

KF8ASM User Manuals

V1.1

Contents

1 ASSEMBLER OVERVIEW.....	4
1.1 ABSOLUTE CODE.....	4
1.2 RELOCATABLE CODE.....	4
2 ASSEMBLE SOURCE CODE SYNTAX.....	4
2.1 MARK.....	5
2.2 MNEMONIC, DIRECTIVES, AND MACROS.....	5
2.3 OPERAND.....	5
2.4 COMMENT.....	6
3 ASSEMBLER INTERFACE.....	6
4 EXPRESSION SYNTAX AND ALGORITHMS.....	7
4.1 TEXT STRING.....	7
4.2 ESCAPE CHARACTERS.....	7
4.3 RESERVED WORDS AND PARAGRAPH NAMES.....	8
4.4 NUMERIC CONSTANTS.....	8
4.5 ARITHMETIC OPERATORS AND PRIORITY.....	9
5 PRETREATMENT.....	10
6 PSEUDO INSTRUCTIONS AND MACROS.....	10
6.1 MACRO.....	10
6.2 PSEUDO INSTRUCTIONS.....	11
6.2.1 <i>__CONFIG</i>	11
6.2.2 <i>BANKSEL</i>	11
6.2.3 <i>.CODE</i>	11
6.2.4 <i>.CONSTANT</i>	12
6.2.5 <i>.DA</i>	12
6.2.6 <i>.DATA</i>	12
6.2.7 <i>.DB</i>	12
6.2.8 <i>.DE</i>	13
6.2.9 <i>.DW</i>	13
6.2.10 <i>.DEFINE</i>	13
6.2.11 <i>.ELSE</i>	13
6.2.12 <i>.END</i>	13
6.2.13 <i>.ENDIF</i>	14
6.2.14 <i>.ENDM</i>	14
6.2.15 <i>.EQU</i>	14
6.2.16 <i>.EXTERN</i>	14
6.2.17 <i>.EXITM</i>	14

6. 2. 18 . GLOBAL.....	15
6. 2. 19 . IDATA.....	15
6. 2. 20 . IF.....	16
6. 2. 21 . IFNDEF.....	16
6. 2. 22 . LOCAL.....	16
6. 2. 23 . MACRO.....	17
6. 2. 24 . ORG.....	17
6. 2. 25 . PROCESSOR.....	17
6. 2. 26 . RADIX.....	17
6. 2. 27 . RES.....	18
6. 2. 28 . SET.....	18
6. 2. 29 . UDATA.....	18
6. 2. 30 . VARIABLE.....	18
APPENDIX A ASSEMBLER INSTRUCTION SET.....	19

1 Assembler Overview

KF8ASM assembler provides a platform for developing assembly language source code for KF series single-chip microcomputer.

The assembler can be used in two ways:

- Generate absolute code that can be directly executed by single-chip microcomputer.
- Generate relocatable code that can be linked with other independently assembled or compiled modules.

1.1 Absolute Code

In absolute code mode, the source file of assembly language is directly replaced by HEX file by assembler, which the mode is absolute because the final address is hard-coded into the source file.

1.2 relocatable code

In relocatable mode, the source file is divided into several modules, and each module is independently compiled by the assembler into **Object file**. The compiled object file can be placed anywhere in the memory of MCU. Linker will solve the problem of symbol application, assigning addresses, and writing the final address where the machine code needs to update the address.

The output of the linker is absolute code.

2 Assemble Source Code Syntax

Assembly language is a programming language that can be used to write source code for applications. You can use any ASCII text editor to create source code files.

Assembly language is a programming language with loose syntax.

There are four types of information for each line of the source file:

- Label
- Mnemonic, pseudo-directives and macros
- Operand
- Notes

The order and location of this information is very important. The labels must be written at the top and start from the first column. Mnemonic symbol cannot be written in the top case. The operand follows the mnemonic symbol. Comments can be followed by operands, mnemonics symbol or label, and can start from any column.

Labels and mnemonics must be separated with whitespace or colon, and mnemonics and operands must be separated with whitespace. Multiple operands must be separated by commas.

Whitespace refers to one or more spaces or tabs. Whitespace is used to segment source code lines. The purpose of using whitespace is that the code is clear and easy to read. Unless inside the character constant, the meaning of multiple whitespace is the same as that of one space.

2.1 Mark

Labels are used to represent a line, a set of codes or a constant value. The jump instruction needs it. The label should start from the first column. It can be followed by a colon (:), a space, a tab, or a newline character. The label must start with one word or an underscore (_), it can contain alphanumeric characters and underscores.

Label Limit:

- It cannot start with two leading underscores, for example: __config.
- It cannot start with a leading underscore and a number, for example: 2NDLOOP.
- It cannot be a reserved word of the assembler (See section 4.3 "Reserved Words and Segment Names").

If you use a colon when defining a label, the colon is treated as a label separator rather than as part of the label itself.

2.2 mnemonic, directives, and macros

Mnemonics tell the assembler which machine instructions to assemble. For example, add, jmp or mov. Unlike labels you create yourself, mnemonics are provided by the assembler. Mnemonics are not case sensitive.

Pseudo-instructions are assembler commands that appear in source code, but it is usually not compiled directly into opcodes. They are used to control the input, output and data distribution of the assembler. Pseudo-instructions are not case sensitive.

Macro is a set of user-defined instructions and pseudo-instructions that are embedded the assembler source code to execute on the same line whenever macro is called.

Assembler instruction mnemonics, pseudo instructions and macro calls should start from the second column or later columns. If same line has a label, and the instruction must be separated from this label by colon or one or more spaces or tabs.

2.3 Operand

Operands give information about the instruction, including the data that should be used and the storage location of the instruction. You must use one or more spaces or tabs to separate operands from mnemonics. Multiple operands must be separated with commas. Attention should be paid to the syntax format of immediate number, i.e. hexadecimal and decimal expressions have

different meanings. For example, “MOV R0, # 3” are compared with “MOV R0, # 0x03”. The previous statement is recognized as register 3 assigning R0, followed by immediate number 3 assigning R0. If immediate number transfer is written in decimal, it should be modified to “MOV R0, ## 3”.

2.4 Comment

The comment is text that explains the operation of one or more lines of code. The assembler treats any text after the semicolon as comments. All characters after the semicolon and up to the end of the line are ignored.

String constants containing semicolons are allowed and will not be confused with comments.

Note: All punctuation marks (colons, commas, semicolons) mentioned above must be corresponding English punctuation marks.

3 Assembler Interface

If you run the assembler from the command line, the general calling syntax is:

Kf8asm [options] asm-file

The available options are as follows:

Options	Means
-a<format>	Select generated file format, it could be one of the following: inhx8m, inhx8s, inhx16, inhx32(default value)
-c	Generate relocatable file
-d	Output debugging information
-Dsymbol[=value]	Equivalent with .define<symbol> <value>
-e[ON/OFF]	Expand macros in list file
-O<file>	Specify name of the output HEX file
-h	Display help information
-i	Ignored the case of the assemble source file
-I<directory>	Specify a header file directory
-l	List supported processors
-M	Output dependent file
-p<processor>	Specify the target processor
-q	Exit
-r<radix>	Setting system, that is the assembler parses unformatted digit.<radix> can be one of the following: oct, dec, hex. Note also the operand syntax qualifiers, #[0-9]+ will is parse as decimal immediate data.
-u	Use absolute path
-v	Output assembler's version information and then exit
-w[0 1 2]	Setting the information level

Unless explicitly specified, the assembler will remove the suffix ". asm" from the source file name and give this name add the suffix ". lst" or ". hex" as the file names of the output list file and HEX file. In the same directory, do not name file names that only depend on the case of characters in the file names. The file names on the file systems of most modern operating systems are case

sensitive (In Windows systems, whether NTFS is case-sensitive or not is optional, and FAT32 is case-insensitive. Depending on case to distinguish files can cause many imperceptible problems).

The assembler always generates an `lst` file, and if there are no errors, it also generates a `hex` file and a `.o` file.

The development tool integrates parameter information, and usually uses the default tool configuration options.

4 Expression Syntax and Algorithms

4.1 Text String

"String" is any valid ASCII character (It is between 0 and 127) sequence. It may include quotation marks or empty characters.

The way to add special characters to a string is to escape with a backslash `\` before these special characters. The escape sequence applied to strings also applies to characters.

If the corresponding post quotation marks are found in the processing of the string, the string ends. If the corresponding closing quotation mark is not found before the end, the string will end at the end of the line. Although you can't extend the string directly to the second line, but usually you can use the `'dw'` Pseudo-instructions again on the next line to achieve this goal.

The `'dw'` Pseudo-instructions stores the entire string in contiguous word units. If the number is odd of a string character (byte), and the `'dw'` and `'data'` Pseudo-instructions fill a byte of 0 (00) at the end of the string.

If a string is used as an immediate operand, it must be one character long, otherwise an error will be generated. Need to note that the manipulation method of a string often ends with `\0`, unless the custom treatment method is based on a given length.

4.2 Escape Characters

The assembler accepts ANSI C escape sequences to represent certain special control characters:

ANSI C escape sequences:

Escape Character	Description	Hexadecimal Value
<code>\a</code>	Alarm character	07
<code>\b</code>	Backspace character	08
<code>\f</code>	Form feed character	0C
<code>\n</code>	Line break	0A
<code>\r</code>	Enter	0D
<code>\t</code>	Horizontal Tab	09
<code>\v</code>	Vertical Tab	0B
<code>\\</code>	Backslash	5C
<code>\?</code>	Question mark	3F
<code>\'</code>	Single quotes	27

\"	Double quotation marks	22
\000	Octal	
\xHH	Hexadecimal digit	

4.3 Reserved words and paragraph names

You cannot use the following words as labels, constants, or variable names:

- Pseudo-instructions (see Section 6.2 "Pseudo-instructions").
- Instructions (see Appendix A "Instruction Set").

In addition, the assembler retains the following segment names:

Segment Name	Application
.code	.code pseudo-instructions default segment name
.idata	.idata pseudo-instructions default segment name
.udata	.udata pseudo-instructions default segment name

4.4 Numeric Constants

The assembler supports the following constant radix formats: hexadecimal, decimal, octal, binary and ASCII. The default cardinality is hexadecimal; When the cardinality is not explicitly specified with the base descriptor, the default cardinality will determine what values are assigned to the constants in the target file.

The following table gives specifications for various cardinality values:

Type	Syntax	Example
Binary	B 'Binary digit'	B '01010101'
Octal	O 'Octal digit'	O '777'
Decimal	D 'Decimal digit' .Digit	D '100' .100
Hexadecimal	H 'Hexadecimal digit' Hexadecimal digit	H '9f' 0x9f
ASCII	A 'Character' 'Character'	A 'C' 'C'

Note:

1. The binary integer is 'b' or 'B' followed by one or more binary digits '01' enclosed in single quotation marks.
2. Octal integer is 'o' or 'O' followed by one or more octal digits enclosed in single quotation marks.
3. The decimal integer is 'd' or 'D' followed by one or more decimal digits '0123456789' enclosed in single quotation marks. Or, the decimal integer is '.' followed by one or more decimal numbers '0123456789'.
4. The hexadecimal integer is 'h' or 'H' followed by one or more hexadecimal numbers enclosed in single quotation marks, i.e. '0123456789abcdefABCDEF'. Alternatively, the

hexadecimal is whole '0x' followed by one or more hexadecimal digit , i.e.

'0123456789abcdefABCDEF'.

5. The ASCII character is 'a' or 'A' followed by a single quotation mark. Or, ASCII character is a character enclosed in single quotation marks.

4.5 Arithmetic Operators and Priority

Arithmetic operators can be used with pseudo instructions and their variables specified in the following table.

Note: These operators cannot be used with program variables, they can only be used with pseudo instructions to control conditional compilation, etc.

The order of operators in the table corresponds to their priority, where the first operator has the highest priority and the last operator has the lowest priority. Priority refers to the execution order of operators in code statements.

Arithmetic Operators (Sorted by Priority)

Operator	Description	Priority	Example
\$	Current program counter	1	goto \$+3
!	Not	2	if !(a==b)
-	Negation (2's complement)	2	-1*length
~	Complement	2	flags=~flags
*	Multiplication	3	a=b*c
/	Division	3	a=b/c
%	Remainder	3	a=b%c
+	Add	4	length=len*8+3
-	Subtraction	4	length=(len-1)*2
<<	Left shift	5	flags=flags<<1
>>	Right shift	5	flags=flags>>1
>=	Greater than or equal to	6	if id>=root
>	Greater than	6	if id>root
<	Less than	6	if id<root
<=	Less than or equal to	6	if id<=root
==	Equal	6	if id == root
!=	Not equal	6	if id!=root
&	Bit And	7	flags=flags & ERR_BIT
^	Bit Xor	7	flags=flags^ERR_BIT
	Bit Or	7	flags=flags ERR_BIT
&&	Logic And	8	if(len==51) && b
	Logic Or	8	if(len==62) b
=	Assignment	9	index=1
+=	Add assignment	9	index +=1
-=	Subtraction assignment	9	index-=1
=	Multiplication assignment	9	index=1
/=	Division assignment	9	index/=1
%=	Remainder assignment	9	index%=1
>>=	Right shift assignment	9	index>>=1
<<=	Left shift assignment	9	index<<=1
&=	And assignment	9	index&=1
=	Or assignment	9	index =1
^=	Xor assignment	9	index ^=1

++	Add Add assignment	9	i++
--	Sub Sub assignment	9	i--

The '++' and '--' operators can only be used on a single line and cannot be nested in other expression evaluations.

5 Pretreatment

In the following code:

```
.INCLUDE foo.inc
```

The assembler will read the foo.inc file until the end of the file, and then the assembler will go to deal with code following the '.INCLUDE foo.inc' line.

The DEFINE pseudo instruction will be specially handled by the assembler.

As long as the assembler processes a line of code similar to the following:

```
.DEFINE X Y
```

Every X encountered later will be replaced by Y. Until the end of the source file or a line of code is encountered:

```
.UNDEFINE X
```

Using .INCLUDE instruction to include the appropriate existing files in your code, and the assembler will automatically search the default directory for the header file with the specified name.

6 pseudo instructions and macros

In absolute code mode, the .ORG pseudo instruction is used to instruct the assembler to allocate the starting storage place of assembly code address. If not specified by .ORG, the assembler will default generate code from 0x0000 storage address.

6.1 Macro

Kf8asm supports simple macro mode, which can be defined as using macros like the following.

Define:

```
SetDate .macro parm1, parm2
```

```
MOV R0, #parm1
```

```
MOV R1, #parm2
```

```
ST [R0], R1
```

```
.endm
```

Use: SetDate 33, 25

Actual result:

```
9821 M MOV R0, #parm1
```

```
9919    M   MOV R1, #parm2
F748    M   ST  [R0], R1
```

It should be noted that the name defined by the macro must be written in the top case, without parameters, and the default data is decimal. It is not supported to change the code to hexadecimal mode with "0x" after "#". The immediate numbers need to be decorated with "#", otherwise they exist as register address meanings.

6.2 pseudo instructions

6.2.1 __CONFIG

__CONFIG <expression>

Note: There are two underscores in front of it.

This pseudo instruction is used as an expression of configuration word value, such as "__config 0x2007, 0x3ffc". Where the preceding value is the address and the following value is the assignment result. The address information should be consistent with the chip information. Other address data does not work. In addition, there should be spaces or tabs in front of it, that is, it cannot be written in freeze frame.

The result of assignment can be expressed, i.e. 0x3ffc&0x0328|0x415A.

6.2.2 BANKSEL

BANKSEL <label>

This pseudo instruction instructs the assembler and linker to generate a store selection code to set the store to contain specifies the store for the label. Only one label should be specified. You must have defined this label earlier.

The linker generates the appropriate memory selection code and generates the appropriate set/clear instruction for the RPx bit in the PSW register, or the BANK register assignment. If the device contains only one memory area of RAM area, no instructions will be generated.

This pseudo instruction is used in the following types of code: absolute code or relocatable code.

Simple example:

```
banksel var      ; Choose the correct bank for var
add var, r0      ; Operating var
```

The pseudo instruction does not explicitly specify the region in which the variable is located, or specifies the region but writes the code without particularly considering the underlying region.

6.2.3 .CODE

<label> .CODE <expression>

If label is not specified, the segment is named .code, but there is at most one unnamed segment in a project. The starting address is initialized to the address specified by expression, and if no address is specified, the starting address is assigned when linking.

This instruction is only available in relocatable mode.

There is no “end code” pseudo instruction. When another code segment or data segment is defined or arrives at the end of file, the code of a segment ends automatically.

For the writing of assembly projects, if there are multiple source files, you should add this pseudo instruction at the beginning of the code, and the address may not be specified.

6.2.4 .CONSTANT

`.CONSTANT <label> = <expression> [, <label> = <expression>]*`

Creates symbols to be used in assembler expressions. Once the constant is initialized, it cannot be change again, and the expression must be completely resolvable and must not contain labels that need to be relocated when assigning values.

This is the main difference between symbols declared as constants and those declared as variables or created by `.set` pseudo instruction. In addition, constants and variables can be used interchangeably in absolute code expressions.

Usage of this pseudo instruction: absolute code or relocatable code.

6.2.5 .DA

`<label> .DA <expression> [, <expression>]*`

Stores compressed strings in program memory.

This pseudo instruction is used in the following types of code: absolute code or relocatable code.

6.2.6 .DATA

`.DATA <expression> [, <expression>]*`

Initialize one or more program memory words with data. The data can be in the form of constant and relocatable or outer labels, or expressions of any of the above.

Note: This pseudo instruction is not a pseudo instruction defining data segments. See `.idata` and `.udata` for pseudo instructions defining data segments.

6.2.7 .DB

`<label> .DB <expression> [, <expression>]*`

The program memory word is reserved with an 8-bit value. Multiple expressions continue to continuously fill bytes until the end of the expression. If there are an odd number of expressions, the last byte will be zero.

This pseudo instruction is used in the following types of code: absolute code or relocatable code.

For example:

`.db 0x0f, 't', 0x0f, 'e', 0x0f, 's', 0x0f, 't', '\n'`

ASCII: 0x0F74 0x0F65 0x0F73 0x0F74 0x0a00

In general, it can be used with pseudo instruction and define initial RAM information. Such as:

`ID_debug_touch_0 .idata`

```
_TOUCH_CH_TRS_EN      ;//Array name
.de 0x01
.db 0x00
.db 0x00
.dw 0x1234
```

6.2.8 .DE

<label> .DE <expression> [, <expression>]*

Define EEPROM data, and the characters in each string will be stored in a separate word.

The pseudo instruction is not currently supported.

6.2.9 .DW

<label> .DW <expression> [, <expression>]*

Define word data, i.e. word length of 16bit, occupying a program address space.

For example, ". DW 0x0000", that is the compile order address drops the value 0x0000, which is a NOP instruction.

6.2.10 .DEFINE

.DEFINE name <string>

Defines text substitution symbols. This pseudo instruction defines a text replacement string. When encounter name in assembly code any bit, it will be replaced by a string. Using pseudo instructions without string will cause the definition of name is internally marked and can be used for conditional detection in ifdef pseudo instructions.

Explicit RAM usage can be defined using this pseudo instruction. Case-insensitive, such as: ".DEFINE aaa 0x183". The 'aaa' exist as variable meaning in code , the address is 0x83 in block 1.

6.2.11 .ELSE

.ELSE

It is used with .if pseudo instruction, provides an alternate path to assembly code if .if evaluates to FALSE.

6.2.12 .END

.END

In any assembler, at least one .end pseudo instruction is required to indicate the end of compilation. In a single assembly file program, only one. end pseudo instruction must be used.

Care should be taken not to include files containing .end, because these files will stop the assembly prematurely.

6.2.13 .ENDIF

`.ENDIF`

Represents the end of the conditional assembly.

Every time you use `.if` pseudo instruction, there must be a corresponding `.endif`. `.if` and `.endif` are not instructions, they are used only for conditional assembly code.

6.2.14 .ENDM

`.ENDM`

Terminates a macro definition that starts with `.macro`.

Every time you use a `.macro` instruction, there must be a corresponding `.endm`.

6.2.15 .EQU

`<label> .EQU <expression>`

Defines an assembler constant. The value of expression is assigned to label. The value of label cannot be changed.

A common expression for Defining pseudo instruction to define variable. Such as:

`"bbb .EQU 0x183"`. But it need to note that must freeze-frame write it.

6.2.16 .EXTERN

`.EXTERN <symbol> [, <symbol>]`

Declares an externally defined label. This pseudo instruction can only be used in relocatable code.

This pseudo-instruction declares a symbol that can be used in the current module but is defined as a global label name in another module. You must include an `.extern` statement before you can use the label. At least one label must be specified on this line.

This directive can be used as long as there is more than one file in the project. When a file uses a label (usually a variable), the file will use `.extern`. `.global` will be used in another file, so that the reference number can be seen by other documents. These two pseudo instructions must be used as specified, otherwise the label will not be visible in other files. This can be a declaration of a variable or a declaration of a function.

Simple example:

`.extern func`

...

6.2.17 .EXITM

`.EXITM`

When assembling, force immediate return from macro extension. The effect is the same as when encountering the `.endm` pseudo instruction.

6.2.18 .GLOBAL

`.GLOBAL <symbol> [, <symbol>]*`

This pseudo instruction declares symbolic names defined in the current module and available to other modules. At least one label must be specified on this line.

This pseudo instruction is used in the following types of code: relocatable code.

When a project uses multiple files, you need to generate linkable object code. When this happens, you can make use `.global` and `.extern` directives. `.global` is used to make labels visible to other files. `.extern` must be used in labeled files so that the labels are visible in the file.

Simple example:

```
.global Var1, Var2
.global AddThree
.ldata
    Var1 .res 1
    Var2 .res 1
.code
AddThree:
    add r0, 3
```

6.2.19 .IDATA

`<label> .IDATA <expression>`

This pseudo instruction declares the beginning of an initialized piece of data. If label is not specified, this section is named `.idata`. The starting address is initialized to the specified address, and if no address is specified, the starting address is assigned when linking. The storage space will be allocated, the initialization data will be placed in ROM, and the user must provide code to load the initialization data into the allocated storage space.

This pseudo instruction is used in the following types of code: relocatable code.

Use this directive to initialize variables, or use the `.ldata` pseudo instruction, and then start to initialize variables with the value in the code. It is recommended that variables always be initialized.

Simple example:

```
.idata
    limitL .dw 0
    limitH .dw D '300'
    gain   .dw D '5'
    flags .db 0
    String .db 'Hi therer!'
```

The content declared with this macro is the value assigned to the required initialization in RAM. Therefore, the code is needed to implement it. If there are multiple idate segments, and the naming distinction of `< label >` should be given separately.

Initialization functions need to be called at the beginning of the code, namely:

```
PAGESEL    _cinit
CALL       _cinit ; deal with initial value, idate segment
PAGESEL$
```

6.2.20 .IF

`.IF <expression>`

Code block to start executing conditional assembly: If the value of expression is TRUE, the assembly will be followed by code after `.if`. Otherwise, subsequent code will be skipped until the `.else` or `.endif` pseudo instruction is encountered. The value is 0 and it is treated as a logical FALSE. An expression whose value is any other value is treated as logical TRUE. The `.if` pseudo instruction runs according to the logical value of the expression.

This pseudo instruction is used in the following types of code: absolute code or relocatable code.

This pseudo instruction is not an instruction, but only used to control which code blocks will be assembled.

Simple example:

```
.if version == 100          ;check current version
    mov r0, 0x0a
    mov io_1, r0
.else
    mov r0, 0x01a
    mov io_2, r0
.endif
.IFDEF
    .IFDEF <symbol>
```

If label has been defined previously (usually by executing the `.define` pseudo instruction). The assembly will continue until a matching `.else` or `.endif` pseudo instruction is encountered.

This directive is used in the following types of code: absolute code or relocatable code.

This pseudo instruction is not an instruction, but only used to control which code blocks will be assembled. You use this Pseudo-instructions delete or add code without commenting on large blocks of code during debugging.

6.2.21 .IFNDEF

`.IFNDEF <symbol>`

If the label has not been defined before or has been undefined by executing the `.undefine` pseudo instruction, the code after the pseudo instruction will be assembled. Assembly will be enabled or disabled until the next matching `.else` or `.endif` pseudo instruction is encountered.

This directive is used in the following types of code: absolute code or relocatable code.

This pseudo instruction is not an instruction, but only used to control which code blocks will be assembled. You use this Pseudo-instructions delete or add code without commenting on large blocks of code during debugging.

6.2.22 .LOCAL

`.LOCAL <symbol> [[=<expression>], [<symbol> [=expression>]]*`

Declares to treat the specified data element as a local element of the macro. Label can be the same as other declaring label outside of the macro definition, so there will be no conflict between the two labels.

If you call macros recursively, each call will have its own local copy.

This pseudo instruction is used in the following types of code: absolute code or relocatable code.

6.2.23 .MACRO

<label> .MACRO [<symbol>, [, <symbol>]*]

The macro is a series of instructions that can be inserted into assembly source code using a macro call. You must first define macro before it can be referenced in subsequent source code.

This pseudo instruction is used in the following types of code: absolute code or relocatable code.

Simple example:

;Define macro

Read .macro device, buffer, count

 mov r0, device

 mov ram_20, r0

 mov r0, buffer ;buffer address

 mov r0, count ;counter

 call sys_21 ;call sub program

.endm

;Use the defined macro above

Read 0x0, 0x55, 0x05

6.2.24 .ORG

.ORG <expression>

Set the program start of subsequent code at the address defined by expression. If the .org pseudo instruction is not specified, code will be generated from address 0.

6.2.25 .PROCESSOR

.PROCESSOR <symbol>

Select the target processor. It doesn't work for the time being.

6.2.26 .RADIX

.RADIX <symbol>

Sets the default cardinality value for the assembler. Symbol is selected from oct, dec, hex, representing octal, decimal, hexadecimal. The assembler will use the cardinality value specified here to explain that the value of cardinality value what is not explicitly specified.

MOV R0, #100 defaults to 100 in decimal and 0x64 in hexadecimal. But if the macro declares the default according to the immediate parameter in hexadecimal, the result is 0x100.

6.2.27 .RES

`.RES <expr>`

Moves the memory cell pointer forward from the current cell by the value specified in `expr`. It is used to preserve data storage space.

The most common use of `.res` is to store data in relocatable code, which needs to be used in conjunction with `udata`.

For example:

```
_MAIN_RAM_2      .udata
    Touch_Flag    .res      1
    Touch_Flag1   .res      1
```

That is, under the RAM segment represented by `_MAIN_RAM_2`, the variable at the first address is `Touch_Flag`. The second address is the variable `Touch_Flag` and `Touch_Flag1` must be in the same RAM partition.

6.2.28 .SET

`<label> .SET <expression>`

Assigns the value of a valid assembler expression specified by `expression` to `label`. The function of the `.set` pseudo instruction is similar to that of the `.equ` pseudo instruction, except that the set value is later changed by other. set directives.

This directive is used in the following types of code: absolute code or relocatable code.

6.2.29 .UDATA

`<label> .UDATA <expression>`

This directive declares the beginning of an uninitialized piece of data. If no label is specified, this paragraph is named `.udata`. The starting address is initialized to the specified address, and if no address is specified, the starting address is assigned when linking. This segment does not generate code. You should use the `.res` pseudo instruction to reserve space for data.

This directive is used in the following types of code: relocatable code.

6.2.30 .VARIABLE

`.VARIABLE <label> [= <expression>, <label> [= <expression>]]*`

Declare a variable named `label`, which can be reassigned. The value of `label` does not have to be specified at declaration time.

Appendix A Assembler Instruction Set

[Mnemonic * is not part of the syntax, there are instructions]

Mnemonic/operand	Instruction Format	Instruction Description	Period	Impact mark
NOP	0000 0000 0000 0000	NOP	1	
NOPZ	1111 1111 1111 1111	NOP	1	
CRET	0000 0000 0000 0000	Subroutine return instruction	2	
RRET Rn, #data *	1011 0rrr kkkk kkkk	Immediate number send to Rn and return	2	
IRET	0000 0000 0000 1001	Interrupt return instruction	2	
CWDT	0000 0000 0110 0100	WDT clear	1	
IDLE	0000 0000 0110 0011	Go into sleep mode	1	
Data transfer instruction				
MOV dir	0000 1111 ffff ffff	Dir←(dir)	1	Z
MOV Rn,dir	0101 rrr0 ffff ffff	Rn←(dir)	1	
MOV dir, Rn	0101 rrr1 ffff ffff	dir←(Rn)	1	
MOV Rn,#data *	1001 1rrr kkkk kkkk	Rn←data	1	
MOV Rn,Rs	1111 1000 11ss srrr	Rn←(Rs)	1	
LD Rn,[Rs]	1111 0111 00ss srrr	Rn←((Rs))	1	
ST [Rn],Rs	1111 0111 01ss srrr	(Rn)←(Rs)	1	
SWAPR Rn,dir	0100 rrr0 ffff ffff	Rn<7:4>=dir<3:0> Rn<3:0>=dir<7:4>	1	
SWAP dir	0100 rrr1 ffff ffff	dir<7:4>=dir<3:0> dir<3:0>=dir<7:4>	1	
MOVB #data *	1110 0001 kkkk kkkk	BANK←data	1	
MOVP #data *	1110 0000 kkkk kkkk	PCH←data	1	
Arithmetic operation instruction				
ADD Rm,dir	0010 0rr0 ffff ffff	Rm←(Rm)+(dir)	1	CY、DC、Z
ADD dir,Rm	0010 0rr1 ffff ffff	dir←(Rm)+(dir)	1	CY、DC、Z
ADD Rn,#data *	1000 0rrr kkkk kkkk	Rn←(Rn)+data	1	CY、DC、Z
ADD Rn,Rs	1111 1000 00ss srrr	Rn←(Rn)+(Rs)	1	CY、DC、Z
SUB Rm,dir	0011 1rr0 ffff ffff	Rm←(dir)-(Rm)	1	CY、DC、Z
SUB dir,Rm	0011 1rr1 ffff ffff	dir←(dir)-(Rm)	1	CY、DC、Z
SUB Rn,#data *	1010 0rrr kkkk kkkk	Rn←data-(Rn)	1	CY、DC、Z
SUB Rn,Rs	1111 1000 01ss srrr	Rn←(Rs)-(Rn)	1	CY、DC、Z
CMP Rn,#data *	1111 0010 1kkk krrr	-	1	CY、DC、Z
CMP Rn,Rs	1111 0001 10ss srrr	-	1	CY、DC、Z
INC dir	0000 1011 ffff ffff	dir←(dir)+1	1	Z
INCR dir	0000 1010 ffff ffff	R0←(dir)+1	1	Z
INC Rn	1111 1111 0001 0rrr	Rn←(Rn)+1	1	Z
DEC dir	0000 0111 ffff ffff	dir←(dir)-1	1	Z
DECR dir	0000 0110 ffff ffff	R0←(dir)-1	1	Z
DEC Rn	1111 1111 0000 1rrr	Rn←(Rn)-1	1	Z
Logical operation instruction				
AND Rm,dir	0010 1rr0 ffff ffff	Rm←(Rm)^(dir)	1	Z
AND dir,Rm	0010 1rr1 ffff ffff	dir←(dir)^(Rm)	1	Z
AND Rn,#data *	1000 1rrr kkkk kkkk	Rn←(Rn)^data	1	Z
AND Rn,Rs	1111 1000 10ss srrr	Rn←(Rn)^(Rs)	1	Z
ORL Rm,dir	0011 0rr0 ffff ffff	Rm←(Rm)^(dir)	1	Z
ORL dir,Rm	0011 0rr1 ffff ffff	dir←(dir)^(Rm)	1	Z
ORL Rn,#data *	1001 0rrr kkkk kkkk	Rn←(Rn)^(data)	1	Z

Mnemonic/operand	Instruction Format	Instruction Description	Period	Impact mark
ORL Rn,Rs	1111_1001_00ss_srrr	$R_n \leftarrow (R_n) \vee (R_s)$	1	Z
XOR Rm,dir	0001_1rr0_ffff_ffff	$R_m \leftarrow (R_m) \oplus (dir)$	1	Z
XOR dir,Rm	0001_1rr1_ffff_ffff	$dir \leftarrow (dir) \oplus (R_m)$	1	Z
XOR Rn,#data *	1010_1rrr_kkkk_kkkk	$R_n \leftarrow (R_n) \oplus data$	1	Z
XOR Rn,Rs	1111_1001_01ss_srrr	$R_n \leftarrow (R_n) \oplus (R_s)$	1	Z
CLR Rn	0000_0010_xxxx_1rrr	$R_n = 0$	1	Z
CLR dir	0000_0011_ffff_ffff	$dir = 0$	1	Z
CPLR dir	0000_0100_ffff_ffff	$R_0 \leftarrow \neg (dir)$	1	Z
CPL dir	0000_0101_ffff_ffff	$dir \leftarrow \neg (dir)$	1	Z
CPL Rn	1111_1111_0000_0rrr	$R_n \leftarrow \neg (R_n)$	1	Z
RRCR dir	0001_0000_ffff_ffff	$R_0 \leftarrow (dir)$ C cycle with carry 1 bit right	1	CY
RRC dir	0001_0001_ffff_ffff	$dir \leftarrow (dir)$ C cycle with carry 1 bit right	1	CY
RRC Rn	1111_1111_0010_0rrr	$R_n \leftarrow (R_n)$ C cycle with carry 1 bit right	1	CY
RLCR dir	0001_0010_ffff_ffff	$R_0 \leftarrow (dir)$ C cycle with carry 1 bit left	1	CY
RLC dir	0001_0011_ffff_ffff	$dir \leftarrow (dir)$ C cycle with carry 1 bit left	1	CY
RLC Rn	1111_1111_0001_1rrr	$R_n \leftarrow (R_n)$ C cycle with carry 1 bit left	1	CY
Bit operation instruction				
CLR dir,b	0110_0bbb_ffff_ffff	Clear the b bit of dir to 0	1	
SET dir,b	0110_1bbb_ffff_ffff	Clear the b bit of dir to 1	1	
CLR Rn,b	1111_1110_00bb_brrr	Clear the b bit of Rn to 0	1	
SET Rn,b	1111_1110_01bb_brrr	Clear the b bit of Rn to 1	1	
Branch instruction				
DECRJZ dir	0000_1000_ffff_ffff	$R_0 \leftarrow (dir) - 1$, Skip next instruction for 0	1/2	
DECJZ dir	0000_1001_ffff_ffff	$dir \leftarrow (dir) - 1$, Skip next instruction for 0	1/2	
DECJZ Rn	1111_1111_0101_1rrr	$R_n \leftarrow (R_n) - 1$, Skip next instruction for 0	1/2	
INCRJZ dir	0000_1100_ffff_ffff	$R_0 \leftarrow (dir) + 1$, Skip next instruction for 0	1/2	
INCJZ dir	0000_1101_ffff_ffff	$dir \leftarrow (dir) + 1$, Skip next instruction for 0	1/2	
INCJZ Rn	1111_1111_0101_0rrr	$R_n \leftarrow (R_n) + 1$, Skip next instruction for 0	1/2	
JNB dir,b	0111_0bbb_ffff_ffff	The b bit of dir skip next instruction for 0	1/2	
JB dir,b	0111_1bbb_ffff_ffff	The b bit of dir skip next instruction for 1	1/2	
JNB Rn,b	1111_0111_10bb_brrr	The b bit of Rn skip next instruction for 0	1/2	
JB Rn,b	1111_0111_11bb_brrr	The b bit of Rn skip next instruction for 1	1/2	
JMP #data12 *	1100_kkkk_kkkk_kkkk	Unconditional branch instruction	2	
CALL #data12 *	1101_kkkk_kkkk_kkkk	Subroutine call instruction	2	

Note: dir is a general register or a special function register; Rn and Rs represent R0 to R7; Rm represents R0 to R3; #data represents an 8-bit immediate number, #data12 represents a 12-bit immediate number, where the immediate number should be written in hexadecimal, and if it is decimal, it should be written as ##data or ##data12; b represents the b-th bit of the register; [Rn] represents the data in the address pointed to by the value in Rn; () represents data in a special function register, general data register, or register bank.

The instruction with *, MOVB #data, MOVP #data and CMP Rn, #data exist for partial chipType, please refer to the chip data manual for details.