ASM Software Development Kit

Standard Edition

Table of Contents

Assembler

User's manual

Assembler: User's manual

Table of Contents

Foreword

This user's manual contains the description of the assembler, describes the variants of its launching, lists error and warning messages.

About this manual

This manual describes the assembler language for the UMC 320 microprocessor, the options of the command line and also instructions, directives, and macros used in the assembler.

Intended audience

This manual will be useful for everybody, who would like to use the assembler language for building information systems for UMC 320 microprocessor.

The structure of this manual

Each Chapter of this manual covers a separate topic, related to the macroassembler.Chapter I describes macroassembler usage. Chapter II gives a review of the assembler language. The following two Chapters (III and IV) give a brief overview of directives and macros.

Feedback

We did our best to make this manual useful and we would like to get your opinion of it. We appreciate your feedback and are looking forward to any e-mails we might receive. You may e-mail to the following address: <about@interstron.ru>

You may also use the feedback link on our web cite: http://www.interstron.ru

Chapter 1. Assembler usage

What is the assembler?

The program in the assembler language we will call *source code* and its representation in machine code – *object code*. The function of the assembler is to transform the source code to object code.

Assembler launching

`asm.exe` [[`options`]] [[`input_file_name,`]]

where

`input_file_name`
The name of the file to translate.

`options`
One or several of the following options:

Table 1.1. Assembler command line options

-h (-help)	Display brief information about the command line options.
-q (-quiet)	"Quiet" mode. Operate without output to the console.
-d (-debuginfo)	Place the debug information of the assembler to the object code.
-l[file]	Generate listing. If the name of the file, to which the listing is to be placed, is not specified, a default file is created. The name of this file should coincide with the name of the input file and should have the extension `.lst`
-ff	Option of listing generation. Should be used only with `-l` option. Directs assembler to include code from negative branches of conditional directives, that do not enter the object file, to the listing.
-fi	Option of listing generation. Should be used only with `-l` option. Inserts all include files to listing.
-fm	Option of listing generation. Should be used only with `-l` option. After processing the procedure-like macro outputs its body (with substituted parameters) to the listing. In case of the absence of this option the body of the macro is not placed to the listing.
-ofile	Defines the file name for the result of translation.
-Ipath	Defines the path (paths) for searching include files. Paths are divided by semicolon and are looked up in the order of their specification in the command line.
-Dsymbol[=text]	C-style definition of the single-line macro.
-M`codepage`	*Compiler messages language and codepage setting* Orders to the compiler to output all messages in the set language and coding. The codepage available volumes:

	"en_US.850" - english lenguage, codepage 850;
	"ru_RU.1251" - russian language, code page 1251 (Windows);
	"ru_RU.866" - russian language, code page 866 (DOS).
	The default setting is "en_US.850" (english lenguage, code page 850).
	Example:
	`-M "ru_RU.1251"`
	`-Men_US.850`

--	Indicates that all the following arguments are not options.

Chapter 2. Assembler language

Translation phases

It it possible to determine three main translation phases of the initial text by the macroassembler: *lexical analysis* – during this phase trigraphs are substituted, logical lines are formed and tokens are determined; *macroprocessor phase* – interpretation of the directives of conditional translation, definition and substitution of macros, interpretation of other directives of the macroprocessor (see Chapter 4, *Macrofacilities*); the third phase – *assembler itself* – the translation of the text transformed by macroprocessor.

Lexical agreements

Assembler to a great extent uses lexical agreements of the C language Standard. This refers to trigraphs processing, numbers representation, symbols encoding in the string literals (symbols “ \ ? ‘ are to be preceded by \), and lines transfer (if the last symbol in the line is “\”, the next line is taken to be the continuation of the current line, thus constituing single logical line). In addition, if the symbol “\” is in the middle of the line, the rest part of the line is transferred to the next logical line, if this symbol is not inside a comment.

During phase of lexical processing, the incoming flow of symbols is divided into tokens. Tokens are identifiers, numeric tokens, string and character literals, operators, signs "##", "#", "#&", comments, spaces and other symbols of the input language, not enumerated above.

Note

The processing of braces in macroparameters or in operators "#&" is performed in the phase of macroprocessor, therefore, in any case, symbols, located before or after braces belong to different tokens.

If two tokens that turned out to be adjacent while macroprocessing, should be concatenated by tentative lexical analysis, a space is placed between them.

```
STR  MACRO p
        string #p
        ENDM

        section data
        STR abc}def         // string "abc def"
        ends
```

Identifier

Identifier is a name, given to program objects (procedures, symbols, labels, etc.)

Identifiers should comply the following requirements:

- Identifiers may consist of Latin both upper-case and lower-case letters (A-Z, a-z), digits (0-9)

and underscore (_).

- Identifier may start with any symbol enumerated above, except digits.

Numeric tokens

A sequence of symbols of input language, which starts from the decimal digit and contains digits and Latin letters, is considered to be a numeric token (or just a number). If the number starts with the prefixes of scale of notation "0b", "0o", "0d" or "0x", the number is interpreted as having binary, octal, decimal, or hexadecimal base and the following digits and letters represent its value. Otherwise, the digit is treated in current scale of notation, the base of which by default is equal to 10 and can be changed by `RADIX` directive (see the section called “ RADIX directive ”), all digits and letters are significant, starting from the first. The significand should contain digits and letters permissible in its scale of notation, otherwise the numeric token is considered to be erroneous.

```
section data
byte    0xk       ; error
byte    0o18      ; error
RADIX    16
byte    0b33      ; error, because prefix "0b"
                  ; specifies binary notation
byte    0xb33     ; ok,  "0x" should be specified, although
                  ; current base of notation is 16
byte    0c33      ; equivalent to "0xc33", "0c" is not a prefix of base
                  ; of notation
ends
```

We will call correct numeric tokens *integer literals*. If the value of the number fits into the 31-bit integer, the value of the number is treated as signed, otherwise it is treated as unsigned. If the value of the number does not fit into the 32-bit integer, the warning is given, and its lowest 32 bits are taken as its value.

Note

Even after using `Radix 16` hexadecimal numbers starting with letter should be used with 0x prefix. Otherwise such token (e.g. "FF") is not numeric literal but symbol name.

String literals

The definition is similar to that in the C language standard, that is a sequence of symbols or the so-called escape sequences, starting with " or L"; and ending with “. Escape sequence is one of the following character sequences:

\'	represents symbol '
\"	represents symbol "
\\	represents symbol \
\?	represents symbol ?

\a	represents symbol BEL (bell)
\b	represents symbol of crossing out (backslash)
\f	represents symbol of format feed (FF)
\n	represents symbol of line feed (LF)
\r	represents symbol of carriage return (CR)
\t	represents symbol of horizontal tabulation
\v	represents symbol of vertical tabulation
octal_number	represents symbol, which code is equal to octal number. Octal number can have values from 0 to 377. Next three digits regard.
\x*hexadecimal_number*	represents symbol, which code is equal to hexadecimal number

In order to include symbol “ or \ to the string literal, it is necessary to precede it with the backslash. String literals are a special kind of data, their inner representation – a sequence of bytes, which contain symbol codes, starting from the first symbol after opening “ and to the symbol before closing ”, taking into consideration substitution of escape-sequences by symbols, which they represent.

For example:

```
section    data
string    "abcd\"ef\\g\?h\""    ; contain abcd"ef\g?h"
ends
```

Note

Contrary to C language,

- Zero byte is not added to the end of string literal.
- At the stage of lexical analysis adjacent string literals are not concatenated, however, you may specify several string literals in sequence in `string` directive.

Character literals

Represent separate symbols. The definition is similar to that in the C language Standard, that is a sequence of symbols or the so-called escape-sequences, starting with ‘ or L’ and ending with ‘.

To include the symbol ‘ or \ to the character literal, it is necessary to precede it with the backslash. Character literals are interpreted as signed integers. If the literal contains one symbol, its value is equal to its symbol code. If it contains two or more symbols, its value is equal to the value of the number, whose low byte contains the first symbol code, and the next byte contains the second symbol code.

Comments

Comments let to insert notes, explanations and other text notes in the text of the program. Assembler ignores them while translating program, but includes them in listing.

Three types of comments are used in the macroassembler:

```
; traditional line comment
// single-line comment in C style
/* multiline comment in C style */
```

Assembler language commands

Every command in source code is represented in the following fashion:
[symbol :] [[instruction] | [directive]] [comment]

Besides, C style comments may be placed in any place and appear to be a separator.

All fields, as you may see, are not necessary.

Empty lines may be used for better readability.

Symbol

Symbol is an identifier. Symbol may be used to define the label, `common` symbol, string or multiline macros or numeric variable. The label cannot be defined in the line, where the directive of macroprocessor is used.

Rules for choosing symbol names:

- Names of symbols are case-sensitive.
- Names of symbols should be unique.
- Names of symbols should not coincide with the names of predefined instructions, macro definitions and directives.

Symbol names should not contain spaces.

The definition of label is the name of the corresponding symbol with the colon after it. Other program commands may refer to labels.

Instruction

The instruction consists of mnemonics (name of microprocessor command) and of one or several operands. The presence of operands is obligatory for some commands, for others the usage of operands is prohibited.

The purpose of operands is to represent information for processing. Operands separate themselves from mnemonics at least by one space or by tabulation. In-between operands are separated by com-

mas.

In instructions with several operands, *operands-receivers* always precede source-operands. For example:

```
ldis r2, 0
```

means, that the meaning of the source-operand (0) is kept in the operand-receiver (r2).

Mnemonics morphology. Mnemonics is made up of mnemonics base, which express the meaning of the command, and optional sequence of single-letter postfixes, which specify modification of the action.

Signedness postfix s specifies that the command operate on signed operands; the absence of this postfix specifies that unsigned variant of the command is used.

```
ldi r0, -2      ; loads -2 to the lowest word of r0
                ; upper word is zeroed
ldis r0, -2     ; loads -2 to the lowest word of r0
                ; and 0xFFFF to the upper word (sign extension)
```

Mnemonics of arithmetic commands, for which signedness is not important (for example, the command of addition in two's complement case), have no signedness postfix.

Commands of reading from memory have the postfix of size: d for a double word, w for a word or b for a byte.

```
ldb r0, [r1]    ; loads a byte to the address, specified in register r1,
                ; to the lowest byte of r0; upper bytes are zeroed
ldbs r0, -2     ; loads -2 to the lowest byte of r0
                ; 0xFFFFFF is written to the upper bytes (sign extension)
ldw r0, [r2]    ; loads word from the address, specified in r2 register,
                ; to the lowest word of r0; upper word is zeroed
ldd r0, [r3]    ; loads double word from the address,
                 ; specified in register r3, to te register r0
```

Operators

Operator is a sign, used for infix or prefix notation. Operator links operands into expressions. The elementary operators are arithmetic: addition, substraction, multiplication, etc. Operators can be unary, binary and ternary. Order of operators execution in expression is defined by the priority and order, in which they are specified.

Constant expressions

The value of any constant expression (or just expression) is a union of optional reference to segment and number. If the reference to segment is present, the value refers to the memory address in the specified segment (the number express the offset from the beginning of the segment).

The expression with the reference to segment is called address expression, without such a reference – numeric, or just an address and a number correspondingly.

The elementary expressions are:

- The name of the label (the value is the address of the label);
- Integer literal;
- $ - the current address.

The name of the label and the current address are considered elementary address expressions.

Any expression is either elementary or is made up of elementary expressions with the help of operations. The value of the expression can be relocatable (defined in the linkage phase) or non-relocatable (in other words, numeric). The name of the label or the current address are relocatable expressions, number – numeric (defined in the assembling phase). The result of any operation is a relocatable expression, if at least one operand is relocatable, and numeric, if all operands are numeric. The exception are the operations of subtraction and comparison: the result of subtraction or two address comparison can be a numeric expression.

Hereinafter the word "number"; is used in the meaning of "numeric expression", and the word "address" – in the meaning of "relocatable expression".

Operators priority

Operators priority is the order of evaluation arithmetic and/or logical expressions

There are strict rules of expressions evaluation:

1. Expressions in parenthesis are calculated first of all.
2. Operators are performed in accordance to the priority.
3. Adjacent unary operators are grouped right to left.
4. Binary operators with equal priority are grouped left to right.

Operators priority as well as their notation are completely taken from C language.

Table 2.1. Operators Priority

Operators Priority
unary operators
* / %
+ - (as binary operators)
<< >>
< <= > >=
== !=
&
^

Operators Priority
|
&&
||

Operators with equal priority are placed in a single line. Operators, placed above, have higher priority. For example:

Expression a + b*c means a+ (b*c), not (a + b)*c, because * has a higher priority.

Expression a + b + c means (a + b) + c.

Unary operators

Unary operators have the highest priority. They are right-associative, that is performed from right to left.

Unary operators are applied to numbers only.

Table 2.2. Unary operators

Operator	Usage	Explanation
+ and -	+ (-) Y	Unary plus and minus
~	~Y	Negation
!	!Y	Unary logical negation

Binary operators

Binary operators have less priority, than unary. If one of the operands contains unsigned number, and the other - signed, the last operand is converted to unsigned.

Multiplicative operators

Multiplicative operators have the highest priority among all binary operators.

Table 2.3. Multiplicative operators

Operator	Usage	Explanation
*	X * Y	Multiplication of X by Y
/	X / Y	Division of X by Y
%	X % Y	Remainder of division of X by Y

Relational operators

Table 2.4. Relational operators

Operator	Usage	Explanation
==	X == Y	X equals to Y
!=	X != Y	X not equals Y
<	X < Y	X less than Y
>	X > Y	X greater than Y
<=	X <= Y	X less or equal to Y
>	X >= Y	X greater or equal to Y

Bitwise operator

Bitwise operators apply to numbers only. The result is a number.

Table 2.5. Bitwise logical operators

Operator	Usage	Explanation
&	X & Y	Bitwise AND
|	X | Y	Bitwise OR
^	X ^ Y	Exclusive bitwise OR (XOR)
>>	X >> Y	Shift to the right by Y bit
<<	X << Y	Shift to the left by Y bit

Logical operators

If evaluation of the first operand determines the result, the second operand is not evaluated.

Table 2.6. Logical operators

Operator	Usage	Explanation
&&	X && Y	Logical "and". The result is equal to 1 if both operands are not zero, otherwise – 0.
||	X || Y	Logical "or". The result is equal to zero if both operands are not zero, otherwise – 1.

Other operators

Table 2.7. Other operators

Operator	Usage	Explanation
+	X + Y	Addition of two numbers or an address and a number
-	X - Y	Subtraction of the number from the address or from the number; subtraction of the two addresses, belonging to the one section and defined

Operator	Usage	Explanation
		by the time of the expression evaluation
?:	X ? Y : Z	Ternary conditional operator. If the first operand is not zero, the value of the expression X ? Y : Z is the value of the second operand, otherwise the third.

Note

If arithmetic operations (addition, substraction, multiplication ect.) result in an overflow , no diagnostics is given.

Data

Recording the data.

The data is characterized by: size, negative numbers representation and endianness.

UMC 320 microprocessor keeps data in little-endian order: it keeps lowest data bits in the byte with the lowest address. For example, when saving the number 0x9876 at some address DEST, the microprocessor places 0X76 to address DEST, and 0x98 - to address DEST + 1. For the code the big-endian scheme is used, the byte sequence, in which the lowest bits of data are kept in the upper byte.

To introduce signed in UMC 320 microprocessor, two’s complement is used.

The datum as a bit image is characterized by size only, so it is enough for data reservation directives to have only operand, which expresses the size. On the other hand, on any of the platforms the size of any datum is a multiple of the size of the byte. Thus, it is possible to generally speak about the size in bytes of target platform.

The rules of immediate data evaluation.

Immediate data is always presented by its arithmetic value (arithmetic quantity, as for example, -5 or 127), upon which arithmetic operations are performed (for example, addition or division).

Bit operations (for example, conjunction and shift) are performed upon bit image of the received arithmetic quantity, taking into account representation of negative numbers.

Chapter 3. Directives

File inclusion

```
INCLUDE "filename"
```

Assembler will further process lines from file `filename` (the include file), as if its contents were at the place of this directive. When the processing of the include file is finished, assembler returns to processing of the initial file. Following folders are searched for the include file:

- folders of already included files of include files chain, starting from the last and moving to the first;
- folder of the main file;
- include folders, declared with the help of the `-I` command line option(see Table 1.1, "Assembler command line options")

Note

Directives of the conditional translation. There should be an `ENDIF` directive corresponding to every `IF` directive in the same file (and vice versa). `IF` directive in the same file should correspond to every `ELSE` and `ELIF` directives. `ENDM` directive should correspond to every `MACRO` or `WHILE` directive (and vice versa) in the same file. No unfinished comments are to be in the include file. If `INCLUDE` directive is met inside the macro definition, while replacing this macro , no parameter substitution takes place in the include file.

The `INCLUDE` directive nesting level can not exceed the maximum level of 60.

Section declaration

There are no limitations for sections order within the module. There are several ways to declare a section.

```
section "section_name" [attribute] [attribute] [attribute]
```

Opens the section.

ro	read-only (`sh_flags` does not contain `SHF_WRITE`)
data	contains data (`sh_flags` does not contain `SHF_EXECINSTR`)
noalloc	the memory is not reserved for the section in the program (`sh_flags` does not contain `SHF_ALLOC`)

The type of the defined section is always equal `SHT_PROGBITS`.

For some predefined sections there is a shorter way of declaration:

```
section section_name
```

The following section names are possible:

data	Analogous to `section ".data" data` This section contains initialized data, which is included to the data segment of the program.
code	Analogous to `section ".text" ro` Contains the instructions of the program.
zidata	Analogous to `section ".bss" data` Only the size of data is included to the object file, the data itself is not stored.
rodata	Analogous to `section ".rodata" data ro` This section contains read-only data, which is added to read-only memory
comment	Analogous to `section ".comment" noalloc` Contains comments to insert to the object file.
alines	Analogous to `section ".debug.alines" noalloc` Section containing debug information about assembler lines
clines	Analogous to `section ".debug.clines" noalloc` Section containing debug information about C/C++ lines

Sections, declared by the two above-mentioned ways, should be closed by directive ends (without parameters).

Note

`section_name` may be used without the key-word *`section`*. The meaning of the command is the same, but `ends` directive is not necessary to close the section – the

section will be closed when another section declaration or end of file will be met.

Symbol declaration

```
global symbol_name
```

Symbol is declared external global. If it is not defined in the module, this directive lets to use it.

```
weak symbol_name
```

Symbol is declared external weak. If it is not defined in the module, this directive lets to use it.

Global and weak symbols have two differences:

- First, if linker meets several already defined symbols, one of which is global and the rest are weak, the preference is given to global, while the two definitions of one global name are prohibited;
- Second, if the weak symbol is not defined, linker does not search for its definitions in libraries; unresolved weak symbol is given the zero value, while the global symbol should always be resolved.

Note

Symbol may be defined with the help of directive of symbol definition (see the section called “Label (symbol) definition”)

```
<name> common [ebyte | eword] space <size>
```

Defines the symbol *name* of the size *size* with binding STB_WEAK в секции SHN_COMMON. Alignment is set by the keywords *ebyte* and *eword*. Their meanings are the following:

ebyte	- 2 bytes
eword	- 4 bytes

.

Data reservation

Data is reserved in order of its enumeration.

ebyte, eword
: Aligns the address counter by word or double word correspondingly. It may lead to the formation of "gap", which is filled with zeros in this case.

byte | word | dword expression [,expression]+
: Reserves the byte, word, double word correspondingly and places the value of expression in it. You may reserve several bytes, words, double words. Operands should be separated by commas.

string "line_of_symbols"
: Allocates string (escape-sequences are allowed). Escape-sequences correspond to the C language Standard, with the exception of sequence of \ooo kind, where o – octal digit. In C language the length of such digit sequence is not limited, and in this assembler it is limited by three digits. The string does not end with zero (in contrast to C-lines). It is possible to use several adjacent strings in the declaration. Strings may be separated one from another by space(s). Strings are taken in double quotes.

space expression
: Reserves bytes number equal to the value of the expression. The expression should be 32-bit unsigned integer.

Example 3.1. Memory Reservation Directives

```
ebyte
eword

byte 0xff
word 0xffff
dword 0xffffffff

string "string\n"

space 1024
```

`Org` directive

```
org numeric_expression
org relocatable_expression
```

The argument of `org` directive declares the offset from the beginning of the current section. In the first case the offset is equal to the value of the numeric expression, in the second case the label in the relocatable expression, should be, by the moment of translation, defined in the current section. The expression is considered to be numeric, the name of the label is changed to its offset from the beginning of the section. The value of the result, treated as 32-bit unsigned integer is then assigned to the current relative address. If the shift is done forward the org directive is analogous to directive `space`.

Example:

```
L:
   byte    0
.....

L1:
   org L
```

```
byte L1 - L
org L1
```

Label (symbol) definition

```
label_name:
```

Defines the label (symbol) with the name *label_name* and local binding (if not otherwise specified by global and weak external symbols declaration directives). The symbol gets the value of the current address.

The range of legal directive values

Table 3.1. The range of legal directive values

Directive	The range of possible values	special conditions
byte	-128..255	
word	-32768..65535	
dword	-2147483648..4294967295	
space, org, common	0..4294967295	Argument expression can not be relocatable

Chapter 4. Macrofacilities

A macro is a sequence of assembler commands, starting by `MACRO` directive and finished by the `ENDM` directive. Like procedures, macros have names and when used they are substituted by this command sequence.

The difference between a procedure and a macro is the following. The procedure code is placed in the program once and microprocessor transfers the control to it when it is necessary . The code of macro may be placed in the program more than once. Assembler substitutes every use of the macro name with its body. Macros have several advantages:

- The code of macro is performed in the phase of translation in contrast to the procedure code, which is performed during the program execution.

  

  Note

  In the phase of translation, evaluation or substitution is performed, depending on the type of the used macro. (see the section called “Multiline Macros”).

- The usage of macros instead of procedures slightly speeds up the performance of the program, as there is no time waste for calling and return from the procedure
- Lets to use one block several times

When macros are used, the resulting code of the program may increase.

The macro expansion nesting level can not exceed the maximum level of 60.

Single-Line Text Macros (C Style Macros)

Single-line macro is defined with the help of keyword EQU as follows:

```
name EQU text_of_substitution
```

and further in the initial text every time (except of the contexts listed below, see the section called “Macros And Numeric Variables Substitution”), when the *`name`* is used, it is substituted with the *`text_of_substitution`*. If in the macroprocessor directive, defining macro or numeric variable, its name coincide with the name of corresponding macro, the substitution is not performed. One should also take into account that the check for the presence of the macroprocessor directive in the line is made before the substitution of the macros values. For example, if the macro contains IF, it will not be processed as a directive of conditional translation. If there are macros in the substituted text , they are also substituted with the corresponding text, however, recursive substitution is not performed (as in C).

`Text_of_substitution` begins with the first symbol, different from the space and comment. Ending spaces in the line are not included in *`text_of_substitution`*.

Note

The text of substitution should not contain keywords `IF`, `ELIF`, `ELSE`, `ENDIF` and

```
        EQU.
```

Multiline Macros

Multiline macro may be procedure-like or function-like:

```
; procedure-like macro
M   MACRO   P1, P2, P3
    ...
    ...
    ENDM

; function-like macro
FM  MACRO (P1, P2, P3)
    ...
    ...
    EXITM replace-text ; exiting and returning "replace-text"
    ...
    ...
    ENDM
```

Function-like macros are substituted with values, specified as arguments of the `EXITM` directive, with the same limitations as for single-line macros apply(see the section called “Single-Line Text Macros (C Style Macros)”). `EXITM` directive should be placed in the body of macro. Function-like macros may contain only macroprocessor commands.

`EXITM` directive can also be placed in procedure-like macros to stop expansion early. In this case arguments of the directive are ignored and are not considered as the result value like in case of function-like macros.

Procedure-like macros may be used only in the position, in which the instruction of assembler is allowed.

In the macro call arguments are separated by commas. When arguments are parsed, macro substitution takes place only if there is a «#&» before the macro name. In this case the argument separation takes place after the macro substitution. In function-like macro call arguments should be enclosed with parenthesis. If there are more arguments than parameters (but there is at least one parameter), part of the string starting with argument corresponding to the last parameter till the end of the line constitutes the last argument. If there are more parameters than arguments, than missing parameters are substituted with empty strings. To pass complex meanings, containing symbols “,”, arguments are taken in braces. In this case to define the end of the argument the closing brace is used. Braces, which are included in argument, should be doubled:

```
    M 1, 2, 3
    M {1,2,3}, {{, }}
```

In the second case macro definition M will be called with three arguments:

```
1,2,3
{
}
```

Starting and ending spaces in arguments are ignored. Spaces in the middle of the argument are transferred as they are.

Example:

```
; two following macros HEAD and TAIL may be used in macros,
; which take variable number of arguments
; function-like macro returns the first element from the list of arguments
HEAD  MACRO  (P1, P2)
      EXITM  P1
      ENDM
; macro returns the "tail" of the list
;(everything, except the first parameter and a comma after it)
TAIL  MACRO  (P1, P2)
      EXITM  P2
      ENDM
; some usual macro with variable number of arguments,
; which are transferred in one parameter
SOME  MACRO  P
      LOCAL  A, PC
      A  EQU  P
      WHILE  !BLANK (A)
      PC  EQU  #&HEAD (#&A)  ; determine next argument
               ; note use of substitution operator
      A  EQU  #&TAIL (#&A)
      ...       ; actions with the next argument
      ENDM                    ; end of WHILE
      ENDM                    ; end of SOME
      ...
      SOME  r0, r1, r2, r3, r4  ; call of macro SOME
```

By replacing macro, all uses of the parameters are substituted with the values of arguments, except arguments of the function `BLANK`. All function `BLANK` calls, in which a macro parameter is specified as argument, are substituted with 0 or 1. After that, the transformed text is processed by the assembler.

For the function-like macro the right parenthesis (")"), if it is not enclosed in braces {}, ends up the current list of arguments. Thus, the balance of parenthesis is not kept.

```
kk EQU #&CATSTR(#&SUBSTR(ab,1,1))  ;=b
; In this example the first closing parenthesis
; ends up the list of arguments SUBSTR
; The list of arguments CATSTR becomes current
; and the second closing parenthesis ends it up.
```

```
kk EQU #&CATSTR((a + b), + c)
; In this example the first closing parenthesis ends up the list
; of arguments CATSTR making «(a + b» its only argument
```

If the argument or its part is enclosed in braces, the first symbol"}" after "{" is considered to be paired to "{". In any case single braces never become part of the argument. For example:

```
M     MACRO    p
        string       #p
        ENDM

    section code
        M    {1,2}
        M    {1,2}} }
        M    { { {1,2}
        M    {1,2} }
    ends
```

listing:

```
Source:
line#     address       data          source
--------- ------------- ------------- ------------
     1                                M     MACRO    p
     2                                        string       #p
     3                                        ENDM
     4                                    section code
     5
     6                                        M    {1,2}
     1      00000000      31  2C              string       "1,2"
     2
     7                                        M    {1,2}} }
     1      00000003      31  2C              string       "1,2} "
     2
     8                                        M    { { {1,2}
     1      00000008      20  20              string       "  1,2"
     2
     9                                        M    {1,2} }
     1      0000000d      31  2C              string       "1,2 "
     2
    10                                   ends
```

If during the multiline macro expansion the value of parameter being substituted is a string literal the destringizing (quotes deletion) occurs.

Note

1. After destringizing, parameter should not contain '\0' symbol.

2. If "#" symbol stands before parameter name in macro definition no destringizing occurs. In this case previous restriction does not take place.

RADIX directive

```
RADIX numeric_expression
```

numeric_expression should have one of the following values: 2, 8, 10,16. Numbers without prefix will be treated using this scale of notation. While transforming numeric variables to text, numeric values will be composed using this scale of notation, besides, if the current base is not equal to 10, a corresponding prefix will be added.

For the numeric_expression a decimal scale of notation is used.

ERROR and WARNING directives

```
ERROR text
```

```
WARNING text
```

Error or warning is produced correspondingly containing the text.

Example:

```
 i=1
IF !i
word !i
  WARNING не i
ELSE
word i
  WARNING !i=0
ENDIF
```

Warning !I=0 is produced.

Directives Of Conditional Translation

IF expression	Strings, starting with the following string and up to directive ENDIF, ELIF or ELSE of the same level will be translated, if (only in case), the meaning of the *expression* is not equal to zero.
ELIF expression	Strings, starting with the next string and up to ENDIF, ELIF or ELSE directive of the same level will be translated, in case (and only in case) the meaning expression is not equal to zero and if none of IF or ELIF blocks of the same level was not translated before.
ELSE	Strings, starting with the next line and up to ENDIF directive of the same level will be translated if (and only in case), none IF or IF blocks of the same level was not translated before.
ENDIF	The sign of the end of sequence of blocks, started by IF directive.

Predefined Macros, Performing Strings Operations

In the following macros the arguments of text type comply to the following rules:

- Text, limited by braces, refers to one argument
- Otherwise, if the argument is a name of text macro, it is substituted with its value
- Otherwise, the text of the argument is used unchanged

INSTR

The search of substring in the string.

Syntax:	`INSTR (str1, str2 [,start])`
Description:	Returns the initial position of *`str2`* inside *`str1`*, the search starting from the position *`start`*. By default *`start`* is equal to zero. If the substring is not found in the string, -1 is returned.

Example:	`ldi r0,INSTR(string,ing)`
	`ldi r1,3`
	`sub r0,r0,r1`

SUBSTR

Returns the substring of the string of the specified length from arbitrary position.

Syntax:	`SUBSTR (str1 [,start], len )`
Description:	Returns the substring from the string *`str1`*, starting from the position *`start`* (by default 0) of the length *`len`*. If the specified substring does not fit in *`str1`*, the part of initial string *`str1`* is taken.
Example:	`str EQU #&SUBSTR(ldis,,3)` `str r0,1 ; will execute command "ldi r0,1"`

CATSTR

Arguments concatenation.

Syntax:	`CATSTR (str1[,str2] ...)`
Description:	Returns the result of the arguments concatenation. The number of arguments is not limited.

Example:	`PREFIX EQU prefix`
	`SUFFIX EQU suffix`
	`bfr CATSTR (#&PREFIX, _, name, _, #&SUFFIX)`
	`; Results in generating branch instruction to the label prefix_name_suffix`

SIZESTR

Obtains length of the string.

Syntax:	`SIZESTR (str1)`
Description:	Returns the number of symbols in the string.

Example:	`label:`
	`string "my string"`
	`END_LABEL = label + #&SIZESTR(my string)`
	`; END_LABEL will point to the place immediately after the "my string" strin`

IDSTR

String comparison.

Syntax:	`IDSTR (str1, str2)`
Description:	Macro `IDSTR` returns 1, if the strings coincide, otherwise zero. The comparison is case-sensitive.

Example:	`IF IDSTR (#&CURRENT, #&SPECIAL)`
	`// some special code`
	`ENDIF`
	`; "Special" code will be translated only in case when`
	`; single-line macros CURRENT and SPECIAL`
	`; are substituted by identical strings.`

Other Predefined Macros

OPATTR

Returns attributes of the argument.

Syntax:

```
OPATTR (arg)
```

Description:

OPATTR macro returns the number, with the attributes of the argument set.

Table 4.1. Meanings of attributes

Bit	Explanation
0x10000	The argument consists of one token. If there are more than one token, than zero is returned.
0x20000	Identifier (including reserved words of macroprocessor and assembler). Set if bit 0x10000 is set.
0x40000	Number
0x10000000	Attribute of erroneous token. Set, if the argument is a number, containing odd symbols at the end, for example 0x123k21.
0x20000000	Attribute of overflow. Set if the argument is a number, which does not fit in 32-bit unsigned integer, for example, 0x33333333333333333
If the bit 0x20000 is set and if the token is used by assembler, the lowest byte contains its code:	
0x00001	General purpose register.
0x00002	System register.
0x00003	Assembler directive.
0x00004	Modifier of jump instruction.
0x00005	Mnemonics .
0x00006	Label
0x00020	The label is defined
0x00040	The label is defined in current section
Symbol code is kept in the 2-nd byte, if it is known to macroprocessor:	
0x00100	Directive of macroprocessor
0x00200	The name of single-line text macro
0x00300	The name of multiline macro
0x00400	The name of numeric variable

Example:

```
q    MACRO
```

```
ENDM

    section code
    ldi r0,LOW(OPATTR(r11))
    ldhi r0,HIGH(OPATTR(r11))
; The value of r0 will be set to 0x30001.
; The bits set have following meanings:
;    0x10000 - argument is a single token
;    0x20000 - argument is an identifier
;    0x1     - argument is a register name

ldi r1,LOW(OPATTR(0x2222222222222))
ldhi r1,HIGH(OPATTR(0x2222222222222))
; The value of r1 will be set to 0x20050000.
; The bits set have following meanings:
;    0x10000   - argument is a single token
;    0x40000   - argument is a number
;    0x2000000 - overflow

ldi r2,LOW(OPATTR(q))
ldhi r2,HIGH(OPATTR(q))
; The value of r2 will be set to 0x20300.
; The bits set have following meanings:
;    0x10000 - argument is a single token
;    0x20000 - argument is an identifier
;    0x300   - argument is a multiline macro

ends
```

LOW

Returns low 16 bit of the expression value.

Syntax:	`LOW (arg)`
Description:	`LOW` macro takes const-expression as an argument and returns low 16 bits of its value.

HIGH

Returns bits from 16 to 31 of the expression value.

Syntax:	`HIGH (arg)`
Description:	`HIGH` macro takes const-expression as an argument and returns bits from 16 to 31 of its value.
Example:	`k=0x12345678`

```
ldhi r1, HIGH(k)          ; loads 0x1234 to the highest word of r1
ldi r1, LOW(k)          ; loads 0x5678 to the lowest word of r1
```

Numeric Variables ("=" directive)

```
name = expression
```

For example:

```
Q = 1
Q = Q + 1
```

Value that is constant expression (see the section called “Constant expressions”) can be assigned to the numeric variable. Use of the numeric variable is substituted by it's value. The names of numeric variables should not coincide with the label names and reserved words.

DEFINED, BLANK and IDENT Function

```
DEFINED (name_of_macro)
```

Returns 1, if *name_of_macro*, is a defined macro or numeric variable, otherwise returns zero. DEFINED can only be used in directives of conditional translation.

BLANK is predicate of one argument. Only a macro parameter or single-line text macro can be its argument. Equals 1, if the parameter is not specified or is an empty text macro.

IDENT (*string_literal*) – is treated as identifier, with the name consisting of symbols in *string_literal*. Allows to refer to labels, which names are not identifiers. For example:

```
global IDENT (".label")
ldi    r0, IDENT (".label")
```

DEFINED and BLANK may only be used in directives of conditional translation.

WHILE Cycles

The WHILE block is executed, until the value of const-expression becomes equal to zero.

```
WHILE const-expression
....
....
ENDM
```

Concatenation (Operator ##)

If the operator “##” of macroprocessor is used in macro, the initial text is processed as if there were no symbols “##” in the initial text after parameter substitution.

Note

While intermediate substitution of macro (in nested macros) only “##”, adjacent to parameters, are processed. During the final substitution all “##” are processed.

Example:

```
CONCAT  MACRO  (P1, P2)
        EXITM  P1##P2
        ENDM
CONCAT  (en, try)
```

Stringalisation Operator (#)

If the name of macro parameter follows #, it is substituted with a string literal, containing the text of the parameter. This string literal may be used in `string` directive.

Macros And Numeric Variables Substitution

In certain contexts macros and numeric variables substitution does not take place:

- While processing the arguments of multiline macro
- While processing the argument of the directive `EXITM`
- In single-line macro definition
- While forced substitution of string macro definition (i.e., when in one of the three above mentioned contexts a single-line macro definition was replaced in the manner described below)

To force macro or variable substitution in these contexts, use operator “#&” before the macro or variable name.

Example:

```
LOAD    EQU  ldi             ; LOAD contain "ldi"
LOAD0   EQU  LOAD r0,        ; LOAD0 contain "LOAD r0,"
LOADR0  EQU  #&LOAD r0,  ; LOADR0 contain "ldi r0,"
```

Local Names

```
        LOCAL identifier {,identifier}
```

May be used inside macro definition or WHILE cycle. During macro substitution or performing next `WHILE` cycle a unique identifier @x is generated for every such name, where x – integer positive number, with which this name is changed.

Example:

```
M       MACRO
LOCAL   LAB
        ...
LAB:         // each time  macro M is called, a unique label
             // is generated instead of LAB
        ...
        br  al, LAB
        ...
        ENDM
```

Appendix A. Reserved words

Table A.1. Keywords

section	ends	code	data
zidata	rodata	comment	alines
clines	common	ebyte	eword
byte	word	string	dword
line	space	global	weak

Table A.2. System registers

psw	pc	rsl	rsh
tba	tdp	dfstc	ccp
srp	rfi	rmi	rci
cwp	scbp	rsp	srsp

Table A.3. Additional code

no	eq	le	lt
ls	cs	hs	mi
vs	al	ne	gt
ge	hi	cc	lo
pl	vc		

Table A.4. Mnemonics

abs	acc	accmul	adc
add	adi	adt	and
br	brf	bri	cl
clf	cli	clo	cls
clz	cmp	entry	exs
exz	flush	halt	int
jp	ld	ldacc	ldb
ldbs	ldd	ldds	ldhi
ldi	ldis	ldsys	ldw
ldws	lib	libs	lid
lids	liw	liws	lrb
lrbs	lrd	lrds	lrw

lrws	max	maxs	min
mins	mul	muls	neg
nop	not	or	reset
resp	ret	rli	rol
ror	rri	rti	sat
sbc	shl	shls	shr
shrs	shrsacc	sib	sid
siw	sli	slis	srb
srd	sri	sris	srw
stacc	stb	std	stsys
stw	sub	trap	xor

Table A.5. Macroprocessor directives

INCLUDE	EQU	IF	ELIF
ELSE	ENDIF	MACRO	ENDM
IDENT	DEFINED	BLANK	LOCAL
RADIX	WARNING	ERROR	WHILE
EXITM	=		

Table A.6. Predefined macros

SUBSTR	INSTR	CATSTR	IDSTR
HIGH	LOW	SIZESTR	OPATTR
ECHO			

Linkage Editor (Linker) Of Object Files

User's Manual

Linkage Editor (Linker) Of Object Files: User's Manual

Table of Contents

Chapter 1. Command Line

Object file linker has two main operating modes. The choice of the operating mode is determined by the presence or absence of the **-l** option.

- The main operating mode of linker is the generation of the executable. The file may be generated in one of the four formats, depending on the options, listed in the Table below. The `ELF` format is generated by default.
- The **–l** option instructs the linker to work in the librarian mode. In this mode the mandatory option **–o** indicates the name of the library, with which the librarian deals.

The name of the output file is specified by optional option **–o**. Default names consist of the name of the first of the linked modules and extension, corresponding to the chosen format (correspondingly `.elf, .hex, .mv` and `.bin`).

Command line syntax of the linkage editor:

`lnk.exe [ options [[ name_of_the_object_file ]|[ name_of_the_module ]] ][ name_of_the_module ]`

`options`
One of the parameters from Table 1.1, “Options”.

`name_of_the_object_file`
The name of the file for linking.

`name_of_the_module`
The name of the module. Used in librarian mode.

Table 1.1. Options

`-h (-help)`	display brief information about command line options.
`-m (-mv)`	generate the executable file in `MV` format.
`-x`	generate the executable file in Intel HEX format.
`-b (-bin)`	generate the executable file in binary format.
`-p[filename]`	to create map (mapfile).
`-q (-quiet)`	«Quiet» mode. Do not send output to console.
`-l (-lib)`	instructs the linker to work in the librarian mode. This option is necessary for one of the following options and should be placed to the left of them: `-f`, `-a`, `-i`, `-e`.
`-f (-forced)`	add module(s) to the library with the attribute of forced linking. Librarian mode option, to use it, option `-l` is required.
`-a (-add)`	add module(s) to the library. Librarian mode option, to use it option `-l` is required.
`-e (-extract)`	extract module and to save it to disk. The module is deleted from the library. Librarian mode option, to use it option `-l` is required.
`-cfile`	Configuration file name. map.cfg is used default.

`-i (-info)`	display the list of library modules and symbols. Librarian mode option, to use it option `-l` is required.
`-ofile`	The name of the executable file in linker mode and the name of created library or library under examination in library mode. The name of file should follow `-o` immediately, with no spaces.
`-nd`	Do not add debugging sections to the executable file. All sections which name starts with `.debug` will not be included in the executable file. Used in linker mode.

-M`codepage`	*Compiler messages language and codepage setting*
	
	Orders to the compiler to output all messages in the set language and coding. The codepage available volumes:
	
	"en_US.850" - english lenguage, codepage 850;
	
	"ru_RU.1251" - russian language, code page 1251 (Windows);
	
	"ru_RU.866" - russian language, code page 866 (DOS).
	
	The default setting is "en_US.850" (english lenguage, code page 850).
	
	Example:
	
	`-M "ru_RU.1251"`
	`-Men_US.850`

`--`	instructs parser of the command line, that all following arguments are not options.

Example 1.1. Example of the command line usage

—

```
lnk.exe main.obj \LIB\cpp.lib
```

Generates the object file with the extension `elf`, using the object file main.obj and the library \LIB\cpp.lib.

```
lnk.exe -x -S.test=0x0,0x1000 main.obj clib.lib cpplib.lib
```

Generates the object file in Intel HEX format and declares the section with the name `.test`, placed at the address 0x0 with sizes of 0x1000 byte. The object file `main.obj` and libraries `clib.lib` and `cpplib.lib` are used.

—

Chapter 2. Libraries Of Object Modules

A library is a file, containing program modules. To use the procedure, contained in the library, it is necessary just to call it.

Creating Libraries

To create a library it is necessary to inform linker about its name and to specify one or several object modules, which are to be placed into the library. The names of the modules should be separated by one or several spaces or tabulation. Use command line like following to perform this operation:

```
lnk.exe -l -a -oname_of_the_library name_of_the_object_module
```

Linker will automatically add the corresponding extension to the library name. Also, you may set your own extension – linker will not change it while file creation.

For example, to create the library with the name `my.lib`, which is to contain one object module with the name `math.obj`, following command will be used:

```
lnk.exe -l -a -omy math.obj
```

Operating Libraries

To add object modules into already existing library, use option `-a`.

Option `-f` may be used instead of option `-a` to add files to the library with an attribute of forced linking. This attribute guarantees the module to be added to the executable during the linking stage even though it will not required directly.

To remove the module from the library use option `-e`. The result of the operation will be the removal of the module from the library and saving it to disc. The name of the module is specified without any extension.

```
lnk.exe -l -e -ocpplib.lib math
```

You can copy the module from the library by succesive use of options `-e` and `-a`.

```
lnk.exe -l -e -ocpplib.lib math

lnk.exe -l -a -ocpplib.lib math.obj
```

Option `-i` will generate list of modules in the library, along with names of symbols, defined in those modules:

```
lnk.exe -l -i -oclib.lib
```

Using Libraries

The names of used libraries are separated by one or several spaces or by tabulation.

Chapter 3. Peculiarities Of Realization

Sections Allocation

When many modules are linked, they may contain different sections with identical names. These sections will be successively allocated in memory, with the exception of cases, when they contain identical symbols with linking not weaker than `STB_WEAK`.

Configuration file

The linker configuration file is used to set address space area by user and to locate sections there. It's syntax is:

<FILE> ::= (<allocation information>**)** *

<allocation information> ::= <start address> **[**<max size> **]** **[**<new attributes> **]** '\n'

default **|** **(** **(**<section name> **|** <attributes> **)** **[** '(' **(**lib name **[** ':' module name **(** ',' module name **)** * **]** **|** module name **)** ')' **]** **)** **[** <new size> **]** '\n'

<attributes> - set of some +ro, +data divided by some number of spaces.

<new attributes> - set of any +ro, +data divided by some number of spaces. In this way it's possible to redefine attributes of sections that belong to the area.

<new size> - setting of the .stack, .heap and .hstack sections size (The default size of this sections is accordingly 10000, 10000 и 1000).

It's acceptably to divide linker configuration file strings by empty strings.

The configuration file is assigned by linker option `-c` or looking for `map.cfg`in the folder containing executable linker file (`lnk.exe`).

Example of the linker configuration file usage

Let us suppose in the library lib1.lib in additional to other modules there are m1.obj, m2.obj. Let there is data section named "DataSection".

Let us assume the command line looks like

lnk m3.obj m4.obj m5.obj m6.obj lib1.lib lib2.lib

And configuration file map.cfg is

```
0x00000000 1000

code    (lib1.lib:m1, m2   m3.obj   m4.obj    lib2.lib)
data    (m1.obj, m2.obj)
"DataSection"

0x00002000 +data +ro
".data" (m5.obj)
```

```
0x00004000
+data
+ro

0x00010000
stack 5000
default
```

This means that next requirements are specified:

1. To allocate in the area that begins from address 0:
   - Code sections (".text") from modules m1, m2 of the library lib1, from all modules of the library lib2 and from modules m3, m4;
   - Data sections from modules m1 и m2;
   - Sections named DataSection

   At that this area size limited by 1000 byte.
2. To allocate in the area that begins from address 0x00002000 data sections (".data") from module m5 and set new attributes for this sections.
3. Sections with attribute = ro and sections with attribute = data (from all modules except already allocated) are allocated from the address 0x00004000. Section that added to previous areas will not be added to this area because section name indication (for example DataSection or .text) has higher priority than attributes indication (for example +ro or +data).
4. All other sections allocated from address 0x00010000. At that stack size is set to 5000 byte.

Starting And Ending Symbols

For easy access to sections, symbols which specify the beginning and the end of the section (the end of section is the address, following the last byte of the section) are introduced. These symbols names are obtained in the following way: First, all symbols “ .” in the section name are replaced with “_”, and “_” – for “__”. Two underscores are then added to the beginning and to the end of thus obtained string (for example, “.text” -> “_text” -> “___text__”).Mentioned above symbol names are obtained by adding words "base" and "limit" to thus transformed section name, depending on whether symbol should specify the indicator of section beginning or end correspondingly, for example: ___text__base. For example of using starting and ending symbols see the following:

Example 3.1. The calculation of the heap size

—

```
#include <stddef.h>

extern "C" char __heap__base;
extern "C" char __heap__limit;

int main ()
{
    size_t heap_size = &__heap__limit - &__heap__base;
    return heap_size;
}
```

—

Sections Allocation In Executable File

Different sections contain program instructions and control information. Sections, listed below, are predefined and have their own types and attributes.

Table 3.1. Predefined section names

Section name	Default Section Size	Section Type	Attributes
.text	0	SHT_PROGBITS	SHF_EXECINSTR + SHF_ALLOC
.data	0	SHT_PROGBITS	SHF_WRITE + SHF_ALLOC
.rodata	0	SHT_PROGBITS	SHF_ALLOC
.bss	0	SHT_NOBITS	SHF_WRITE + SHF_ALLOC
.init	0	SHT_PROGBITS	SHF_WRITE + SHF_ALLOC
.fini	0	SHT_PROGBITS	SHF_WRITE + SHF_EXECINSTR + SHF_ALLOC
.stack	0x10000	SHT_NOBITS	SHF_WRITE + SHF_ALLOC + SHF_NOINIT
.hstack	0x1000	SHT_NOBITS	SHF_WRITE + SHF_ALLOC + SHF_NOINIT
.heap	0x10000	SHT_NOBITS	SHF_WRITE + SHF_ALLOC + SHF_NOINIT

.text
: This section contains text or executable program instructions.

.data
: This section contains initialized data, added to the data segment of the program.

.rodata
: This section contains read-only data, which are placed in the segment of read-only memory.

.bss

This section contains uninitialized data, added to the data segment of the program. During the program launch, this area of memory is initialized with zeros.

.init
: This section contains addresses of the functions which should be called for program initialization. During the program execution these functions are called before the main function of the program (the `main()` function for C).

.fini
: This section contains addresses of the functions which should be called at the end of program execution. When the program ends normally, these functions are called after the main function.

.stack
: During the program execution, this memory segment is used for stack.

.hstack
: Used by processor only when its registers are overflowed. One should not use it explicitly. When the program is launched, the address of this stack is passed to the processor.

.heap
: segment of memory, called “heap”, is used for dynamic work with memory.

Failing the configuration file sections will be loaded each following other from zero address. At first the `.text` section is loaded, after that the `.data` one and so on in order, in which they are specified in Table 3.1, “Predefined section names”. After that sections with unpredefined names are loaded. If the configuratoin file is assigned sections will be loaded in the same order but from the specified in the configuration file address.

Note

If the section parameter `SHF_ALLOC` is not specified, the section will be allocated at zero address. Besides, linker has a special attribute for the last three sections. They should not be zero-initialized, so Intel HEX, binary and MV formats of executable files do not contain these sections.

Peculiarities Of `WEAK` Linking

When the linkage editor (linker) links several object files, multiple declaration of global symbols with identical names is not allowed. On the other hand, if there is a declared global symbol, the appearance of `WEAK` symbol with an identical name is not a mistake. Linker identifies these symbols and sections, in which they are located. This is also correct for two and more symbols with `STB_WEAK` linking.

Peculiarities Of `COMMON` Linking

If the name of the symbol(s), declared in section `SHN_COMMON`, coincides with the name of the symbol, declared in ordinary section, the symbol from `SHN_COMMON` will point to the symbol from the normal section. If all symbols with this name are in the section `SHN_COMMON`, the memory for them will be allocated in section `.bss`. The size of the given memory will be taken from the `SH_SIZE` attribute of the symbol.

Mapfile

Mapfile lists all sections and symbols, used in the source code. The structure of the file is the following:

- All sections are listed
- All symbols are listed

Type and attributes are specified for sections in the following form:

Type

P Section has the type `SGT_PROGBITS`

N Section has the type `SGT_NOBITS`

Attributes

D The section has no `SHF_EXECINSTR` property (data section).

RO The section has no `SHF_WRITE` property (read-only section).

NA There section has no `SHF_ALLOC` property (space is not allocated for the section in the executable image in memory).

Debugger

User's Manual

Debugger: User's Manual

Table of Contents

Foreword

The present user's manual contains the description of the debugger, describes the principles of its usage, the methods of its launching.

Chapter 1. Debugger Operation

Main Principles Of Debugger Operation

To start debugging one should create an object file. For this purpose initial file is translated by assembler `asm.exe`. Besides, depending on the assembler options, files of different object formats are created. Detailed information about assembler you may get from here.

After creating an object file you may debug it.

Launching Of Debugger

The syntax of the command line of the debugger is the following:

gdb.exe [-b number] name_of_the_source_file [options], where:

The `name_of_the_source_file` is the name of the object file to debug. The option `-b` argument shows the entry point for .bin-file. This option should be present to load .bin-file.

The debugger uses the object file extension to recognize the format:

-elf executable file in ELF format

-hex executable file in Intel HEX format

-bin executable file in binary format

-mv executable file in MV format

`options` – one of the following options:

`/?` – display brief information about command line options;

`/x name_of_script_file`

– use `name_of_script_file` for debugging. Script file is formed according to the following rule: each line can contain either a debugging command or a comment, starting with '#' symbol

`/l name_of_the_log_file` – use log file for debugging;

In log file with the name `name_of_the_log_file` the contents of the whole debugging session are written – used debugger commands and their output data.

`/r` – working in batch mode. In this mode the debugger just launches the program for execution.

Debugger options may also be written in the special file `gdb.cfg`, which should be placed in the same directory with `gdb.exe`. It should contain the only line, in which the necessary debugger options are enumerated.

Chapter 2. Commands Of Debugger

Notation And Terminology

This Chapter uses the following notation:

| separates two alternatives.

[something] - indicates, that something is not mandatory: it may be specified and may be not.

(group)* means, that group in brackets can be repeated zero or more times.

(group)+ means, that group in brackets can be repeated one or more times.

"line" defines the text “line”.

“Format” is one of the following modes of information output.

x hexadecimal

o octal

t binary

d decimal

r without transformation

N natural

Commands Of Breakpoints Operation

-break-after

```
-break-after number number_of_times
```

Breakpoints number ‘number’ do not work, until it is reached 'number_of_times' times.

-break-delete

```
-break-delete
```

Delete breakpoints, which numbers are specified in the list of arguments.

-break-enable

```
-break-enable
```

Enable (disabled before) breakpoints.

-break-disable

```
-break-disable
```

Disable the mentioned breakpoints.

-break-insert

```
-break-insert [-exec | -read | -write] line
```

Insert a breakpoint or watchpoint into the place, indicated in the ‘line'. Options specify that breakpoint should be set correspondingly on execution, reading and writing. 'A line' may have one of the following forms:

name_of_the_function

name_of_the_variable

A variable should be visible at the moment of setting the watchpoint on it.

name_of_the_file:name_of_the_function

name_of_the_file:name_of_the_line

***address**

***address:length**

'Length' means ”length” of the breakpoint action in bytes.

-break-list

```
-break-list
```

Display the list of set breakpoints.

Commands Of Program Resources Operation

-data-write-memory

```
-data-write-memory address word-format word-size value
```

Write to memory at the address “address” value “value”, occupying the “word-size” byte.

`address'

An expression, defining the address in the memory of the first word to write.

`word-format'

A format, which should be used to write the value to the memory.

`word-size'

A size of the new value in bytes.

`value'

A new value.

-data-read-memory

`-data-read-memory` **`address word-format word-size lines-number columns-number`**

To display the contents of memory as a table consisting of lines-number by columns-number words, each word occupying word-size bytes. In general lines-number*columns-number*word-size bytes are read (returns as “total-bytes”).

`address'

An expression, defining address in the memory of the first word to read.

`word-format'

A format, to use for printing the words from memory.

`word-size'

Size in bytes of every word in memory.

`lines-number'

A number of lines in output table.

`columns-number'

Columns number in the output table.

-data-list-register-names

`-data-list-register-names` **`[ ( reg-num )+ ]`**

Display the list of register names. If the arguments are not given, the list of all register names is displayed. If arguments are integers, the command will print the list of register names, corresponding to arguments.

`reg-num'

Register number

-data-list-register-values

`-data-list-register-values` **`format [ ( reg-num )* ]`**

Display the contents of general registers. "Format" is a format according to which the contents of registers should be printed. An optional list of numbers, specifying registers, which should be displayed, follow specification of format. The absence of the list of register numbers means, that the contents of all registers should be displayed.

`format'

Format to display the register contents.

`reg-num'

Register number.

-data-list-stackregister-values

```
-data-list-stackregister-values  format [ ( reg-num )* ]
```

Display the contents of the stack registers. "Format" is the format in accordance to which the contents of registers should be printed. An optional list of numbers, indicating registers, which should be displayed, follow specification of the format. The absence of the list of numbers means, that the contents of all registers should be displayed.

`format'

Format, to use to display the contents of registers.

`reg-num'

Register number.

-data-list-bufregister-values

```
-data-list-bufregister-values  format [ ( reg-num )* ]
```

To display contents of cyclic buffer registers. "Format" is the format according to which registers contents should be printed. Optional list of numbers, following format specification, indicate registers, which should be displayed . The absence of the list of numbers means, that the contents of all registers should be displayed.

`format'

Format, to use to display the contents of registers.

`reg-num'

Register number.

-data-write-register-values

```
-data-write-register-values  format [reg-num1 value1 ..reg-numN valueN]
```

Store value “valueN” in the general register number "reg-numN" in accordance with the format "format".

`format'

Format, to use to display the contents of registers.

`reg-numN'

Register number.

`valueN'

A new value of the corresponding register.

-data-write-stackregister-values

`-data-write-stackregister-values` **`format [reg-num1 value1 ..reg-numN valueN]`**

Store value “valN” in the stack register number "reg_numN" in accordance with the format "format".

`format'

Format, to use to display the contents of registers.

`reg-numN'

Register number.

`valueN'

A new value of the corresponding register.

-data-write-bufregister-values

`-data-write-bufregister-values` **`format [reg-num1 value1 ..reg-numN valueN]`**

Store value “valN” in the register of the cyclic buffer number "reg_numN" in accordance with the format "format".

`format'

Format, to use to display the contents of registers.

`reg-numN'

Register number.

`valueN'

A new value of the corresponding register.

-data-list-changed-registers

```
-data-list-changed-registers
```

Display list of registers, underwent changes after the latest address to them of the debugger.

-data-evaluate-expression

```
-data-evaluate-expression expression
```

Evaluate expression, specified in the single argument of the command.

-stack-list-frames

```
-stack-list-frames  [first_level last_level]
```

Display stack of functions, starting from the level number 'first_level' and ending with the level 'last_level'. The absence of arguments means the representation of the whole stack.

-stack-select-frame

```
-stack-select-frame number_level
```

To make the level of functions stack with the number 'number_level' current. The upper level of stack is considered to be current by default.

-stack-list-locals

```
-stack-list-locals [mode]
```

To display a list of local variables of the current stack level. With the mode equal to 0 it displays the names of variables only; with the mode equal to 1 – values only.

-stack-list-arguments

```
-stack-list-arguments disp_value [lower_frame upper_frame]
```

Display argument list for frames from the 'lowest-frame' to the 'highest-frame' (inclusively). If the “lowest-frame” and the “highest-frame” are not specified, enumerate arguments for the whole call stack. The argument 'disp_value' should have the value 0 or 1. Value 0 means, that only argument names are displayed, and 1, that argument names and values are printed.

-clear-clock

```
-clear-clock
```

To zeroize cycle-number-counter.

Commands, responsible for execution

-exec-continue

```
-exec-continue [-t ms]
```

Continue the program execution. The program is executed until it meets the breakpoint or until it is over. Argument “ms” of the option "-t" is the time in milliseconds, after which the program will stop its execution.

-exec-run

```
-exec-run [-t ms]
```

Start the program execution. It is executed until it meets the breakpoint or until it is over. Argument “ms” of option "-t" is the time in milliseconds after which the program will be paused.

-exec-until

```
-exec-until filename:line_number [-t ms]
```

Executes the program until the line, specified in the argument. Option "-t" declares with the assistance of argument "ms" time in milliseconds, upon which the program will be paused.

-exec-step

```
-exec-step [-t ms]
```

Restart the program, pausing it upon reaching the beginning of the next line of the source code, if this line is not a function call. If the next line is a function call, the program stops at the first line of that function. Option "-t" declares with the assistance of the argument "ms" time in milliseconds, upon which the program should be paused.

-exec-next

```
-exec-next [-t ms]
```

Restart the program, pausing it on reaching the beginning of the next line of the source code. Option "-t" declares with the assistance of the argument "ms" time in milliseconds, upon which the program should be paused.

-exec-step-instruction

```
-exec-step-instruction [-t ms]
```

Restarts the program execution, executing one microprocessor instruction. Option "-t" declares with the assistance of the argument "ms" time in milliseconds, upon which the program should be paused.

-exec-next-instruction

```
-exec-next-instruction [-t ms]
```

Restarts the program execution, executing one microprossessor instruction. If the instruction is a function call, the execution continues until function return. Option "-t" declares with the assistance of the argument "ms" time in milliseconds, upon which the program should be paused.

-exec-finish

```
-exec-finish [-t ms]
```

Restarts the program execution until the current function is finished. Option "-t" declares with the assistance of the argument "ms" time in milliseconds, upon which the program should be paused.

-exec-abort

```
-exec-abort
```

Aborts program execution.

-exec-reverse

```
-exec-reverse file_name line_number
```

"Reversed" program execution. Declares the command execution point to be line “line_number” of the file “file_name”.

-file-exec-file

```
-file-exec-file -e|-h|-b|-m file_name
```

Specifies the executable file for debugging. Depending on the options, file may have the following format:

-e executable file in ELF format

-h executable file in Intel HEX format

-b executable file in binary format

-m executable file in MV format

General Purpose Commands

Debugger exit

```
quit, -gdb-exit, q
```

Debugger exit

Help display

```
help line
```

Display reference information about the debugger. The “line” may be a command name or the name of the command category.

Note

All commands have the common key '/?', according to which the reference information is displayed.