Migration guide

Migrating from the CS+ CA78K0R toolchain for RL78 to IAR Embedded Workbench® for RL78

Use this guide as a guideline when converting source code written for the CS+ CA78K0R toolchain for RL78 to IAR Embedded Workbench® for RL78.

	Product	Version number
Migrating from	CS+ CA78K0R (CA78K0R)	1.20 to 1.7x
Migrating to	IAR Embedded Workbench for RL78 (EWRL78)	2.x

Migration overview

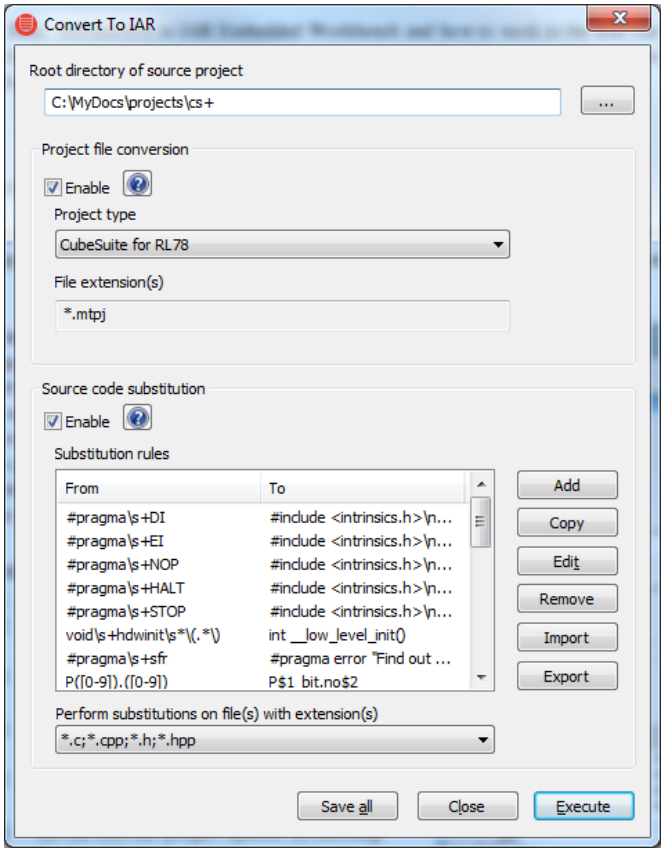
Migrating an existing project from Renesas toolchain for RL78 requires that you collect information about your current project and then apply this information to the new IAR EWRL78 project. In addition, you need to make some changes in the actual source code. The information in this document is intended to simplify this process.

Note: If you are new to using IAR Embedded Workbench, we suggest that you first look at the user guides and tutorials which you can find in the IAR Information Center.

Project conversion

To migrate existing CS+ applications to IAR EWRL78 there is a tool called **Convert To IAR**. This is a GUI application included with IAR Embedded Workbench, available via the **Tools** menu.

The **Convert To IAR** tool converts CS+ project files into EWRL78 project files without changing the original project file. Information about source files, include paths, defined symbols and build configuration is transferred. As an option, also source code text substitutions are performed and you can add your own substitution rules including support for regular expressions.



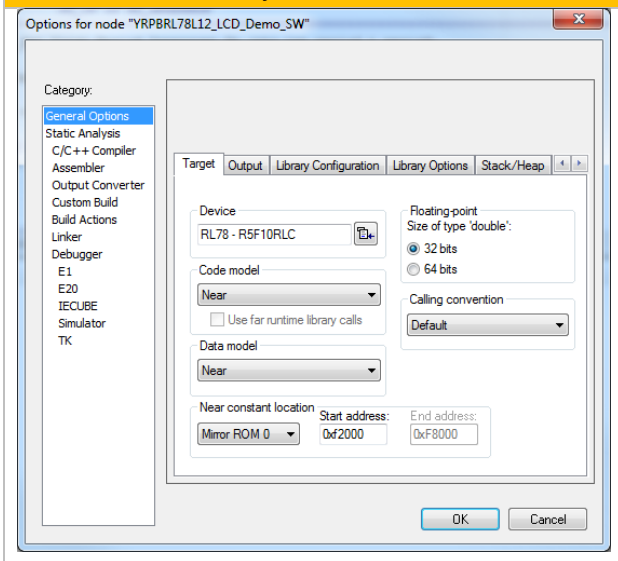
Procedure

1. Start EWRL78.
2. Start **Convert To IAR** available in the **Tools** menu.
3. Navigate to the CS+ project to convert by clicking the browse button.
4. Click the **Execute** button and a new EWRL78 project file will be created.
5. Add the new project to a EWRL78 workspace by choosing **Project>Add Existing Project…**.
6. Set the relevant project options by choosing **Project>Options…**.
   Hint: Open the original project in CS+, walk through the options and set the corresponding options in EWRL78 as suggested in the section *Important tool settings* below.

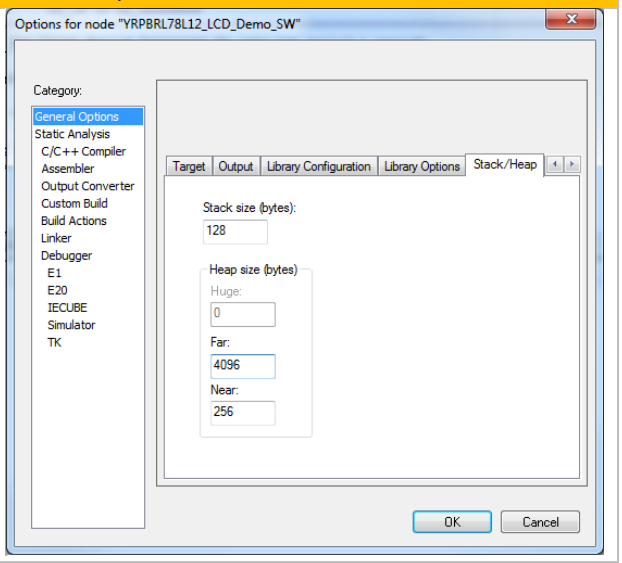
Important tool settings

To change project settings, choose **Project>Options...**. Below is an overview of the most important tool settings.

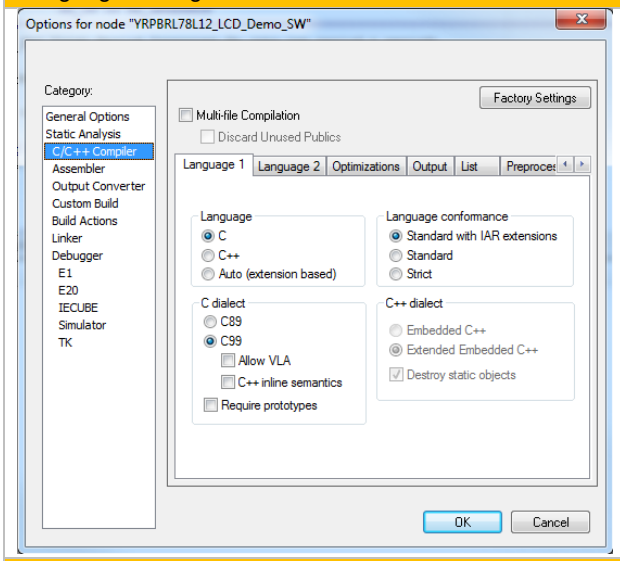
Device selection and Byte-order



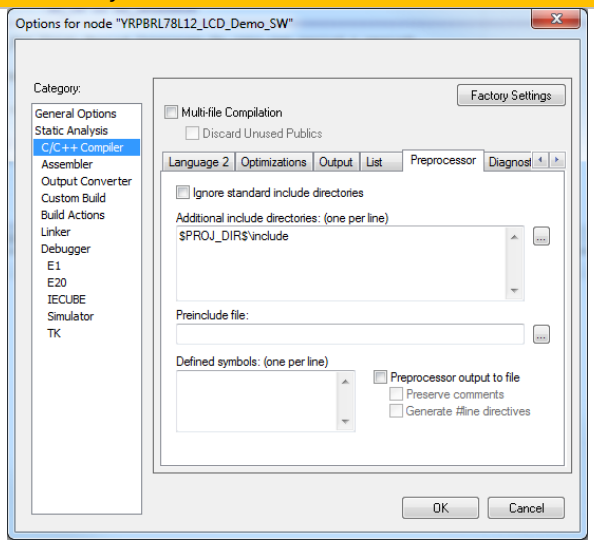
Stack/Heap size



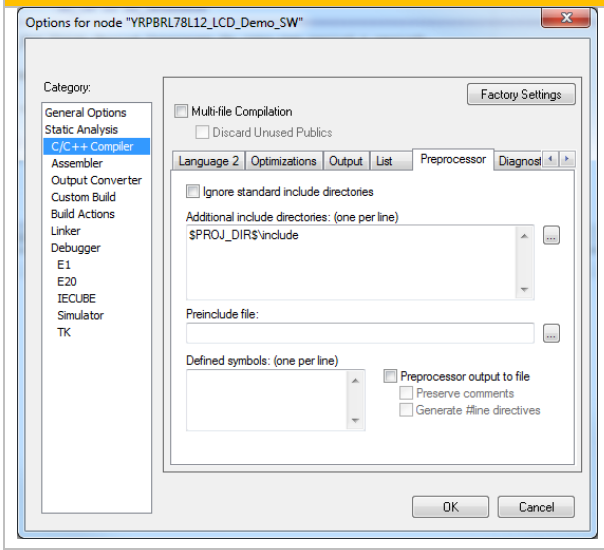
Language settings



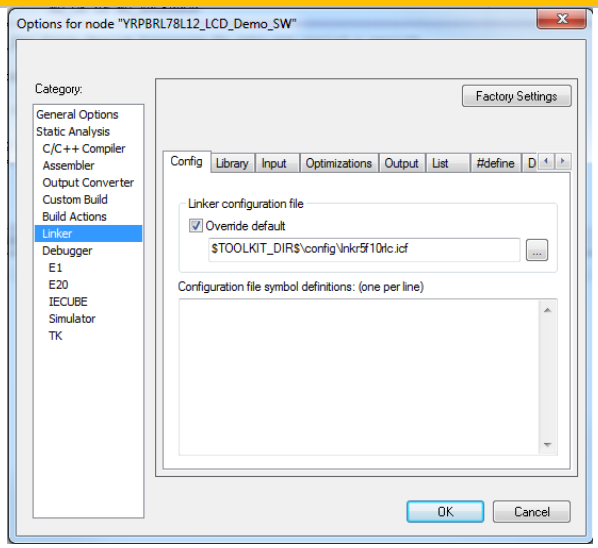
Defined symbols and include directories

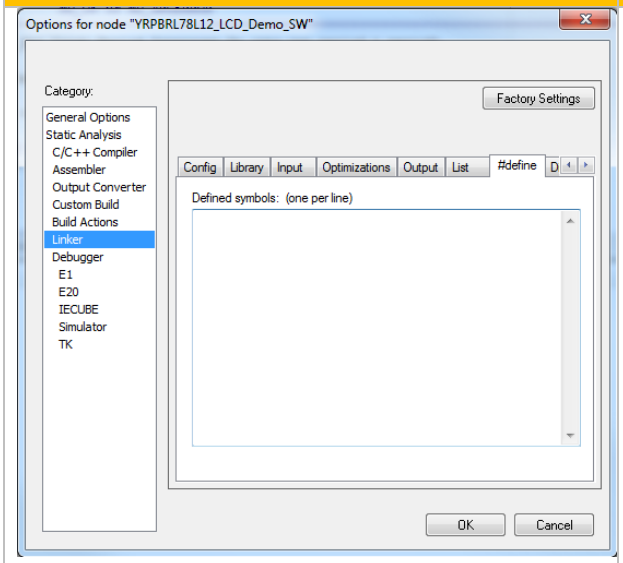
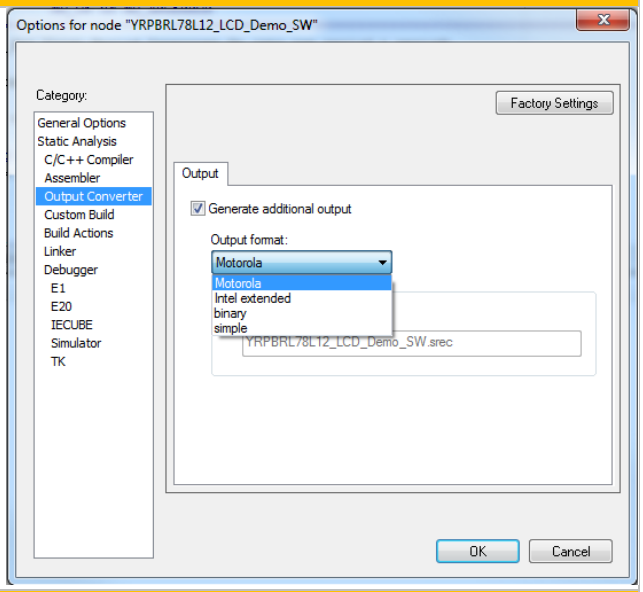


Include directories



Linker configuration file



Linker symbols	Additional output format
	

Note: We recommend that you verify all settings to make sure they match your project needs.

Building your project

After successfully converting the Renesas project and considered the basic code differences described above, you will still most likely need to fine-tune parts of the source code so that it follows the EWRL78 syntax.

1. Select your device under **Project>Options>General Options**.
2. Choose **Project>Make**.
3. To find the different errors/warnings, press **F4** (Next Error/Tag).
   This will bring you to the location in the source code that generated this error/warning.
4. For each error/warning, modify the source code to match the EWRL78 syntax.
   Note: See the **Reference information** section below for this step.
5. After correcting one or more errors/warnings, repeat the procedure.

Note: It is always a good idea to correct the first couple of errors/warnings in different source files first.
This is because errors and warnings later in the source code might just be effects of faulty syntax at the beginning of the source.

Reference information

Locate a feature in the left-hand column; then you can find the IAR Systems counterpart to the right. For detailed information about this feature specific to IAR Embedded Workbench®, see the relevant documentation. For a complete list of guides, see IAR Information Center in the IDE.

Compiler-specific details

CS+ (CA78K0R)	IAR Systems
Programming languages	
Assembler, ANSI C	Supported programming languages: assembler, C, Embedded C++, Extended Embedded C++, and C++.
Processor configuration	
• All RL78 devices(Core 0, 1 and 2 automatically selected by device) • Little endian • Bit order (in bit fields) left or right (option -rb)	`--core={s1|s2|s3}` `s1` Generates code for S1, the RL78 core with only one register bank and a multiplexed 8-bit bus. `s2` Generates code for S2, the core without instructions to support a hardware multiplier/divider. `s3` (default) Generates code for S3, the core with instructions to support a hardware multiplier/divider.
Memory models/Data models/Code models	
• Small model (-ms option) Data model: near (64KB address range) Code model: near (64KB address range) • Medium model (-mm option) Data model: near (64KB address range) Code model: far (1MB address range) • Large model (-ml option) Data model: far (1MB address range) Code model: far (1MB address range)	Supported code models (option `--code_model`): near (default): Function calls reach the first 64 Kbytes of memory. far: Function calls reach the entire 1 Mbyte memory.
	Supported data models (option `--data_model`): near (default): Data is by default placed in the highest 64 Kbytes of memory far: Data is by default placed in the entire 1 Mbyte of memory
The linker automatically selects appropriate libraries.	The linker automatically selects appropriate libraries.
Overriding default placement of given code/data model	

To override default placement of the selected code model, use any of these memory attributes:
`__far`
`__near`
 	
Example (function placement to the far area):
`__far void my_func(void)`
`{`
`}`
	To override default placement of the selected code model, use any of these memory attributes:
	`__callt`
	`__near_func` (default)
	`__far_func`

`__far`
`__near`
 	
Example (variable placement to the far area):
`__far int my_var;`
	To override default placement of the selected data model, use any of these memory attributes:
	`__near` (default) : The highest 64 Kbytes
	`__far` : The entire 1 Mbyte of memory
	
	For example:
	`__near int i = 3;`
	`__far unsigned u;`

Absolute placement of variables	

Use `__directmap` for absolute placement. Please note that this variables will be treated as static variables and they also cannot be initialized.
 	
`__directmap char c = 0xffe00 ;`
`__directmap struct x {`
`char a ;`
`char b ;`
`} c = { 0xffe00 } ;`
 	
or
	`__no_init char a @0x80;`
	
	or
	
	`#pragma location=0x80`
	`__no_init const int a;`

Memory model: SMALL/MEDIUM
`#pragma section @@DATA MY_DAT AT 0x0FFE00`
`volatile char c;`
 	
Memory model: LARGE
`#pragma section @@DATAL MY_DAT AT 0x0FFE00`
`volatile char c;`
 	
Please note, that the segment name length is max. 8 characters
 	
In case of constants:
Memory model: SMALL/MEDIUM
`#pragma section @@CNST MY_DAT AT 0x03000`
`const char my_const;`
 	
Memory model: LARGE
`#pragma section @@CNSTL MY_DAT AT 0x03000`
`const my_const;`
 	
Please note, that
• the segment name length is max. 8 characters
• Compiler option -s/-sa has to be used in order to allow segment switching. If this option is not used the #pragma has to be used at the beginning of the file which is valid for all data within the file.	

Absolute placement of functions	

Memory model: SMALL
 	
`#pragma section @@CODE MY_SEG AT  0x2400`
 	
or without address (will be defined in linker file)
 	
`#pragma section @@CODE MY_SEG`
 	
Memory model: MEDIUM/LARGE
 	
`#pragma section @@CODEL MY_SEG AT  0x2400`
 	
or without absolute address (will be defined in linker file)
 	
`#pragma section @@CODEL MY_SEG`
 	
Please note, that
• user has to define this #pragma at the beginning of the file before any C code.
• all functions within one file will be placed to the defined segment. Switching of segments within one file is not possible.
• segment name length is max. 8 characters
	`void f(void) @ "MyFunctions";`
	or
	`void f(void) @ "MyFunctions"`
	`{`
	`}`
	or
	`#pragma location="MyFunctions"`
	`void f(void);`

The section `MY_SEG` must be placed by customizing the linker configuration file in case the absolute address is not used within the #pragma.	The section `MyFunctions` must be placed by customizing the linker configuration file. See the compiler guide section *Customizing the linker configuration file*.
Constants in ROM	
`const unsigned short constants[] = {0x1234, 0x5678}`	`const unsigned short constants[] = {0x1234, 0x5678}`
Interrupt functions	

`#pragma interrupt INTWDTI r_wdt_interrupt`
 	
`__interrupt static void r_wdt_interrupt(void)`
`{`
`/* Do something here.*/`
`}`
	`#pragma vector = 0x17`
	`__interrupt void MyInterruptRoutine(void)`
	`{`
	`  /* Do something here.*/`
	`}`
	or
	`#pragma vector = UART1_R_RXNE_vector /*`

`#pragma vect INTWDTI r_wdt_interrupt`
 	
`__interrupt static void r_wdt_interrupt(void)`
`{`
`/* Do something here.*/`
`}`
 	
Please note, that the #pragma shall be declared at the beginning of the file.
	`Symbol from I/O header file */`
	`__interrupt void MyInterruptRoutine(void)`
	`{`
	`  /* Do something here. */`
	`}`
	Note that an interrupt function must have the return type `void`, and it cannot specify any parameters.

Inline assembler	

`__asm("NOP");`
 	
`OR`
 	
`#asm`
`...`
`NOP`
`...`
`#endasm`
	`asm["movw ax, sp"];`
	`asm["mov a, 0xff"];`

CS+ (CA78K0R)		IAR Systems
Sizes on integers and floating-point		
8 bits	char	8 bits
16 bits	int	16 bits
16 bits	short	16 bits
32 bits	float	32 bits
32 bits	long	32 bits
Not available	long long	32 bits
32 bits	double	32 bits (treated as `float`)
Extended keywords		
• `__callt` • `callt` (Only if non ANSI functions are allowed. See –za option.)	Call functions via callt table	`__callt`
• `__sreg` • `sreg` (Only if non ANSI functions are allowed. See –za option.)	Allocate variables in saddr area	`__saddr`
• `__boolean` • `boolean` (Only if non ANSI functions are allowed. See –za option)	Variables defined with this attribute will be placed within SADDR or SFR and are accessible via 1bit access.	-
• `bit` (Only if non ANSI functions are allowed. See –za option)	Variables defined with this attribute will be placed within SADDR or SFR and are accessible via 1bit access.	-
• `__interrupt`	Hardware interrupt	`__interrupt`
• `__interrupt_brk`	Software interrupt	
• `__asm`	Inline assembler	`asm, __asm`
• `__rtos_interrupt`	RTOS interrupt handlers. For RI78V4 RTOS.	-

• `__directmap`	Absolute placement of variables	`__no_init char a @0x80;`
		
		or
		
		`#pragma location=0x80`
		`__no_init const int a;`

• `__near`	data:0F0000H to 0FFFFFH code:000000H to 00FFFFH	`__near` `__near_func`

• `__far`	data:000000H to 0FFFFFH code:000000H to 0FFFFFH	`__far, __huge` `__far_func`
Pragma directives		
`#pragma sfr`	Use SFR names in C source files.	
`#pragma vector (or interrupt) <interrupt-request-name> <function-name>`	Write interrupt service routines in C. See above.	

`#pragma di`
`#pragma ei`
 	
`void main (void) {`
`  DI();`
`  EI();`
`}`
	Allows the usage of the intrinsic functions enable and disable interrupts in C.	`#include <intrinsics.h>`
		`void main (void){`
		`  __disable_interrupt();`
		`  __enable_interrupt();`
		`}`

`#pragma halt`
`#pragma stop`
`#pragma brk`
`#pragma nop`
 	
`void main (void) {`
`  HALT ();`
`  STOP ();`
`  BRK ();`
`  NOP ();`
`}`
	Allows the usage of the intrinsic functions halt, stop, brk and nop in C.	`#include <intrinsics.h>`
		
		`void main (void) {`
		`  __halt();`
		`  __stop();`
		`  __no_operation();`
		`  __break();`
		`}`

`#pragma section <compiler-output-section-name> <news-section-name> [AT startaddr]`	Switches sections	-
`#pragma name <module-name>`	Change the module name.	-
`#pragma rot`	Use the inline rotation functions. e.g. #pragma rot unsigned char rorb (x, y) ; unsigned char x ; unsigned char y ; Rotates x to right for y times unsigned int rorw (x, y) ; unsigned int x ; unsigned char y ; Rotates x to right for y times.	-
`#pragma div` unsigned int divuw (x, y) ; unsigned int x ; unsigned char y ; Performs unsigned division of x and y and returns the quotient. unsigned char moduw (x, y) ; unsigned int x ; unsigned char y ; Performs unsigned division of x and y and returns the remainder.	Use optimized division functions.	Some RL78 microcontrollers have a hardware multiplier/divider unit. To use the optimized division set **General Options >Library Configuration>Use Hardware Multiplier/Divider Unit**.
`#pragma mul` unsigned int mulu (x, y) ; unsigned char x ;	Use the inline multiplication function.	`See above.`

unsigned char y ; Performs unsigned multiplication of x and y. unsigned long muluw (x, y) ; unsigned int x ; unsigned int y ; Performs unsigned multiplication of x and y.		
`#pragma mac` e.g. unsigned long macuw (x, y, z) ; unsigned long x ; unsigned int y ; unsigned int z ; Performs unsigned sum-of-products calculation of x + (y * z) and returns the result. signed long macsw (x, y, z) ; signed long x ; signed int y ; signed int z ; Performs signed sum-of-products calculation of x + (y * z) and returns the result.	Use optimized sum-of-products calculation functions.	Some RL78 microcontrollers have a hardware multiplier/divider unit. To include runtime support for this unit, use the sample library code provided in the `rl78\src\lib\hw_multiply_division_units` directory. Use the files *hwmac.h* and *hwmac.s*.

`#pragma opc`	Insert data at the current code address.
	e.g.
	void main (void) {
	__OPC (0xa7) ;
	}
		Insert the opcode with inline assembler.
		`asm["opcode"];`

`#pragma rtos_interrupt`	Write RI78V4 (real-time OS) interrupt handlers in C.	-
`#pragma rtos_task`	Write RI78V4 (real-time OS) tasks in C.	-
`#pragma ext_func`	Call flash area functions by using a branch table from the boot area.	-
`#pragma inline`	Inline the standard library functions memcpy and memset in order to improve performance.	-
Intrinsic functions		
`HALT();`	Activate halt mode	`__halt();`
`STOP();`	Activate stop mode	`__stop();`
`BRK();`	Use software interrupt	`__break();`
`NOP();`	Add NOP instruction	`__no_operation();`
`EI();`	Enable interrupts	`__enable_interrupt();`
`DI()`	Disable interrupts	`__disable_interrupt();`
`Preprocessor symbols`		
`__LINE__`	Current source line number of the file being compiled	`__LINE__`
`__FILE__`	File name of the file being compiled	`__FILE__`
`__DATE__`	Date of compilation	`__DATE__`
`__TIME__`	Translation time of source file	`__TIME__`
`__STDC__`	Conformance to the ANSI	`__STDC__` `__STDC_VERSION__`

	standard	
__K0R_SMALL__ __K0R_MEDIUM__	Specifies which model is used.	__DATA_MODEL_NEAR__ __DATA_MODEL_FAR__ __CODE_MODEL_NEAR__ __CODE_MODEL_FAR__
__CHAR_UNSIGNED__	Char treated as unsigned. When the -qu option was specified	-
__RL78__	RL78 family is specified	-
__RL78_1__	Core 1 is selected	__S1__
__RL78_2__	Core 2 is selected	__S2__
__RL78_3__	Core 3 is selected	__S3__
__CA78K0R__	Compiler identification	__ICCRL78__
Compiler options		
-cdevice-type	Target device definition. Example: R5F100LE device -cf100le	Specfied with the linker file and device header files.
-o[output-file-name]	Specify object output file including path information	--output {filename|directory} -o { filename|directory }
-no	Specify not to output an object file. See above –o option	-
-rprocess-type process-type: • *a* = Performs indirect reference in 1-byte units. • *b* = Assigns a bit field from the most significant bit (MSB). • *d[n][m]* = assigns an external variable/external static variable (except for the const-type variable) automatically to the saddr area n = 1: char, unsigned char n = 2: char, unsigned char, short, unsigned short, int, unsigned int, enum, near pointer n = 4: char, unsigned char, short, unsigned short, int, unsigned int, enum, long, unsigned long, pointer *m* = Structure, union, array n/m not defined = All variables are assigned to saddr if n/m not defined • *s[n][m]* = Assigns a static auto variable automatically to the saddr area n = 1: char, unsigned char n = 2: char, unsigned char, short, unsigned short, int, unsigned int, enum, near pointer n = 4: char, unsigned char, short, unsigned short, int, unsigned int, enum, long,	Specification of program assignment to a memory	--code_model{near|n|far|f} --data_model{near|n|far|f}

unsigned long, pointer m = Structure, union, array n/m not defined = All variables are assigned to saddr if n/m not defined • *c* = Performs indirect reference in 1-byte units. Packs a structure and aligns the structure members to 1 byte. Since the compiler handles data within arrays as pointers, byte access is used when the -rc option is specified. • *f* = Assigns ROM data in the far area. • *n* = Assigns ROM data in the near area		
-nr The -nr option disables the -r option. Process types are interpreted as follows: a = Does not perform indirect reference in 1-byte units. b = Assigns a bit field from the least significant bit (LSB). d = Does not automatically assign any variable to the saddr area. s = Does not automatically assign any variable to the saddr area. c = Do not perform indirect reference in 1-byte units. Does not pack a structure and does not align the structure members to 1 byte.	Use default program assignment to the memory. See option –r.	-
-g*[n]* n = 1: Add debug information to the object module file only. No debug information is added to the assembler source file. n = 2: Adds debug information to the object module file and the assembler source module file.	Debug information can be added to the object files by using this option.	--debug -r
-n*g*	Disable adding of debug information to the object files. See –g option.	-
-p[output-file-name]	Specify preprocess list file	`--preprocess[=[c][n][l]] { filename|directory }`
-k[process-type] process-type: not specified = will be set as default –kfln	Specify contents of the preprocess list file. See option –p.	`-l[a|A|b|B|c|C|D][N][H] { filename|directory }`

c = Delete comments d = Expand definitions defined by #define f = Performs conditional compilations of #if, #ifdef, and #ifndef. i = Expands #include l = Performs #line processing n = Performs line number and paging processing.		
`-dmacro-name[=definition-name][,macro-name[=definition-name]] ...`	Define macro	`-D symbol[=value]`
`-umacro-name[,macro-name] ...`	Un-define macro	-
`-ifolder[,folder] ...`	Add search path for include files	`-I path`
`-a[output-file-name]`	Specify the output of assembler source file	`-la`
`-sa[output-file-name]`	C source code will be added as comment within the generated assembler source files. See option –a.	`-lA`
`-e[output-file-name]`	Specify error list file	`-l[c|C|D]`
`-se[output-file-name]`	C source code will be added to error list file. See option –a.	`-l[c|C|D]`
`-x[output-file-name]`	Specify the output of a cross reference list file	Information can be seen in the assembler output.
`-lw[number-of-characters]`	Specify the character counter per line for each list file.	-
`-ll[number-of-lines]`	Specify the number of lines per page for each list file.	-
`-lt[number-of-characters]`	Specify number of spaces to be used instead of tabulator.	Not using tabs.
`-lf`	Add form feed to the end of each list file.	-
`-li`	Add the C source code to the include files used by the assembler. See also –sa option	-
`-w``[level]` level: 0 = No warning messages are output. 1 = Normal warning messages are output. 2 = Detailed warning messages are output.	Specify whether to output the warning message to the console or not.	`--no_warnings`
`-v`	Output the execution state of the current compilation to the console	Default
`-nv`	Disable option –v.	`--silent`
`-f``file-name`	Use input file for passing options to the compiler.	`-f filename`
`-t``folder`	Define destination folder for	-

	temporary files.	
-ztype type: p = The characters after "//" before the line feed code are interpreted as a comment. c = Nesting of comments is permitted. s = Interprets the kanji code in comments as SJIS e = Interprets the kanji code in comments as EUC. n = Interprets comments as not containing kanji codes. b = char-/unsigned char-type argument and return value are not int-extended. a = Functions not in the ANSI standard are invalid. f = Outputs objects for flash. *taddress* = Specifies the start address of the flash area branch table. z*address* = Specifies the start address of the flash area. x = Outputs the object for the RAM allocation	Enables extended functions.	-
-nz	Disable option –z.	-
-yfolder	Define search path for device files	User includes the device file. Its either under INSTALL_DIR/rl78/inc or in a directory specified as an include path with `-I path`
-mtype type: s = small model m = medium model l = large model	Memory model specification	`--code_model` `--data_model`
-mi[MAA] MAA: 0 = 0 to FFFF→F0000 to FFFFF 1 = 10000 to 1FFFF→F0000 to FFFFF	Specify mirror area	`--near_const_location[RAM|ROM0|ROM1]` RAM: Constants are located in RAM, in the range `0xF0000–0xFFFFF` ROM0: Constants are located in ROM, in the range `0x00000–0x0FFFF`, and are mirrored by hardware to RAM, in the range `0xF0000–0xFFFFF` ROM1: Constants are located in ROM, in the range `0x10000–0x1FFFF`, and are mirrored by hardware to RAM, in the range `0xF0000–0xFFFFF`

`-common`	output common object file for 78K0R and RL78	-
`-ma``file-name[`**`-ma`**`file-name]` **`-ma`**`file-name[,file-name]`	Specify variables and functions by using an specific file	-
`--` **`-?`** **`-h`**	Help information on options for the command line.	Call iccrl78.exe with no input.

Assembler-specific details

CS+ (CA78K0R)	IAR Systems
Limitations in source code structure	
Interrupt functions in assembler	

To insert an entry in the interrupt vector table, define the destination with the `DW` directive, for example like this:
 	
`@@BASE CSEG BASE`
`_r_wdt_interrupt:`
`...`
`...`
`RETI`
 	
`@@VECT04 CSEG AT 0004H`
`DW _r_wdt_interrupt`
	Interrupt functions should be declared as `ROOT` so that they cannot be discarded by the linker even if no symbols in the segment are referred to. To insert an entry in the interrupt vector table, define the destination with the `DW` directive, for example like this:
	`COMMON INTVEC:CODE:ROOT(1)`
	`ORG 0x08 ;INTP0`
	`branchToInter0:`
	`DW inter0`

Code segments and data segments are defined by using the following assembler directive: CSEG: In internal or external ROM address area DSEG: In internal or external RAM address area BSEG: In internal RAM saddr area	Use the section control directive SECTION alias RSEG to place your code and data in sections. A section is *relocatable*.

`[segment-name] CSEG [relocation-attribute]`
 	
Possible relocation attributes:
• CALLT0
Place segments to the CALLT area of the device
Default segment: ?CSEGT0
• `FIXED`
Place segment within the range 0x000C0 to 0x0FFFF
Default segment: ?CSEGFX
• `BASE`
Place segment within the range 0x000C0 to 0x0FFFF
Default segment: ?CSEGB
• `AT absolute-expression`
Place the segment to an absolute segment
Default segment: -
• `UNIT`
Place the segment on odd or even address within the address range 0x000C0 to 0xEFFFF
Default segment: ?CSEG
• `UNITP`
Place the segment on even address within the address range 0x000C0 to 0xEFFFF
Default segment: ?CSEGUP
• `IXRAM`
Place the segment on odd or even address within the address range 0x000C0 to 0xEFFFF
Default segment: ?CSEGIX
• `SECUR_ID`
	`SECTION` *`section`* `[:`*`type`*`] [:`*`flag`*`] [(`*`align`*`)]`
	
	- *`align`*
	The power of two to which the address should be aligned. The default align value is `0`.
	
	- *`flag`*
	
	`ROOT`, `NOROOT`
	`ROOT` (the default mode) indicates that the section fragment must not be discarded.
	`NOROOT` means that the section fragment is discarded by the linker if no symbols in this section fragment are referred to. Normally, all section fragments except startup code and interrupt vectors should set this flag.
	`REORDER`, `NOREORDER` `NOREORDER` (the default mode) starts a new fragment in the section with the given name, or a new section if no such section exists.
	`REORDER` starts a new section with the given name.
	- *`section`*
	The name of the section. The section name is a user-defined symbol.
	- *`type`*
	The memory type, which can be either `CODE`, `CONST`, or `DATA`.
	- *`value`*
	Byte value used for padding, default is zero.
	- *`type-expr`*
	A constant expression identifying the ELF type of the section.

Place the segment within the security id address range 000C4H to 000CDH.
Default segment: ?CSEGSI

- `PAGE64KP`
  Place a segment within a 64KB page. Same named segment used in different files will not be combined.
  Default segment: ?CSEGP64
- `UNIT64KP`
  Place a segment within a 64KB page. Same named segment will be combined.
  Default segment: ?CSEGU64
- `MIRRORP`
  Place the segment to the mirror area.
  Default segment: ?CSEGMIP
- `OPT_BYTE`
  Place the segment within the option byte address range 000C0H to 000C3H.
  Default segment: ?CSEGOB0

```
[segment-name]  DSEG [relocation-attribute]
```

Possible relocation attributes:

- `SADDR`
  Place segment to the saddr area of the device 0x0FFE20 to 0xFFEFF
  Default segment: ?DSEGS
- `SADDRP`
  Place segmentsto the saddr area of the device 0x0FFE20 to 0xFFEFF on even address
  Default segment: ?DSEGSP
- `AT absolute-expression`
  Place the segment to an absolute segment
  Default segment: -
- `UNIT`
  Place the segment on odd or even address within the `RAM`
  Default segment: ?DSEG
- `UNITP`
  Place the segment on even address within the RAM
  Default segment: ?DSEGUP
- `BASEP`
  Place the segment on even address within the RAM (except saddr area)
  Default segment: ?DSEGBP
- `PAGE64KP`
  Place a segment within a 64KB page in RAM. Same named segment used in different files will not be combined.
  Default segment: ?DSEGP64
- `UNIT64KP`
  Place a segment within a 64KB page in RAM. Same named segment will be combined.
  Default segment: ?DSEGU64

```
[segment-name]  BSEG [relocation-attribute]
```

Possible relocation attributes:

- `AT absolute-expression`
  Place the segment to an absolute segment

- *`flags-expr`*
  A constant expression identifying the ELF flags of the section.

The section name can be referenced in the linker file in order to place the section at a specific address otherwise it will get a default placement.
Ex:
Assembler file:

```
SECTION my_section:CODE:NOROOT(2)
__start
;code...
```

Link file:

```
define symbol start_of_my_section = 0x0160;
place at address mem:start_of_my_section{
readonly section my_section};
```

Default segment: - • UNIT Place the segment on odd or even address within the RAM (0xFFE20 to 0xFFEFF) Default segment: ?BSEG		
Bit segments can be defined by using the BSEG assembler directive. See above.		Bit segments cannot be defined explicitly, but can easily be defined using bit operators in code or data segments. As a byte is the smallest allocable memory segment, no memory is lost or gained using either tool.
Binary representation		
		Not supported, should be replaced by `0x0f`.
CS+ (CA78K0R)		**IAR Systems**
Integer constants		
`1010B, 1010Y`	Binary	`1010b, b'1010`
`1234O, 1234Q`	Octal	`1234q, q'1234, 01234`
`1234, -1, 1234D, 1234T`	Decimal	`1234, -1, d'1234, 1234d`
`8FFFH, 0FFFFH`	Hexadecimal	`0FFFFh, 0xFFFF, h'FFFF`
Operand modifiers in assembler		
`+`	Addition of values of first and second terms e.g. `BR !$ + 6`	`+`
`-`	Subtraction of value of first and second terms e.g. `BACK : BR BACK - 6H`	`-`

`*`	Multiplication of value of first and second terms.
	
	e.g.
	`TEN EQU 10H`
	`MOV A, #TEN * 3`
		`*`

`/`	Divides the value of the 1st term of an expression by the value of its 2nd term and returns the integer part of the result. `MOV A, #256 / 50`	`/`
`MOD`	Obtains the remainder in the result of dividing the value of the 1st term of an expression by the value of its 2nd term. e.g. `MOV A, #256 MOD 50`	`MOD` `%`
`+sign`	Returns the value of the term as it is. e.g. `FIVE EQU +5`	`+sign`
`-sign`	The term value 2 complement is sought. e.g. `NO EQU -1`	`-sign`
`NOT`	Obtains the logical negation (NOT) by each bit. e.g. `MOVW AX, #LOWW ( NOT 3H )`	`BINNOT` `~`

AND	Obtains the logical AND operation for each bit of the first and second term values. e.g. `MOV A, #6FAH AND 0FH`	BINAND &
OR	Obtains the logical OR operation for each bit of the first and second term values. e.g. `MOV A, #0AH OR 1101B`	BINOR |
XOR	Obtains the exclusive OR operation for each bit of the first and second term values. e.g. `MOV A, #9AH XOR 9DH`	BINXOR ^

EQ (=)	Compares whether values of first term and second term are equivalent. Return true (0xFF) if equal and false (0x00) if not.
	
	e.g.
	`A1 EQU 12C4H`
	`A2 EQU 12C0H`
	
	`MOV A, #A1 EQ ( A2 + 4H )`
		EQ
		=
		==

NE(<>)	Compares whether values of first term and second term are not equivalent. Return true (0xFF) if not equal and false (0x00) if equal.
	
	e.g.
	`A1 EQU 12C4H`
	`A2 EQU 12C0H`
	
	`MOV A, #A1 EN A2`
		NE
		<>
		!=

GT(>)	Compares whether value of first term is greater than value of the second. Return true (0xFF) if value of first operand is greater than the second one and false (0x00) if not.
	
	e.g.
	`A1 EQU 12C4H`
	`A2 EQU 12C0H`
	
	`MOV A, #A1 GT A2`
		GT
		>

GE(>=)	Compares whether value of first term is greater than or equivalent to the value of the second term. Return true (0xFF) if value of first operand is greateror equal than the second one and false (0x00) if not.
	
	e.g.
	`A1 EQU 12C4H`
	`A2 EQU 12C0H`
	
	`MOV A, #A1 GE A2`
		GE
		>=

LT(<)	Compares whether value of first term is	LT

	smaller than value of the second. Return true (0xFF) if value of first operand is less than the second one and false (0x00) if not.
	
	e.g.
	`A1 EQU 12C4H`
	`A2 EQU 12C0H`
	
	`MOV A, #A1 LT A2`
		`<`

`LE(<=)`	Compares whether value of first term is smaller than or equivalent to the value of the second term. Return true (0xFF) if value of first operand is less than or equal the second one and false (0x00) if not.
	
	e.g.
	`A1 EQU 12C4H`
	`A2 EQU 12C0H`
	
	`MOV A, #A1 LE A2`
		`LE`
		`<=`

`SHR`	Shift right. e.g. `MOV A, #01AFH SHR 5`	`SHR` `>>`
`SHL`	Shift left. e.g. `MOV A, #21H SHL 2`	`SHL` `<<`
`HIGH`	Returns the high-order 8-bit value of a term. e.g. `MOV A, #HIGH 1234H`	`HIGH`
`LOW`	Returns the low-order 8-bit value of a term. e.g. `MOV A, #LOW 1234H`	`LOW`

`HIGHW`	Returns the high-order 16-bit value of a term.
	
	e.g.
	`MOVW AX, #HIGHW 12345678H`
	`MOV ES, #HIGHW LAB`
	`MOVW AX, ES:!LAB`
		`HWRD`

`LOWW`	Returns the low-order 16-bit value of a term. e.g. `MOVW AX, #LOWW 12345678H`	`LWRD`
`MIRHW`	Obtains the 16 higher-order bits of an address in the mirror destination area specified as the operand in the mirror source area. e.g. `MOVW RP0, #MIRLW PM0`	-
`MIRLW`	Obtains the 16 lower-order bits of an address in the mirror destination area specified as the operand in the mirror	-

	source area. e.g. `MOVW RP0, #MIRLW PM0`	

`DATAPOS`	Obtains the address part of a bit symbol.
	
	e.g.
	`SYM EQU 0FE68H.6`
	`MOV A, !DATAPOS SYM`
	
	`; return value 0FE68H`
		-

`BITPOS`	Obtains the bit part of a bit symbol.
	
	e.g.
	`SYM EQU 0FE68H.6`
	`CLR1 [HL].BITPOS SYM`
		-

`MASK`	Obtains a 16-bit value in which the specified bit positions are 1 and all others are 0. e.g. `MOVW AX,#MASK (0,3,0FE00H.7,15)`	-
`()`	Prioritizes the calculation within ( ) e.g. `MOV A, #(4+3)*2`	`( )`
Assembler directives		
`CSEG`	Code segment placement. See above.	`RSEG MY_SECTION:CODE` `SECTION MY_SECTION:CODE`
`DSEG`	Data segment placement. See above.	`RSEG MY_SECTION:DATA` `SECTION MY_SECTION:DATA`
`BSEG`	Bit segment placement. See above.	-
`EQU` `name EQU expression`	Defines a symbol with numerical data	`EQU`
`SET` `name SET expression`	Defines a symbol with numerical data. Bit symbol cannot be defined	`SET` `VAR`

`DB`
`label: DB size`
`label: DB initial-value`
	Initialize byte area	`DS`
		`DS8`

`DW`
`label: DW size`
`label: DW initial-value`
	Initialize word area	`DS16`

`DG`
`label: DG size`
`label: DG initial-value`
	Initialization of 20 bit area in 32 bits (4 bytes)	-

`DS`
`label: DS absolute-expression`
	Reserve number of bytes specified by the operand.	`DS expr [,expr]...`
		`Ex:`
		`buffer DS 25`

`DBIT` `[name] DBIT`	Reserve one bit of memory area in bit segment	-

`EXTRN`
`[label:] EXTRN symbol-name[,]`
`[label:] EXTRN BASE(symbol-name[,...])`
	External symbol definition. Meaning of the BASE attribute is that the symbol is located within 64KB area (0x0 to 0xFFFF)	`EXTERN symbol [,symbol]...`

`EXTBIT` `[label:] EXTBIT bit-symbol-name[,...]`	External bit definition.	-
`PUBLIC` `[label:] PUBLIC symbol-name[,...]`	Define symbol to be referenced by other module.	`PUBLIC symbol [,symbol]...` `PUBWEAK symbol [,symbol]...`

NAME [label:] NAME object-module-name[,...]	Define object module name.	MODULE symbol PROGRAM symbol NAME symbol
BR [label:] BR expression	Tells the assembler to automatically select a 2-, 3-, or 4-byte BR branch instruction according to the value range of the expression specified in the operand field.	-
CALL [label:] CALL expression	Tells the assembler to automatically select a 3- or 4-byte CALL branch instruction according to the value range of the expression specified in the operand field.	-
MACRO [macro-name:] MACRO formal-parameter[,...]	Executes a macro definition by assigning the macro name specified in the symbol field to a series of statements described between MACRO directive and the ENDM directive.	name MACRO [argument] [,argument]... ENDMAC
LOCAL LOCAL symbol-name	Define symbol which is valid within a macro body only.	LOCAL symbol [,symbol] ...
REPT [label:] REPT absolute-expression	Tells the assembler to repeatedly expand a series of statements described between this directive and the ENDM directive the number of times equivalent to the value of the expression specified in the operand field.	REPT *expr* REPTC *formal,actual* REPTI *formal,actual* [,*actual*] ...
IRP [label:] IRP formal-parameter, [actual-parameter]	Tells the assembler to repeatedly expand a series of statements described between IRP directive and the ENDM directive the number of times equivalent to the number of actual parameters while replacing the formal parameter with the actual parameters (from the left, the order) specified in the operand field.	-
EXITM [label:] EXITM	Forcibly terminates the expansion of the macro body defined by the MACRO directive and the repetition by the REPT-ENDM or IRP-ENDM block.	ENDR EXITM
ENDM	Instructs the assembler to terminate the execution of a series of statements defined as the functions of the macro.	ENDM
END	Declares termination of the source module	END
$PROCESSOR	Specifies in a source module file the assemble target type. e.g. $PROCESSOR (f1166a0)	-
$DEBUG	Adds local symbol information in the object module file.	compiler option --debug -r
$NODEBUG	Does not add local symbol information in the object module file.	-
$DEBUGA	Adds assembler source debug information in the object module file.	compiler option --debug -r
$NODEBUGA	Does not add assembler source debug	-

	information in the object module file.	
`$XREF`	Outputs a cross-reference list to an assemble list file.	`LSTXRF+`
`$NOXREF`	Does not output a cross-reference list to an assemble list file.	`LSTXRF-`
`$SYMLIST`	Outputs a symbol list to a list file	`LSTOUT+`
`$NOSYMLIST`	Does not output a symbol list to a list file.	`LSTOUT-`
`$INCLUDE(filename)`	Include a file.	`#include {"`*`filename`*`" | <`*`filename`*`>}`
`$EJECT`	Indicates an assembly list page break.	-
`$LIST`	Indicates starting location of output of assembly list.	-
`$NOLIST`	Indicates stop location of output of assembly list.	-
`$GEN`	Outputs macro definition lines, reference line and also macro-expanded lines to assembly list.	`LSTMAC+`
`$NOGEN`	Does not output macro definition lines, reference line and also macro-expanded lines to assembly list.	`LSTMAC-`

`$COND`	Outputs approved and failed sections of the conditional assemble to the assembly list.	Lists only the source code within positive condition blocks:
		`LSTCND+`
		Lists all source code:
		`LSTCND-`

`$NOCOND`	Does not output approved and failed sections of the conditional assemble to the assembly list.	-
`$TITLE('title-string')`	Prints character strings in the TITLE column at each page header of an assembly list, symbol table list, or cross-reference list.	-
`$SUBTITLE('title-string')`	Prints character strings in the SUBTITLE column at header of an assembly list.	-
`$FORMFEED`	Outputs form feed at the end of a list file.	-
`$NOFORMFEED`	Does not output form feed at the end of a list file.	-
`$WIDTH`	Specifies the maximum number of characters for one line of a list file.	-
`$LENGTH`	Specifies the number of lines for 1 page of a list file	-
`$TAB`	Specifies the number of characters for list file tabs.	-

`$IF(switch-name)`
`...`
`$ELSEIF (switch-name)`
`...`
`$ELSE`
`...`
`$ENDIF`
	Sets conditions in order to limit the assembly target source statements.	-

`$_IF conditional-expression`
`...`
`$_ELSEIF conditional-expression`
`...`
`$ELSE`
`...`
`$ENDIF`
	Sets conditions in order to limit the assembly target source statements. The IF and ELSEIF control instructions are used for true/false condition judgment with switch name(s), whereas the _IF and _ELSEIF control instructions are used for true/false condition judgment with a conditional expression. See	`#if cond`
		`...`
		`#elif cond`
		`...`
		`#else`
		`...`
		`#endif`

	above.	
`$SET(switch-name)`	Sets value 0xFF for switch name specified by IF/ELSEIF control instruction.	-
`$RESET(switch-name)`	Sets value 0x00 for switch name specified by IF/ELSEIF control instruction.	-
`$KANJICODE kanji-code`	Interprets Kanji character code for specified Kanji characters described in the comment.	-

`$RAM_ALLOCATE (segment-name)`	Allocate the segment with the specified segment name to the memory area name "RAM".	`RSEG MY_SECTION:DATA`
		`SECTION MY_SECTION:DATA`
		or in the linker file.

Assembler options		
-c`device-type`	The -c option specifies the target device for performing assembly	Can only specify core. `--core={s1|s2|s3}`
-o`[output-file-name]`	Specifies the output of an object module file	`--output {`*`filename|directory`*`}` `-o {`*`filename|directory`*`}`
-no	Disables the –o, -j, -g and –ga option	-
-j	The -j option specifies that the object module file can be output even if a fatal error occurs	-
-nj	Disables the –j option	-
-g	The -g option specifies that debug information (local symbol information) is to be added to an object module file	`--debug` `-r`
-ng	Disables –g option	-
-ga	The -ga option specifies that assembler source debug information is to be added to an object module file.	`--debug` `-r`
-nga	The -nga option disables the -g and -ga option	-
-i`path-name`	Include files path definition	`-Ipath`

-p`[output-file-name]`	The -p option specifies the output of an assemble list file.	`-l[a][d][e][m][o][x][N][H]`
		`{`*`filename|directory`*`}`
		
		• `a` Assembled lines only.
		• `d` The `LSTOUT` directive controls if lines are written to the list file or not. Using `-ld` turns the start value for this to off.
		• `e` No macro expansions.
		• `m` Macro definitions.
		• `o` Multiline code.
		• `x` Includes cross-references.
		• `N` Do not include diagnostics.
		• `H` Includes header file source lines.
		• *`filename`* The output is stored in the specified file.
		• *`directory`* The output is stored in a file (filename extension `i`) which is stored in the specified directory.

-np	The -np option disables the -p, -ka, -ks, -kx, -lw, -ll, -lh, -lt, and -lf option	-
-ka	The -ka option outputs an assemble list into an assemble list file.	See above.
-nka	The -nka option disables the -ka option.	-
-ks	The -ks option outputs a symbol list	See above.

	followed by an assemble list into an assemble list file.	
-nks	The -nks option disables the -ks option.	-
-kx	The -kx option outputs a cross reference list followed by an assemble list into an assemble list file.	`-lx`
-nkx	The -nka option disables the -kx option.	-
-lw[number-of-characters]	The -lw option specifies the number of characters per line in a list file	-
-ll[number-of-lines]	The -ll option specifies the number of lines per page in an assemble list file.	-
-lhcharacter-string	The -lh option specifies the character string printed in the title column of the header of an assemble list file	-
-lt[number-of-characters]	Spaces used for tabulator.	Always spaces.
-lf	The -lf option inserts a form feed (FF) code at the end of an assemble list file	-
-nlf	The -nlf option disables the -lf option	-
-e[output-file-name]	The -e option specifies the output of an error list file.	Part of the assembler list file.
-ne	The -ne option disables the -e option.	-
-ffile-name	File with options to be used from command-line.	`-f filename`
-tpath-name	Folder for temporary files.	-
-zs **-ze** **-zn**	Allow comments with following codes: -zs: Shift-JIS code -ze: EUC code -zn: Not interpreted as kanji	-
-ypath-name	Device file path	-
-dsymbol-name[=value][,symbol-name[=value] ...]	The -d option defines symbols.	`-D`*`symbol`*`[=`*`value`*`]`
-common	Generate common object file for RL78 and 78K0R	-
-mirchk	The -mirchk option checks the range of the address for a label in the mirror area. (From v1.60)	-
--	The -- option outputs a help message on the display.	Call iasmrl78.exe with no input.

Linker and library details

CS+ (CA78K0R)	IAR Systems
Device-specific header files	
All SFRs are accessible by adding the “#pragma sfr” to the file.	All SFRs are defined in `ioxxx.h` files.

CS+ (CA78K0R)		IAR Systems
Linker options		
-o[output-file-name]	Define the debug file.	`--output/-o {filename/directory}`
-no	The -no option disables the -o, -j, and -g option.	-
-j	Generate debug file even if a fatal error occurs.	`--force_ouput`
-nj	The -nj option disables the -j option	-
-g	The -g option specifies that debug information (local symbol information) is to be added to a load module file	Debug information is included by default and removed by `-strip`. Debug information with terminal: `--debug_lib`
-ng	The -ng option disables the -g, -kp,	`--strip`

	and -kl option.	
-s[area-name]	The -s option generates the stack decision public symbols "_@STBEG" and "_@STEND".	Specified in the linker file.
-ns	The -ns option disables the -s option.	-
-dfile-name	The -d option specifies that the specified file is to be input as a link directive file.	--config *filename*
-p[output-file-name]	The -p option specifies the output of a link list file. It also specifies the location to which it is output and the file name.	--log *topic*[,*topic*,...] --log_file *filename*
-np	The -np option disables the -p, -km, -kd, -kp, -kl, -ll, and -lf option.	-
-km	The -km option outputs a map list into a link list file.	--map {*filename*|*directory*} Use this option to produce a linker memory map file. The map file has the default filename extension map.
-nkm	The -nkm option disables the -kd and -km option.	-
-kd	The -kd option outputs a link directive into a link list file.	-
-nkd	The -nkd option disables the -kd option.	-
-kp	The -kp option outputs a public symbol list into a link list file.	-log sections
-nkp	The -nkp option disables the -kp option.	-
-kl	The -kl option outputs a local symbol list into a link list file.	-log sections
-nkl	The -nkl option disables the -kl option.	-
-ll[number-of-lines]	The -ll option specifies the number of lines per page in a link list file	-
-lf	The -lf option inserts a form feed (FF) code at the end of a link list file.	-
-nlf	The -nlf option disables the -lf option.	-
-e[file-name]	The -e option specifies the output of an error list file. It also specifies the location to which it is output and the file name	-
-ne	The -ne option disables the -e option.	-
-bfile-name	The -b option specifies that the specified file is to be input as a library file.	No separate option. Enter library names separated with blanks.
-ipath-name[,path-name] ...	The -i option specifies that a library file is to be input from the specified path.	No separate option. Enter library names separated with blanks.
-ffile-name	Command line options to be passed via file	-f filename
-tpath-name	The -t option specifies a path in which a temporary file is created.	-
-ypath-name	Device file path	-
-w[level] level: 0: No warning message is output. 1: A normal warning message is	Define warning level.	--no_warnings --warnings_affect_exit_code --warnings_are_errors

output. 2: A detailed warning message is output.		
-zbaddress	Define flash start address in case e.g. a bootloader is used. e.g. bootloader located in flash from address 0x0000 to 0x1FFFF Application could use the option –z2000h in order to start from the address 0x2000	Done in the linker. Can only half way be done with option: `--place_holder` *`symbol`*`[,`*`size`*`[,`*`section`*`[,`*`alignment`*`]]]` *`symbol`* The name of the symbol to create *`size`* Size in ROM; by default 4 bytes *`section`* Section name to use; by default `.text` *`alignment`* Alignment of section; by default `1`
-gocontrol-value, start-address[,size] control-value = value of option byte C3 start-address = OCD monitor start address size = OCD monitor size	Configure on-chip debugging.	-
-gisecurity-id	Security ID specification.	Defined as section `.security_id` in the linker.
-gbuser-option-byte-value	Definition of user option byte 0xC0 to 0xC2	Defined as section `.option_byte` in the linker.
-mi[MAA] MAA: 0 = 0 to FFFF→F0000 to FFFFF 1 = 10000 to 1FFFF→F0000 to FFFFF	Specify mirror area	Done at the compiler level.
-ccza	Use the -ccza option to specify whether to allocate a segment to the last byte of each 64 KB boundary area.	-
-nccza	Disable option -ccza	-
-self/-selfw	The -self and -selfw options specify whether to restrict allocation to the self RAM area. Self RAM is used by code/data flash programming libraries.	-
-ocdtr/-ocdtrw	Use the -ocdtr and -ocdtrw options to specify whether to restrict allocation to the trace RAM area. (From v1.40)	-
-ocdhpi/-ocdhpiw	The -ocdhpi and -ocdhpiw options specify whether to restrict allocation to the hot plug-in RAM area (From v1.40)	-
-rcaddress	Use the -rc option to specify the address that the copy routine for expanding ROMized segments in RAM area is allocated.	-
-rastart-address,end-address	The -ra option specifies the ROMization target area.	-
-rrmstart-address	The -rrm option specifies whether to reserve the work area for the RRM/DMM function.	-

--	The -- option outputs a help message on the display.	Call ilinkrl78.exe with no input.
Segments/Sections		
`@@CODE`	Segment for code portion (allocated to near area)	`.text`
`@@CODEL`	Segment for code portion (allocated to far area)	`.textf`
`@@CODER`	Segment for code portion (allocated to RAM)	Must create a user defined section and in the linker file use: `initialize by copy { MY_SECTION };`
`@@LCODE`	Segment for library code (allocated to near area)	`.text`
`@@LCODEL`	Segment for library code (allocated to far area)	`.textf`
`@@LCODER`	Segment for library code portion (allocated to RAM)	-
`@@CNST`	ROM data (allocated to near area within mirror)	`.const`
`@@CNSTR`	Segment for ROM data portion (allocated to RAM) (allocated to near area within mirror).	`.data_init`
`@@CNSTL`	ROM data (allocated to far area)	`.constf`
`@@CNSTLR`	Segment for ROM data portion (allocated to RAM) (allocated to far area)	`.dataf_init`
`@@R_INIT`	Segment for near initialized data (with initial value)	`.data`
`@@RLINIT`	Segment for far initialized data (with initial value)	`.dataf`
`@@R_INIS`	Segment for initialized data (sreg variable with initial value)	-
`@@CALT`	Segment for callt function table	`.callt0`
`@@VECT`*nn* The value of *nn* changes depending on the interrupt types	Segment for vector table	`.intvec`
`@@BASE`	Segment for callt function and interrupt function	`.text`
`@@LBASE`	Segment for library and callt function	`.text`
`@@INIT`	Segment for data area (with initial value, allocated to near area)	`.data`
`@@INITL`	Segment for data area (with initial value, allocated to far area)	`.dataf`
`@@DATA`	Segment for data area (without initial value, allocated to near area)	`.bss`
`@@DATAL`	Segment for data area (without initial value, allocated to far area)	`.bssf`
`@@INIS`	Segment for data area (sreg variable with initial value)	-
`@@DATS`	Segment for data area (sreg variable without initial value)	-
`@@BITS`	Segment for boolean type and bit type variables	-

Runtime environment

CS+ (CA78K0R)	IAR Systems
Calling convention	

Parameters passed on the stack	
The second and following arguments are passed to functions on the stack.	• Local variables and parameters not stored in registers • Temporary results of expressions • The return value of a function (unless it is passed in registers) • Processor state during interrupts • Processor registers that should be restored before the function returns (callee-save • registers).
Parameters passed in registers	
`AX`	8-bit values in: `A`, `B`, `C`, `X`, `D`, `E`
`AX`	16-bit values in: `AX`, `BC`, `DE`
`AX, BC`	24-bit values in: Stack
`AX, BC`	32-bit values in: `BC:AX`
`AX, BC`	Floating-point values in: `BC:AX`
Return values	
`CY`	1-bit
`BC`	8-bit values in: `A`
`BC`	16-bit values in: `AX`
`BC` (lower), `DE` (upper)	24-bit values in: `A:HL`
`BC` (lower), `DE` (upper)	32-bit values in: `BC:AX`
`BC` (lower), `DE` (upper)	Floating-point values in: `BC:AX`
Preserved registers	
`HL`	`BC` and `DE`
Scratch registers	
`AX, BC, DE, ES, CS`	-The registers `AX`, `HL`, `CS` and `ES`. -Registers that are used as register parameters and for returning values by a function.
System startup and exit code	
The system startup code is provided as a pre-compiled library and is automatically included within the project. However, the user has the possibility to include the ready-made `cstart.asm` file and adapt it according to the needs. Usually the user will use the standard library and just implement the function hdwinit which will be automatically called by the cstart library. Here the user can configure the hardware before the application runs to the main function.	The system startup code is located in the ready-made `cstartup.s` file. In addition, you specify additional settings, for example for the stack and heap size. It is likely that you need to customize the code for system initialization. For example, your application need to initialize memory-mapped special function registers, or omit the default initialization of data segments performed by cstartup. You can do this by providing a customized version of the routine `__low_level_init`, which is called from cstartup before the data segments are initialized. Modifying cstartup directly should be avoided.
Global variable initialization	
Static and global variables are initialized: zero-initialized variables are cleared and the values of other initialized variables are copied from ROM to RAM memory. This initialization will be done by the so called ROMization process.	Static and global variables are initialized: zero-initialized variables are cleared and the values of other initialized variables are copied from ROM to RAM memory. This initialization can be overrided by returning 0 from the `__low_level_init` function. Variables declared `__no_init` which are not initialized at all: `__no_init int i;`
Reentrancy and recursive functions	
Most of the standard library functions are reentrant. Please check the documentation for details.	The compiler is always reentrant when using the DLIB library.
Other operations	