

RL-ST6

Linker for ST6



Reference manual

Revision 01/2000

RAISONANCE

SOFTWARE LICENSE FOR RL-ST6

RAISONANCE grants to the CUSTOMER a license to use the RL-ST6 software (herein referred to as the SOFTWARE) subject to the following provisions :

- The SOFTWARE can be only used on one computer. The CUSTOMER may use the SOFTWARE on as many computers as purchased licenses. CUSTOMER may use the SOFTWARE on a multi-user or network system if one copy of the software is purchased for each node or terminal on which the SOFTWARE is to be simultaneously used.

- The SOFTWARE is copyrighted by and shall remain the property of RAISONANCE. The CUSTOMER is permitted to make a unique copy of the SOFTWARE only for backup purpose.

REFERENCE MANUAL: No part of this document may be copied or reproduced in any form or by any means without the prior written consent of RAISONANCE S.A. except for personal use by the CUSTOMER. The information contained herein is subject to change. RAISONANCE S.A. assumes no responsibility for the information contained in this document.

Copyright © 1998 RAISONANCE S.A. All right reserved

Contents

1. INTRODUCTION	5
1.1 What is the Linker used for ?	7
1.2 Format of the processed files	8
1.3 Linking process	9
1.4 Generated files	10
1.5 Bin/Hex Files	10
2. RL-ST6 CONFIGURATION	11
2.1 General Syntax	12
2.2 Use with RC-ST6	13
2.3 RL-ST6 and RIDE	14
2.4 Control directives	15
2.5 Target specification	16
2.6 Control of the absolute file	18
2.7 Control of the listing	20
3. APPENDICES	21
3.1 Linking mechanisms	22
3.2 Overlaying variables	24
3.3 LARGE memory model module mapping and bridges	25
3.3.1 INTRODUCTION	25
3.3.2 MODULE MAPPING	25
3.3.3 BRIDGES	25
3.3.4 INSERT CONSTRAINTS FOR MODULE MAPPING	26
3.4 Linking with RIDE	27
3.5 List of directives	30
3.6 List of errors	32
3.7 Revision History	39
4. INDEX	41

1. INTRODUCTION

1.1 What is the Linker used for ?

1.2 Format of the processed files

1.3 Linking process

1.4 Generated files

1.5 Bin\Hex files

The Linker RL-ST6 is used for linking the files (relocatable modules) assembled (or compiled) separately, in order to produce a single file, which can be used for programming of an EPROM (ROM), or for loading onto a debugging tool such as SIMICE-ST6 simulator.

This documentation indicates how to use RL-ST6 independently of the assembler or compilation software.

The manual is divided into two parts :

1. the present **introduction**,
2. configuration of RL-ST6.

Finally, a description of the internal mechanisms of the Linker, and a summary of the control directives and their abbreviations, can be found in the appendices at the end.

To use RL-ST6 integrated in another Raisonance program, please see also the chapter dedicated to RL-ST6 in the documentation related to these products.

1.1 What is the Linker used for ?

To increase the legibility and the program structure, it is often useful to divide a program into several files. Two types of separations are possible :

- inclusion of files (*INCLUDE* directive) allowing a file, at assembly, (or compilation), to be integrated within another, and for the entirety to be translated. It is simply an editing facility and during compilation it is treated as if there were only one file.
- separate assembly or compilation. The different files are assembled independently, then grouped together using the Linker.

In this mode, a file can use declarations of variables or sub-programs defined in another file. To establish these references between files, the variable must be defined with the attribute *EXTERNAL* in the files where the variable is used without really being defined. In the files where the variable is defined, (i.e., the file where the memory space is actually reserved), the variable must be defined with the attribute *PUBLIC*.

This mode is mandatory as soon as we start to work with a compiler. It is then possible to mix parts written in a high-level language with others written in assembler, provided that the intermediate formats be compatible, (i.e., they can be processed by the same Linker).

1.2 Format of the processed files

RL-ST6 only processes files in relocatable *OMF-ST6* format, generated by *MA-ST6* assemblers and *RC-ST6* compilers in files with the extension *OBJ*. The ultimate file, produced by RL-ST6, is also in *OMF-ST6* format, but contains only absolute segments (rather than *relocatable* ones). This file has the suffix *AOF*.

The format *OMF-ST6*, is compatible with all RAISONANCE's tools.

1.3 Linking process

The stages are :

1. Choice of the necessary modules. Modules containing non useful information will simply be ignored.
2. Allocation of necessary memory space for each of the segments declared in the selected modules, and translation of intra-module references.
3. Resolution of the *EXTERNAL* references between modules.
4. Generation of absolute code.
5. Detection of errors or anomalies arising from the preceding stages, and creation of a list file.

In most situations, the internal mechanisms of each of the different stages are of no interest to the user. However, some explanations, concerning essentially the priority rules of memory space allocation, are supplied in the appendices (*LINKING MECHANISMS*).

1.4 Generated files

RL-ST6 creates two types of file : absolute code files and list files. These files have the same name, but different suffixes: *MS6* for the list, *AOF* for absolute code.

The “.MS6” file includes 2 parts for the memory mapping. The first one concerns the DATA space (with or without bank switching) and the second one concerns the CODE space (with or without bank switching).

The format of the output files is the same as that of the input files (*that is, the OMF-ST6 files*). As these output files no longer contain relocatable segments, they can be directly loaded by RAISONANCE's debugging tools : **SIM- ST6**. These output files keep the tables of symbols in order to allow symbolic debugging.

The list file is an ASCII file which reports the list of errors encountered during the Linking, the list of the physical spaces (indicating the addresses where the relocatable segments have been placed) and a table of symbols.

1.5 Bin/Hex Files

From absolute file (.aof) it is possible to obtain .bin and .hex files. With Ride see section **Erreur! Source du renvoi introuvable.** With the command line use the ohst6.exe application (in ride\bin directory):

- For .hex file, enter "ohst6.exe c:\ride\examples\rcst6\bits\bits.aof"
- For .bin file, enter "ohst6.exe c:\ride\examples\rcst6\bits\bits.aof bin".

2. RL-ST6 configuration

2.1 *General Syntax*

2.2 *Use with RC-ST6*

2.3 *RL-ST6 and RIDE*

2.4 *Control Directives*

2.5 *Target specification*

2.6 *Control of the absolute file*

2.7 *Control of the listing*

2.1 General Syntax

The configuration of the Linker is carried out entirely on the DOS command line.

This command line must contain, in this order :

1. the key *RLST6* (for *RLST6.EXE*),
2. the list of files to be linked, with a comma as a separator between each file,
3. if desired, the name of the output module, preceded by the keyword *TO*,
4. the list of control directives, simply separated by spaces.

The syntax can, therefore, be summarized as follows :

RLST6 fic1 [[,ficN]...] [to fic_out] [[directive]...]

RL-ST6 offers an editing facility, which deals with command lines which are too long : all or part of the command line (except the keyword *RLST6*) can be transferred to an external file created under an editor. The name of the file is then indicated by placing the character '@' before it.

EXAMPLE :

rlST6 @list1.fic @list2.dir

2.2 Use with RC-ST6

RL-ST6 will automatically load the RC-ST6 libraries as they are required.

RL-ST6 will analyze the project and will select the most appropriate library functions for the application e.g. print or the startup function.

IMPORTANT:

It is better not to specify the RC-ST6 libraries on the command line, and « let RL-ST6 do what it wants »,

2.3 RL-ST6 and RIDE

Integrated Development Interface RIDE is very powerful for RL-ST6.

The file and library lists are built automatically for the user. Default options allow results to be generated quickly.

The file order is exactly the same than the one displayed in the window « Project ».

When user libraries are added to the project, it is important to put them at the end of the list.

2.4 Control directives

The control directives are divided into three groups :

1. Control directives for target specification,
2. Control directives for the generation of an output file,
3. Control directives for the generation of an output listing.

In general, for each directive, there is a corresponding keyword and an abbreviation which can be interchangeable. All the control directives are optional; if not defined, a default control is carried out, except for target specification. Note that all the control directive are given by RIDE directly to the RL-ST6 linker using the configuration dialog boxes.

In practice, the use of control directives is rare.

The following chapters describe the two groups of control directives.

Note:

GLOBAL is a general directive available for all the Raisonance tools. It allows to use a directive whatever the toolchain is. If the used tool implements the specified directive it will take it into account, if the used tool does not implement the directive, it will ignore it. It allows to keep the same code for different tools.

Syntax: global(directive)

2.5 Target specification

Each ST6 derivative has a specific mapping for CODE memory and DATA memory. This mapping must be given as parameter to the RL-ST6 linker for a complete and correct relocation. In this way three control directives have been added for CODE space, three others for DATA space and two for DRBR mask and location.

The three following ones are for CODE specifications :

CODESTART

This control directive indicates the starting CODE address. For example, with a ST6200C, the control directive must be CODESTART(0B00h), with a ST6201C, the control directive must be CODESTART(0800h) and with a ST6218C the control directive must be CODESTART(0000h).

CODESIZE

This control directive indicates the size of CODE space. For example, with a ST6200C, the control directive must be CODESIZE(1024), with a ST6201C it must be (1824) and with a ST6218C it must be CODESIZE(8192)

CODERESERVED

This control defines the CODE area which are reserved inside the space. The control directive syntax is the following one :

CODERESERVED (startaddr-endaddr [, startaddr-endaddr])

For example, to specify that the space from 0000h to 007Fh, and from 0FA0h to 0FEFh are reserved, the control directive must be :

CODERESERVED(0000h-007Fh,0FA0h-0FEFh)

The three following ones are for DATA specifications :

DATASTART

This control directive indicates the starting DATA address. For example, with a ST6200C, the control directive must be DATASTART(084), with a ST6201C, the control directive must be DATASTART(084) and with a ST6218C the control directive must be DATASTART(000).

DATASIZE

This control directive indicates the size of DATA space. For example, with a ST6200C, the control directive must be DATASIZE(064), with a ST6201C it must be (064) and with a ST6218C it must be DATASIZE(192)

DATARESERVED

This control defines the DATA area which are reserved inside the space. The control directive syntax is the following one :

DATARESERVED (startaddr-endaddr [, startaddr-endaddr])

For example, to specify that the space from 0000h to 001Fh, the control directive must be :

DATARESERVED(0000h-001Fh)

The two following ones are for DRBR specification :

DRBRADDR

This control directive defines the address of DRBR register. For example, with a ST6218C, the control directive must be :

DRBRADDR(0CBh)

DRBRMASK

This control directive defines the mask of DRBR register. In other words, it indicates which bits into DRBR are used to defined the DATA banks. For example, with a ST6218C, the control directive must be :

DRBRMASK(018h)

2.6 Control of the absolute file

The generated file regroups all the information needed for debugging, contained in the input files : symbols and line numbers (for high-level languages). It is sometimes necessary to reduce the dimension of the generated file in order to load it onto development tools with limited memory space.

The first solution is to specify the *NODEBUG* option to the assembler (or compiler). This solution has the advantage of being able to retain the symbols of certain files selectively.

The second solution, quicker, is to ask RL-ST6 to delete this information globally for all the files. However, RL-ST6 allows distinction between symbols that are local to a module, and those declared with the attribute PUBLIC, and line numbers (for high-level languages). This is done via the following directives :

DEBUGSYMBOLS(DS), NODEBUGSYMBOLS(NODS)

For local symbols.

DEBUGPUBLICS(DP), NODEBUGPUBLICS(NODP)

For public symbols.

DEBUGLINES(DL), NODEBUGLINES(NODL)

For line numbers.

The debug directives (*DS/DP/DL*) are those taken by default.

There are two controls for filling the reserved and unused CODE areas with a specific value. Those controls are :

ROMFILLRESERVEDVALUE

For the reserved areas. For example, to fill the reserved areas with the value 04h, the correct control is *ROMFILLRESERVEDVALUE(04h)*

ROMFILLUNUSEDVALUE

For the unused areas. For example, to fill the reserved areas with the value 0FFh, the correct control is *ROMFILLUNUSEDVALUE(0FFh)*.

In case of use of RC-ST6 compiler, you may want to initialize a part of the static DATA memory. This can be done inside the startup routine via the control *RAMSIZEINIT*. The initialized DATA space always starts at address 084h. For example, if you want to initialize 10h bytes from 084h to 094h, you can use the control directive *RAMSIZEINIT(010h)*

Again in case of use of RC-ST6 compiler and the printf routine. You have to specify what is the maximum size needed for printf parameters. By default this size is set to 2. The needed size is 2 for a character, a short integer or a long integer or a generic pointer. The needed size is specified with *PRSTATICSIZE(04h)* to reserved 4 bytes.

Finally, the directive *NAME (NA)* allows the name of an output module to be changed. By default, the name of the first module processed, (the first module of the first file indicated in the list), is chosen. This directive is useful when the first module has a generic name, for example, *MAIN* for programs written in C language.

2.7 Control of the listing

The generated listing file is often quite large, due to the number of symbols or line numbers of certain applications.

The aim of the following directives is to reduce the amount of information contained in the list file:

PRINT(PR) / NOPRINT(NOPR)

NOPRINT means no list file will be created. PRINT is the default value.

SYMBOLS(SB) / NOSYMBOLS(NOSB)

Creation (or not) of the table of local symbols. SYMBOLS is the default value.

PUBLICS(PL) / NOPUBLICS(NOPL)

Creation (or not), of the table of public symbols. PUBLICS is the default value.

LINES(LI) / NOLINES(NOLI)

Creation (or not) of the table of line numbers (created by a high-level language compiler). LINES is the default value.

IXREF (IX) / NOIXREF(NOIX)

Creation (or not) of the table of cross references, showing in which segment(s) each public symbol is declared and used. NOIXREF is the default value.

CALLTREE / NOCALLTREE

Creation (or not) of the call tree of the project. Each node is labeled with the necessary stack levels for a complete execution, with the data size needed and the class of the function (0=standard tree, 1=interrupt tree and 2=NMI tree). The call tree is inserted inside the list file, in this way, if no list file is generated, no call tree can be generated. CALLTREE is the default value.

MODULEMAP / NOMODULEMAP

Creation (or not) of the module mapping when LARGE memory model of the RC-ST6 compiler is used. The module mapping indicates the CODE page and the DATA page for each module and the number of calls between modules. The number of bridges generated is also indicated. The module mapping is inserted inside the list file, in this way, if no list file is generated, no module mapping can be generated. MODULEMAP is the default value.

3. APPENDICES

3.1 *Linking mechanisms*

3.2 *Overlaying variables*

3.3 *LARGE memory model module mapping and bridges*

3.4 *Linking with RIDE*

3.5 *List of directives*

3.6 *List of errors*

3.1 Linking mechanisms

A relocatable file (suffix .OBJ) can contain one or several concatenated modules. If the user has taken care to specify precisely one or several of these modules, only the specified modules will be processed. If however, he has indicated only the filename, all the modules included in the file will be processed.

In the case of a library, which is also a concatenation of modules, only modules with public symbols corresponding to external symbols defined in a previously processed module, will be processed. This condition is important, as it means that the order of specification of the libraries is not indifferent : if library L1 refers to a symbol declared in PUBLIC in library L2, FIRST L1, THEN L2, must be specified in the file list.

The first stage of the allocation, is to group together in one single module, all relocatable segments from different modules which have :

- the same name,
- the same type (DATA,...),
- the same allocation attributes (INPAGE, INBLOCK, INWINDOW).

The second stage, is to take all the relocatable segments in order, and place them in the "holes" left free by the absolute segments.

The "seating order" of the relocatable segments is precise. It takes into account the following rules :

- the segments are treated separately according to their physical allocation space : ROM or RAM.
- in general, the order of priority for placing the segments is as follows (descending order) :
 - ◆ Absolute segments,
 - ◆ Relocatable segments specified in a relocation control directive,
 - ◆ Relocatable segments in the same order as modules.
- the search for an available "hole", always begins at the address 0, and goes on until a suitably-sized available space is found.

Once the segments have been located, the addresses of the PUBLIC symbols are deduced by adding to the segment address, the difference between the symbol and the beginning of the segment.

The totality of the external references can be run through, and the correct address can be introduced into the code.

Saving the absolute code is a second pass, in which each module is taken again, and concatenated in an output file, taking care :

- to ignore the PUBLIC or EXTERNAL lists,
- to ignore, or not, the lists of symbols following the specified control directives.

The Linker directly takes the DRBR mask from the Target specifications, when Data bank switching is enabled. This mask can also be given with the DRBRMask (value) directive where value is the hexadecimal value of the mask.

3.2 Overlaying variables

- RL-ST6 linker generates a call tree file of the project. According to this tree and the link between CODE and DATA segments (a group CODE and DATA segment defines a function), it determines which DATA segment can be overlaid. The overlay process creates three groups of overlaid data. The first one `_DGROUP00_` is for standard functions, means functions in the reset vector tree, the second one `_DGROUP01_` is for interrupt functions, means functions in the interrupt vectors tree, the third and last one `_DGROUP02_` is for NMI functions, means functions in the NMI vector tree.
- If the pragma `NOVERLAY [NOOL]` is used, the program structure analysis is inhibited and the local variables are not overlaid. It is highly recommended to let the linker performing the overlay process.

3.3 LARGE memory model module mapping and bridges

3.3.1 Introduction

The automatic module mapping mechanism takes in account modules and not segments. This means that mapping is done for a set of segments (CODE and DATA) at one time. The aim of the process is to minimize the number of bridges, in other words, the process tries to set in the same memory page (a couple of CODE page and DATA page) the module that have a lot of calls.

The library modules are set in STATIC code page because by definition they may be called by all user functions. Those modules do not use global data.

3.3.2 Module mapping

First of all, you must keep in mind that the mapping algorithm has a complexity of n^n where n is the module to map. This means that to map 5 modules, there are $5^5 = 3125$ possibilities. In other words, for example, for 10 modules, the number of possibilities is 10,000,000,000 possibilities. This number is too big to we developed in a correct time.

So 5 is a correct value for developing the algorithm and is the default. If there are more than 5 modules that are no constrained, the process takes the 5 biggest and set the other ones in the static CODE page. But the best solution is to constrain manually as many modules as possible and let only the 5 biggest to the process. You can also modify the number of modules to be mapped by the process. See chapter 3.3.4 for information about constrain.

3.3.3 Bridges

Each time a call is necessary between two functions which are not in the same CODE page and the called one is not in the STATIC CODE page, a bridge is generated. The bridge structure is the following one :

```
Bridge_from_fct1_to_fct2 :
    CALL ?C?SETPAGE_fct2
    CALL fct2
    CALL ?C?SETPAGE_fct1
    JP    fct1_call_addr + 2
```

Function fct1 has performed a jump to the bridge.

You can see, due to the bridge structure, that a different bridge is needed for each call from fct1 to fct2. The bridge size is 8 bytes. The ?C?SETPAGE_xxx function sets the PRPR and DRBR registers to the correct values corresponding to the environment of 'xxx'. While in the biggest ST6 derivative, there are three different values for PRPR and 2 different values for DRBR, there are 12 different ?C?SETPAGE_xxx functions.

You must notice that when a function from a CODE banked area calls a function in STATIC CODE page which only calls functions in STATIC CODE page or in the caller CODE banked area, no bridge is necessary. However if a function from a CODE banked area calls a function in STATIC CODE page which calls a function in another CODE banked area, a bridge is necessary between the caller and the function in STATIC CODE page.

3.3.4 Insert constraints for module mapping

There are two ways to insert constraint for module mapping.

The first one is to set the CODE page and the DATA page into the source code of the module. If the source code is a MA-ST6 assembler code, you can use \$PRPAGE(val) and \$DTPAGE(val) directives to specify the segments, or \$MODULE(val) directive to specify the module (group of CODE and DATA pages). If the source code is a RC-ST6 compiler code, you can use #pragma PRPAGE(val) and #pragma DTPAGE(val) directives to specify the segments, or #pragma MODULE(val) directive to specify the module (group of CODE and DATA pages). Please refer to the respective user manuals for more information about those controls.

The second one is to use RIDE interface. In the section “Options | Project | RLST6 | Banking”, you must select the different modules and set the module you want to specify. The module number is as follows :

Module Index	CODE page	DATA page
0	0	1
1	STATIC	1
2	2	1
3	3	1
4	0	2
5	STATIC	2
6	2	2
7	3	2
8	0	STATIC
9	STATIC	STATIC
10	2	STATIC
11	3	STATIC
--	AUTO	AUTO

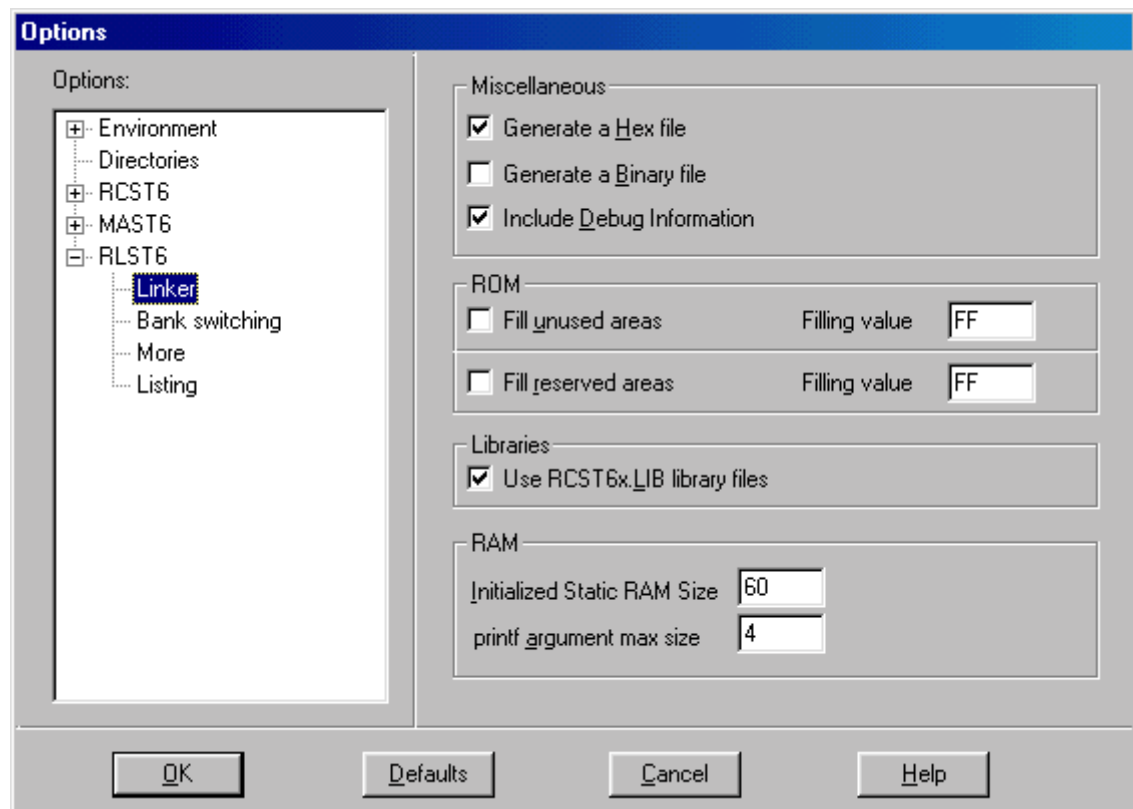
You can also modify the number of modules to be processed in the section “Options | Project | RLST6 | Banking | Number of modules to process”.

3.4 Linking with RIDE

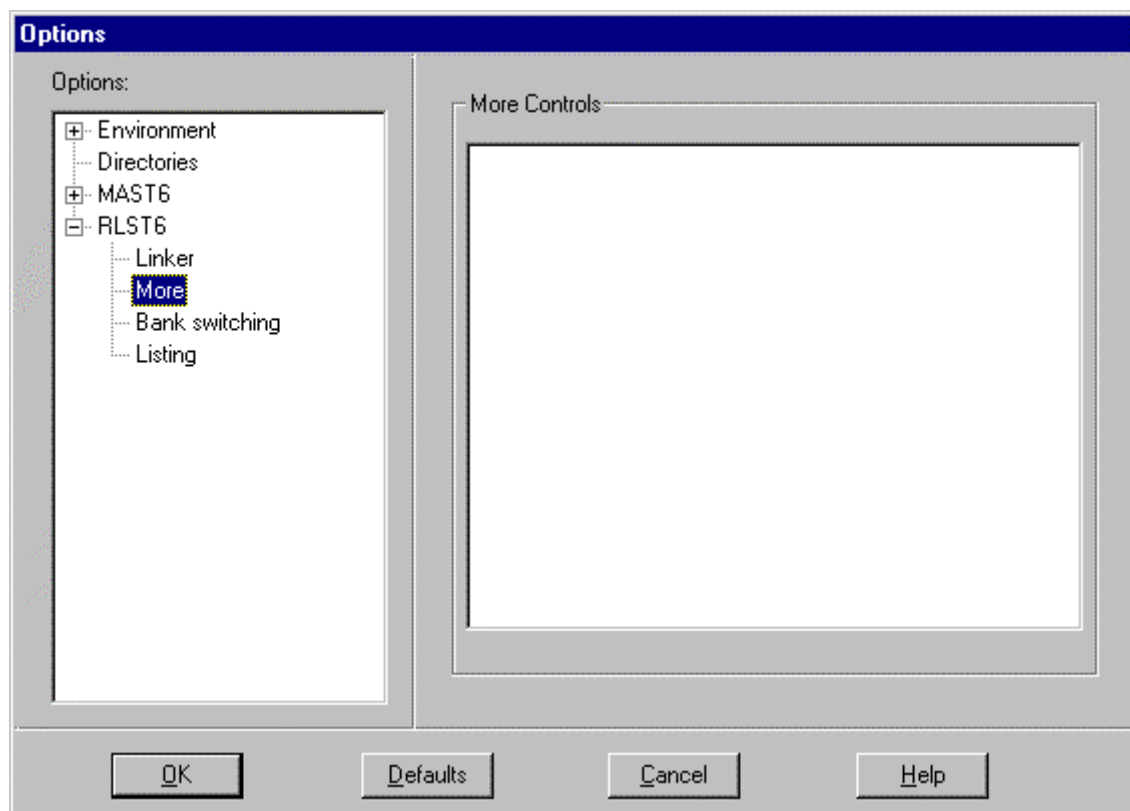
The RIDE integrated environment allows the compilation, the assembling and linking of applications built in a project, with the commands MAKE and BUILD ALL, LINK.

The Linker options dialog box allows to the libraries used during the linking, and the linking directives to be defined:

the box is opened with the **Options|Project|RLST6** command.

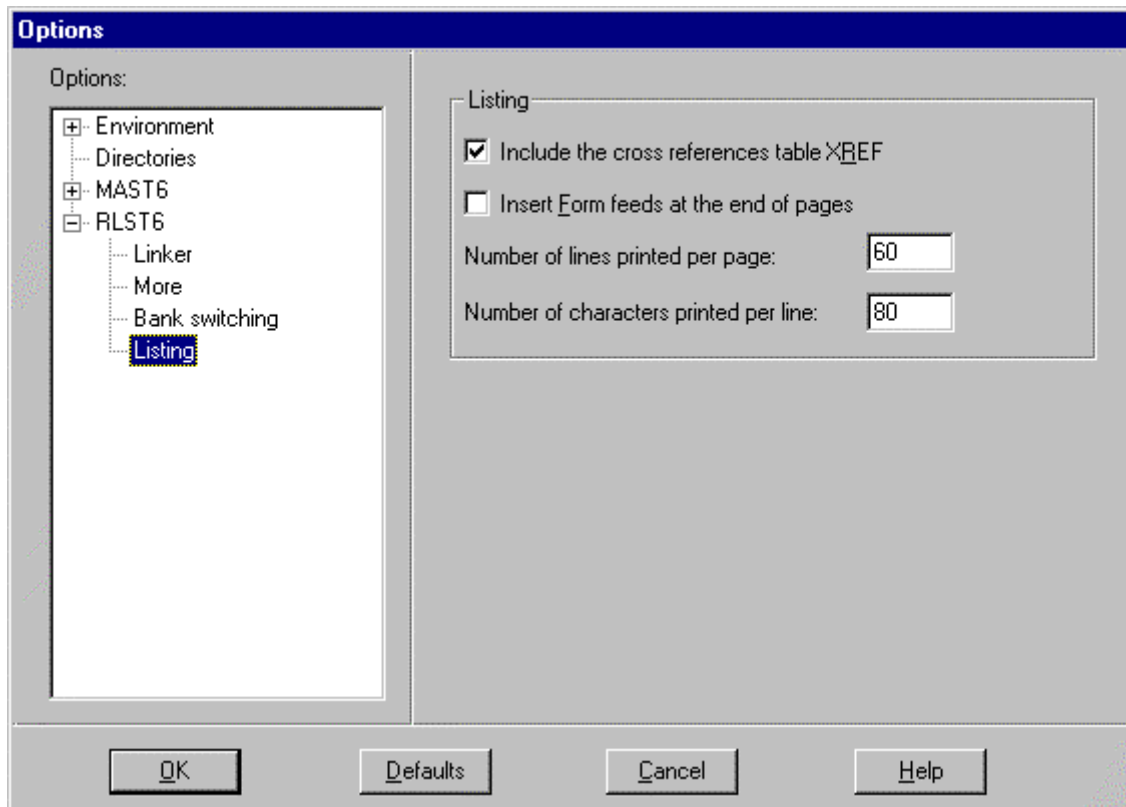


- **Intel Hex File** (RLST6|Linker|Generate an Intel Hex File): generate an Intel executable file.
- **Binary File** (RLST6|Linker|Generate a Binary file) : generate a binary executable file.
- **DEBUG** (RLST6|Linker|Include Debug information) : Debug information can be kept in the output file (default option).
- **ROM:** Fill or not the unused areas and/or the reserved areas of the ROM
- **Libraries:** Use or not the RCST6x.lib library
- **RAM:**
 - **Initialized Static RAM Size:** Value from 0 to 64.
 - **Printf argument max size:** Number of bytes necessary in the static stack segment of printf for the argument passing.



RL-ST6 / More RIDE options

- **More controls...** (*RLST6|More*) : This control allows adding linker directives or specific commands or commands file to be added e.g. : @list1.file).



RL-ST6 / Listing RIDE options

- **XREF** (*RLST6|Listing|Include cross-references XREF*) : A cross reference table can be generated (default option).
- **EJECT** (*RLST6|Listing|Insert Form Feed at end of pages*) : This control specifies the insertion of form feed at the end of each page of the listing file.
- **PL** (*RLST6|Listing|Number of lines printed per page*) : This control specifies the number of lines printed per page in the listing file.
- **PW** (*RLST6|Listing|Number of characters printed per lines*) : This control specifies the number of characters printed per lines in the listing file.

3.5 List of directives

ABBREVIATION	DIRECTIVE
AOB	AUTOOPTBANK
CALLTREE	CALLTREE
CODEBANKAREA	CODEBANKAREA
CODERESERVED	CODERESERVED
CODESIZE	CODESIZE
CODESTART	CODESTART
DATABANKAREA	DATABANKAREA
DATASIZE	DATASIZE
DATASTART	DATASTART
DL	DEBUGLINES
DP	DEBUGPUBLICS
DRBRADDR	DRBRADDR
DRBRMASK	DRBRMASK
DS	DEBUGSYMBOLS
IX	IXREF
LI	LINES
MODULEMAP	MODULEMAP
NA	NAME
NOCALLTREE	NOCALLTREE
NODL	NODEBUGLINES
NODP	NODEBUGPUBLICS
NODS	NODEBUGSYMBOLS
NOIX	NOIXREF
NOLI	NOLINES
NOMODULEMAP	NOMODULEMAP
NOOL	NOOVERLAY
NOPR	NOPRINT
NOPU	NOPUBLICS
NOSB	NOSYMBOLS

OL	OVERLAY
PL	PAGELength
PW	PAGEWIDTH
PR	PRINT
PRSTATICSIZE	PRSTATICSIZE
PU	PUBLICS
RAMSIZEINIT	RAMSIZEINIT
ROMFILLRESERVEDVALUE	ROMFILLRESERVEDVALUE
ROMFILLUNUSEDVALUE	ROMFILLUNUSEDVALUE
TO	TO

3.6 List of errors

RL-ST6 generates 3 types of errors, each with an increasing level of seriousness :

WARNING : Signals an anomaly of minor importance. The linking will continue and finish as normal, but the programmer is strongly advised to check that everything is all right.

ERROR : Signals a more serious problem, but the code will still be generated.

FATAL : The type of the error prevents absolute code generation by RL-ST6.

The following list reiterates the three error categories, and supplies some explanations as to the probable cause. In most cases, the signaling of an error is supplied with additional information : name of symbol, of segment, of module,...

WARNINGS

ASSIGNED ADDRESS NOT COMPATIBLE WITH ALIGNMENT

The address indicated in the relocation directive is not compatible with the segment type.

DATA SPACE MEMORY OVERLAP

An internal RAM space is occupied by several overlaid segments. These overlays are sometimes voluntarily created by the user.

CODE SPACE MEMORY OVERLAP

A code space is occupied by several overlaid segments.

MODULE NAME NOT UNIQUE

A module has been found several times, in several different files. Only the first module has been processed.

EMPTY ABSOLUTE SEGMENT

The absolute segment indicated is empty. No memory space will be allocated to it, and its absolute address even runs the risk of being occupied by another segment.

ERRORS

UNRESOLVED EXTERNAL SYMBOL: symbol [file]

The external symbol « symbol » found in « file » has not been declared.

SEGMENT COMBINATION ERROR

The recombination of partial segments has been faced with a conflict of type.
The segment is ignored.

EXTERNAL ATTRIBUTES MISMATCH

The same EXTERNAL symbol has been found in different modules with different attributes. The symbol is ignored.

EXTERNAL ATTRIBUTES DO NOT MATCH PUBLIC

The attributes of a PUBLIC symbol do not correspond to the attributes of an EXTERNAL symbol bearing the same name.

MULTIPLE PUBLIC DEFINITIONS

Several definitions of the same PUBLIC symbol have been encountered in different modules. This symbol is ignored.

SEGMENT OVERFLOW

The segment indicated, after concatenation of the partial segments, is of a size greater than the maximum allowed.

ADDRESS SPACE OVERFLOW

It is impossible to allocate a space to the indicated segment, which will be ignored.

SEGMENT IN LOCATING CONTROL CANNOT BE ALLOCATED.

The segment indicated cannot be located at the address stipulated by the relocation directive.

EMPTY RELOCATABLE SEGMENT

The indicated segment is empty.

CANNOT FIND SEGMENT

The indicated segment cannot be found.

SEGMENT TYPE NOT LEGAL FOR COMMAND

The segment indicated, specified in a directive, is of a type incompatible with the directive.

SEGMENT DOES NOT FIT

The segment indicated is too big to be placed at the address indicated in the directive.

CANNOT FIND MODULE

The module indicated cannot be found.

MODULE OR SEGMENT DOES NOT FIT IN ONE CODE PAGE

The code in the module or segment causing the error (see what is appended to error) is greater than 2 Kbytes. Therefore, it does not fit into one ST6 code page.

Split your .c file into several .c files (module) or your function into several functions (segment).

MODULE DOES NOT FIT IN ONE DATA PAGE

The global data in your module is greater than 64 Bytes. Therefore, it does not fit into one ST6 data page.

Distribute your global data in several .c files.

INWINDOW SEGMENT IS GREATER THAN 64 BYTES

Your in-window segment does not fit into one window. One window contains 64 Bytes. Your segment is greater than that.

Split your segment into several segments which will be all smaller than 64 Bytes.

SEVERAL BLOCK ALIGNMENTS WITHIN ONE MODULE

Block alignment means that the segment is starting at the beginning of the ST6 page. In this case, several segments are supposed to start at the same spot.

Reduce the number of block alignments in the module or distribute the block-aligned segments over several modules.

WINDOW- OR PAGE-ALIGNED SEGMENT COULD NOT BE PLACED

Window-aligned or page-aligned segments could not be placed.

Split your modules in several pieces.

INWINDOW SEGMENT COULD NOT BE PLACED

In-page constrained segments could not be placed.

Split your module in several pieces.

ABSOLUTE SEGMENTS OVERLAP

An absolute segment is starting at an address which is occupied by another absolute segment

Place the segments taking into account their length.

ABSOLUTE SEGMENTS MUST BE IN THE SAME PAGE BANK FOR ONE MODULE

One module is supposed to occupy one ST6 page. Its segments have to be placed on the same page (exception: for the code, a module can contain segments located on the static page and another page different from the static page).

Place segments in the same module on the same page.

ADDRESS AND PAGE SELECTIONS DO NOT MATCH

The range of addresses for the static code page is from 0x800-0xFEF, and the range of the bank-switched pages is from 0x000 to 0x7FF. The static data page ranges from 0x80 to 0xFF, and the switched pages from 0x00 to 0x3F. From these addresses, take off the reserved pieces which vary from component to component. Page and address selections have to correspond.

Choose corresponding page and address selections.

INVALID SECTION NUMBER

The section number you indicated is not valid. It might be out of bounds, or the section might not exist for your component.

Choose a valid section number. For the code, 0, 2 and 3 are the banked sections and 1 is the static section. For the data, 8 is the static section and 0 and 1 are the banked sections.

FATAL ERRORS***INVALID COMMAND LINE SYNTAX***

The command line contains a major syntactical error.

EXPECTED ITEM MISSING

An operand indispensable to a directive is missing.

INVALID KEY WORD

An unrecognized directive has been encountered.

INVALID CONSTANT

A numerical value has an invalid format.

INVALID NAME

The name indicated is invalid, (invalid characters,...)

INVALID INPUT MODULE

The segment indicated cannot be located at the address stipulated by the relocation directive.

3.7 Revision History

DATE	SECTION	DESCRIPTION

4. INDEX

AOF,8, 10	library,22 LINES,20 listing,20
BIN,10	
command line,12 Command line,10	NAME,19 NODEBUGLINES,18 NODEBUGPUBLICS,18 NODEBUGSYMBOLS,18 NOIXREF,20 NOLINES,20 NOPRINT,20 NOPUBLICS,20 NOSYMBOLS,20
DEBUGLINES,18 DEBUGPUBLICS,18 DEBUGSYMBOLS,18	
ERROR,32, 34	OBJ,8 OMF-ST6,8 output file,10 output module,19 overlay,33
FATAL,32, 38	
GLOBAL,15	PRINT,20 PUBLICS,20
HEX,10	seating order,22 SIM-ST6,10 symbols,10, 18, 20 syntax,38
INBLOCK,22 INPAGE,22 INWINDOW,22 IXREF,20	WARNING,32, 33