

RC-ST6

ANSI-C Compiler for the ST6 family



Reference Manual

April 2005

RAISONANCE

Information in this document is subject to change without notice and does not represent a commitment on the part of the manufacturer. The software described in this document is furnished under license agreement or nondisclosure agreement and may be used or copied in accordance with the terms of the agreement. It is against the law to copy the software on any medium except as specifically allowed in the license or nondisclosure agreement. No part of this manual may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or information storage and retrieval systems, for any purpose other than the purchaser's personal use, without prior written permission.

Every effort was made to ensure the accuracy in this manual and to give appropriate credit to persons, companies and trademarks referenced herein.

Copyright 2000, 2005 RAISONANCE SAS. All rights reserved

Microsoft® and Windows™ are trademarks or registered trademarks of Microsoft Corporation.

PC® is a registered trademark of International Business Machines Corporation.

Raisonance is a registered trademark of RAISONANCE S.A.S.

CONTENTS

1. INTRODUCTION	7
1.1 Presentation	8
1.2 RC-ST6 Environment	9
1.2.1 “Windows” versus “DOS” tools	9
1.2.2 Invocation	9
1.3 Installation	10
1.3.1 “Install” program	10
1.3.2 Organization of the directories	10
1.4 First programs	11
1.4.1 “Hello !”	11
1.4.2 Counter	12
1.5 Provided Source Files	12
2. DIFFERENCES FROM ANSI-C	13
2.1 Restrictions	14
2.1.1 Types	14
2.1.2 “Object” Sizes	14
2.1.3 Functions of the “ANSI” Libraries	14
2.1.4 Reentrance and Recursivity	14
2.2 Extension to the ANSI Standard	15
2.2.1 global directive	15
2.2.2 Reserved Keywords	15
2.2.3 New Type Qualifiers	16
3. IMPLEMENTATION	23
3.1 Memory Models	24
3.1.1 How to select your Memory Model	24
3.1.2 How the Memory Model influences the Code Generation	24
3.2 Linker	25
3.3 The Concept of “Module”	25
3.4 Base Data Types	26
3.4.1 List of base types	26
3.4.2 Standard word (int) width	26
3.5 Data Accesses in the LARGE Model	27
3.5.1 Using Keywords to specify Variable Location	27
3.5.2 Interrupt Handlers	29

3.6 Pointers	30
3.6.1 General	30
3.6.2 Syntax	30
3.6.3 Space qualified Pointers	31
3.6.4 Pointers Coding	32
3.6.5 Use of Pointers	34
3.6.6 Pointers to Functions	36
3.7 Semantical Considerations	37
3.8 Parameter Passing Convention	38
3.8.1 Rules	38
3.8.2 Interfacing C Programs to Assembler	38
3.9 Predefined Macros	39
3.10 Startup code	39
 4. PRAGMAS	 41
4.1 Using Pragmas	42
4.1.1 What is a Pragma?	42
4.1.2 Indicating a Pragma	42
4.1.3 List of Pragmas	44
4.2 Environment Pragmas	46
4.2.1 Reference Directory	46
4.2.2 Listing File	47
4.2.3 Object or Assembler Source File	51
4.3 Implementation Controls	53
4.3.1 Memory	53
4.3.2 Optimization	57
4.4 Exercising Control through RIDE	63
4.4.1 The Compiler RCST6	63
4.4.2 The linker RLST6	64
 5. LIBRARIES	 65
5.1 string.h	66
5.2 stdio.h	67
5.2.1 putchar	68
5.2.2 getchar	68
5.2.3 ungetchar	68
5.2.4 sprintf and printf	68
5.2.5 sscanf	69
5.3 stdlib.h	70
5.4 ctype.h	71
5.5 regst6x.h	72
 6. APPENDICES	 73

6.1 Error Messages	74
6.1.1 Error types	74
6.1.2 Causes	74
6.1.3 System errors	75
6.1.4 Preprocessor errors	76
6.1.5 Compilation errors	79
7. INDEX	100
8. BIBLIOGRAPHY	102
8.1 About C language	102
8.2 About ST6 microcontrollers	102

1. INTRODUCTION

1.1 Presentation

1.2 RC-ST6 Environment

1.3 Installation

1.4 First programs

1.5 Provided Source Files

1.1 Presentation

If you are using the compiler for the first time, please read this manual before compiling your first programs. The authors of this compiler have made every effort to comply with the ANSI standard for the C language, but have added a number of restrictions and extensions to cater for the unique architecture of the ST6.

This manual describes differences from ANSI-C. It is not intended to be a C language tutorial nor an introduction to the grammar. For information on the language, “The C language” by B.W. Kerninghan and D.M. Ritchie is highly recommended and is the standard reference in this field.

This manual is not intended to provide details of the ST6 architecture. Previous experience of the ST6 architecture is desirable but not mandatory, and knowledge of the ST6 instruction set is not necessary.

The contents are given at the beginning of the manual.

The introduction contains some examples as a guide to the implementation.

Section 2 describes how to install and run RC-ST6.

Section 3 describes the extensions and differences between the RC-ST6 and ANSI-C specifications.

Section 4 explains the implementation needed to optimize the generated code and link programs written in assembler language with those written for RC-ST6.

Section 5 describes the compiler controls.

Section 6 gives a list of the functions available in RCST6x.LIB.

Finally, the appendices discuss performances and list error messages.

1.2 RC-ST6 Environment

1.2.1 “Windows” versus “DOS” tools

The Raisonance RC-ST6 compiler is a part of RKIT-ST6, which is a Windows Integrated Development Environment (IDE) containing a set of development tools for ST6 microcontroller applications. This development environment includes a syntax highlighting editor, a project manager, on-line help, and provides all the facilities to call the different Raisonance development tools.

The following software products are supplied as a software KIT:

the ANSI-C RC-ST6 compiler, with its libraries

the MA-ST6 macro assemblers

the LX-ST6 linker

the SIMICE-ST6 integrated debugger/simulator

For a standard application, it is highly recommended that you use RCST6 in the RAISONANCE IDE. But, as for the other Raisonance tools, a command line version (DOS) is available to allow an automatic invocation (from a “make” utility for example).

1.2.2 Invocation

From the Raisonance IDE:

Within the RIDE IDE, the RC-ST6 C compiler is invoked:

to compile a C file, with the **Project|Translate** command.

to build a project (.PRJ file), with the **Project|Build All** command.

Please refer to the reference manual named « **RIDEPGM. Integrated Development Environment** » to learn about the RIDE project manager.

From a “Command Prompt” windows:

Run the RCST6.EXE executable file followed by the filename, then the list of directives.

1.3 Installation

1.3.1 “Install” program

The installation program **Install.exe**, in the CD allows the installation of the RAISONANCE RKIT-ST6 tools, and especially RC-ST6. An autorun file starts the installation automatically, you just have to follow the instructions.

When the installation program is run, all the tools of RKIT-ST6 are installed including the RC-ST6 C compiler. This requires about 8 MB of hard disk space.

1.3.2 Organization of the directories

Upon installation, the following structure is created :

The **INC** directory contains:

- the standard ANSI .h files and special function register declarations REGxxx.h
- special function register declarations REGxxx.inc (for the macro assembler)
- the declaration files for the kernel

This directory also contains a sub-directory which holds the source files of all library modules needed to carry out hardware related adaptations and for the initialization of the application.

The **BIN** directory holds the executable files (.EXE, .DLL).

The **LIB** directory holds the different run time libraries of the C Compiler and the kernel.

The **EXAMPLES** directory and its sub-directories holds examples of complete applications.

The **DOC** directory holds the user manuals in .pdf files.

The **HELP** directory holds the help files (.HLP).

1.4 First programs

Here are some simple examples with comments, as an introduction to the Raisonance RC-ST6 compiler:

1.4.1 “Hello !”

```
#include <stdio.h>
#include <regST6.h>

void main(void)
{
    printf ("\nHello world !");
    while (1);
}
```

The RCST6S.LIB library contains the standard functions, including the ANSI standard printf function which, by default, transmits data on the ST6 UART. It is not necessary to specify them in the command line as they are automatically searched by the linker.

The initialization function ensures automatic initialization and execution of the main() function. But, because there is no operating system, the function must include an infinite « while (1) » loop at the end of the program.

The string “\nHello world!” is automatically placed in CODE memory space.

The include file stdio.h concerns ANSI libraries functions declarations, adapted to the ST6.

The include file regST6.h concerns ST6 sfrs (Special Functions Registers) declarations. This declaration was useless in this example as the “main” function does not refer to sfrs.

1.4.2 Counter

```
#include <stdio.h>
#include <regST6.h>

unsigned counter = 0;

/* External interrupt 0 */
void it_ext0 (void) interrupt 1
{
    /* Increment counter */
    counter++;
}

void main(void)
{
    unsigned n = counter;
    //Enable interrupt
    while (1) if ( n!= counter )
    printf ("\nCount : %ld", n = counter );
}
```

New keywords such as **interrupt** have been added to the language which is consequently a superset of ANSI-C.

The `it_ext0 ()` function is an interrupt routine. The compiler automatically generates the interrupt vector.

1.5 Provided Source Files

Included with RC-ST96 are some source files useful to develop some specific applications:

- `Initstat.st6` concerns global and static variables initialization routine for the RC-ST6
- `Putchar.st6` concerns I/O routines at character level.
- `Startup.st6` is useful to provide the initial starting code for the ST6 from power-up to the initial execution of the `main()` routine.

2. DIFFERENCES FROM ANSI-C

2.1 Restrictions

2.2 Extensions to the ANSI standard

2.1 Restrictions

A typical ST6 derivative contains a few kilobytes of ROM (up to 8KB), and a hundred bytes in RAM. Thus, the ST6 family is certainly not dedicated to the largest embedded applications, and the RCST6 compiler has been designed to optimize what is supposed to be a standard ST6 application.

Considering both the real constraints and the goal of optimizing the typical applications, introduction of restrictions appear to be mandatory:

Physical Constraint	Restrictions of the Implementation
Small code size	Functions length limited to 2KB
Small RAM size	Object size limited to 64 bytes, NO floating point
6 levels of Stack	No recursivity

Often, the choice is directed by the intention to provide compact libraries, matching with the requirements of most of the ST6 applications.

2.1.1 Types

For the arithmetic types, only the 8-bit and 16-bit types are implemented: “long int” and “unsigned long int” are reduced to 16-bit. All floating types have been discarded.

2.1.2 “Object” Sizes

The declared variables (for both in code space or in data space) are limited in size at 64 bytes. Functions are limited to the size of one section (2KB) in the LARGE model. However, the module size for the SMALL model is larger (up to 4KB) but only one section is available.

2.1.3 Functions of the “ANSI” Libraries

As in every cross compiler, several functions (memory allocator, file management...) are not proposed.

The “printf” function is available, but in a simplified version: no floating points and some complicated format have been discarded.

The “scanf” function is not missing.

2.1.4 Reentrance and Recursivity

Recursivity is not possible, but reentrance from a higher level of interrupt is allowed.

This restriction has been made considering the fact that the hardware stack is limited to 6 levels.

2.2 Extension to the ANSI Standard

This section describes the differences between the Raisonance RC-ST6 compiler and a standard ANSI-C compiler which generates code for execution by a computer. Details of the new keywords and grammatical changes are included.

The RC-ST6 compiler language is globally a superset of that adopted by the ANSI/ISO standard (Document 9899:1990).

It has some restrictions but also some extensions. A program that operates on a standard computer, such as a PC, can therefore be compiled by the RC-ST6 and is executable, even if it does not use system specific resources such as the file manager.

RC-ST6 C Compiler is dedicated to embedded systems, that is why some minor rules of the ANSI standard are not taken into account. The two main points are:

- wchar type is not supported. It is considered as a normal char.
- the floating types are not supported.

Some new keywords have been added. They consist of memory space qualifiers (to specify the variable allocation space) and function attributes (to indicate that a function is to be processed as an interrupt routine for example). The rules resulting from the addition of new keywords are discussed in the next section. Details of the grammar which defines the syntax of the new keywords is given in the appendix (it is an extension to that given in the Kernighan & Ritchie standard).

The ANSI standard gives an accurate definition of the C language, but also allows compiler authors to make a choice concerning the implementation. Further details are given in the appendix.

2.2.1 global directive

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)

2.2.2 Reserved Keywords

Reserved keywords specified by the standard are given in Table 1. The new reserved keywords specific to Raisonance RC-ST6 are given in Table 2. None of these keywords can be redefined in your assembly files.

auto	break	Case	char	const	continue	default	do
double	else	Enum	extern	float	for	goto	if
int	long	Register	return	short	signed	sizeof	static
struct	switch	Typedef	union	unsigned	void	volatile	while

Table 1: Standard reserved keywords.

asm	at	code	data	generic
interrupt	intrinsic	scode	sdata	sfr

Table 2: Raisonance RC-ST6 specific keywords

2.2.3 New Type Qualifiers

2.2.3.1 Space Qualifiers

In the RC-ST6 the notion of space qualifiers as type qualifiers has been added. The rules are similar to those applicable to **const** and **volatile**, which are both type qualifiers specified in the ANSI standard.

Type qualifier: **const, volatile**

Space qualifier: **code, data, sdata, sfr**

Space qualifiers are linked to the memory model:

	SMALL	LARGE	Example
Data	For every declaration, the variable is directly accessible.	For the static and global declarations: in the current DRBR. For the automatic declarations (local): in the pseudo-stack. For the external declarations: supposed to be in a distinct DRBR.	<code>char data c1; data char c2; extern char data c3;</code>
Code	In the current PRPR	For the static and global declarations: in the current PRPR. For the automatic declarations (local): in the pseudo-stack, in the current PRPR. For the external declarations: supposed to be in a distinct PRPR	<code>char code c1 = 5; code char c2 = 'A'; extern char code c3;</code>
Sdata	=data	Non banking data area. The variables have to be on the static data page which is always visible.	<code>char sdata c1; sdata char c2; extern char sdata c3;</code>

code,..., sdata spaces are the various areas that are addressable from the ST6 instructions set.

Semantic rules:

- The RC-ST6 compiler allows local variables to be space qualified.
- A function cannot be « space qualified » as it will always be located in the code space. Similarly, a pointer to a function is implicitly regarded as a pointer to code.
- CODE space can only be read, and not written to. However, a write operation on a variable qualified by code will not result in an error (the execution of such an instruction will have no effect). To « protect » a variable, the const attribute must be used (even for the code space).

Note: const and code qualifiers are independent concepts: the code qualifier does not assume the const qualifier, whereas const can be assigned to any memory type.

```
Example 1:
char code c;
unsigned code int i;
typedef struct { char line;
                int (*fct[3])();
                char *choice[3];
            } smenu;
code smenu menu0, menu1;
```

A character string is qualified by code. The initialization given in example 2 is correct because it corresponds to the assignment of a pointer to a char in « code », to a generic pointer to a « char » (see the section « Implementation / pointers »).

Note: This initialization would not have been allowed if the code qualifier had assumed the const qualifier (ANSI specifies that a pointer to const cannot be assigned to non-const pointer).

```
Example 2:
char *p = "Hello!";
```

2.2.3.2 sfr (Special Function Registers)

The space qualified by `sfr` contains the registers used to address the microcontroller internal peripherals, and is equivalent to the REG space in assembler. Sfrs have restrictions because the space is only directly addressable. This means that:

a `sfr` variable cannot be of a compound type. `sfr` arrays, structures and union, pointer to `sfr` (and `sfr` pointers) are not allowed.

an `&` operator (address of) cannot be applied to a **sfr** variable.

a `sfr` variable cannot be initialized. Example 1 will generate an error.

the `sfr` declaration must be given an absolute storage class attribute that includes the `at` instruction and register address (see the following section).

Example 1:
`sfr at 0xc0 PA = 0xe6;     /* forbidden */`

Example 2:
`sfr at 0xc0 PA;`

Restrictions:

A `sfr` variable cannot be an external variable, since only one storage class can be used.

Only **char** is permitted (long, int, signed, etc. are not allowed). Moreover, **char** type is assigned by default to `sfr`.

The sfrs of most ST6 derivatives are defined in the supplied files. To use them, they only need to be included at the beginning of the program. The `sfr` keyword will rarely have to be used directly.

2.2.3.3 generic

generic applies to pointers exclusively. This keyword is used to indicate that the pointer can point an object in either code or data space. The size of such an object is three bytes because, in addition to the two address bytes, a third byte is used to specify the required memory space.

Example:

```
char generic *ptr1;  /*Generic is the opposite of 'space
qualified'*/
char code ptr2;    / This is pointer to code only */
```

Note: The default qualifier implicitly assigned to any declared pointer can be configured by GENERIC control (see section 5). If a GENERIC control is used all pointers will be processed as GENERIC pointers, until NOGENERIC control is used.

2.2.3.4 Absolute Allocation by at

The **at** keyword allows the absolute address of a variable (or constant) to be specified. It is particularly useful when addressing a **sfr**.

As it is a storage class specifier, **at** cannot be associated with another storage class. The use of **at** with **extern** is not allowed. Absolute allocation applies to global declarations only. The **at** keyword must be followed by an integer constant expression, which must not contain variable addresses. Therefore, the **&** operator is not allowed.

Syntax: **at** integer_value

Example 1:

```
/* declaration using AT statement */
typedef struct { char unsigned
lcr,dll,dlm,lsr,iir,ier,mcr,msr,rbr,thr;
}tUART;
data at 0x10 tUART ImageOfTheUart;
```

If the **at** keyword is followed by a declaration list, the current address is incremented (example2).

Example 2:

```
at 0x02 data char byte0, byte1; /* byte0 at 0x2, byte1 at 0x3 */
```

The **at** keyword can also be used to place a function at a fixed location by defining the address of the first byte of its executable code (example 3).

Example 3:

```
at 0x800 char HighFunc() { /*...*/};
```

2.2.3.5 Function Attributes

Interrupt

The interrupt attribute causes the defined function to be interpreted as an interrupt routine. If an interrupt procedure invokes sub-routines, these do not need to be assigned with the interrupt attribute.

Interrupts must be followed by a constant expression specifying the interrupt number.

Syntax: interrupt interrupt_number

Example:

```
void fct_timer0 (void) interrupt 1 {counter++;}
```

Grammatically, interrupt functions may be set up or typed (void is mandatory).

The associated interrupt vector address is 0xFFC for interrupt #0 (NMI), and:

$$0xFF8 - 2 * IT_number,$$

as shown in the following table:

Vector ID	Address	Description
0	0FFC	NMI
1	0FF6	Port A Pins
2	0FF4	Port B Pins
3	0FF2	

Note:

1. To exit, the RETI instruction is used in place of RET,
2. Main registers are automatically saved at the entry of the interrupt,
3. If an interrupt uses some “common subroutines”, the code of this subroutine will be duplicated if the function is not reentrant (most cases). The only situation for not duplicating a function happens when the function has no local variables/registers.

2.2.3.6 asm instruction

The **asm** instruction allows hexadecimal code to be placed at the current address of the executed code, and therefore provides a limited in-line assembly. Initializations must be of the constant type and the variable addresses are regarded as constants. Each object is assumed to be of 1 byte size, except for external variable addresses which require two bytes. This source text can be thought of as an in-line assembly. However, it is output to the source file generated only when using the SRC control. The source text is not assembled and output to the object file.

Syntax: **asm** assembler_block

```
Example:
extern int fct_error();

void main () {
asm {0x6D}; /* Stop Operations */
...
}
```

Note: Including assembler code can be risky. Great care should be exercised and all necessary checks should be carried out.

Additional information: How to deal with assembler includes within RIDE:

- First, translate the .c file with the .src file as output (select 'src' in 'Options | Project | RCST6 | Object').
- After the translation, include the .src file in your project *after* the .c file.
- In order to link the whole project, have the .src node translated with MAST6 (click the right mouse button and select 'Node properties').

2.2.3.7 Intrinsic functions

Intrinsic functions are an extension to ANSI C, and are provided to give the user a way to access some instructions offered by the architecture that would otherwise never be generated by the compiler (typically bit or byte management stuff).

Intrinsic functions are always inserted ‘in line’ in the generated code (as opposed to accessed via call/ret).

The list of intrinsic functions can be found in the file “intrins.h”, which is located in the “RIDE\INCST6” directory.

Here is a list of the intrinsic functions implemented in RCST6

`void _nop_ (void)`, inserts a nop instruction in the generated code.

The following intrinsic functions do not exist anymore, due to the fact that now “management of the bit” is available:

`_testbit_( b )`, sends back the value of the bit b (from the bit space).

`_chkfloat_( f )`, checks if the float f is correct (!= NaN)

`._?ror_( )`, rotates right the first parameter the number bit position indicated by the second parameter.

? = c for unsigned char

? = i for unsigned int

? = l for unsigned long

`._?rol_( )`, rotates left the first parameter the number bit position indicated by the second parameter

? = c for unsigned char

? = i for unsigned int

? = l for unsigned long

`_getbit( a, n )`, sends back the nth bit of the byte to the address a.

`_putbit( unsigned char BitVal, unsigned char ByteAddr, unsigned char BitOffs )`, sets value BitVal to bit BitOffs of character ByteAddr. BitVal must be a constant and can only be 0 or 1

`_swapnibble_`, swap the 2 nibbles of a char. Thus, 0xCA will become 0xAC

`_swapbyte_`, swap the 2 bytes of a 16-bit word. Thus, 0xCAFE becomes 0xFECA

3. IMPLEMENTATION

3.1 *Memory Models*

3.2 *Linker*

3.3 *The Concept of “Module”*

3.4 *Base Data Types*

3.5 *Data Accesses in the Large Model*

3.6 *Pointers*

3.7 *Semantical Considerations*

3.8 *Parameter Passing Convention*

3.9 *Predefined Macros*

3.10 *Startup code*

3.1 Memory Models

3.1.1 How to select your Memory Model

The ST6 family is quite wide, and presents important variations of the size of the memory spaces such as “ROM/RAM/EEPROM”. However, a quick analysis allows these derivatives to be put in one of the two following groups with:

1. Those that implement a bank switching mechanism (for both data and code).
2. Those that include less than 128 bytes for the RAM and less than 4 KB for the ROM. Here, bank switching is not necessary.

Bank switching for the ROM involves bank switching for the RAM and vice versa. Then, “crossed” cases (i.e. data bank switching without code bank switching) do not have to be taken into account.

This difference concerning the bank switching leads to two memory models:

MEMORY MODEL	DATA BANK SWITCHING	CODE BANK SWITCHING
SMALL	NO	NO
LARGE	YES	YES

Note: The presence of an EEPROM may disturb the above consideration... Indeed, it happens that the management of the EEPROM on small derivatives requires a DATA bank switching without CODE bank switching. In that case, SMALL is the adequate model: The EEPROM will be managed through some dedicated drivers (subroutines that switch temporary the current DATA page).

3.1.2 How the Memory Model influences the Code Generation

In the compiler, the difference in code generation between the LARGE and the SMALL model is relatively small. Different library routines are called to access pointers, or to access data that leads to bank switching. Furthermore, these routines are not different at all if they access pointers to data in code. CODE data is retrieved the same way in the SMALL and the LARGE model. For data stored in the DATA space, however, there is a write-only register specifying the data bank which must be set in the LARGE model. Interrupts must be handled so the register is not lost. These procedures are omitted for data stored in DATA space with the SMALL model.

The most distinct difference in code generation is in the linker. There, depending on the modules, the code is located on different pages. Only two pages are visible at a time: the static page which is visible all the time, plus the page which is currently selected. In order to handle the calls and returns when switching from the currently selected page to a hidden page, *bridges* have to be created through the static page.

3.2 Linker

The RLST6 Linker is more than a simple relocater and it has some specific functions:

Choice of the segmentation (when bank switching is necessary), with creation of gateways.

Organization of the “pseudo-stack”, and duplication of modules to ensure reentrance.

Supervision of the use of the hardware stack (limited to 6 levels).

Global optimization (post in-lining,...).

3.3 The Concept of “Module”

In the LARGE model, the ST6 bank switching has two forms: the ROM (register PRPR) and the RAM (register DRBR). In case of an interrupt, these registers must be saved, set and then restored. But this operation is quite difficult because they all are “read only” registers.

To make the management of these registers easier, the RCST6 compiler gathers the segments DRBR and PRPR in the Module concept. A module means:

- A module is defined by a couple (DRBR, PRPR). Up to 8 modules (8KB of ROM and 192 bytes of RAM) can be present.
- A module is also a group of functions and declarations resulting from (at least) one source file. This means that the user will manage the bank switching by adding in a same file functions that seem to be linked. A module becomes an object including functions and declarations. This could be compared to the notion of “class” in C++.
- When an interrupt occurs, only the Module Identity will be saved and restored.

In the SMALL model, there is only one module. The ROM limit of the modules is:

1. 4 KB of ROM in the SMALL model and
2. 2 KB of ROM in the LARGE model.

The capacity of the LARGE model is, however, 8 modules or 4 code pages. Therefore, there are more modules allowed in the LARGE model, although they have to be smaller than the only module in the SMALL model.

The modules will be grouped, and relocated by the optimizing RLST6 linker.

3.4 Base Data Types

The implementation of base data types is not specified in detail by the C language, and the ANSI standard provides a large choice of formats and word widths. This section lists conventions taken by Raisonance RC-ST6 compiler.

3.4.1 List of base types

- **char** or **char signed**: 8 bits

- **char unsigned**: 8 bits

By default, char are considered as unsigned. This means in particular that `((char)0x40 < (char)0x80)` is true.

- **int** or **signed int**: 16 bits

By default, they are interpreted as **signed**.

- **int unsigned**: 16 bits

3.4.2 Standard word (int) width

The C language assumes that the int type is used as the standard width of words dealt with by the processor.

- This choice is not so well adapted to the ST6 which is an 8-bit microcontroller but has been originally adopted because it corresponds to the standard defined for compilers operating on PC systems, and was influenced by constraints set by the ANSI standard. Specifically, a variable of the char type is first converted to int before it is used in an expression. Theoretically, this conversion (also called promotion) causes the compiler to deal with only 16-bit words, which results in more code generated for lower values.

To simplify this, a 'NOINTPROMOTE' mode is included in which conversions are not systematically performed (see section 4.3.2.1), allowing 8-bit arithmetic.

Furthermore, the usual conversion from 8 to 16 bits often produces useless code that will then be suppressed in optimizing procedures for the generated code, even if the INTPROMOTE control is activated.

3.5 Data Accesses in the LARGE Model

In the LARGE model, access is regulated through several location-specific keywords. Keywords specify exactly where a variable is declared.

3.5.1 Using Keywords to specify Variable Location

3.5.1.1 Syntax

A variable declaration looks like:

type_specifier_list name;

type_specifier_list may contain:

one of : *far, near* (*space attribute*)

one of : *code, data, sdata*, (*space qualifier*)

const, volatile

The default for the space attribute is:

near.

The default for the space qualifier is:

data.

The default for a variable declared *extern* is:

far, data

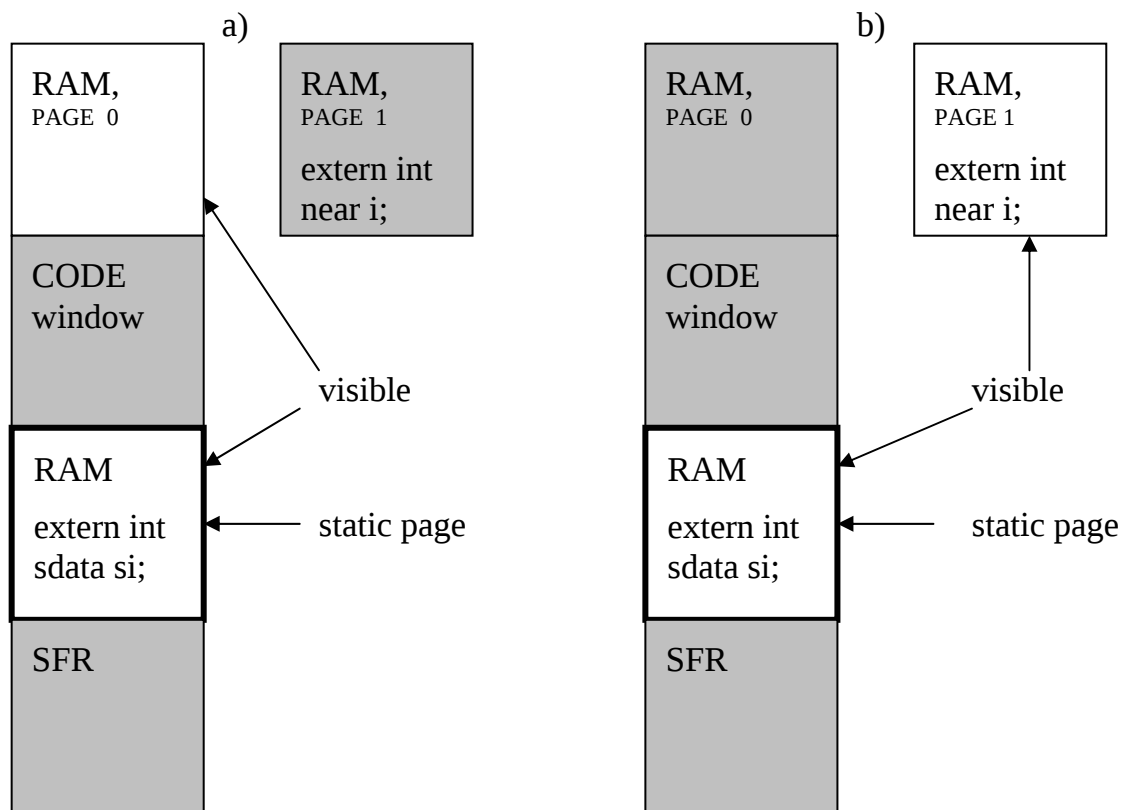
The keywords *far* and *near* refer to the bank switching mechanism. *Far* means that the variable could be located on any page and therefore preliminary page switching has to be done. *near* means that the variable is supposed to be on the currently visible page. Care has to be taken with this supposition in case of *extern* declarations (see the next sections).

Code means that the variable is in the ROM space, *data* and *sdata* mean that it is in the RAM space. *Sdata* specifies in addition where the variable is located in RAM space.

Far code and *near code* do not put any additional meaning into *code*, because the *code* variables are viewed through a *code* window in *data* space, for which the DWR register has to be set anyway.

3.5.1.2 The Meaning of sdata

In the ST6, the data space is split into four parts: The first part is RAM and banking area. It can be switched with another page, set by the DRBR register. The second part is a window into code space which is determined by the DWR register. The third part is RAM again, and it is a slice which is always visible, called the *static page*. This is the area where sdata variables can be found. The last part of data space is reserved to special function registers.



The above drawing shows the data space at two instances. At the instance a), page 0 is selected whereas at the instance b), page 1 is selected. Only in the second instance, the variable *i* is visible. **The variable *si* on the static page is always visible. If a variable is defined with sdata, it is always located on the static page.**

3.5.1.3 The Meaning of `near`

Note: `sdata` and `near` data produce exactly the same compiler output.

In case of local variables which are `near` by default, there is no problem: The linker automatically puts the local or `near` variables on the current module.

In case of `extern` declarations, however, it has to be certain that the variable is on the page the linker decided to put for the current module. Consider the drawing of the previous section. If the variable is not on the current page and it is defined with '`near`', the variable is not accessed correctly (case b)). Which module is chosen by the linker depends on several optimization factors, except it is forced with the pragma `DRSECT('pagenum')`.

The following example could lead to scenario b):

<u>File 1</u>	<u>File 2</u>	
<pre>extern int sdata si; extern int near data i;</pre>	<pre>int sdata si; int near data i;</pre>	←* ok * ←dangerous, maybe not on the same page

This example leads to scenario a):

<u>File 1</u>	<u>File 2</u>	
<pre>#pragma DTSECT(1) extern int sdata si; extern int near data i;</pre>	<pre>#pragma DTSECT(1) int sdata si; int near data i;</pre>	←* ok * ←* ok *, on the same page

3.5.2 Interrupt Handlers

In Large memory model, a specific module must be generated for all interrupt handlers. This is necessary because the interrupt handlers need to put their local variables in static data page due to interrupt process. Therefore, interrupt handlers may not be in the same module as any standard function because this will implicate that the other functions will set their local variables also in static data page without any specific need for this.

3.6 Pointers

3.6.1 General

Two types of pointers are available: space qualified pointers (which point to a given addressable space such as CODE) and generic pointers which can point to any address in any addressable space.

3.6.2 Syntax

A pointer declaration looks like:

```
type_specifier_list_of_pointed_object * type_qualifier_list_of_pointer name ;
```

where:

type_specifier_list_of_pointed_object concerns the pointed object.

type_qualifier_list_of_pointer concerns the pointer itself.

Examples:

```
code int * data cptr;
```

The pointer is in the “data” memory space and points to an int that is in the code space.

```
char * volatile ptr;
```

The pointer is volatile, and points to a character (undefined space).

type_specifier_list_of_pointed_object may contain:

one of: `far`, `near` (*space attribute*)

one of: `code`, `data`, `sdata`, `generic` (*space qualifier*)

`const`, `volatile`

The default for the space attribute is:

1. far for the LARGE model and
2. near for the SMALL model.

The default for the space qualifier is: generic.

If the pragma NOGENERIC is defined, the default is: data.

On the other hand, type_qualifier_list_of_pointer may contain:

one of: `code`, `data`, `sdata` (*space qualifier*)

The default for the space qualifier is data.

Note: Pointers to functions are not concerned by this syntax (see section 3.6.6).

3.6.3 Space qualified Pointers

A pointed object may be qualified by a space qualifier. Four space qualifiers are possible:

1. `code` which represents the internal ROM,
2. `data` which represents the internal RAM,
3. `sdata` representing a part of the internal RAM which is always visible and
4. `generic` which stands for either code or data. Such a pointer will point dynamically onto any address of ROM/RAM.

Case 3 requires some more explication: The writable data space (as well as the code space) is divided into two parts. One part is always visible, and in the LARGE model the other part can be switched with another page. Pointers qualified by `sdata` always occupy space in the visible, the static **data** space.

The data space actually contains a third and a fourth part: the third part is a window into the code space, selected by the DWR register and the fourth part is occupied by special function registers.

In addition, a pointed object may be qualified by two space attributes:

1. `near` where data is accessed on the same code or data page and
2. `far` where data is accessed on a different code or data page.

This qualification is only valid in a model where the data page or the code page can be switched. In the LARGE model there is bank switching and therefore the distinction between near and far exists. In the SMALL model, all pointers are near.

In the LARGE model, a pointer to `data_` can be specified as being either `far` or `near`. If it is far, the right `data_page` has to be selected first. For a pointer to code, however, the choice of the right code page is meaningless: The DWR register has to be set anyway. This register chooses a small slice of code space (64 bytes) which is visible in data space. It is able to address the whole code space, regardless of the page number. Therefore, for a pointer to code, the cases `far code`, `near code` and `code` are equivalent.

3.6.4 Pointers Coding

The distinction between far and near pointers raises the question how much space a pointer address occupies. It occupies either:

1. 1 Byte or
2. 2 Bytes.

The first byte serves for coding bank switching, whereas the second byte sets the offset for indirect addressing. If the pointer points to code space, the DWR register occupies the whole first byte. If it is a pointer to data space, bit number 2 (the 3rd bit) of the first byte chooses the data page (the DRBR register). The second byte represents the offset within the data page or code slice. This offset occupies the 6 least significant bits of the second byte. Bit number 6 (the 7th bit) reveals if the pointer points to code or to data space (code = 1, data = 0).

Considering the pointer coding described above, it is obvious that code pointers need 2 bytes of coding because the first byte sets the DWR register. Pointer to far data also need 2 bytes of coding because the first byte sets the data page. The only case which needs only 1 byte of coding is the case of a pointer to near data.

The registers used for ‘double-registers’ coded with two bytes are X-T or Y-U. Their use is strictly similar. The registers used for pointers coded with one byte are: X and Y. The reason is that X and Y are the only registers which allow indirect access.

The coding can be summarized in the following table:

	Size	Syntax	Description	Model		Usage												
				S	L													
ND	1	char near data*p_nd;	Data of the module, static area.	✓	✓	(X) (Y)												
C	2	char code * p_fc;	Code address: DWR + offset	✓	✓	(X-T) or (Y-U) where T/U = DWR X/Y = 0x40 + offs												
			<table><tr><td>X/Y</td><td>0</td><td>1</td><td>X5</td><td>X4</td><td>X3</td><td>X2</td><td>X1</td><td>X0</td></tr><tr><td>T/U</td><td>0</td><td>0</td><td>D5</td><td>D4</td><td>D3</td><td>D2</td><td>D1</td><td>D0</td></tr></table>				X/Y	0	1	X5	X4	X3	X2	X1	X0	T/U	0	0
X/Y	0	1	X5	X4	X3	X2	X1	X0										
T/U	0	0	D5	D4	D3	D2	D1	D0										
FD	2	char far data * p_fd;	Data address: DRBR + offset		✓	(X-T) or (Y-U) where T/U = coded DRBR X/Y = 0x40 + offs												
			<table><tr><td>X/Y</td><td>0</td><td>0</td><td>X5</td><td>X4</td><td>X3</td><td>X2</td><td>X1</td><td>X0</td></tr><tr><td>T/U</td><td colspan="8">same as generic (module ref)</td></tr></table>				X/Y	0	0	X5	X4	X3	X2	X1	X0	T/U	same as generic (module ref)	
X/Y	0	0	X5	X4	X3	X2	X1	X0										
T/U	same as generic (module ref)																	
G	2	char generic* p_fc;	Global pointer (every banked code and data).		☑	(X-T) or (Y-U) where C / D selects Code: 1 or data: 0 M is either DWR (if code) or the module ref in bit 2 (if data).												
			<table><tr><td>X/Y</td><td>0</td><td>C/D</td><td>05</td><td>04</td><td>03</td><td>02</td><td>01</td><td>00</td></tr><tr><td>T/U</td><td>M7</td><td>M6</td><td>M5</td><td>M4</td><td>M3</td><td>M2</td><td>M1</td><td>M0</td></tr></table>				X/Y	0	C/D	05	04	03	02	01	00	T/U	M7	M6
X/Y	0	C/D	05	04	03	02	01	00										
T/U	M7	M6	M5	M4	M3	M2	M1	M0										

Note: The NULL Pointer is coded “X=0, T=0”

One wonders now how generic pointers can be coded. In fact, as generic pointers can point to either code or data, they always need 2 bytes to describe them. Therefore, in the LARGE model, near generic pointers would be coded similarly to far generic pointers. For these reasons of code size reduction we decided to suppress the category `near generic` and to use `far generic` in the LARGE model. In the SMALL model, of course, far is meaningless. If a generic pointer is pointing to data in the SMALL model, the higher byte (MSB) is meaningless.

3.6.5 Use of Pointers

In fact, as one of our main goals is to reduce the size of the executable at the end, we created only two kinds of accesses to pointers in both the LARGE and the SMALL model: either a direct access to near pointer data (model SMALL and LARGE), coded in-line for now, an access to far generic pointers (model LARGE) with the functions IPF.. and XPF.., or an access to near generic pointers (model SMALL) through the functions IPN.. and XPN... In this manner, the code does not have to be duplicated if there are calls to several kinds of pointers.

This leads to the following table:

	LARGE		SMALL		Representation (Bytes)
Sdata	x	in-line code	x		1
Near Data	x	no call	x		1
Far Data	x	call to	-	call to	2
Far Generic	x	IPF..	-	IPN.. or	2
Near Generic	-	XPF..	x	in-line code	2
[Near Far] Code	x		x		2

with the library functions having the following syntax:

I = imported, X= exported, P = pointer, F = far, N = near, G = generic

	Read (import)	Write (export)	Function
LARGE	IPFG..	XPFG..	sets DWR (import), sets DRBR, interrupt handling
SMALL	IPNG..	in-line	sets DWR (import)

Although there is only one function call for the classes far data, far generic, near generic, [near|far] code in the SMALL and LARGE model respectively, the cases are still semantically different:

```
code char * pc;
generic char * pg;
1. pg = pc; // allowed
2. pc = pg; // not allowed!!
```

The semantic difference is accounted for in the internal representation of the pointers, i.e. the one or two bytes described above. Inside the function, a bit in a test variable is set to one or zero if the pointer points to code (bit 6 of the offset byte) or to data, and according to the bit the preparation for the indirect access is different if the pointer points to code or to data.

Other cases of distinction are treated with wrapping functions: a wrapping function sets or resets a bit in the same test variable and calls the main function. The main function proceeds according to the bit configuration of the test variable. In this way we optimized the code size at the expense of a tiny loss of execution speed.

The test variable contains the following cases:

1. bit 6: code (1) or data (0)
2. bit 4: word (1) or byte (0) (set in wrapping function)
3. bit 5: post-increment (1) of offset or not (0) (set in wrapping function)

As mentioned in the last table, interrupt handling has to be taken care of. This is the case if the data page (DRBR register) is changed and an interrupt occurs. In the case of interrupts, the internal module reference indicating the code and data page is stored in order to be recovered after the interrupt. So if the DRBR register is changed, the module reference has to be updated and the old module reference stored in some other register. At the end of the routine, the old module reference has to be recovered. This is taken care of in the library routines. At one point, the module reference is stored in the three low-order bits of the test variable.

Up to now, all the keywords we described concerned the pointed object. `Extern` is a keyword which describes the pointer itself. In fact, a declaration containing `extern` should rather be written as:

```
int    * extern np;
```

The `extern` keyword expresses that the pointer is in far memory. In the LARGE model, this means that the address of `np` is taken with `#GENPTRH` and `#GENPTRL` and input to `IPF..` for access, or `XPF..` for write. In the SMALL model the DWR register has to be dealt with and `IPN..` is called, or, if the pointer is in data space (`extern data *np`), an in-line access is performed.

If `extern` is called with a variable only instead of a pointer, the same function calls are used for the model LARGE. In the SMALL model, a difference shows because the variable is in data space by default except the keyword `code` is used. If the variable is in data space, it is accessed directly. If it is in code space, the DWR register is changed directly through the `#WINDOW` function, and the data is accessed through the `DATA` function.

3.6.6 Pointers to Functions

3.6.6.1 Implementation

The ST6 does not allow an indirect call to be performed. Nevertheless, the RCST6 compiler mimics indirect calling by implementing a “switch table”. A function vector is implemented with a byte (it makes this mechanism more efficient).

3.6.6.2 Restrictions

The very special way of indirect call simulation sets several limitations:

Not all types of functions are possible. Only functions with less than two bytes for parameters are indirectly callable.

Indirect calls cannot be performed from an interrupt handler.

Up to 255 “indirectly callable” functions are possible (this limitation is due to the in-byte coding).

3.7 Semantical Considerations

In general, assignments can be made from the less general to the more general variable or pointed object. For example, `sdata` is less general than `data`. `data` or `code` is less general than `generic`. `near` is less general than `far`. Different concepts at the same level are not compatible, for example `data` and `code`.

```
char code * pc;
char data * pd;
char sdata * psd;
char generic * pg;
char near * pn;
char far * pf;
```

```
pg = pc;    //allowed
pc = pg;   // not allowed!!
pc = pd;   // not allowed!!
pd = psd;   // allowed
psd = pd;  // not allowed!!
pn = pf;   // not allowed!!
pf = pn;    // allowed
```

Note: From a syntactic point of view, the space qualifiers are managed as “const” and “volatile”. However, the semantic rules are completely different. Indeed, the ANSI specifies that a pointer to a “const” object cannot be assigned to a more general pointer:

```
char const    * pconst;
char          * p;
```

```
p = pconst;    // not allowed!!
pconst = p;      //allowed
```

3.8 Parameter Passing Convention

3.8.1 Rules

A DATA segment is used to pass parameters. This segment contains the automatic variables as well as the parameters. This segment is overlayable, and will be relocated by the RLST6 linker that will analyze the structure of the program to optimize the mechanism of overlay.

Assembly routines need to know the name of this data segment. Names are built automatically in the following way:

Object	Name	Description
Local Segment	?DT??xxxx?mmmm	xxxx is the function name, mmmm the module name
First parameter location	??xxxx?BYTE	xxxx is the function name,

For example,

```
void MyTest ( void )
{
    extern int fct (int);
    fct(3);
}
```

will generate the following code:

```
LDI    ??fct?BYTE, #000H
LDI    ??fct?BYTE + 01H, #003H
CALL   ?fct
RET
```

3.8.2 Interfacing C Programs to Assembler

void func2 (void) will generate the symbol ?FUNC2. The question mark character (?) is prefixed to the function name.

For functions that return a value with a type of a size identical to that of the base type, the value is placed in a register set as illustrated in the next Table.

char (1 byte)	A
int (2 bytes)	A (LSB) and B (MSB)

Location place for values returned by functions

3.9 Predefined Macros

Several identifiers are predefined:

__LINE__	Produces a decimal constant containing the current source line number.
__FILE__	Produces a string literal containing the name of the file being compiled.
__DATE__	Produces a string literal containing the date of compilation, in the form “mm dd yyyy”.
__DATE2__	Produces a string literal containing the date of compilation in the form “mm dd yy”.
__DATE3__	Produces a string literal containing the date of compilation in the form “dd mm yy”.
__TIME__	Produces a string literal containing the time of compilation, in the form “hh:mm:ss”.
__STDC__	Produces the constant « 1 ».
__RCST6__	Produces the version number. Ex: if the version number is 1.02 it returns “102”.
__MEMORY_MODEL__	Produces the constant « 0 » if the current memory model is SMALL , the constant « 1 » if the model is LARGE .

Concerning the **__MEMORY_MODEL__** macro, there exist some definitions in the « include » files of RCST6 about the value associated to the different memory models. These values are: **_SMALL** and **_LARGE**.

3.10 Startup code

The Startup Code contains initializations needed to make your C program work correctly. It is executed at power-up and, when finished, it jumps to the “main” function of your application. An appropriate version of the startup code (depending on some parameters like memory model and others) is linked with the rest of the application when the “main” symbol is found by the compiler and performs a variety of initialization tasks including:

- Initialize the internal memory to 0
- Initialize Static variables, as required by ANSI
- Initialize the timer to obtain a UART frequency of 9600 bauds with typical quartz.

The Startup Code is written in assembler and can be found in the file `ride\incst6\sources\rcst6\startup.st6`. It is recommended not to modify this file; if you need a specific initialization for your project we suggest you perform it at the beginning of the “main” function.

4. PRAGMAS

4.1 Using Pragmas

4.2 Environment Pragmas

4.3 Implementation Controls

4.4 Exercising Control through RIDE

4.1 Using Pragmas

4.1.1 What is a Pragma?

Pragmas are controls passed to the compiler to activate special modes or to drive the code generation. They can be divided into two categories depending on the compilation process they affect:

- environment pragmas that specify the compilation context,
- implementation pragmas that affect the content of the generated code.

4.1.2 Indicating a Pragma

These controls can be passed in the following three ways:

- by indicating them as arguments in the command line,
- by entering them in the source file after the preprocessor `#pragma` command,
- by specifying them in dialogue boxes (RIDE).

In the command line mode, the list of controls should follow the name of the file to be compiled.

Example 1:

```
RCST6 raison.c LARGE PIN(C:\RC\INCLUDE)
```

In the current Raisonance RC-ST6 version, you may specify several controls after the `#pragma` statement.

Example 2:

```
#pragma SMALL  
#pragma SMALL NOINTPROMOTE  
/* several controls may be specified on the same line */
```

Note: Some controls are processed only if they are entered in the command line, or at the beginning of the file prior to any declaration.

Some controls exist in both the positive and negative form (such as PRINT and NOPRINT, INTPROMOTE and NOINTPROMOTE). If both forms are specified in a file or command line, the last one encountered will be assumed, as the command line controls are read before the beginning of the file.

Most of controls have no parameter. But some controls that set definitions accept a parameter to be placed between parenthesis.

Example 3:

```
#pragma pin("\include")
/* Directory including .h files */
```

Note: Previous versions of Raisonance RC-ST6 compiler used a different syntax for control parameters which is still supported by the current version. With this former syntax, example 3 becomes:

```
#PRAGMA PIN="\INCLUDE"
```

Dialogue boxes allow you to automatically generate control by marking boxes. For example, the following dialogue box will generate controls given in example 4.

Example 4:

```
#pragma PRINT
#pragma COND
#pragma SYMBOLS
#pragma CODE
#pragma PAGEWIDTH(60)
#pragma PAGELENGTH(80)
```

4.1.3 List of Pragmas

Where both positive and negative forms exist, the underlined controls are the default ones. The controls with an asterisk (*) can be used only in the command line or at the beginning of the source file, the others can be included within the source code, but always at the global declaration level, that is before a function, and not inside it.

Control name	Abbr.	Description
[END]ASM	-	Copies a block of RC-ST6 file in the assembler source file.
[NO]CODE	[NO]CD	Displays the assembly code in the source file.
[NO]COND	[NO]CO	Displays conditional blocks in the source file not affected by compilation in the listing file.
[NO]DEBUG	[NO]DB or [NO]DG	Displays debugging information in the object file.
EJECT	EJ	Adds a form feed in the listing file.
LISTINCLUDE	-	Displays the content of the include files in the listing file.
OBJECT(ofn)	OJ(ofn)	Specifies the object file name. The file name is optional.
[NO]OBJECTEXTEND	[NO]OE	Includes additional information in the object file.
PAGELength(Num)	PL(Num)	Specifies line numbers per page in the listing file.
PAGEWIDTH(Num)	PW(Num)	Specifies the number of characters per line in the listing file.
PATHINCLUDE(dirlist)	PIN(dirlist)	Defines the default directory of include files.
[NO]PRINT	[NO]PR	Specifies the creation of a listing file.
SRC(fn)	-	Generates an assembler source file rather than an object file. The SRC file name is optional.
[NO]SYMBOLS	[NO]SB	Displays a list of symbols in the listing file.

Environment pragmas

Control name	Abbr.	Description
DEFINE	DF	Macro definition and macro parameters definition.
DEFINEJOKER	DEFJ	Allows the definition of public symbols of the number type.
[NO]DUPLICATESTRING	[NO]DS	Identical strings are duplicated.
[NO]GENERIC	[NO]GEN	Specifies that all pointers with no space qualifier are regarded as generic pointers.
[NO]INITSTATICVAR	[NO]IS	Specifies that global or static variables must be initialized.
[NO]INTPROMOTE	[NO]IP	Specifies the use of ANSI integer promotion rules.
LARGE	LA	Specifies the use of the large memory type model.
OPTIMIZE	OT	Performs optimization.
[NO]OPTICONSTFOLD	[NO]OCSF	Same as OT(xx,0)
[NO]OPTIPEEPHOLE	[NO]OPPH	Same as OT(xx,6)
[NO]OPTISUBX	[NO]OSBX	Same as OT(xx,5)
[UN]SIGNEDCHAR	-	Specifies: default char type is signed.
SMALL	SM	Specifies the use of the small memory type model.
RESTORE	-	Restores control settings.
SAVE	-	Saves control settings.

Implementation pragmas

4.2 Environment Pragmas

Environment pragmas are used to control the generation and the structure of the listing file and the source file.

4.2.1 Reference Directory

The PATHINCLUDE control redefines the reference directory which contains the include (.h) files. The reference directory must be specified by the string immediately following PATHINCLUDE statement. The default directory is “\RIDE\include” which is created during the installation procedure. When a relative path is used, remember that the current directory is the directory from which compilation has been invoked.

Syntax: **PATHINCLUDE**(path)

Abbreviation: PIN

Example: #pragma PIN(\include)

Note: This control comes from previous Raisonance RC-ST6 compilers. It may as well be used with previous Raisonance compiler syntax.

Thus, the example above becomes:

#pragma PIN(\include) RCST6 raison.c PIN=\include
--

4.2.2 Listing File

4.2.2.1 Generating Listing Files

The PRINT control specifies that a listing file is created. Its name may be specified by the string following PRINT statement with .LST extension. The default listing file name is the source file name with .LST extension. If the NOPRINT control is entered at the beginning of the file (or on the command line), the listing file will not be generated even if the PRINT control occurs in the file. Default value is PRINT.

Syntax: **PRINT**(listing_file_name.LST)
 PRINT
 NOPRINT

Abbreviation: **PR**
 NOPR

Example 1: RCST6 raison.c NOPRINT

Example 2: #pragma PRINT(\list\lst2.lst)

Note: In former Raisonance RC-ST6 compilers the listing file was generated (resp. not generated) when using PRINTINCLUDE control (resp. NOPRINTINCLUDE). This syntax is still supported.

4.2.2.2 Listing of Include Files and Conditional Blocks

The LISTINCLUDE control is used to add the contents of the include files to the listing file. By default the listing file does not include those files.

Syntax: LISTINCLUDE

```
Example 1:  
RCST6 raison.c LISTINCLUDE
```

```
Example 2:  
#pragma LISTINCLUDE
```

The COND control is used to write source file conditional blocks (blocks built on **#if #endif**) not affected by compilation to the listing file. The NOCOND control has the opposite affect and thus prevents source file conditional blocks not compiled from appearing in the listing file.

Syntax: COND
NOCOND

Abbreviation: CO
NOCO

```
Example:  
/* source file */  
#if 0  
printf("displayed if COND");  
#endif  
#if 1  
printf("Always displayed ");  
#endif  
/* command line and corresponding listing file */  
RCST6 raison.c COND  
#if 0  
printf("displayed if COND");  
#endif  
#if 1  
printf("Always displayed ");  
#endif  
/* command line and corresponding listing file */  
RCST6 raison.c NOCOND  
#if 1  
printf("Always displayed ");  
#endif
```

4.2.2.3 Listing Mnemonics and Symbols

The SYMBOLS control is used to display the list of all symbols used in and by the module compiled in the source file. For each symbol is given its class, memory type, offset and size.

Syntax: **SYMBOLS**

Abbreviation: **SB**

Example 1: RCST6 raison.c SYMBOLS

The NOSYMBOLS control has the opposite effect.

Syntax: **NOSYMBOLS**

Abbreviation: **NOSB**

The CODE control is used to display the assembly code corresponding to the source file in the listing file.

Syntax: **CODE**

Abbreviation: **CD**

Example 2: RCST6 raison.c CD

The opposite control prevents the assembly code display.

Syntax: **NOCODE**

Abbreviation: **NOCD**

Note: In the former versions of Raisonance RC-ST6 compiler, the CODE control was used to generate an object file. Object file generation is now performed by the OBJECT control (see section 4.3.2.3).

4.2.2.4 Controlling the Listing File Format

The PAGEDLENGTH control is used to specify the number of lines per page in the listing file. PAGEDLENGTH must thus be followed by a number chosen between 1 and 65535 and must be enclosed in parenthesis. The default number is 60.

Syntax: **PAGEDLENGTH**(line_number)

Abbreviation: PL

PAGEDLENGTH may be used with 0 as argument which means that no formatting will be done.

Example 1: #pragma PL(55)

The PAGEDWIDTH control is used to specify the number of characters per line in a listing file page. PAGEDWIDTH must be followed by a number chosen between 78 and 132 and must be enclosed in parenthesis. The default number is 120.

Syntax: **PAGEDWIDTH**(number)

Abbreviation: PW

Example 2: RCST6 raison.c PW(90)

The EJECT control is used to add a form feed at the current line in the listing file. It cannot be used in the command line.

Syntax: **EJECT**

Abbreviation: EJ

Example 3: #pragma EJECT

4.2.3 Object or Assembler Source File

An object file is systematically generated by the Raisonance RC-ST6 compiler (which calls Raisonance MA-ST6 assembler) unless SRC control is used. In this latter case an assembler source file is generated instead of the object file and you may then assemble this one using Raisonance MA-ST6 assembler.

4.2.3.1 Assembler Source File

When using the SRC control an assembler source file name is thus generated. The source file name with .SRC extension may be specified by a string following SRC statement. The current module base name with .SRC extension is used by default if no name is specified.

Syntax: **SRC**(assembler_source_file_name)
 SRC

```
Example 1:  
RCST6 raison.c SRC(\as\assourc.src)
```

```
Example 2:  
#pragma SRC
```

You may explicitly rewrite a block of your current RCST6 file in the assembler source file by using ASM control. It allows you to write some parts of your RC file in assembler. This assembler part must begin with ASM control and end with ENDASM.

Syntax: **ASM**
 ENDASM

```
Example 3:  
/* example of RCST6 file using ASM control */  
main ( )  
{ printf("This program will indefinitely loop");  
#pragma ASM  
  jmp $;  
#pragma ENDASM  
}
```

4.2.3.2 Object file

By default (when SRC control is not used) an object file is automatically generated. Its name may be specified with the OBJECT control followed by the name you have chosen, ended by .OBJ extension. The RC-ST6 current file name with .OBJ is used by default.

Syntax: **OBJECT(object_file_name)**

Abbreviation: OJ

```
Example 1:  
RCST6 raison.c OBJECT(\objec\rais.obj)
```

```
Example 2:  
#pragma OJ("\\objec\rais.obj")
```

Debugging information may be added to the object file by using the DEBUG control. This information is: local and global variable names and addresses, function names and line numbers.

Syntax: **DEBUG**

Abbreviation: DG

```
Example 4:  
RCST6 raison.c DEBUG
```

To avoid debugging information to be added to object file, NODEBUG control may be used. The default value is NODEBUG.

Syntax: **NODEBUG**

Abbreviation: NODG

```
Example 5:  
#pragma NODEBUG
```

4.3 Implementation Controls

4.3.1 Memory

4.3.1.1 Models

Two memory models are available. Selecting the right model is not a complex question, it only corresponds to the question:

“Does the target derivative implement bank switching?”

If NO, SMALL model will be the best solution.

If YES, LARGE model is mandatory

To sum up, consider that the LARGE model should be selected for any derivative that offers more than 4 KB of CODE (or more than 128 bytes of DATA).

The memory model will be selected with a pragma, but it depends on the chosen derivative (bank switching or not). However, it is possible to select the SMALL model and to manually manage the bank switching mechanism.

A few additional remarks:

Fortunately, CODE banking always goes with DATA banking. Therefore, we can move aside any crossed case.

DATA banking does not mean that “DRBR exists”. Indeed, we consider here 128 bytes of DATA memory space, 64 bytes of static memory and 64 bytes of internal RAM used for variables. We do NOT consider as “DATA” the EPROM (if any), nor the RAM used by the LCD controller.

SMALL model

The SMALL model is reserved for derivatives that do not require bank switching mechanism. PRPR and DRBR registers (when they exist) will be preset by the startup program. The compiler will assume that they keep for ever the same value.

Note: Accessing the EEPROM contents is still possible, but will require a special driver which must assume (when the DRBR is modified) that interrupts will be masked. The case of a NMI service also has to be handled with care.

LARGE model

LARGE model allows CODE and DATA bank switching simultaneously. Nevertheless, the RCST6 compiler will assume that both DRBR and PRPR are constant inside a Module (a Module corresponds to the contents of a source file). It means that the user will be partially responsible for the banking map by organizing his files... Several modules should be merged by the linker to optimize both the code size and the execution speed (by reducing the number of bridges between the banks).

Model	Description	
SMALL	<u>Bank Switching</u>	No. Only DWR is managed.
	<u>Memory</u>	4K code 128 bytes DATA Those spaces are supposed to be “permanent”
	<u>EEPROM</u>	If it exists, it has to be driven by the user in import/export functions to replace DRBR on the RAM (that is supposed to be permanent).
LARGE	<u>Bank Switching</u>	Both on code (PRPR) and data (DRBR)
	<u>Memory</u>	512 bytes RAM+EEPROM 4 X 2KB of code (up to 8KB)
	<u>EEPROM</u>	If it exists, it has to be driven by the user in import/export functions to replace DRBR on the RAM (that is supposed to be permanent).

Concerning the access to the EEPROM, this is not managed by the RC-ST6 Compiler. It is advised to write a specific driver including a reading function a writing function.

Every interrupt must be inhibited or you must save DRBR and restore it afterwards. This register is write-only, in the SMALL memory model, the value to save is the value selecting the unique RAM page if it exists. In the LARGE memory model, the global variable CUR_MOD, defining both PRPR and DRBR must be restored. Do to do this, call the ?C?SET_PAGE function (i.e.: “CALL ?C?SET_PAGE”).

Note: The compilation model indicates the stack location for the automatic variables (refer to the above chart). However, the temporary savings of registers (required, for example, for the calculation of complex expressions) are always carried out in internal stack.

The Raisonance RC-ST6 compiler does NOT support combined memory models. The linker will automatically detect the selected model, and will act differently depending on the model. If a different model is used, an error will be generated.

Libraries are automatically chosen according to their type but also to the memory model. The following table gives the name of those libraries.

Memory model	Name
SMALL	RCST6S
LARGE	RCST6L

Libraries

4.3.1.2 Variables and Constants

A char type variable, where none of the attributes signed and unsigned are specified, has a default type which is not defined by the ANSI standard. On RC-ST6, this default type is unsigned. It is possible to change this default type by using the `SIGNEDCHAR` or `UNSIGNEDCHAR` controls. The control `SIGNEDCHAR` imposes signed as the default type for the char whereas the `UNSIGNEDCHAR` control keeps unsigned as the default value.

Syntax: **SIGNEDCHAR**
 UNSIGNEDCHAR

Example 1: #pragma SIGNEDCHAR

Note: By default, char variables are interpreted as unsigned, which means that `((char)0x80 > (char)0x40)` is true.

The ANSI standard imposes that a global or static variable must be initiated to 0 when no initiator is specified. `INITSTATICVAR` and `NOINITSTATICVAR` controls allow you to control and carry out those initializations.

Syntax: **INITSTATICVAR**

Abbreviation: **IS**

Example 2: RCST6 raison.c IS

As the ANSI standard requires, the IS control is active by default.

Syntax: **NOINITSTATICVAR**

Abbreviation: **NOIS**

The MA-ST6 assembler allows the definition of `PUBLIC` symbols of the type `NUMBER`. In contrast to other symbols, the `NUMBER` symbol is not associated

with a physical microcontroller segment and does not cause memory space to be allocated. It is regarded as a numerical constant that can be shared by several files.

Another advantage of the NUMBER symbols is that they can be used to replace external labels. The linker processes them as « wild cards », because they can resolve external references to any physical space.

As the C language does not normally provide this useful feature, the DEFINEJOKER control eliminates the need to use the assembler, therefore removing this disadvantage.

Syntax: **DEFINEJOKER**(name=value)

Abbreviation: **DEFJ**(name=value)

The value must be an integer constant expression.

Example 5: RCST6 raison.c DEFINEJOKER(MY_CST=0xE8)\

Note: The former versions of the Raisonance RC-ST6 compiler used a different syntax which is still supported in the current version.

With this former syntax, **example 5** becomes:

RCST6 raison.c DEFINENUMBERPUBLIC=MY_CST/0xE8
--

4.3.2 Optimization

4.3.2.1 Propagation

In order to create a more portable program, it may be interesting to enable ANSI integer promotion rules, which are allowed by INTPROMOTE control.

Syntax: **INTPROMOTE**

Abbreviation: **IP**

Example 1:

```
{
  char   unsigned c1= 0x80, c2=0x81;
  int    i;
  i = c1+c2;
}
```

/* In this example, i will be 0x101 on exiting the block, if the INTPROMOTE option is selected and 1, if not, since the c1+c2 operation is an operation between 8-bit characters.*/

The NOINTPROMOTE control prevents integer promotion rules from functioning. This is imperious with the ST6 microcontroller which is a 8-bit microcontroller.

Syntax: **NOINTPROMOTE**

Abbreviation: **NOIP**

Note: The former versions of Raisonance RC-ST6 compiler used PROMOTE and NOPROMOTE control to specify integer promotion rules. This syntax is still supported.

RC-ST6 language is a superset of C current language and you may want to disable all language extension. NOEXTEND control provides you with this possibility.

Syntax: **NOEXTEND**

Example 2: RCST6 raison.c NOEXTEND

The opposite control exists and is taken as the default.

Syntax: **EXTEND**

4.3.2.2 General

Optimization may be performed employing the OPTIMIZE control. The optimization level is specified by a number following the OPTIMIZE statement. Each optimization level contains lower optimization characteristics. The optimization type may be controlled and may be chosen between SPEED and SIZE.

Syntax: **OPTIMIZE**(opt_level [,opt_type])

Abbreviation: OP

Optimization level	Optimization description
0	1. Expressions are reduced to constants when possible. 2. Internal data addresses access is optimized. 3. Jumps optimization.
1	1. Unused blocks are deleted. 2. Conditional jump optimization.
2	Overlayable segments are identified for the linker.
3	Complex moving operations are optimized.
4	1. Function arguments are located in registers when possible. 2. Elimination of intermediate registers for code operations. 3. Elimination of repetitive local sub-expression. 4. Optimization of case and switch statements.
5	1. Elimination of repetitive global sub-expression. 2. Loop optimization.
6	Loop modification to improve performance.

Optimization levels for OPTIMIZE control

Example 1:
 RCST6 raison.c OPTIMIZE(6,SPEED)

Note: The former versions of Raisonance RC-ST6 compiler used a different syntax which is still supported in the current version. In this former versions only 2 types of optimizations were performed: size or speed optimizations which were respectively obtained by OPTIMIZESIZE and

OPTIMIZESPEED. The opposite controls NOOPTIMIZESIZE and NOOPTIMIZESPEED are still supported as well.

The DUPLICATESTRING control allows you to specify that strings of identical characters will be duplicated in the CODE space.

Syntax: **DUPLICATESTRING**

Abbreviation: DS

A single allocation will be assigned and the address will be re-used if you use NODUPLICATESTRING. By default, strings will not be duplicated.

Syntax: **NODUPLICATESTRING**

Abbreviation: NODS

```
Example 2:
#pragma NODUPLICATESTRING
char *p1 = "Hello";
char *p2 = "Hello";
/* p1 and p2 will be initialized with the same value*/
```

The GENERIC control (activated by default) allows any pointer with no space qualifier to be regarded as a generic pointer.

Syntax: **GENERIC**

Abbreviation: GEN

If the NOGENERIC control is activated, an unqualified pointer will be assumed to be pointing to the default space. In this case, the generic keyword should be entered to obtain a generic pointer. To avoid errors it is best to always work in the same mode. The GENERIC option is an easier, but less efficient mode.

Syntax: **NOGENERIC**

Abbreviation: NOGEN

```
Example 3:
#pragma SMALL
#pragma GENERIC    /*default value*/

char * data ptr1
/* ptr1 will be treated as a generic pointer (3-byte length)
The pointer itself will be assigned to the data memory */
```

4.3.2.3 Defining the Values of Macro Parameters

The `DEFINE` control may be used either in the source file or in the command line and does not have the same signification in both cases.

In the source file the `DEFINE` control allows the definition of macros (without arguments) which could have been defined by the preprocessor `#define` directive. It cannot be used in the command line and is not preceded by a `#pragma` statement.

Syntax: **define**

Abbreviation: **df**

Example 1:
`#define p(val) printf(#val "\n")`

In the command line this control is used to externally control conditional compilation (for example `#ifdef`).

Syntax: **DEFINE**(param_name/param_value)
 DEFINE(param_name=param_value)

Abbreviation: **DF**

Example 2:
/* command line */
RCST6 raison.c DEFINE(MICRO, 0x6201)

/* contents of raison.c */ #ifdef (MICRO)
#define MICRO 0x6203
#endif

#if (MICRO == 0x6201)
/* Port initialization */
/* ... */
#elif (MICRO == 0x6203)
/* other initialization */
/* ... */
#endif

4.3.2.4 Saving and Restoring Current Settings

Current settings of OPTIMIZE and GENERIC controls may be saved using SAVE control.

Syntax: *SAVE*

The use of both SAVE and RESTORE controls allows you to isolate blocks within your source file.

Syntax: *RESTORE*

Example 1:
#pragma SAVE
... /* isolated block */
#pragma RESTORE

4.4 Exercising Control through RIDE

In the environment RIDE, some of the pragmas mentioned above can be defined through dialog boxes and fields. Also, some constants and variables can be set at the beginning of a debugging session which are usually not accessible through pragmas. Let us consider some important dialog boxes here. For a complete documentation, see the on-line help of RIDE.

- 1) Under “Options | TarGet”, a window pops up with choices of microcontrollers and their availability of RAM and ROM.
- 2) Under “Options | Project ”, a window pops up with choices concerning the environment, the directories, the compiler (RCST6) , the assembler (MAST6) and the linker (RLST6).

4.4.1 The Compiler RCST6

- “Options | Project | RCST6 | Source” determines if struct, union or enum is optional in a variable declaration.
 - Under “ Options | Project | RCST6 | Code generation | Generic”, the ‘Generic’ box corresponds to the GENERIC/NOGENERIC pragma for the default space of pointed objects. If the ‘Generic’ box is not crossed, the default is ‘Data’.
 - The “ Options | Project | RCST6 | Code generation | Unsigned character” box shows if char variables are declared signed or unsigned by default. On the RCST6, the box is checked by default. This corresponds to the SIGNEDCHAR/UNSIGNEDCHAR pragma.
- “Options | Project | RCST6 | Defines” defines constants used in #ifdef with DEF or DEFINE(NAME=VALUE), separated by semicolons.
- The directory “Options | Project | RCST6 | Listing” contains several controls concerning the output format.
- “Options | Project | RCST6 | Object” determines the object output.
- “Options | Project | RCST6 | Memory Model” chooses the memory model, LARGE or SMALL. This choice is important as it allows bank switching or not!
 - “Options | Project | RCST6 | Optimizer” chooses the optimization model.
- “Options | Project | RCST6 | Messages” sets the warning level and the number of errors and warnings which make the compilation stop.
 - 0 = no warning
 - 1 = shows some important warnings
 - 2 = shows all warnings

- Sometimes the only reason the compilation stops is that there are too many warnings! In this case, the number of warnings has to be set higher.
- “Options | Project | RCST6 | QCW” is an internal pragma.

4.4.2 The linker RLST6

- In the linker, an important variable to customize to one’s own needs is the maximum number of printf arguments in “Options | Project | RLST6 | Linker”. This is the maximum number of arguments except the first string containing the formatting characters, times two. If this variable is not adjusted, printf will not work properly.

5. LIBRARIES

5.1 string.h

5.2 stdio.h

5.3 stdlib.h

5.4 ctype.h

5.5 regst6x.h

5.1 string.h

List of the implemented functions:

```

char *strchr (char *s, int c);
/* returns a POINTER on last occurrence of c in the character
string s, or NIL if c is not found */
char *strcat(char *p1,const char *p2 );
/* concatenation of p2 at the end of p1, returns p1 */
char *strncat (char *p1,const char *p2,size_t n);
/* concatenation of maximum n characters at the end of p1. The
character '\0' is added and p1 is returned */
size_t strpos (const char *s,int c);
/* search for the first occurrence of the character c in s
returns its position (or -1 if not found) */
char *strncpy(char *p1, const char *p2,size_t n);
/* copies a maximum of n characters of p2 into p1. If the length
of p2 is smaller than n, the string is completed by '\0'
characters. Returns p1. */
char *strcpy (char *p1,const char *p2);
/* copies p2, including '\0', into p1. Returns p1 */
char signed strcmp (const char *p1,const char *p2);
/* compares p1 and p2. Returns a negative value if p1<p2, and a
positive value if p1>p2 and 0 if p1=p2 */
char signed strncmp (const char *p1,const char *p2, size_t n);
/* compares the first n characters of p1 and p2. Returns a
negative value if p1<p2, a positive value if p1>p2, and zero if
p1=p2 */
size_t strlen (const char *s);
/* returns the length of s */

char *strchr(char *s,int c);
/* returns a pointer on the first occurrence of c in s, and NULL
if character is not found */
char *strrchr(char *s,int c);
/* returns a pointer on the last occurrence of c in s, and NULL
if character is not found */

#define memcmp(s1, s2, n)strcmp((char*)s1, (char*)s2,n)
#define memcpy(dest,src,n) strncpy ((char*)dest, (char*)src,n)
#define memchr(s,c,n)strchr ( (char*) s, c, n)

```

All pointers to the characters defined above are **generic** pointers.

5.2 stdio.h

List of implemented functions:

```
typedef unsigned size_t;
int getchar ( void );
int ungetchar ( int );
char *gets ( char *s );
int putchar ( const int c );
int puts ( const char *s );
int printf(const char *format, ...);
int sprintf(char *buffer, const char *format, ...);
```

These functions use the default input/output mode defined by:

- getchar
- putchar
- ungetchar
- _C_INITIO

At start up, the input / output for the library functions is performed via the ST6 UART serial port which is initialized by the _C_INIT_IO function.

The standard input/output mode can be redefined by creating a set of four functions with the same names, in a file. The name must be placed before RCST6x.LIB in the file list given to the linker.

Example:
RLST6 my_file.obj, my_flux.obj

5.2.1 putchar

putchar sends a character in the standard input/output mode.

5.2.2 getchar

getchar is used for data input and returns a character received in the standard input/output mode. A buffer of one character allows the use with the ungetchar function.

5.2.3 ungetchar

ungetchar replaces the ungetc function. It replaces a character read by getchar in a buffer. This character can then be read by getchar (see getchar source in section 6.2.2).

5.2.4 sprintf and printf

5.2.4.1 General

The printf function invokes sprintf.

The usual formatting rules are observed.

To construct the formatted element, the sprintf function requires a stack buffer. The buffer size is set by "SIZEBUFPRINTF" NUMBER. The number is initialized at 25 in the RCST6S.LIB and RCST6T.LIB (SMALL and TINY models) and at 50 in the RCST6L.LIB (LARGE and HUGE models). It is possible to change the value by re-declaring another value in one file only (see DEFINEJOKER control).

Example: RCST6 serie.c LARGE OE defnp=SIZEBUFPRINTF/15

5.2.5 sscanf

The sscanf function is NOT implemented in the RCST6 libraries.

5.3 stdlib.h

The following standard functions are available:

```
int atoi (const char *s);  
/* converts s into int */  
int abs (int n);  
/* returns the absolute value*/
```

5.4 ctype.h

The standard functions are defined by the following macros:

```
/* Macros corresponding to the first table*/
#define isalnum@ (_flag1@ & (_IS_DIG | _IS_UPP | _IS_LOW))
/* returns a value other than zero on an alphabetic or numerical
character */
#define isalpha@ (_flag1@ & (_IS_UPP | _IS_LOW))
/* returns a value other than zero on an alphabetic character */
#define isascii@ ((unsigned)@ < 128)
/* returns a value other than zero on an ASCII character*/
#define isdigit@ (_flag1@ & _IS_DIG)
/* returns a value other than zero on a numerical character*/
#define isupper@ (_flag1@ & _IS_UPP)
/* returns a value other than zero on an upper case alphabetic
character */
#define islower@ (_flag1@ & _IS_LOW)
/* returns a value other than zero on a lower case alphabetic
character */
#define isxdigit@ (_flag1@ & (_IS_DIG | _IS_HEX))
/* returns a value other than zero on a numerical hexadecimal
character*/
#define isspace@ (_flag2@ & _IS_SPA)
/* returns a value other than zero on a space character*/
#define iscntrl@ (_flag2@ & _IS_CNT)
/* returns a value other than zero on a control character*/
#define isprint@ (@ >= 0x20 && @ <= 0x7e)
/* returns a value other than zero on a printable character */
#define ispunct@ (_flag2@ & _IS_P)
/* returns a value other than zero on a punctuation character */
#define toascii@ (@ & 0x7f)
#define toupper@ (@ + ('A' - 'a'))
```

These macros use two functions (-flag1 and -flag2) which return a value provided by a reference array located in CODE. To optimize the space of this array, it has been partitioned in two 128-byte blocks. Each block consists of an array of 256 ‘nibbles’. The first block (associated with -flag1) corresponds to the most commonly used functions. The second block will only be loaded if either of the isspace, ispunct or iscntrl functions is called.

5.5 regst6x.h

These files contain the sfr declarations of many popular ST6 derivatives. If a component is not included in this list, a new file can easily be created by copying and adapting one of the existing files such as regST6.h.

6. APPENDICES

6.1 Error Messages

6.1 Error Messages

The error messages are classified depending on the type and cause.

6.1.1 Error types

A warning is generated if any abnormalities are detected in the program that may have been caused by a programming error. The warning indicates forms allowed by the standard but:

- with a style that does not conform to good programming rules. For example: assignment of an integer to a variable of the enumeration type.
- with a highly probable programming error. For example: the “if(a=b)” expression often corresponds to an entry error where ‘=’ is replaced by “==”.

An error does not abort the compilation, but suspends code generation.

A fatal error causes immediate termination of the compilation. The most likely cause is generally a system problem such as a full disk or memory space.

6.1.2 Causes

System errors refer to the environment. Perhaps a file cannot be found, or the memory or hard disk is full. System errors are often fatal errors.

Preprocessor errors are either lexical errors (a character not allowed by the language is encountered) or errors within a preprocessor control (macro redefinition etc.).

Compilation errors can be both syntactic and semantic errors (for example, two structures cannot multiply each other).

6.1.3 System errors

Fatal S000: Insufficient memory

PC-Base memory is used up.

Fatal S001: 'xxx' file not found

The indicated file cannot be found either because it does not exist or because the access path is invalid.

Fatal S002: Write error on file

The disk space may be used up.

Fatal S003: Read error on file

Fatal S004: No argument list

The name of the file to be compiled may exist in the command line as an argument and no argument has been found.

Fatal S005: Illegal Extension 'xxx'

The files to be compiled cannot have the OBJ and LST extensions.

Fatal S006: Cannot open 'xxx' file

Check that FILE exists in the CONFIG.SYS file. Check the space available on the disk, and the number of files in the current directory.

Fatale S007: Insufficient EMS memory

EMS memory size on your system is insufficient.

Fatale S008: User break

You stop the compile process with CTRL-C or Cancel (RIDE).

6.1.4 Preprocessor errors

Error P000: **Line too long**

The length of the source file line cannot exceed 256.

Error P001: **Incorrect end of line**

A character string is not closed (“i” is missing).

Error P002: **Invalid character**

The character is invalid for an identifier, keyword etc.

Error P003: **Invalid ‘xxx’ character**

The character (hexadecimal value) is invalid for an identifier, keyword etc.

Error P004: **Bad character constant definition**

An incorrect use of the quotation marks or ‘\xxx’ sequence.

Error P005: **Control unknown**

An unrecognized command line control or pragma. Refer to the list of valid controls.

Error P006: **Bad file specification**

The file to be included is incorrectly specified.

Error P007: **Unexpected ENDIF**

#endif out of scope.

Error P008: **Unexpected ELSE**

#else out of scope.

Error P009: **Unexpected ELIF**

#elif out of scope.

Error P010: Incorrect EOF

The 'include' file is incorrectly terminated.

Error P011: Interleaved 'mac' macro

Recursive macros are not allowed.

Error P012: Wrong number of parameters

At invocation, the number of parameters does not match the definition.

Error P013: Predefined function unknown

The only authorized function is defined().

Error P014: Different redefinition of macro

Only identical redefinition are permitted.

Error P015: Redefinition or suppression of predefined macro not allowed

Predefined macros can be neither suppressed nor redefined.

Error P016: Invalid numerical constant

Incorrect numerical value.

Error P017: Incorrect number format**Error P018: Invalid octal character**

A non-octal character has been encountered.

Warning P019 : Integer constant out of range

Values exceeding the accepted numerical value range.

Error P020: \x not followed by an hexadecimal digit**Error P021: Incorrect EOF**

An end of file is encountered in a comment, character string etc.

Error P022: **Use of DEFINED without argument**

Error P023: **Incorrect use of control ‘control’**

Verify syntax associated to ‘control’.

Error P024: **Missing macro name after define**

Error P026: **Expected string**

Error P028: **Duplicate formal parameter ‘symb’**

One of the formal parameters in the macro definition is duplicated.

Error P029: **Cannot remove predefined macro ‘name’**

‘name’ identifies a predefined macro that it is not allowed to remove.

Error P030: **Macro body cannot start or end with ‘##’**

Error P031: **New line expected extra characters found.**

6.1.5 Compilation errors

Error C000: ':' missing

A colon is missing in a labeled instruction.

A colon may be missing after the operator.

',' missing

An instruction or declaration must be terminated by a semi-colon.

A for instruction must have two semi-colons.

{' missing

A left brace in an initialization list or at the beginning of a block is missing.

}' missing

A right brace in an initialization list, at the end of an instruction block or structure field declaration is missing.

(' missing

A left parenthesis in the control expression of a conditional or iterative instruction is missing.

A left parenthesis in a parameter list is missing.

)' missing

A left parenthesis in the control expression of a conditional or iterative instruction is missing.

A left parenthesis in a parameter list is missing.

,' missing

The elements of an initialization list must be separated by a comma.

The function parameters must be separated by a comma.

] ' missing

A right square bracket in the array subscript expression is missing.

Error C001: ‘type’ specifier not allowed in the context

Type specifiers cannot be combined.

A structure, union or enumeration declaration must not include a type specifier.

Error C002: Too many types

Most of the type specifiers are non compatible: long char is not allowed (but long int is permitted).

Error C003: Undefined ‘symb’ union or structure

The symb union or structure is not defined.

The reference to a structure type that without structure type definition is only allowed for a pointer definition.

Error C004: Undefined ‘symb’ enumeration

The symb enumeration is not defined.

Error C005: Type error in the ‘symb’ redeclaration

The variable is redeclared with a type different from the first declaration.

The function is re-declared or defined with a type different from the first declaration.

Error C006: ‘symb’ defined more than once

The designated identifier has been declared more than once.

An identifier cannot be declared as both extern and static.

An identifier cannot be declared both as extern and local variable.

Error C007: Invalid type in the ‘symb’ definition

A function can return neither a function nor an array.

The type returned by the function does not match the function declaration type.

Error C008: Parameter list longer than the sample

The number of parameters in the function definition is higher than in the sample declaration.

Error C009: Parameter list smaller than the sample

The number of parameters in the function definition is smaller than in the sample declaration.

Error C010: ‘symb is not a parameter

The identifier exists in the parameter list but there is no corresponding declaration.

Error C014: Undefined ‘symb’

The symbol encountered is not declared.

The symbol is declared as a type but is used as a variable.

Error C015: ‘Op’ unary operators not followed by an operand

The ++ and --- unary operators must immediately be followed by an operand.

Error C017: Cast not followed by an operand

A type cast must immediately be followed by an operand.

Error C018: ‘symb’ variable cannot be subscripted

symb is not of the array type.

Error C019: Bad subscript of ‘symb’ array

The subscript expression for symb array is invalid.

Error C022: ‘->’ Requires a structure or a union

The -> operator can apply to a pointer to a structure or union only.

‘.’ Requires a structure or union

The dot ‘.’ operator can only apply to a structure or union.

Error C023: A field must be an identifier

The (.) point operator must be followed by an identifier.

Error C024: Non existent ‘symb’ field

The selected field does not belong to the structure.

Error C025: Wrong number of arguments

At function invocation, the number of actual arguments does not match the number of formal arguments in the function declaration.

Error C026: Bad actual argument type

At function invocation the actual parameter cannot be converted to the formal parameter type given in a previous declaration.

Error C029: ‘switch’ requires integer operand(s)

The switch instruction test expression must be of the integer type : char, short, int, long or enum.

‘oper’ requires integer operand (s)

The binary <<, >>, &, ^, | operators require operands of the integer type.

Error C030: ‘oper’ requires arithmetic operands

The binary *, /, %, +, -, ==, != operators require operands of the arithmetic type.

A sum of pointers is not allowed.

Pointer/non-pointer comparison is not allowed (except for 0 constant).

Error C031: Types invalid for ‘oper’ operator

The * indirection operator cannot be used with a pointer to void.

The switch instruction control expression is invalid.

The & operator cannot be used with a variable qualified by sfr.

The operand of the ~ operator must be of the integer type.

The operand of the !, ++ and—operators must be of the scalar type.

The operand of the + and - unary operators must be of the arithmetic type.

The operand of the ++ and—unary operator cannot be qualified by const or code.

Error C032: Divide by zero

The fraction denominator in a constant expression is evaluated as a null constant.

Error C033: Type not permitted for bit field

A bit field declaration can be of unsigned type only.

The default type of a bit field declaration is char unsigned.

Error C034: Bit field too long

The size of a bit field is limited to 1 byte.

Fatal C036 : Too many errors

The number of errors is limited to 40 by default. This value can be modified using the MAXERR control.

Error C037: Too many warnings

The number of warnings is limited to 40 by default. This value can be modified using the MAXWAR control.

Error C038: Type qualifier not allowed with a function

A function declaration cannot include a type qualifier.

Error C039: Type returned by 'symb' is invalid

A function can return neither a function nor an array. The type of the expression returned by the function does not match the type of the declaration.

Error C040: Type conversion impossible

Type cast cannot be performed.

In an assignment, the right operand type cannot be converted to left operand type.

Warning C042: Condition always false!

The control expression of a conditional or iterative instruction is evaluated as a null constant expression.

Warning C043: Condition always true!

The control expression of a conditional or iterative instruction is evaluated as a non null constant expression.

Error C045: Invalid function call

A bad function invocation.

Error C046: Type qualifier not allowed on automatic variables

Automatic variables cannot be qualified.

Error C047: Invalid initialization

A static variable must be initialized by a constant expression.

A list can initialize an array, a structure or a union only.

The initialization type does not match the type of the variable.

Error C048: Too many initializations

The number of initializations is higher than the number of elements in the array or structure.

The array is initialized by too long a character string.

Error C049: Assignment of a constant

In an assignment, the left operand cannot be a constant.

Error C050: Invalid operation for pointers

An operation on pointers with different space qualifiers.

Error C052: Operator requires an operand value

In an assignment, the left operand of an operator must be addressable.

The operand of an & address unary operator must be addressable.

The operand of a ++ or—unary operator with prefix or suffix must be addressable.

Error C053: No absolute allocation

Absolute allocation by at requires an address.

Error C054: Too many storage classes in the declaration

A declaration must include one storage class specifier only.

Error C055: Requires a constant expression

The array size must be a constant expression.

The case instruction value must be a constant expression.

The bit field length in the structure declaration must be a constant expression.

The explicit enumeration values must be constant expressions.

The initializations of static variables must be constant expressions.

Error C059: Requires a void function

The interrupt and using (setting) attributes can be used only with functions of the void type.

Error C060: Too many type qualifiers

A declaration can include one type qualifier only.

Error C061: No initialization

A variable declared in code or const must be initialized.

Error C062: Illegal type qualifier

sfr qualifiers are not permitted with compound types.

The type specifiers permitted for the sfr qualifier are char and int only. Default type is char.

Error C063: Illegal operation for generic pointers

Addition and subtraction on pointers to void are not allowed.

Error C064: Premature EOF

Premature end of file.

Possible cause :

*/ at the end of a comment is missing.

A quotation mark (“) at the end of a character string is missing
A conditional compilation (beginning by #if) is not closed (by #endif).

Error C066: Case duplicated

The case instruction can have one value only.

Error C067: ‘default’ repeated

A switch instruction must contain one default instruction only.

Error C068: Unexpected ‘break’

Break outside of a switch instruction or loop.

Unexpected ‘continue’

Continue outside of a switch instruction or loop.

Error C069: Unexpected ‘case’

case outside of a switch instruction or loop.

Unexpected ‘default’

Default outside of a switch instruction or loop.

Error C070: Requires a function without parameter

The interrupt and using (setting) attributes can be used only with a function without parameters.

Error C071: A void function cannot return a value

The function is declared as void but it contains a return instruction with a value.

Error C072: Void type not allowed

A variable or parameter declaration of the void type is not allowed.

An array declaration of the void type is not allowed.

A value cannot be assigned to a void variable.

Error C073: 'symb' redefinition

A parameter cannot have more than one declaration.

Error C074: Invalid declaration syntax

A syntax error in the declaration.

Error C075: '(' missing in 'instr'

A misplaced left parenthesis in the instruction test expression.

Error C076: ')' missing in 'instr'

A misplaced right parenthesis in the instruction test expression.

Error C077: Invalid expression syntax

A syntax error in the analysis of an expression.

Error C078: Invalid 'case' instruction

The constant expression which must follow the case instruction is missing.

Invalid 'default' instruction

The default instruction cannot be followed by an expression.

Invalid 'goto' instruction

The goto instruction must be immediately followed by an identifier.

Error C079: Invalid type for a declaration in a register

An array, structure or union in register cannot be declared.

Warning C080: Addressing of a 'register' variable

Addressing a variable declared in a register will cancel this specification (the & operator cannot be applied to a register variable).

Error C081: An identifier is missing

A variable declaration must include an identifier.

A type declaration must include an identifier.

Error C082: Undefined or null array size

The size of a single dimension array may be omitted in an extern declaration.

The size of an array may be omitted in a function parameter declaration.

If multidimensional arrays are used, the limit of all dimensions except the first one must be indicated.

Error C084: Non initialize-able variables

You cannot initialize :

a function variable

an extern variable

a typedef.

parameters

variables qualified by sfr.

Error C085: Invalid type qualifier

In an assignment or initialization, both operands for a local variable do not have the same type qualifier.

The actual and formal parameters of a function do not have the same type qualifier.

The type of the expression returned by a function does not have the same type qualifier as the function type.

Differences or comparisons on pointers with different type qualifiers are not allowed.

Warning C086: Assignment of integer to enumeration

Initialization of a enum variable with an integer type value.

Assignment of an enum variable with an integer type value.

Error C087: Identified bit field is of null size

A bit field declaration that contains an identifier cannot be of null size.

Error C088: 'symb' function defined more that one

A function cannot have more than one definition.

Error C089: A function must return a value

A non **void** function must return a value.

Warning C090: Call of ‘symb’ function without a sample

The function has been invoked without prior definition or declaration.

Error C091: Invalid parameter declaration

A syntax error in the parameter declaration of a function.

Warning C092: ‘symb’ is declared but badly not used

The identifier is declared at the beginning of the block but not used within the block body.

Error C093: ‘do’ without ‘while’

The do instruction must be followed by a while instruction.

Error C094: An expression is missing

The constant expression that declares the size of a bit field is missing.

The constant expression that declares the enumeration constant values is missing.

The constant expression that must immediately follow the attributes is missing.

Error C095: ‘instr’ without ‘;’

An instruction must be terminated by a semi colon.

Warning C096: The function must return a value

A function must contain a return instruction.

Error C097: Undefined ‘symb’ label

The block contains a goto at symb label, but the label definition has not been found.

Warning C098: ‘goto symb’ causes piping of the initializations

The symb label is defined in a block that contains initializations. A goto symb instruction that is out of scope will ignore these initializations.

Error C099: 'symb' label duplicated

A label cannot be defined more than once.

Error C100: Operand not allowed with '&' operator

A bit field cannot be addressed.

Error C101: Invalid type control expression

The control expression of a conditional or iterative instruction must be of the scalar type.

Error C102: Invalid indirection on void pointer

The (*) indirection cannot be used on a void pointer.

Error C103: Misplaced 'else' control

An **else** instruction without a matching if is invalid.

Error C104: Invalid parameter declaration for 'symb' function

The parameter type in the sample declaration of the function is missing.

The parameter declaration or name in a function definition is missing.

Warning C105: Probable incorrect assignment

The control expression of an iterative or conditional instruction is an assignment (this may be a write error on '=' symbol).

Error C106: Operand of undefined or null size

The operand of the sizeof operator is of undefined or null size. Change the operand declaration to determine the size.

Error C107: 'sizeof' operator cannot be applied to a function

You cannot invoke sizeof with a function

'sizeof' operator cannot be applied to a bit field

You cannot invoke sizeof with a bit field.

Error C108: Unexpected '}'

The source file contains one right brace too much.

Fatal C109: Unexpected instruction

Warning C110: Assignment of generic pointer from code

In some cases, it is dangerous to assign a code pointer to a generic pointer.

Warning C111: Operation with void type

It is dangerous to do operation between *void* pointers.

Warning C112: Control ‘control’ duplication ignored

Control ‘symb’ is duplicated. This occurrence is ignored.

Error C113: Control ‘control’ parameter must be an integer

Fatal C114: Unknown control

Unknown control found on command line. Compile process is stopped.

Fatal C115: ‘(’ after control expected

Fatal C116: ‘)’ after control expected

Fatal C117: Respecified or conflicting control

Fatal C118: Bad decimal number

A parameter value given to a control on command line is an invalid numerical constant.

Fatal C119: Out of range number

A parameter value given to a control on command line is out of range.

Fatal C120: Identifier expected

Fatal C122: Macros too nested

Compiler cannot expand macros.

Fatal C123: **Cannot have general control on invocation line**

Warning C124: **Unknown #pragma, line ignored**

Error C125: **asm/endasm requires src-control to be active**

Error C126: **Misplaced primary control, line ignored**

This control cannot be there in the source file. It must be placed before any inclusion of file.

Error C127: **Misplaced control 'control', line ignored**

This control cannot be there in the source file. It must be placed before any inclusion of file.

Warning C128: **'symb' warning unreachable, control ignored**

Error C129: **Unclosed string**

Error C130: **Unclosed comment**

Error C131: **Unbalanced #if-endif controls**

Number of #if and number of #endif are not identical.

Error C132: **Bad integer expression**

Error C133: **'param' unexpected in a formal list**

'param' is not valid here.

Error C134: **'symb' : Pointer to field**

Warning C135: **Interrupt() may not receive or return value(s)**

interrupt service functions must be VOID et cannot have any parameter. Those two conditions are assumed to continue the compile process.

Error C137: Bit members not allowed in struct/union

A field of a structure or union cannot be of bit type. Use bit field specifiers.

Error C138: ‘symb’ self-referential structure or union

Error C139: type follows void

You cannot specify parameters after keyword void in a function declaration.

Error C140: void invalid

void type cannot be found there.

Error C141: ‘symb’ : function returns function

A function cannot return a function.

Error C142: ‘symb’ : function returns array

A function cannot return an array.

Error C143: Struct/union comparison illegal

Comparison between two structures or two unions is illegal. Uses memcmp function.

Error C144: ‘symb’ : duplicate struct/union/enum tag

struct/union/enum tag is already used for something else

Error C145: ‘symb’ not an union tag

Error C146: ‘symb’ not a struct tag

Error C147: ‘symb’ : duplicate struct/union member

Error C148: ‘=’ : Illegal type conversion from/to ‘void’

Type conversions from or to void have strict rules. Refer to a C language documentation.

Error C149: **Non-address/-constant initializer**

You cannot initialize with something else than a constant or an address.

Error C150: **String out of bounds**

Error C1ST6: **Switch expression has illegal type**

A switch condition must be integer type.

Error C152: **Mspace illegal on struct/union member**

Error C154: **asm/endasm not allowed in include file**

Error C155: **Cannot open 'file'**

Error C156: **Not a lvalue**

Error C157: **'*' illegal indirection**

Error C158: **sizeof returns zero**

Size of sizeof parameter is unknown.

Error C159: **Missing 'c'**

Error C162: **Syntax error near 'symb'**

Error C164: **'typedef' and 'at' illegal combination**

You cannot use in the same declaration typedef and at.

Error C165: **'num' : absolute specifier illegal**

num is out of bounds to qualify the space.

Error C169: Conflicting memory models

Definition and declaration of the function have not the same memory model.

Error C173: Function member in struct/union

A struct/union member cannot be a function. Use function pointers.

Error C177: Illegal pointer conversion**Warning C179: 'symb' mspace on parameters ignored**

Space qualifier on a parameter has no sense, because it is the memory model of the function that fixed the memory space.

Error C180: Unable to create temporary file

Too many files are open on your PC. Close those that are not useful. If it is not the case, verify that the environment variable CST6TMP has a correct value.

Error C181: Unable to write temporary file

Too many files are open on your PC. Close those that are not useful.

Error C182: 'symb' : function redefinition**Error C183: 'symb' : function definition requires ANSI style parameter list****Fatal C184: File 'File' doesn't exist**

Compiler cannot found the file.

Error C186: 'symb' : function redefinition (different attribute)

Check if declaration and definition of 'symb' have the same attributes.

Error C187: 'symb' : function redefinition (different memory models)

Check if declaration and definition of 'symb' have the same memory models.

Fatal C188: System error 'message'

Message must be helpful for the system error.

Warning C190: 'code' qualifier not allowed here : ignored

Warning C191: **Inconsistent comparison with ‘unsigned’**

Warning C192: **Assignment between pointers to char signed and unsigned**

You try an assignment between a pointer to signed char and one to unsigned char. You must check this operation.

Warning C193: **Assignment from generic to typed pointer**

In this operation, you will loose the genericity, is this what you really want?

Error C194: **‘symb’ is not a valid identifier**

Error C195: **‘DATA’ : segment too large**

Error C197: **‘CODE’ : segment too large**

Fatal C200: **Evaluation version : maximum object size reached (val Kb)**

You are using an evaluation version which maximum object size is reached. Contact us to have a complete version.

Error C201: **Missing type qualifier**

The variable type is not specified.

Warning C202: **‘symb’ : pointer different mspace**

This is an operation between qualified pointers on data. Is this really what you want ?

Error C203: **‘symb’ : pointer : different mspace**

This is an operation between qualified pointers on two memory spaces not compatible.

Error C204: **Pointer : const to non-const assignment**

A const pointer cannot be assigned to a non-const pointer.

Error C206: Cannot modify 'code' variable

By definition a code variable is read only. So it cannot be modified.

Warning C207: Suspicious pointer conversion

Perhaps do you need to check this conversion?

Error C208: 'symb' a function without parameter is required**Error C209: 'symb' : return value via register bank****Error C210: Illegal pointer conversion**

7. INDEX

- ?C?SET_PAGE,54
- Absolute allocation,19
- ANSI**,13, 15
- asm,21, 51
- Base data types,26
- CD**,49
- char,26
- char signed,26
- char unsigned,26
- CO,48
- code,16, 49
- COND,48
- const,16, 17
- ctype.h,71
- CUR_MOD,54
- data,16
- DEBUG**,52
- DEFINEJOKER**,56
- DEFJ**,56
- DF**,61
- DG**,52
- DRBR,54
- DS**,60
- DUPLICATESTRING**,60
- EEPROM,54
- EJ**,50
- EJECT**,50
- ENDASM**,51
- Error Messages,74
- examples,11
- EXTEND,58
- function,20
- GEN**,60
- generic,19, 60
- generic pointers,30
- Getchar,68
- global,12
- GLOBAL**,15
- IMPLEMENTATION,23
- INITSTATICVAR**,55
- int,26
- int unsigned,26
- Interrupt,12
- INTPROMOTE**,57
- Intrinsic functions
 - _chkfloat_(f),22
 - _crol_(), _irol_(), _lrol_(),22
 - _cror_(), _iror_(), _lror_(),22
 - _getbit(a, n),22
 - _nop_,22
 - _putbit(v, a, n),22
 - _swapbyte_,22
 - _swapnibble_,22
 - _testbit_(b),22
- IP**,57
- IS**,55
- Keywords,15
- Reserved keywords,15
- LIBRARIES,65
- Linker**,25

Macros,39
 Memory Model,53
 LARGE,53
 SMALL,53
 Memory Models,24
Module,25

NOCD,49
NOCO,48
NOCODE,49
NOCOND,48
NODEBUG,52
NODG,52
NODS,60
NODUPLICATESTRING,60
NOEXTEND,58
NOGEN,60
NOGENERIC,60
NOINITSTATICVAR,55
NOINTPROMOTE,57
NOIP,57
NOIS,55
NOOPTIMIZESIZE,59
NOOPTIMIZESPEED,59
NOPR,47
NOPRINT,47
NOPROMOTE,57
NOSB,49

OBJECT,52
OJ,52
OP,59
OPTIMIZE,59
OPTIMIZESIZE,59
OPTIMIZESPEED,59

PAGELength,50
PAGEWIDTH,50
 Parameter passing convention,38
PATHINCLUDE
 PRAGMA,46
PATHINCLUDE,46
PIN,46
PL,50
 Pointer
 Pointer in ‘code’ memory,17
 Pointers,30

PR,47
pragmas,41, 42
 Pragmas,42
PRINT,47
 Printf,68
PROMOTE,57
 Pragma,26
 putchar,12, 68
PW,50

regst6x.h,72
 RESTORE,62

SAVE,62
SB,49
 sdata,16
 sfr,16, 18
 Space qualifier,18
 signed int,26
SIGNEDCHAR,55
 Pragma,55
 sources,12
 space qualified pointers,30
 Space qualifiers,16
SRC,51
 sscanf,69
 startup,12
Startup code,39
 static,12
stdio.h,65
 stdlib.h,70
string.h,65, 66
SYMBOLS,49

ungetchar,68
UNSIGNEDCHAR,55

version,7
 volatile,16

Warning,74
 wchar,15

8. Bibliography

8.1 About C language

The reference book for C language is traditionally « The C programming language » by Kernighan and Richie.

The C programming language

Kernighan & Richie

Prentice-Hall

ISBN: 2-225-82070-8

This book is made up of an analytical description of the language as well as an educational one. Nevertheless many books, written by other authors, certainly more educational, are as well available.

8.2 About ST6 microcontrollers

For more details on a particular device, refer to data books available at ST distributors, or on the WEB at: <http://www.st.com>