, but otherwise "iterate" defines just a plain text substitution.

The parameters defined by instructions like "iterate" or "repeat" are processed everywhere in the text of associated block, but with some limitations if the block is defined partly by the text of macroinstruction and partly in other places. In that case the parameters are only accessible in the parts of the block that are defined in the same place as the initial command.

Every time a parameter is defined, its name can have the "?" character attached to it to indicate that this parameter is case-insensitive. However when parameters are recognized inside the preprocessed line, it does not matter whether they are followed by "?" there. The only modifier that is recognized by preprocessor when it replaces the parameter with its value is the "`" character.

The repeating instructions together with "if" belong to a group called control directives. They are the instructions that control the flow of assembly. Each of them defines its own block of subordinate instructions, closed with corresponding "end" command, and if these blocks are nested within each other, it always must be a proper nesting - the inner block must always be closed before the outer one. All control directives are therefore the unconditional instructions - they are recognized even when they are inside an otherwise skipped block.

The "postpone" is another control directive, which causes a block of instructions to be assembled later, when all of the following source text has already been processed.

```
dw final_count
postpone
        final_count = counter
end postpone
counter = 0
```

The above sample postpones the definition of "final_count" symbol until the entire source has been processed, so that it can access the final value of "counter" variable.

The assembly of the source text that follows "postpone" includes the assembly of any additional blocks declared with "postpone", therefore if there are multiple such blocks, they are assembled in the reverse order. The one that was declared last is assembled first when the end of the source text is

reached.

When the "postpone" directive is provided with an argument consisting of a single "?" character, it tells the assembler that the block contains operations which should not affect any of the values defined in the main source and thus the assembler may refrain from evaluating them until all other values have been successfully resolved. Such blocks are processed even later than the ones declared by "postpone" with no arguments. They may be used to perform some finalizing tasks, like the computation of a checksum of the assembled code.

The "irpv" is another repeating instruction and an iterator. It has just two arguments, first being a name of parameter and second an identifier of a variable. It iterates through all the stacked values of symbolic variable, starting from the oldest one (this applies only to the values defined earlier in the source).

```
var equ 1
var equ 2
var equ 3
var reequ 4
irpv param, var
        db param
end irpv
```

In the above example there are three iterations, with values 1, 2, and 4.

"irpv" can effectively convert a value of symbolic variable into a parameter, and this can be useful all by itself, because the symbolic variable is only evaluated in the expressions inside the arguments of instructions (labeled or not), while the parameters are preprocessed in the entire line before any processing of command is started. This allows, for example, to redefine a regular value that is linked by symbolic variable:

```
x = 1
var equ x
irpv symbol, var
        indx %%
        symbol = 2
        break
end irpv
assert x = 2
```

The combination of "indx" and "break" was added to the above sample to limit the iteration to the latest value of symbolic variable. In the next section a better solution to the same problem will be presented.

When a variable passed to "irpv" has a value that is not symbolic, the parameter is given a text that produces the same value upon computation. When the value is a positive number, the parameter is replaced with its decimal representation (similarly how the "%" parameter is processed), otherwise the parameter is replaced with an identifier of a proxy symbol holding the value from stack.

The "outscope" directive is available while any macroinstruction is processed, and it modifies the command that follows in the same line. If the command causes any parameters to be defined, they are created not in the context of currently processed macroinstruction but in the context of the source text that called it.

```
macro irpv?! statement&
        display 'IRPV wrapper'
        esc outscope irpv statement
end macro
```

This allows not only to safely wrap some control directives in macroinstructions, but also to create additional customized language constructions that define parameters for a block of text. Because "outscope" needs to be present in the text of a specific macroinstruction that requires it, it is recommended to use it in conjunction with "esc" as in the example above, this ensures that it is handled the same way even when the entire definition is put inside another macroinstruction.

12. Matching parameters

The "match" is a control directive which causes its block of instructions to be assembled only when the text specified by its second argument matches the pattern given by the first one. A text is separated from a pattern with a comma character, and it includes everything that follows this separator up to the end of line.

Every special character (except for the "," and "=", which have a specific meaning in the pattern) is matched literally - it must be paired with identical token in the text. In the following example the content of the first block is assembled, while the content of the second one is not.

```
match +,+
        assert 1        ; positive match
end match

match +,-
        assert 0        ; negative match
end match
```

The quoted strings are also matched literally, but name tokens in the pattern are treated differently. Every name acts as a wildcard and can match any sequence of tokens which is not empty. If the match is successful, the parameters with such names are created, and each is assigned a value equal to the text the wildcard was matched with.

```
match a[b], 100h[3]
        dw a+b          ; dw 100h+3
end match
```

A parameter name in pattern can have an extra "?" character attached to it to indicate that it is a case-insensitive name.

The "=" character causes the token that follows it to be matched literally. It allows to perform matching of name tokens, and also of special characters that would otherwise have a different meaning, like "," or "=", or "?" following a name.

```
match =a==a, a=8
        db a            ; db 8
end match
```

If "=" is followed by name token with "?" character attached to it, this element is matched literally but in a case-insensitive way:

```
match =a?==a, A=8
        db a            ; db 8
end match
```

When there are many wildcards in the pattern, each consecutive one is matched with as few tokens as possible and the last one takes what is left. If the wildcards follow each other without any literally matched elements between them, the first one is matched with just a single token, and the second one with the remaining text:

```
match car cdr, 1+2+3
        db car          ; db 1
        db cdr          ; db +2+3
end match
```

In the above sample the matched text must contain at least two tokens, because each wildcard needs at least one token to be not empty. In the next example there are additional constraints, but the same general rules applies and the first wildcard consumes as little as possible:

```
match first:rest, 1+2:3+4:5+6
        db `first       ; db '1+2'
        db 13,10
        db `rest        ; db '3+4:5+6'
end match
```

While any whitespace next to a wildcard is ignored, the presence or absence of whitespace between literally matched elements is meaningful. If such elements have no whitespace between them, their counterparts must contain no whitespace between them either. But if there is a whitespace between elements in pattern, it places no constraints on the use of whitespace in the corresponding text - it can be present of not.

```
match ++,++
        assert 1        ; positive match
end match

match ++,+ +
```

```
        assert 0        ; negative match
end match

match + +,++
        assert 1        ; positive match
end match

match + +,+ +
        assert 1        ; positive match
end match
```

The presence of whitespace in the text becomes required when the pattern contains the "=" character followed by a whitespace:

```
match += +, ++
        assert 0        ; negative match
end match

match += +, + +
        assert 1        ; positive match
end match
```

The "match" command is analogous to "if" in that it allows to use the "else" or "else match" to create a selection of blocks from which only one is executed:

```
macro let param
        match dest+==src, param
                dest = dest + src
        else match dest-==src, param
                dest = dest - src
        else match dest++, param
                dest = dest + 1
        else match dest--, param
                dest = dest - 1
        else match dest==src, param
                dest = src
        else
                assert 0
        end match
end macro

let x=3                 ; x = 3
let x+=7                ; x = x + 7
let x++                 ; x = x + 1
```

It is even possible to mix "if" and "match" conditions in a sequence of "else" blocks. The entire construction must be closed with "end" command corresponding to whichever of the two was used last:

```
macro record text
        match any, text
                recorded equ `text
        else if RECORD_EMPTY
                recorded equ ''
        end if
end macro
```

The "match" is able to recognize symbolic variables and before the matching is started, their identifiers in the text of the second argument are replaced with corresponding values (just like they are replaced in the text that follows the "equ" command):

```
var equ 2+3

match a+b, var
```

```
        db a xor b
end match
```

This means that the "match" can be used instead of "irpv" to convert the latest value of a symbolic variable to parameter. The sample from the previous section, where "irpv" was used with "break" to perform just one iteration on the last value, can be rewritten to use "match" instead:

```
x = 1
var equ x
match symbol, var
        symbol = 2
end match
assert x = 2
```

The difference between them is that "irpv" would execute its block even for an empty value, while in the case of "match" the "else" block would need to be added to handle an empty text.

When the evaluation of symbolic variables in the matched text is undesirable, a symbol created with "define" can be used as a proxy to preserve the text, because the replacement is not recursive:

```
macro drop value
        local temporary
        define temporary value
        match =A, temporary
                db A
                restore A
        else
                db value
        end match
end macro

A equ 1
A equ 2

drop A
drop A
```

A concern could arise that "define" may modify the meaning of text by equipping it with a local context. But when the value for "define" comes from a parameter of macroinstruction (as in the above sample), it already carries its original context and "define" does not alter it.

The "rawmatch" directive (with a synonym "rmatch") is very similar to "match", but it operates on the raw text of the second argument. Not only it does not evaluate the symbolic variables, but it also strips the text of any additional context it could have carried.

```
struc has instruction
        rawmatch text, instruction
                namespace .
                        text
                end namespace
        end rawmatch
end struc

define x
x has a = 3
assert x.a = 3
```

In the above sample the identifier of "a" would be interpreted in the context effective for the line calling the "has" macroinstruction if it was not converted back into the raw text by "rmatch".

13. Output areas

The "org" instruction starts a new area of output. The content of such area is written into the destination file next to the previous data, but the addresses in the new area are based on the value specified in the argument to "org". The area is closed automatically when the next one is started or when the source

ends.

```
        org 100h
        start:                  ; start = 100h
```

The "$" is a built-in symbol of expression class which is always equal to the value of current address. Therefore definition of a constant with the value specified by "$" symbol is equivalent to defining a label at the same point:

```
        org 100h
        start = $               ; start = 100h
```

The "$$" symbol is always equal to the base of current addressing space, so in the area started with "org" it has the same value as the base address from the argument of "org". The difference between "$" and "$$" is thus the current position relative to the start of the area:

```
        org 2000h
        db 'Hello!'
        size = $ - $$           ; size = 6
```

The "$@" symbol evaluates to the base address of current block of uninitialized data. When there was no such data defined just before the current position, this value is equal to "$", otherwise it is equal to "$" minus the length of said data inside the current addressing space. Note that reserved data no longer counts as such when it is followed by an initialized one.

The "section" instruction is similar to "org", but it additionally trims all the reserved data that precedes it analogously to how the uninitialized data is not written into output when it is at the end of file. The "section" can therefore be followed by initialized data definitions without causing the previously reserved data to be initialized with zeros and written into output. In this sample only the first of the three reserved buffers is actually converted into zeroed data and written into output, because it is followed by some initialized data. The second one is trimmed because of the "section", and the third one is cut off since it lies at the end of file:

```
        data1 dw 1
        buffer1 rb 10h          ; zeroed and present in the output

        org 400h
        data dw 2
        buffer2 rb 20h          ; not in the output

        section 1000h
        data3 dw 3
        buffer3 rb 30h          ; not in the output
```

The "$%" is a built-in symbol equal to the offset within the output file at which the initialized data would be generated if it was defined at this point. The "$%%" symbol is the current offset within the output file. These two values differ only when they are used after some data has been reserved - the "$%" is then larger than "$%%" by the length of unitialized data which would be generated into output if it was to be followed by some initialized one.

```
        db 'Hello!'
        rb 4
        position = $%%          ; position = 6
        next = $%               ; next = 10
```

The values in the comments of the above sample assume that the source contains no other instructions generating output.

The "virtual" creates a special output area which is not written into the main output file. This kind of area must reside between the "virtual" and "end virtual" commands, and after it is closed, the output generator comes back to the area it was previously operating on, with position and address the same as there were just before opening the "virtual" block. This allows also to nest the "virtual" blocks within each other.

When "virtual" has no argument, the base address of this area is the same as current address in the outer area. An argument to "virtual" can have a form of "at" keyword followed by an expression defining the base address for the enclosed area:

```
        int dw 1234h
        virtual at int
                low db ?
```

```
        high db ?
end virtual
```

Instead of or in addition to such argument, "virtual" can also be followed by an "as" keyword and a string defining an extension of additional file where the initialized content of the area is going to be stored at the end of a successful assembly.

The "load" instruction defines the value of a variable by loading the string of bytes from the data generated in an output area. It should be followed by an identifier of symbol to define, then optionally the ":" character and a number of bytes to load, then the "from" keyword and an address of the data to load. This address can be specified in two modes. If it is simply a numeric expression, it is an address within the current area. In that case the loaded bytes must have already been generated, so it is only possible to load from the space between "$$" and "$" addresses.

```
virtual at 100h
        db 'abc'
        load b:byte from 101h   ; b = 'b'
end virtual
```

When the number of bytes is not specified, the length of loaded string is determined by the size associated with address.

The other variant of "load" needs a special kind of label, which is created with "::" instead of ":". Such label has a value that cannot be used directly, but it can be used with "load" instruction to access the data of the area in which this label has been defined. The address for "load" has then to be specified as the area label followed by ":" and then the address within that area:

```
virtual at 0
        hex_digits::
        db '0123456789ABCDEF'
end virtual
load a:byte from hex_digits:10  ; a = 'A'
```

This variant of "load" can access the data which is generated later, even within the current area:

```
area::
db 'abc'
load sub:3 from area:$-2        ; sub = 'bcd'
db 'def'
```

The "store" instruction can modify already generated data in the output area. It should be followed by a value (automatically converted to string of bytes), then optionally the ":" character followed by a number of bytes to write (when this setting is not present, the length of string is determined by the size associated with address), then the "at" keyword and the address of data to replace, in one of the same two modes as allowed by "load". However the "store" is not allowed to modify the data that has not been generated yet, and any area that has been touched by "store" becomes a variable area, forbidding also the "load" to read a data from such area in advance.

The following example uses the combination of "load" and "store" to encrypt the entire contents of the current area with a simple "xor" operation:

```
db "Text"
key = 7Bh
repeat $-$$
        load a : byte from $$+%-1
        store a xor key : byte at $$+%-1
end repeat
```

If the final data of an area that has been modified by "store" needs to be read earlier in the source, it can be achieved by copying this data into a different area that would not be constrained in such way. This is analogous to defining a constant with a final value of some variable:

```
load char : byte from const:0

virtual
        var::
        db 'abc'
        .length = $
end virtual
```

```
store 'A' : byte at var:0

virtual
        const::
        repeat var.length
                load a : byte from var:%-1
                db a
        end repeat
end virtual
```

The area label can be forward-referenced by "load", but it can never be forward-referenced by "store", even if it refers to the current output area.

The "virtual" instruction can have an existing area label as the only argument. This variant allows to extend a previously defined and closed block with additional data. The area label must refer to a block that was created earlier in the source with "virtual". Any definition of data within an extending block is going to have the same effect as if that definition was present in the original "virtual" block.

```
virtual at 0 as 'log'
        Log::
end virtual

virtual Log
        db 'Hello!',13,10
end virtual
```

If an area label is used in an expression, it forms a variable term of a linear polynomial. The metadata of such term is the base address of the area. The metadata of an area label itself, accessible with "sizeof" operator, is equal to the current length of data within the area.

There is an additional variant of "load" and "store" directives that allows to read and modify already generated data in the output file given simply an offset within that output. This variant is recognized when the "at" or "from" keyword is followed by ":" character and then the value of an offset.

```
checksum = 0
repeat $%
        load a : byte from : %-1
        checksum = checksum + a
end repeat
```

The "restartout" instruction abandons all the output generated up to this point and starts anew with an empty one. An optional argument may specify the base address of newly started output area. When "restartout" has no argument, the current address is preserved by using it as the base for the new area.

The "org", "section" and "restartout" instructions cannot be used inside a "virtual" block, they can only separate areas that go into the output file.

14. Source and output control

The "include" instruction reads the source text from another file and processes it before proceeding further in the current source. Its argument should be a string defining the path to a file (the format of the path may depend on the operating system). If there is a "!" between the instruction and the argument, the other file is read and processed unconditionally, even when it is inside a skipped block (the unconditional instructions from the other file may then get recognized).

```
include 'macro.inc'
```

An additional argument may be optionally added (separated from the path by comma), and it is interpreted as a command to be executed after the file has been read and inserted into the source stream, just before processing the first line.

The "eval" instruction takes a sequence of bytes defined by its arguments, treats it as a source text and assembles it. The arguments are either strings or the numeric values of single bytes, separated with commas. In the next example "eval" is used to generate definitions of symbols named as a consecutive letters of the alphabet:

```
repeat 26
        eval 'A'+%-1,'=',`%
end repeat
```

```
assert B = 2
```

The "display" instruction causes a sequence of bytes to be written into standard output, next to the messages generated by the assembler. It should be followed by strings or numeric values of single bytes, separated with commas. The following example uses "repeat 1" to define a parameter with a decimal representation of computed number, and then displays it as a string:

```
macro show description,value
        repeat 1, d:value
                display description,`d,13,10
        end repeat
end macro

show '2^64=',1 shl 64
```

The "err" instruction signalizes an error in the assembly process, with a custom message specified by its argument. It allows the same kind of arguments as the "display" directive.

```
if $>10000h
        err 'segment too large'
end if
```

The "format" directive allows to set up additional options concerning the main output. Currently the only available choice is "format binary" followed by the "as" keyword and a string defining an extension for the output file. Unless a name of the output file is specified from the command line, it is constructed from the path to the main source file by dropping the extension and attaching a new extension if such is defined.

```
format binary as 'com'
```

The "format" directive, analogously to "end", uses an identifier that follows it to find an instruction in the child namespace of case-insensitive symbol named "format". The only built-in instruction that resides in that namespace is the "binary", but additional ones may be defined in form of macroinstructions.

The built-in symbol "__time__" (with legacy synonym "%t") has the constant value of the timestamp marking the point in time when the assembly was started.

The "__file__" is a built-in symbol whose value is a string containing the name of currently processed source file. The accompanying "__line__" symbol provides the number of currently processed line in that file. When these symbols are accessed within a macroinstruction, they keep the same value they had for the calling line. If there are several levels of macroinstructions calling each other, these symbols have the same value everywhere, corresponding to the line that called the outermost macroinstruction.

The "__source__" is another built-in symbol, with value being a string containing the name of the main source file.

The "retaincomments" directive switches the assembler to treat a semicolon as a regular token and therefore not strip comments from lines before processing. This allows to use semicolons in places like MATCH pattern.

```
retaincomments
macro ? line&
        match instruction ; comment , line
                virtual
                        comment
                end virtual
                instruction
        else
                line
        end match
end macro

var dd ?  ; bvar db ?
```

The "isolatelines" directive prevents the assembler from subsequently combining lines read from the source text when the line break is preceded by a backslash.

The "removecomments" directive brings back the default behavior of semicolons and the "combinelines" directive allows lines from the source text to be combined as usual.

15. CALM instructions

The "calminstruction" directive allows to define new instructions in form of compiled sequences of specialized commands. As opposed to regular macroinstructions, which operate on a straightforward principle of textual substitution, CALM (Compiled Assembly-Like Macro) instructions are able to perform many operations without passing any text through the standard preprocessing and assembly cycle. This allows for a finer control, better error handling and faster execution.

All references to symbols in the text defining a CALM instruction are fixed at the time of definition. As a consequence, any symbols local to the CALM instruction are shared among all its executed instances (for example consecutive instances may see the values of local symbols left by the previous ones). To aid in reusing these references, commands in CALM are generally operating on variables, routinely rewriting the symbols with new values.

A "calminstruction" statement follows the same rules as "macro" declaration, including options like "!" modifier to define unconditional instruction, "*" to mark a required argument, ":" to give it a default value and "&" to indicate that the final argument consumes all the remaining text in line.

However, because CALM instruction operates outside of the standard preprocessing and assembly cycle, its arguments do not become preprocessed parameters. Instead they are local symbolic variables, given new values every time the instruction is called.

If the name of defined instruction is preceded by another name enclosed in round brackets, the statement defines a labeled instruction and enclosed name is the argument that is going to receive the text of the label.

In the definition of CALM instruction, only the statements of its specialized language are identified. The initial symbol of every line must be a simple name without modifiers and it is only recognized as valid instruction if a case-insensitive symbol with such name is found in the namespace of CALM commands (which, for the purpose of customization, is accessible as the namespace anchored at the case-insensitive "calminstruction" symbol). When no such named instruction is found, the initial name may become a label if it is followed by ":", it is then treated as a case-sensitive symbol belonging to a specialized class. Symbols of this class are only recognized when used as arguments to CALM jump commands (described further down).

An "end calminstruction" statement needs to be used to close the definition and bring back normal mode of assembly. It is not a regular "end" command, but an identically named instruction in the CALM namespace, which only accepts "calminstruction" as its argument.

The "assemble" is a command that takes a single argument, which should be an identifier of a symbolic variable. The text of this variable is passed directly to assembly, without any preprocessing (if the text came from an argument to the instruction, it already went through preprocessing when that line was prepared).

```
calminstruction please? cmd&
        assemble cmd
end calminstruction

please display 'Hi!'
```

The "match" command is in many ways similar to the standard directive with the same name. Its first argument should be a pattern following the same rules as those for "match" directive. The second argument must be an identifier of a symbolic variable, whose text is going to be matched against the pattern. The name tokens in pattern (except for the ones made literal with "=" symbol) are treated as names of variables where the matched portions of text should be put if the match is successful. The same variable that is a source of text can also be used in pattern as a variable to write to. When there is no match, all variables remain unaffected.

```
calminstruction please? cmd&
        match (cmd), cmd
        assemble cmd
end calminstruction

please(display 'Hi!')
```

Whether the match was successful can also be tested with a conditional jump "jyes" or "jno" following the "match" command. A "jyes" jump is taken only when the match succeeded.

```
calminstruction please? cmd&
        match =do? =not? cmd, cmd
        jyes done
```

```
        assemble cmd
    done:
end calminstruction

please do not display 'Bye!'
```

To further control the flow of processing, the "jump" command allows to jump unconditionally, and with "exit" it is possible to terminate processing of CALM instruction at any moment (this command takes no arguments).

While the symbols used for the arguments of the instruction are implicitly local, other identifiers may become fixed references to global symbols if they are seen as accessible at the time of definition (because in CALM instruction all such references are treated as uses, not as definitions). A command like "match" may then write to a global variable.

```
define comment

calminstruction please? cmd&
        match cmd //comment, cmd
        assemble cmd
end calminstruction

please display 'Hi!' // 3
db comment                      ; db 3
```

To enforce treatment of a symbol as local, a "local" command should be used, followed by one or more names separated with commas.

```
calminstruction please? cmd&
        local comment
        match cmd //comment, cmd
        assemble cmd
end calminstruction
```

A symbol made local is initally assigned a defined but unusable value.

If a pattern in CALM instruction has a "?" character immediately following the name of a wildcard, it does not affect how the symbol is identified (whether the used symbol is case-insensitive depends on what is present in the local scope at the time the instruction is defined). Instead, modifying the name of a wildcard with "?" allows it to be matched with an empty text.

Since the source text for "match" is in this variant given by just a single identifier, this syntax allows to have more arguments. An optional third argument to "match" may contain a pair of bracket characters. Any wildcard element must then be matched with a text that has this kind of brackets properly balanced.

```
calminstruction please? cmd&
        local first, second
        match first + second, cmd, ()
        jyes split
        assemble cmd
        exit
    split:
        assemble first
        assemble second
end calminstruction

please display 'H',('g'+2) + display '!'
```

The "arrange" command is like an inverse of "match", it can build up a text containing the values of one or more symbolic variables. The first argument defines a variable where the constructed text is going to be stored, while the second argument is a pattern formed in the same way as for "match" (except that it does not need to precede a comma with "=" to have it included in the argument). All non-name tokens other than "=" and tokens preceded with "=" are copied literally into the constructed text and they do not carry any recognition context with them. The name tokens that are not made literal with "=" are treates as names of variables whose symbolic values are put in their place into the constructed text.

```
calminstruction addr? arg
        local base, index
```

```
        match base[index], arg
        local cmd
        arrange cmd, =dd base + index
        assemble cmd
end calminstruction

addr 8[5]                       ; dd 8 + 5
```

With suitably selected patterns, "arrange" can be used to copy symbolic value from one variable to another or to assign it a fixed value (even an empty one).

If a variable used in pattern turns out to have a numeric value instead of symbolic, as long as it is a non-negative number with no additional terms, it is converted into a decimal token stored into the constructed symbolic value (an operation that outside of CALM instructions would require use of a "repeat 1" trick):

```
digit = 4 - 1

calminstruction demo
        local cmd
        arrange cmd, =display digit#0h
        assemble cmd
end calminstruction

demo                            ; display 3#0h
```

This is the only case when a non-symbolic value is converted to symbols that may be put into text composed by "arrange", other types are not supported.

The "compute" command allows to evaluate expressions and assign numeric results to variables. The first argument to "compute" defines a target where the result should be stored, while the second argument can be any numeric expression, which is becomes pre-compiled at the time of definition. When the expression is evaluated and any of the symbols it refers to turns out to have symbolic value, this text is parsed as a new sub-expression, and its calculated value is then used in the computation of the main expression.

A "compute" therefore can be used not only to evaluate a pre-defined expression, but also to parse and compute an expression from a text of a symbolic variable (like one coming from an argument to the instruction), or a combination of both:

```
a = 0

calminstruction low expr*
        compute a, expr and 0FFh
end calminstruction

low 200 + 73                    ; a = 11h
```

Because symbolic variable is evaluated as a sub-expression, its use here has no side-effects that would be caused by a straightforward text substitution.

The "check" command is analogous to "if". It evaluates a condition defined by the logical expression that follows it and accordingly sets up the result flag which may be tested with "jyes" or "jno" command. The values of symbolic variables are treated as numeric sub-expressions (they may not contain any operators specific to logical expression).

```
calminstruction u8range? value
        check value >= 0 & value < 256
        jyes ok
        local cmd
        arrange cmd, =err 'value out of range'
        assemble cmd
    ok:
end calminstruction

u8range -1
```

All commands that are not explicitly said to set the flag that is checked by "jyes" and "jno", keep the value of this flag unchanged.

The "publish" command allows to assign a value to a symbol identified by the text held in a variable. This allows to define a symbol with a name constructed with a command like "arrange", or a name that was passed in an argument to an instruction. The first argument needs to be the symbolic variable containing the identifier of the symbol to define, the second argument should be the variable holding the value to assign (either symbolic or numeric). The first argument may be followed by ":" character to indicate that the symbol should be made constant, or it can be preceded by ":" to make the value stacked on top of the previous one (so that the previous one can be brought back with "restore" directive).

```
calminstruction constdefine? var
        local val
        arrange val,
        match var=  val, var
        publish var:, val
end calminstruction

constdefine plus? +
```

The above instruction allows to define a symbolic constant, something that is not possible with standard directives of the assembler.

The purpose of "transform" command is to replace identifiers of symbolic variables (or constants) with their values in a given text, which is the same operation as done by "equ" directive when it prepares the value to assign. The argument to "transform" should be a symbolic variable whose value is going to be processed this way and then replaced by the transformed text.

```
calminstruction (var) constequ? val
        transform val
        publish var:, val
end calminstruction
```

A "transform" command updates the result flag to indicate whether any replacement has been done.

```
calminstruction prepasm? cmd&
    loop:
        transform cmd
        jyes loop               ; warning: may hang on cyclic references
        assemble cmd
end calminstruction
```

The result flag is modified only by some of the commands, like "check", "match" or "transform". Other commands keep it unchanged.

Optionally, "transform" can have two arguments, with second one specifying a namespace. Identifiers in the text given by the first argument are then interpreted as symbols in this namespace regardless of their original context.

The "stringify" is a command that converts text of a variable into a string and writes it into the same variable, specified by the only argument. This operation is similar to the one performed by "`" operator in preprocessing, but it produces a value of string type, not a quoted string.

```
calminstruction (var) strcalc? val
        compute val, val        ; compute expression
        arrange val, val        ; convert result to a decimal token
        stringify val           ; convert decimal token to string
        publish var, val
end calminstruction

p strcalc 1 shl 1000
display p
```

While most commands available to CALM instructions replace the values of variables when writing to them, the "take" is a command that allows to work with stacks of values. It removes the topmost value of the source symbol (specified by the second argument) and gives it to the destination symbol (the first argument), placing it on top of any existing values. The destination argument may be empty, in such case the value is removed completely and the operation is analogous to "restore" directive. This command updates the result flag to indicate whether there was any value to remove. If the destination symbol is the same as source, the result flag can be used to check whether there is an available value without affecting it.

```
calminstruction reverse? cmd&
        local tmp, stack
```

```
            collect:
                match tmp=,cmd, cmd
                take stack, tmp
                jyes collect
            execute:
                assemble cmd
                take cmd, stack
                jyes execute
        end calminstruction

        reverse display '!', display 'i', display 'H'
```

A symbol accessed as either destination or source by a "take" command can never be forward-referenced even if it could otherwise.

Defining macroinstructions in the namespace of case-insensitive "calminstruction" allows to add customized commands to the language of CALM instructions. However, they must be defined as case-insensitive to be recognized as such.

```
        macro calminstruction?.asmarranged? variable*, pattern&
                arrange variable, pattern
                assemble variable
        end macro

        calminstruction writeln? text&
                asmarranged text, =display text,10
        end calminstruction

        writeln 'Next!'
```

Such additional commands may even be defined as CALM instructions themselves:

```
        calminstruction calminstruction?.initsym? variable*,value&
                publish variable, value
        end calminstruction

        calminstruction show? text&
                local command
                initsym command, display text
                stringify text
                assemble command
        end calminstruction

        show :)
```

The command "initsym" in this example is used to assign text to the local symbolic variable at the time when "show" instruction is defined. Similarly to "local" (and unlike "stringify" and "assemble") it does not produce any actual code that would be executed when the "show" instruction is called. The arguments to "initsym" retain their original context, therefore symbols in the text assigned to the "command" variable are interpreted as in the local namespace of the "show" instruction. This allows the "display" command to access the "text" even though it is local to the CALM instruction and therefore normally visible only in the scope of the definition of "show". This is similar to the use of "define" to form symbolic links.

The "call" command allows to directly execute another CALM instruction. Its first argument must provide an identifier of an instruction-class symbol, and at the execution time this symbol must be defined as CALM (it is not possible to call a macroinstruction or a built-in instruction this way). The execution then proceeds directly to the entry point of that instruction, and only returns after the called instruction finishes.

```
        define Msg display 'Hi'

        calminstruction showMsg
                assemble Msg
        end calminstruction

        calminstruction demo
                call showMsg
```

```
        arrange Msg, =display '!'
        call showMsg
end calminstruction

demo
```

When looking up the instruction symbol, the assembler skips the local namespace of the CALM instruction, as it is not expected to contain instruction definitions.

Additional arguments to "call" should be identifiers of variables (or constants) whose values are going to be passed as arguments to the called instruction. The values of these symbols are assigned directly to the argument variables, without any additional validation - this allows to pass to the CALM instruction some values that otherwise would be impossible to pass directly, like numeric ones (because when the instructions are called normally, the arguments are treated as text and assigned as symbolic values). An argument may be omitted when the definition of called instruction allows it, in such case the default value for that argument is used.

```
calminstruction hex_nibble digit*, command: display
        compute digit, 0FFh and '0123456789ABCDEF' shr (digit*8)
        arrange command, command digit
        assemble command
end calminstruction

calminstruction display_hex_byte value: DATA
        compute value, value
        local   digit
        compute digit, (value shr 4) and 0Fh
        call    hex_nibble, digit
        compute digit, value and 0Fh
        call    hex_nibble, digit
end calminstruction

DATA = 0xedfe

calminstruction demo
        call    display_hex_byte
        compute DATA, DATA shr 8
        call    display_hex_byte
end calminstruction

demo
```

16. Assembly commands in CALM instructions

An additional sets of commands for CALM instructions makes it possible to use them for more than just pre-processing, but also to directly generate and process output. They perform elementary operations, mostly on a single unit of data, but at the same time they can perform many calculations in-place, because their arguments, with few exceptions, are pre-compiled expressions, similar to the second argument to "compute".

The "display" command presents a string of bytes as a message in the standard output, just like the regular directive with the same name. It takes a single argument, an expression giving either a string or a numeric value of a single byte.

The "err" command signalizes an error, analogously to its namesake in base language. It takes a single argument, specifying a custom message to present. The argument is expected to evaluate to string value.

The "emit" command generates data of length specified by the first argument and the value specified by the second. Both arguments are treated as pre-compiled expressions. The second argument is optional, if it is omitted, the data of specified length is generated as uninitialized. When the second argument is a string, it must fit within the specified size (a "lengthof" operator may be useful in this case).

The "load" and "store" commands allow to inspect or modify values in the already generated output or in the virtual blocks. While they are similar to their counterparts in base language, they have a different syntax, both always taking three comma-separated arguments. Unlike their cousins, they do not operate on addresses associated with output areas, but on raw offsets. To point to the first byte of an area, stated offset must be zero.

The arguments to "load" are, in order: target variable, offset to load from, number of bytes to load. The first argument must be an identifier of a symbol, the latter two are pre-compiled expressions. The second argument may contain a label of the area, followed by ":" and then offset, or just a plain numeric expression, in which case it is an offset within entire output generated up to this point. The loaded value is always a string of the specified length.

The arguments to "store" are, in order: offset to store at, number of bytes to store, the value to store (numeric or string). The last two arguments are analogous to the arguments to "emit", at the same time the first two arguments are like the last two arguments to "load". The offset may be prepended with the label of an area with ":" as the separator.

To convert between the addresses used by classic "load" and "store" and the raw offsets expected by the CALM commands, it suffices to add or subtract the base address of the area. If the base address is not known, it can be obtained with help of "1 metadataof" operator applied to an area label.

Main index Download Documentation Examples Message board