C51 User Manual

Introduction

About C51

C51 is an open source optimizing C compiler for the 8051/8052 family of microcontrollers. C51 is part of CALL51, which includes also the assembler A51, the linker L51, and the librarian Lib51. C51 is a fork of another open source C compiler, SDCC version 2.9.7 from May 2010.

Disclaimer

This document is derived form the manual of SDCC version 2.9.7 from May 2010, therefore a significant part of the original SDCC manual is used unchanged or paraphrased in this document, and I don't claim those parts as my own work, except of course, for those parts of the SDCC manual that I wrote myself. Otherwise,

Typographic conventions

Throughout this manual, the following conventions are used: commands typed in by the user are printed in **bold.** Code samples are printed in `fixed` width font. Relevant words are printed in *italic.*

System Requirements

As of today C51 runs only under Microsoft Windows XP, Vista, and 7.

Using C51

Supported Data Types

C51 supports the following data-types:

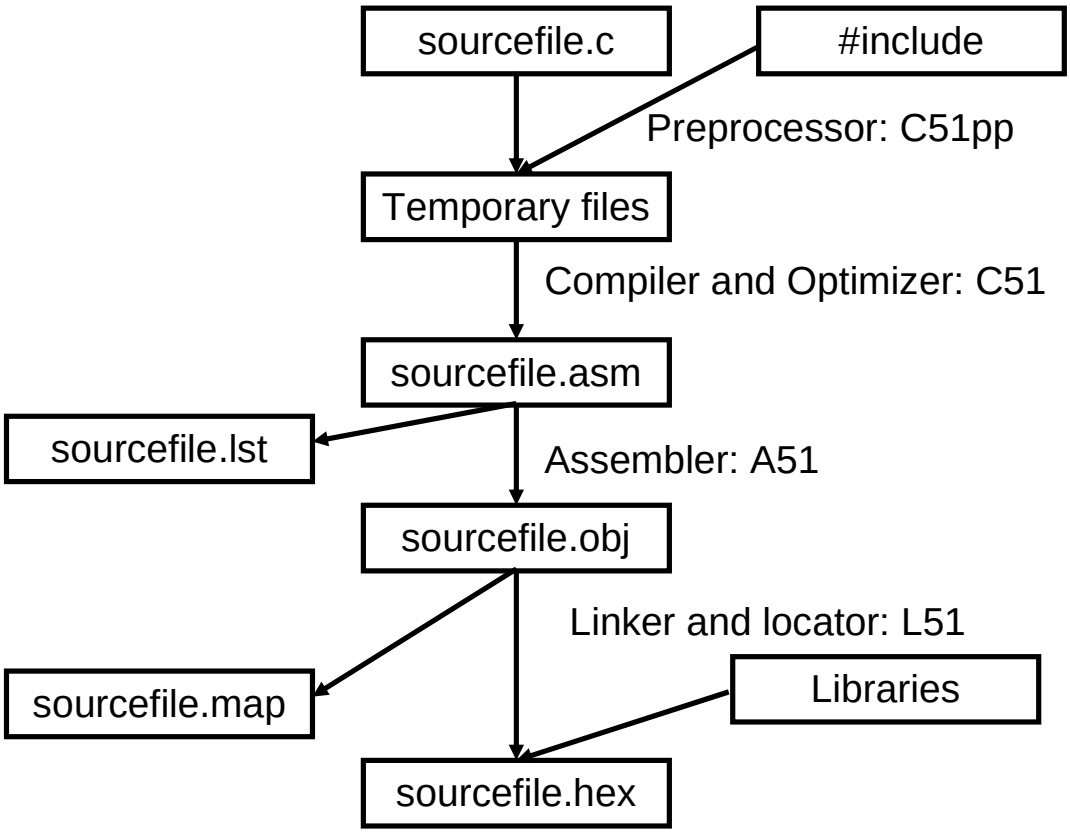
type	width	default	signed range	unsigned range
bit	1 bit	unsigned	-	0, 1
char	8 bits, 1 byte	signed	-128, +127	0, +255
short	16 bits, 2 bytes	signed	-32.768, +32.767	0, +65.535
int	16 bits, 2 bytes	signed	-32.768, +32.767	0, +65.535
long	32 bits, 4 bytes	signed	-2.147.483.648, +2.147.483.647	0, +4.294.967.295
float	4 bytes IEEE 754	signed		1.175494351E-38, 3.402823466E+38
pointer	1, 2, or 3 bytes	generic		

Single Source file Projects

For single source file 8051/8052 projects the process is very simple. Compile the C source code file with the following command **"c51 sourcefile.c".** This will compile, assemble, and link **"sourcefile.c"**. The compilation process generates several output files:

- *sourcefile.asm* – Assembler source file created by the compiler.
- *sourcefile.lst* – Listing file created by the Assembler.
- *sourcefile.obj* – Object file created by the assembler, input to Linker.
- *sourcefile.map* –The memory map for the load module, created by the Linker.
- *sourcefile.hex* –The load module in Intel hex format.
- *sourcefile.lkr* – Linker script.

The figure below shows the compilation process of "sourcefile.c".



Projects with Multiple Source files

C51 can compile only one file at a time. Assume for example that you have a project containing the following files:

foo1.c (contains some functions)
foo2.c (contains some more functions)
foomain.c (contains more functions and the function main)

The first two files will need to be compiled separately with the commands:

C51 -c foo1.c
C51 -c foo2.c

Then compile the source file containing the *main()* function and link the files together with the following command:

C51 foomain.c foo1.obj foo2.obj

Alternatively, *foomain.c* can be separately compiled as well:

C51 -c foomain.c
C51 foomain.obj foo1.obj foo2.obj

The file containing the *main()* function should be the first file specified in the command line, since the linkage editor processes file in the order they are presented to it. The linker is invoked from C51 using a script file with extension .lkr. You can check this file to troubleshoot linking problems such as those arising from missing libraries.

Projects with Additional Libraries

Some reusable routines may be compiled into a library. Libraries created in this manner can be included in the command line. Make sure you include the -L <library-path> option to tell the linker where to look for these files if they are not in the current directory. Here is an example, assuming you have the source file *foomain.c* and a library *foolib.lib* in the directory *mylib* (if that is not the same as your current project):

C51 foomain.c foolib.lib -L mylib

Note here that *mylib* must be an absolute path name.

The most efficient way to use libraries is to keep separate modules in separate source files. To create a library file use Lib51. To display a list of options supported by Lib51 type:

Lib51 -?

To create a new library file, start by compiling all the required modules. For example:

C51 -c divsint.c
C51 -c divuint.c
C51 -c modsint.c
C51 -c moduint.c
C51 -c mulint.c

This will create files divsint.obj, divuint.obj, modsint.obj, moduint.obj, and mulint.obj. The next step is to add the .obj files to the library file:

Lib51 libint.lib divsint.obj
Lib51 libint.lib divuint.obj

Lib51 libint.lib modsint.obj
Lib51 libint.lib moduint.obj
Lib51 libint.lib mulint.obj

Or, if you preffer:

Lib51 libint.lib divsint.obj divuint.obj modsint.obj moduint.obj mulint.obj

If the file already exists in the library, it will be replaced. If a list of .obj files is available, you can tell Lib51 to add those files to a library. For example, if the file 'myliblist.txt' contains

divsint.obj
divuint.obj
modsint.obj
moduint.obj
mulint.obj

Then using Lib51, add the list to a library, for example:

Lib51 -l libint.lib myliblist.txt

Additionally, you can instruct Lib51 to compile the files before adding them to the library. This is achieved using the environment variables LIB51_CC and/or LIB51_AS. For example:

set LIB51_CC=C51 -c
Lib51 -l libint.lib myliblist.txt

To see what modules and symbols are included in the library, options -s and -m are available. For example:

Lib51 -s libint.lib

```
divsint.OBJ:
  __divsint,R_CSEG,0000,0000

divuint.OBJ:
  __divuint_PARM_2,R_OSEG,0000,000
  __divsint_PARM_2,R_OSEG,0000,000
  __divuint,R_CSEG,0000,0000
```

```
modsint.OBJ:
  __modsint,R_CSEG,0000,0000

moduint.OBJ:
  __modsint_PARM_2,R_OSEG,0000,000
  __moduint_PARM_2,R_OSEG,0000,000
  __moduint,R_CSEG,0000,0000

mulint.OBJ:
  __mulint,R_CSEG,0000,0000
  __mulint_dummy,R_CSEG,0000,0000
  __mulint_PARM_2,R_OSEG,0000,0002
```

Command Line Options

Preprocessor Options

C51 uses an adapted version of the GNU preprocessor *cpp* (*gcc* http://gcc.gnu.org/). Some of the most commonly used pre-processor options are listed below. If you need more specific options than those listed below please refer to the CPP manual at:

http://gcc.gnu.org/onlinedocs/cpp/

-I<path>

The additional location where the preprocessor will look for <.h> or ".h" files.

-D<macro[=value]>

Command line definition of macros.

-Aquestion(answer)

Assert the answer 'answer' for question, in case it is tested with a preprocessor conditional such as '#if #question(answer)'. '-A-' disables the standard assertions that normally describe the target machine.

-U<macro>

Undefine macro. ‘-U’ options are evaluated after all ‘-D’ options, but before any ‘-include’ options.

-Wp Option[,Option]...

Pass the Option to the preprocessor c51pp.

Linker Options

-L --lib-path <absolute path to additional libraries>

This option is passed to the linkage editor’s additional libraries search path. The path name must be absolute. Additional library files may be specified in the command line. See section on compiling programs for more details.

--xram-loc <Value>

The start location of the external ram, default value is 0. The value entered can be in Hexadecimal or Decimal format, e.g.: --xram-loc 0x8000 or --xram-loc 32768.

--code-loc <Value>

The start location of the code segment, default value 0. Note when this option is used the interrupt vector table is also relocated to the given address. The value entered can be in Hexadecimal or Decimal format, e.g.: --code-loc 0x8000 or --code-loc 32768.

--data-loc <Value>

The start location of the internal ram data segment. The value entered can be in Hexadecimal or Decimal format, eg. *--data-loc 0x20* or *--data-loc 32*. By default, the start location of the internal ram data segment is set as low as possible in memory, taking into account the used register banks and the bit segment at address 0x20. For example if register banks 0 and 1 are used without bit variables, the data segment will be set, if *--data-loc* is not used, to location 0x10.

--idata-loc <Value>

The start location of the indirectly addressable internal ram of the 8051, default value is 0x80. The value entered can be in Hexadecimal or Decimal format, for example: --*idata-loc 0x88* or --*idata-loc* 136.

--bit-loc <Value>

The start location of the bit addressable internal ram of the 8051/8052.

Compiler/Linker Options

--model-small

Generate code for Small Model programs, see section Memory Models for more details. This is the default model.

--model-medium

Generate code for Medium model programs, see section Memory Models for more details. If this option is used all source files in the project have to be compiled with this option. It must also be used when invoking the linker.

--model-large

Generate code for Large model programs, see section Memory Models for more details. If this option is used all source files in the project have to be compiled with this option. It must also be used when invoking the linker.

--iram-size <Value>

Causes the linker to check if the internal ram usage is within limits of the given value.

--xram-size <Value>

Causes the linker to check if the external ram usage is within limits of the given value.

--code-size <Value>

Causes the linker to check if the code memory usage is within limits of the given value.

--acall-ajmp

Replaces the three byte instructions lcall/ljmp with the two byte instructions acall/ajmp. Only use this option if your code is in the same 2k block of memory. You may need to use this option for some 8051 derivatives which lack the lcall/ljmp instructions. Also, if the derivative you are using lacks the acall/ajmp instructions, you will need to rebuild the libraries with this function.

Optimization Options

--nogcse

Will not do global sub-expression elimination, this option may be used when the compiler creates undesirably large stack/data spaces to store compiler temporaries (spill *loc*ations, sloc). A warning message will be generated when this happens and the compiler will indicate the number of extra bytes it allocated. It is recommended that this option NOT be used, #pragma nogcse can be used to turn off global sub-expression elimination for a given function only.

--noinvariant

Will not do loop invariant optimizations; this may be turned off for reasons explained for the previous option. It is recommended that this option NOT be used; #pragma noinvariant can be used to turn off invariant optimizations **for** a given function only.

--noinduction

Will not do loop induction optimizations; see section strength reduction for more details. It is recommended that this option is NOT used; #pragma noinduction can be used to turn off induction optimizations for a given function only.

--nojtbound

Will not generate boundary condition check when switch statements are implemented using jump-tables. It is recommended that this option is NOT used; #pragma nojtbound can be used to turn off boundary checking for jump tables for a given function only.

--noloopreverse

Will not do loop reversal optimization.

--nolabelopt

Will not optimize labels (makes the dumpfiles more readable).

--no-xinit-opt

Will not memcpy initialized data from code space into xdata space. This saves a few bytes in code space if you don't have initialized data.

--nooverlay

The compiler will not overlay parameters and local variables of any function, see section Parameters and local variables for more details.

--no-peep

Disable peep-hole optimization with built-in rules.

--overlayto <segment_name>

The compiler will overlay parameters and variables into the overlayable data segment <segment_name> for all functions and variables in the current file. Make sure that none of the functions in the file calls a function that uses the same overlayable segment name. This option is intended to be used with libraries or projects where files contain a single function.

--peep-file <filename>

This option can be used to use additional rules to be used by the peep hole optimizer. See section 'Peep Hole optimizations' for details on how to write these rules.

--peep-asm

Pass the inline assembler code through the peep hole optimizer. This can cause unexpected changes to inline assembler code, please go through the peephole optimizer rules defined in the source file tree '<target>/peeph.def' before using this option.

--opt-code-speed

The compiler will optimize code generation towards fast code, possibly at the expense of code size.

--opt-code-size

The compiler will optimize code generation towards compact code, possibly at the expense of code speed.

Other Options

-v –version

Displays the C51 version.

-c --compile-only

Will compile and assemble the source, but will not call the linker L51.

--c1mode

Reads the preprocessed source from standard input and compiles it. The file name for the assembler output must be specified using the -o option.

-E

Run only the C preprocessor. Pre-process all the C source files specified and output the results to the standard output.

-o <path/file>

The output path where everything will be placed or the file name used for all generated output files. If the parameter is a path, it must have a trailing slash (or backslash for the Windows binaries) to be recognized as a path. If the path contains spaces, it should be surrounded by quotes. The trailing backslash should be doubled in order to prevent escaping the final quote, for example: *-o ”F:\Projects\test3\output 1\\”* or put after the final quote, for example: *-o ”F:\Projects\test3\output 1”*\. The path using slashes for directory delimiters can be used too, for example: *-o ”F:/Projects/test3/output 1/”*.

--stack-auto

All functions in the source file will be compiled as *reentrant*, i.e. the parameters and local variables will be allocated on the stack.

--callee-saves function1[,function2][,function3]....

The compiler by default uses a caller saves convention for register saving across function calls, however this can cause unnecessary register pushing and popping when calling small functions from larger functions. This option can be used to switch the register saving convention for the function names specified. The compiler will not save registers when calling these functions, no extra code will be generated at the entry and exit (function prologue and epilogue) for these functions to save and restore the registers used by these functions, this can substantially reduce code size and improve run time performance of the generated code. Do not use this option for built-in functions such as _mulint(); if this option is used for a library function the appropriate library function needs to be recompiled with the same option. If the project consists of multiple source files then all the source files should be compiled with the *same* --*callee-saves* option string. Also see *#pragma callee_saves.*

--all-callee-saves

Function of --*callee-saves* will be applied to all functions by default.

--int-long-reent

Integer (16 bit) and long (32 bit) libraries have been compiled as reentrant. Note by default these libraries are compiled as non-reentrant. See section Installation for more details.

--cyclomatic

This option will cause the compiler to generate an information message for each function in the source file. The message contains some *important* information about the function. The number of edges and nodes the compiler detected in the control flow graph of the function, and most importantly the *cyclomatic complexity.*

--float-reent

Floating point library is compiled as reentrant. See section Installation for more details.

--printf-float

Links a C run time library where the printf() and scanf() functions were compiled with floating point support enabled.

--funsigned-char

The default signedness for every type is signed. In some embedded environments the default signedness of char is unsigned. To set the signess for characters to unsigned, use the option -funsigned-char. If this option is set and no signedness keyword (unsigned/signed) is given, a char will be signed. All other types are unaffected.

--main-return

This option can be used if the code generated is called by a monitor program or if the main routine includes an endless loop. This option results in slightly smaller code and saves two bytes of stack space. The return from the 'main' function will return to the function calling main. The default setting is to lock up i.e. generate a 'sjmp $'.

--nostdinc

This will prevent the compiler from passing on the default include path to the preprocessor.

--nostdlib

This will prevent the compiler from passing on the default library path to the linker.

--verbose

Shows the various actions the compiler is performing. Shows the actual commands the compiler is executing.

--no-c-code-in-asm

The C code will not be used as comments into the asm file.

--fverbose-asm

Include code generator and peep-hole comments in the generated asm files.

--no-peep-comments

Don't include peep-hole comments in the generated asm files even if --fverbose-asm option is specified.

--i-code-in-asm

Include i-codes in the asm file. Sounds like noise but is most helpful for debugging the compiler itself.

--less-pedantic

Disable some of the more pedantic warnings. For more details, see the less_pedantic pragma.

--disable-warning <nnnn>

Disable specific warning with number <nnnn>.

--Werror

Treat all warnings as errors.

--print-search-dirs

Display the directories in the compiler's search path

--vc

Display errors and warnings using MSVC style, so you can use C51 with the visual studio IDE. With C51 both offering a GCC-like (the default) and a MSVC-like output style, integration into most programming editors should be straightforward.

--use-stdout

Send errors and warnings to stdout instead of stderr.

-Wa asmOption[,asmOption]...

Pass the asmOption to the assembler.

--std-c51-89

Generally follow the C89 standard, but allow C51 features that conflict with the standard (default).

--std-c89

Follow the C89 standard and disable C51 features that conflict with the standard.

--std-c51-99

Generally follow the C99 standard, but allow C51 features that conflict with the standard (incomplete support).

--std-c99

Follow the C99 standard and disable C51 features that conflict with the standard (incomplete support).

--short-is-8bits

Treat short as 8-bit.

Intermediate Dump Options

The following options are provided for the purpose of debugging the compiler. They provide a means to dump the intermediate code (iCode) generated by the compiler in human readable form at various stages of the compilation process.

--dumpraw

This option will cause the compiler to dump the intermediate code into a file of named <*source filename*>.*dumpraw* just after the intermediate code has been generated for a function, i.e. before any optimizations are done. The basic blocks at this stage ordered in the depth first number, so they may not be in sequence of execution.

--dumpgcse

Will create a dump of iCodes, after global subexpression elimination, into a file named <*source filename*>*.dumpgcse.*

--dumpdeadcode

Will create a dump of iCodes, after deadcode elimination, into a file named <*source filename*>*.dumpdeadcode.*

--dumploop

Will create a dump of iCodes, after loop optimizations, into a file named <*source filename*>*.dumploop.*

--dumprange

Will create a dump of iCodes, after live range analysis, into a file named <*source filename*>*.dumprange.*

--dumlrange

Will dump the life ranges for all symbols.

--dumpregassign

Will create a dump of iCodes, after register assignment, into a file named <*source filename*>*.dumprassgn.*

--dumplrange

Will create a dump of the live ranges of iTemp's --dumpall Will cause all the above mentioned dumps to be created.

Redirecting Output on Windows Command

By default C51 writes its error messages to the "standard error". To redirect all messages to the "standard output" instead, use the option --*use-stdout*. Additionally, if you happen to have visual studio installed in your Windows computer, you can use it to compile your sources using a custom build and the C51 --*vc* option. Something like this should work:

c:\Call51\bin\C51.exe --vc --model-large -c $(InputPath)

Environment variables

C51 recognizes the following environment variables:

C51_LEAVE_SIGNALS

C51 installs a signal handler to be able to delete temporary files after a user break (^C) or an exception. If this environment variable is set, C51 won't install the signal handlers. This is done in order to be able to debug C51.

TMP, TEMP, TMPDIR

Path, where temporary files will be created. The order of the variables is the search order.

C51_HOME

The path location of the programs used by C51.

C51_INCLUDE

The path location of the include files used by C51.

C51_LIB

The path location of the library files used by C51.

Storage Class Language Extensions

In addition to the normal C storage classes, C51 allows the following 8051/8052 specific storage classes:

data / near

This is the default storage class for the *Small Memory* model (*data* and *near* or the more ANSI-C compliant forms *data* and *near* can be used instead). Variables declared with this storage class will be allocated in the directly addressable portion of the internal RAM of a 8051, e.g.:

```
data unsigned char test_data;
```

xdata / far

Variables declared with this storage class will be placed in the expanded RAM. This is the default storage class for the Large Memory model, e.g.:

```
xdata unsigned char test_xdata;
```

idata

Variables declared with this storage class will be allocated into the indirectly addressable portion of the internal ram of an 8051/8052, for example:

```
idata unsigned char test_idata;
```

Please note, the first 128 byte of idata physically access the same RAM as the data memory. The original 8051 had only 128 byte idata memory; nowadays most devices have 256 bytes idata memory. The stack is located in idata memory.

pdata

Paged xdata access is just as straightforward as using the other addressing modes of an 8051. It is typically located at the start of xdata and has a maximum size of 256 bytes. Please note, pdata access physically xdata memory. In the original 8051/8052 the high byte of the address is determined by port P2 but in newer microcontroller variants the high byte is determined by a separate Special Function Register. This is the default storage class for the Medium Memory model, e.g.:

```
pdata unsigned char test_pdata;
```

If the --xstack option is used, the pdata memory area is followed by the xstack memory area and the sum of their sizes is limited to 256 bytes.

code

Constants declared with this storage class will be placed in the code memory, for example:

```
code unsigned char test_code;
code char test_array[] = {“Hello”};
```

bit

This is a data-type and a storage class specifier. When a variable is declared as a bit, it is allocated into the bit addressable memory of 8051, e.g.:

```
bit test_bit;
```

The bit addressable memory consists of 128 bits which are located from locations 0x20 to 0x2f in data memory.

sfr / sfr16 / sfr32 / sbit

Like the bit keyword, *sfr / sfr16 / sfr32 / sbit* signify both a data-type and storage class, they are used to describe the *Special Function Register* and special function *bit* variables of a 8051/8052, for example:

```
// special function register P0 at location 0x80:
sfr at (0x80) P0;
// 16 bit special function register combination
// for timer 0 with the high byte at location 0x8C
// and the low byte at location 0x8A:
sfr16 at (0x8C8A) TMR0;
sbit at (0xd7) CY; // CY (Carry flag)
```

Special function registers which are located on an address dividable by 8 are bit-addressable; a *sbit* addresses a specific bit within these sfr. 16-bit and 32-bit special function register combinations which require a certain access order are better not declared using *sfr16* or *sfr32*. Although C51 usually accesses *sfr16* or *sfr32* using the Least Significant Byte (LSB) first, this is not always guaranteed.

Please note, if you use a header file which was written for another compiler then the sfr / sfr16 / sfr32 / sbit storage class extensions will most likely be *not* compatible. Specifically the syntax sfr P0 = 0x80; is compiled *without warning* by C51 to an assignment of 0x80 to a variable called P0.

Pointers to 8051/8052 specific memory spaces

C51 allows pointers (via language extensions) to explicitly point to any of the memory spaces of the 8051. In addition to the explicit pointers, the compiler uses (by default) generic pointers which can be used to point to any of the memory spaces. Below there are some pointer declaration examples:

```
// pointer in internal ram pointing to object in external ram
xdata unsigned char * data p;
```

```
// pointer in external ram pointing to object in internal ram
data unsigned char * xdata p;

// pointer in code rom pointing to data in xdata space
xdata unsigned char * code p;

//pointer in code space pointing to data in code space
code unsigned char * code p;

// generic pointer physically located in xdata space
unsigned char * xdata p;

// generic pointer located in default memory space
unsigned char * p;

// the following is a function pointer physically located in data space
char (* __data fp)(void);
```

All unqualified pointers are treated as 3-byte *generic* pointers.

The highest order byte of the *generic* pointers contains the data space information. Assembler support routines are called whenever data is stored or retrieved using *generic* pointers. These are useful for developing reusable library routines. Explicitly specifying the pointer type will generate the most efficient code.

The 8051/8052 memory layout

The 8051 family of microcontrollers have a minimum of 128 bytes of internal RAM memory which is structured as follows:

-Bytes 00-1F -32 bytes to hold up to 4 banks of the registers R0 to R7,
-Bytes 20-2F -16 bytes to hold 128 bit variables and,
-Bytes 30-7F -80 bytes for general purpose use.

Additionally, most recent derivatives of the 8051 family of microcontrollers may have up to 128 bytes of additional, indirectly addressable, internal RAM memory (*idata*). Furthermore, some microcontrollers may have some built in external memory (*xdata*) which should not be confused with the internal, directly addressable RAM memory (*data*).

Normally C51 will only use the first bank of registers (register bank 0), but it is possible to specify that other banks of registers (keyword *using)* should be used, for example, in interrupt service routines. By default, the compiler will place the stack after the last byte of allocated memory for variables. For example, if the first 2 banks of registers are used, and only four bytes are used for *data* variables, it will position the base of the internal stack at address 20

(0x14). This implies that as the stack grows, it will use up the remaining register banks, the 16 bytes used by the 128 bit variables, and 80 bytes for general purpose use. If any bit variables are used, the data variables will be placed in unused register banks and after the byte holding the last bit variable. For example, if register banks 0 and 1 are used, and there are 9 bit variables (two bytes used), *data* variables will be placed starting from address 0x10 to 0x20 and continue at address 0x22. You can also use --dataloc to specify the start address of the *data* and --iram-size to specify the size of the total internal RAM (*data*+*idata*).

The 8051/8052 linker L51 will define the public variable *stack_start* with a value equal to the last byte available in the *idata* memory space. The default star-up code (crt0.c) uses this variable to assign the stack during start-up. If you want to place the beginning of stack somewhere else, use the *sfr* SP in your custom start-up code either in assembly or C as shown below:

```
mov SP, #(0A0H-1) ; Setting the stack in assembly language

SP=0xA0-1; // Setting the stack in C
```

You may also need to use --xdata-loc to set the start address of the external RAM (*xdata*) and --xram-size to specify its size. Same goes for the code memory, using --code-loc and --code-size. If in doubt, don't specify any options and see if the resulting memory layout is appropriate in the .map file, then you can adjust it.

The linker L51 generates a file with memory allocation information. This file, with extension .map, shows all the variables and segments as well as the final memory layout. The linker will complain either if memory segments overlap or there is not enough memory. If you get any linking warnings and/or errors related to segments allocation, check the .map file to find out what the problem is.

Absolute Addressing

Data items can be assigned an absolute address with the *at <address>* keyword, in addition to a storage class, e.g.:

```
xdata at (0x7ffe) unsigned int chksum;
```

In the above example the variable chksum will be located at 0x7ffe and 0x7fff of the external ram. The compiler does *not* reserve any space for variables declared this way as they are implemented with a literal assignment (equ) in the

assembler. Therefore, it is left to the programmer to make sure there are no overlaps with other variables that are declared without the absolute address. The assembler listing file (.lst) and (.map) are good places to look for such overlaps.

If however you provide an initializer, actual memory allocation will take place and overlaps will be detected by the linker. For example:

```
code at (0x7ff0) char Id[5] = ”C51”;
```

In the above example the variable Id will be located from 0x7ff0 to 0x7ff3 in code memory. In case of memory mapped I/O devices the keyword *volatile* has to be used to tell the compiler that accesses might not be removed (optimized out):

```
volatile xdata at (0x8000) unsigned char PORTA_8255;
```

Absolute addresses can be specified for variables in all storage classes, for example:

```
bit at (0x02) bvar;
```

The above example will allocate the variable bvar at offset 0x02 in the bit-addressable space. There is no real advantage to assigning absolute addresses to variables in this manner, unless you want strict control over all the variables allocated. One possible use would be to write hardware portable code. For example, if you have a routine that uses one or more of the microcontroller I/O pins, and such pins are different for two different hardware configurations, you can declare the I/O pins in your routine using:

```
extern volatile bit MOSI; /* master out, slave in */
extern volatile bit MISO; /* master in, slave out */
extern volatile bit MCLK; /* master clock */
/* Input and Output of a byte on a 3-wire serial bus.
If needed adapt polarity of clock, polarity of data and bit
order*/

unsigned char spi_io(unsigned char out_byte)
{
    unsigned char i=8;
    do {
        MOSI = out_byte & 0x80;
        out_byte <<= 1; MCLK = 1;
        if(MISO) out_byte += 1;
```

```
        MCLK = 0;
    } while(--i);
    return out_byte;
}
```

Then, someplace in the code for the first hardware you would use

```
bit at (0x80) MOSI; /* I/O port 0, bit 0 */
bit at (0x81) MISO; /* I/O port 0, bit 1 */
bit at (0x82) MCLK; /* I/O port 0, bit 2 */
```

Similarly, for the second hardware you would use

```
bit at (0x83) MOSI; /* I/O port 0, bit 3 */
bit at (0x91) MISO; /* I/O port 1, bit 1 */
bit at (0x92) MCLK; /* I/O port 1, bit 2 */
```

By using this approach, you can recycle the same hardware dependent routine without changes, as for example in a library. This is somehow similar to sbit, but only one absolute address has to be specified in the whole project.

Parameters and Local Variables

Automatic (local) variables and parameters to functions can either be placed on the stack or in data-space. The default action of the compiler is to place these variables in the internal RAM (for small model) or external RAM (for medium or large model). This in fact makes them similar to *static variables* so by default functions are non-reentrant.

Automatic variables can be placed on the stack by using the *--stack-auto* option, by using *#pragma stackauto* or by using the *reentrant* keyword in the function declaration, e.g.:

```
unsigned char foo(char i) __reentrant
{
...
}
```

Since stack space on 8051 is limited, the reentrant keyword or the --stack-auto option should be used sparingly. Note that the reentrant keyword just means that the parameters and local variables will be allocated in the stack; it does not mean that the function is register bank independent.

Local variables can be also assigned storage classes and absolute addresses, for example:

```
unsigned char foo(xdata int parm)
{
    xdata unsigned char i;
    bit bvar;
    data at (0x31) unsigned char j;
    ...
}
```

In the above example the parameter *parm* and the variable *i* will be allocated in the external ram, *bvar* in bit addressable space and *j* in internal ram. When compiled with *--stack-auto* or when a function is declared as *reentrant* this should only be done for static variables.

It is however allowed to use bit parameters in reentrant functions and also non-static local bit variables are supported. Efficient use is limited to 8 semi-bit registers in bit space. They are pushed and popped to/from the stack as a single byte just like regular registers.

Overlaying

For non-reentrant functions, C51 will try to reduce internal ram space usage by overlaying parameters and local variables of a function (if possible). Parameters and local variables of a function will be allocated to an overlayable segment if the function has *no other function calls and the function is non-reentrant and the memory model is small.* If an explicit storage class is specified for a local variable, it will NOT be overlaid.

Note that the compiler (not the linker) makes the decision for overlaying the data items. Functions that are called from an interrupt service routine should be preceded by a #pragma nooverlay if they are not reentrant.

Also note that the compiler does not do any processing of inline assembler code, so the compiler might incorrectly assign local variables and parameters of a function into the overlay segment if the inline assembler code calls other C functions that might use the overlay. In that case the #pragma nooverlay should be used.

Parameters and local variables of functions that contain 16 or 32 bit multiplication or division will NOT be overlaid since these are implemented using external functions, e.g.:

```
#pragma save
#pragma nooverlay
void set_error(unsigned char errcd)
{
   P3 = errcd;
}
#pragma restore
void some_isr () interrupt (2)
{
   ...
   set_error(10);
   ...
}
```

In the above example the parameter *errcd* for the function *set_error* would be assigned to the overlayable segment if the #pragma nooverlay was not present, this could cause unpredictable runtime behavior when called from an interrupt service routine. The #pragma nooverlay ensures that the parameters and local variables for the function are NOT overlaid.

Forced Overlaying

In C51 it is possible to force several functions to share the same memory locations for variables and arguments by using the option '--overlayto <segment_name>' or the pragma 'overlayto <segment_name>'. For this forced overlaying to work some conditions must be met:

1) Functions are non-reentrant.

2) Variables are used only locally: no global variables.

3) Functions do not call other functions using the same <segment_name>.

4) Variables are non-volatile.

By using forced overlaying, the internal memory requirements are greatly reduced. This is very useful when building library of functions that are mostly independent of each other.

Interrupt Service Routines

C51 allows *Interrupt Service Routines* (ISRs) to be coded in C, by using a predefined format, for example:

```
void timer_isr (void) interrupt (1) using (1)
{
   ...
}
```

The number following the *interrupt* keyword is the interrupt number the routine will service. The compiler will insert a jump to this routine in the interrupt vector table for the interrupt number specified. The optional keyword *using* can be applied to tell the compiler to use a specific register bank when generating code for this function.

Interrupt Service Routines can be in any module including libraries, but must be unique: there must be only one ISR for a given interrupt number or the linker will report an overlapping error in code memory. Also, C51 does not allow for ISR function prototypes: all ISRs must have a body. Finally, ISRs should not be called directly by user code.

If the interrupt service routine is defined without *using* a register bank or with register bank 0 (*using* 0), the compiler will save the registers used by itself on the stack upon entry and restore them at exit, however if such an interrupt service routine calls another function then the entire register bank will be saved on the stack. This scheme may be advantageous for small interrupt service routines which have low register usage.

If the interrupt service routine is using a specific register bank then only *a*, *b*, *dptr* & psw are saved and restored. If such an interrupt service routine calls another function (using another register bank) then the entire register bank of the called function will be saved on the stack. This scheme is recommended for larger interrupt service routines.

Interrupt numbers and the corresponding address and descriptions for the original 8051/8052 microcontrollers are listed below. Newer microcontrollers usually have many more interrupt sources as described in their datasheets.

Interrupt #	Description	Vector Address
0	External 0	0x0003
1	Timer 0	0x000b

2	External 1	0x0013
3	Timer 1	0x001b
4	Serial	0x0023
5	Timer 2	0x002b

Interrupt service routines are prone to some common pitfalls. Here are the most frequent:

Common interrupt pitfall: *variable not declared volatile*

If an interrupt service routine changes variables which are accessed by other functions these variables have to be declared *volatile*. See

http://en.wikipedia.org/wiki/Volatile_variable .

Common interrupt pitfall: *non-atomic access*

If the access to these variables is not *atomic* (i.e. the processor needs more than one assembly instruction to implement the access and could be interrupted while accessing such variable) some action should be taken to prevent invalid data to be retrieved or saved. Access to 16 or 32 bit variables is obviously not atomic on 8 bit CPUs such as the 8051/8052 family of microcontrollers, and should be protected by either disabling interrupts or using an atomic variable to verify the correct access to the variable. The only two atomic variables in the 8051/8052 are bit and data char variables.

Common interrupt pitfall: *stack overflow*

The return address and the registers used in the interrupt service routine are saved on the stack so there must be sufficient stack space. If there isn't enough stack space, variables or registers (or even the return address itself) will be corrupted. This stack overflow is most likely to happen if the interrupt occurs during the "deepest" subroutine when the stack is already in use.

Common interrupt pitfall: *use of non-reentrant functions*

A special note here, int (16 bit) and long (32 bit) integer division, multiplication & modulus as well as floating-point operations are implemented using external support routines. If an interrupt service routine needs to do any of these operations then the support routines (as mentioned in a following section) will have to be recompiled using the *--stack-auto* option and the source file will need to be compiled using the *--int-long-reent* compiler option.

Calling other functions from an interrupt service routine is not recommended, and should be avoid it if possible. Note that when some function is called from an interrupt service routine it should be preceded by a *#pragma nooverlay* if it is not reentrant. Furthermore non-reentrant functions should not be called from the main program while the interrupt service routine might be active. They also must not be called from low priority interrupt service routines while a high priority interrupt service routine might be active. You could use semaphores or make the function *critical* if all parameters are passed in registers.

Interrupt Service Routine Example

```
#include <p89v51rd2.h>
#include <stdio.h>

volatile unsigned char count; // atomic variable is volatile

#define CLK 22118400L
#define BAUD 115200L
#define TIMER_2_RELOAD (0x10000L-(CLK/(32L*BAUD)))
// Set timer 0 to interrupt every 10 milliseconds (1/100Hz)=10 ms
#define TIMER_0_RELOAD (65536L-((CLK)/(12*100L)))

// This Interrupt Service Routine is called automatically
// when timer 0 overflows.  That happens every 10ms
void Timer0_ISR (void) interrupt 1 using 1
{
    // Reload the timer
    TR0=0; // Stop timer 0
    TH0=TIMER_0_RELOAD/0x100;
    TL0=TIMER_0_RELOAD%0x100;
    TR0=1; // Start timer 0
    TF0=0; // Clear timer 0 overflow flag
    // 200H square wave at pin P1_1
    P1_1=!P1_1;
    count++;
}

void SetTimer0 (void)
{
    TR0=0; // Stop timer 0
    TMOD=(TMOD&0x0f0)|0x01; // 16-bit timer
    TH0=TIMER_0_RELOAD/0x100;
    TL0=TIMER_0_RELOAD%0x100;
    TR0=1; // Start timer 0
    ET0=1; // Enable timer 0 interrupt
```

```
    EA=1; // Enable global interrupts
    P1_1=1;
}

void main (void)
{
    unsigned int sec=0;
    // Initialize serial port using timer 2
    setbaud_timer2(TIMER_2_RELOAD);
    // Initialize timer 0 to interrupt every 10 ms
    SetTimer0();
    while(1)
    {
         count=0;
         while(count<100); // Wait one second
         sec++;
         printf("%5d\r", sec);
    }
}
```

Enabling and Disabling Interrupts

Critical Functions and Critical Statements

A special keyword may be associated with a block or a function declaring it as *critical*. C51 will generate code to disable all interrupts upon entry to a critical function and restore the interrupt enable to the previous state before returning. Nesting critical functions will need one additional byte on the stack for each call.

```
int foo () critical
{
    ...
    ...
}
```

The critical attribute maybe used with other attributes like *reentrant*. The keyword *critical* may also be used to disable interrupts more locally:

```
critical{ i++; }
```

Several statements can be included in the same *critical* block.

Enabling and Disabling Interrupts Directly

Interrupts can also be disabled and enabled directly (8051):

```
EA = 0; // or:
EA_SAVE = EA;
EA=0;
...
EA=1;
...
EA = EA_SAVE;
```

Note: it is sometimes sufficient to disable only a specific interrupt source like for example a timer or serial interrupt by manipulating an *interrupt mask* register. Usually the time during which interrupts are disabled should be kept as short as possible. This minimizes both *interrupt latency* (the time between the occurrence of the interrupt and the execution of the first code in the interrupt routine) and *interrupt jitter* (the difference between the shortest and the longest interrupt latency). These really are something different, for example, a serial interrupt has to be served before its buffer overruns so it cares for the maximum interrupt latency, whereas it does not care about jitter. On a loudspeaker driven via a digital to analog converter which is fed by an interrupt a latency of a few milliseconds might be tolerable, whereas a much smaller jitter will be very audible.

Semaphore locking

The 8051/8052 microcontroller has an atomic bit test and clear instruction. These type of instructions are typically used in pre-emptive multitasking systems, where a routine claims the use of a data structure (’acquires a lock on it’), makes some modifications and then releases the lock when the data structure is consistent again. The instruction may also be used if interrupt and non-interrupt code have to share a resource. With the atomic bit test and clear instruction interrupts don’t have to be disabled for the locking operation.

C51 generates this instruction if the source follows this pattern:

```
volatile bit resource_is_free;
if (resource_is_free)
{
    resource_is_free=0;
    ...
    resource_is_free=1;
}
```

Note that the 8051/8052 supports only an atomic bit test and *clear* instruction (as opposed to atomic bit test and *set).*

Functions using dedicated register banks

The 8051/8052 microcontroller family can quickly change register banks. C51 supports this feature with the *using* attribute. The using attribute tells the compiler to use a register bank other than the default bank zero. It is normally applied to *interrupt* functions or any functions called only by that interrupt function. This will, in most circumstances, make the generated ISR code more efficient since it will not have to save registers on the stack.

An *interrupt* function using a non-zero bank will assume that it can freely modify that register bank without restoring it, and will not save it. Since high-priority interrupts can interrupt low-priority ones on the 8051/8052 microcontroller family, this means that if a high-priority ISR *using* a particular bank occurs while processing a low-priority ISR *using* the same bank, the registers in such bank will be corrupted upon returning back to the low-priority interrupt. To prevent this, a register bank should be only used by ISRs with the same priority level. If ISRs are allowed to change priorities at runtime, it will be safer to use the default bank zero and take the small performance penalty.

It is most efficient if ISRs do not call any other functions. If your ISR must call other functions, it is more efficient if those functions use the same bank as the ISR; the next best option is if the called functions use bank zero. It is very inefficient to call a function using a different, non-zero register bank from an ISR.

If a function is called ONLY from 'interrupt' functions using a particular bank, it can be declared with the same 'using' attribute as the calling 'interrupt' functions. For instance, if you have several ISRs using bank one, and all of them call memcpy(), it might make sense to create a dedicated version of memcpy() 'using 1', since this would prevent the ISR from having to save bank zero to the stack on entry and switch to bank zero before calling the function

Start-up Code

The first assembly instruction executed by program built with C51 will execute a jump to label *_crt0*. This jump instruction is placed at address zero (or at the address specified by the *–code-loc* option), which is where 8051/8051 microcontrollers start running after reset. The tasks performed in *_crt0* include: initialization of the stack, initialization of memory to its default values,

clearing up un-initialized data to zero, calling the main function, and remain in a ‘forever’ loop after returning from the main() function to prevent any further code execution. The default libraries of C51 include a *_crt0* function (file crt0.c), but you can write and link your own *_crt0* to fit your specific needs.

The default _crt0 provided in the libraries include an “*lcall*” to “_c51_external_startup”. You can include this C function in your project to perform any initialization required by the target hardware. For example, the following “_c51_external_startup” was used to initialize a C8051F330 microcontroller from Silabs:

```
char _c51_external_startup (void)
{
       PCA0MD&=(~0x40) ; // FIRST DISABLE WDT: clear Watchdog Enable bit
       VDM0CN=0x80;      // enable VDD monitor
       RSTSRC=0x02|0x04; // Enable reset on missing clock detector and VDD

       P0MDOUT|=0x10; // Enable Uart TX as push-pull output
       P1MDOUT|=0x31; // Initialize C2CK, RLED, and GLED as push-pull outputs
       XBR0=0x01;     // Enable UART on P0.4(TX) and P0.5(RX)
       XBR1=0x40;     // Enable crossbar and weak pull-ups
       OSCICN|=0x03;  // Configure internal oscr for its maximum frequency

       // configure serial port and baud rate
       TR1=0;      // Disable timer 1
       TMOD=0x20;  // Set timer 1 as 8-bit auto reload
       CKCON=0x08; // T1M = 1; SCA1:0 = xx
       TH1=0x96;   // for 115200 baud; check table 16.1 in C8051F330 manual
       TR1=1;      // Enable timer 1
       SCON=0x52;

       return 0;
}
```

Inline Assembly Code

You can directly place assembly instructions into you C source file by enclosing the instructions between the *_asm ... _endasm;* keywords. The inline assembly code can contain any valid code understood by the assembler A51, this includes any assembler directives, controls, and comment lines. The compiler does not do any validation of the code within the *_asm ... _endasm;* keyword pair. Specifically it will not know which registers are used and thus register pushing/popping has to be done manually.

It is required that each assembly instruction be placed in a separate line. When the *--peep-asm* command line option is used, the inline assembler code will be passed through the peephole optimizer. There are only a few (if any) cases

where this option makes sense; it might cause some unexpected changes in the inline assembly code.

Several library functions are written using inline assembly code. You can use them as example/base to write your own inline assembly functions.

Inline Assembly Code Example

The example below shows how to use a delay function that was originally written in assembly but it is used in C.

```
#include <p89v51rd2.h>
#include <stdio.h>

#define CLK 22118400L
#define BAUD 115200L
#define TIMER_2_RELOAD (0x10000L-(CLK/(32L*BAUD)))

void Wait1s (void)
{
      _asm
          push ar0
          push ar1
          push ar2
      ;For a 22.1184MHz crystal one machine cycle
      ;takes 12/22.1184MHz=0.5425347us.
      ;The number of machine cycles for this routine is:
      ;(((((183*2)+3)*250)+3)*20+4) for a time of 1.001s
      Wait1s:
          mov R2, #20
      L3: mov R1, #250
      L2: mov R0, #183
      L1: djnz R0, L1 ; 2 cycles-> 2*0.543us*184=200us
          djnz R1, L2 ; 200us*250=0.05s
          djnz R2, L3 ; 0.05s*20=1s
          pop ar2
          pop ar1
          pop ar0
          ret
      _endasm;
}

void main (void)
{
    unsigned int sec=0;
```

```
    // Initialize serial port using timer 2
    setbaud_timer2(TIMER_2_RELOAD);
    while(1)
    {
         Wait1s(); // Wait one second
         sec++;
         printf("%5d\r", sec);
    }
}
```

Naked Functions

The *naked* function modifier attribute prevents the compiler from generating prologue and epilogue code for that function. This means that the user is entirely responsible for such things as saving any registers that may need to be preserved, selecting the proper register bank, generating the *return* instruction at the end, etc. Practically, this means that at least some of the contents of the function would be written in inline assembly.

Interfacing with Assembler Code

Global Registers used for Parameter Passing

The compiler always uses the global registers *DPL, DPH, B* and *ACC* to pass the first (non-bit) parameter to a function, and also to pass the return value of the function according to the following scheme:

- One byte return value in *DPL*.
- Two byte values in *DPL* (LSB) and *DPH* (MSB).
- Three byte values (generic pointers) in *DPH*, *DPL* and *B*. Generic pointers contain type of address space in register *B* as:
    - 0x00 – *xdata*
    - 0x40 – *data* and *idata*
    - 0x60 – *pdata*
    - 0x80 – *code*

- Four byte values in *DPH*, *DPL*, *B* and *ACC*.

The second parameter onwards is either allocated on the stack (for reentrant routines or if *--stack-auto* is used) or in *data/xdata* memory (depending on the memory model).

Bit parameters are passed in a virtual register called ’bits’ in bit-addressable space for reentrant functions or allocated directly in bit memory otherwise.

Functions (with two or more parameters or with bit parameters) that are called through function pointers must therefore be re-entrant so the compiler knows how to pass the parameters.

Registers usage

Unless the called function is declared as *_naked*, or the *--callee-saves/--all-callee-saves* command line option or the corresponding *callee_saves pragma* are used, the caller will save the registers (*R0-R7*) around the call, so the called function can modify their content freely.

If the called function is not declared as _naked, the caller will swap register banks around the call, if caller and callee use different register banks (having them defined by the *using* modifier).

The called function can also use *DPL*, *DPH*, *B* and *ACC* observing that they are used for parameter/return value passing.

Assembler Routine (non-reentrant)

In the following example the function c_func calls an assembler routine asm_func, which takes two parameters.

```
extern int asm_func(unsigned char, unsigned char);
int c_func (unsigned char i, unsigned char j)
{
   return asm_func(i,j);
}
int main()
{
   return c_func(10,9);
}
```

The corresponding assembler function is:

```
$optc51 --model-small
R_CSEG segment code
R_OSEG segment data overlay
public _asm_func_PARM_2, _asm_func
rseg R_OSEG
_asm_func_PARM_2:
     ds 1
rseg R_CSEG
_asm_func:
     mov a,dpl
     add a,_asm_func_PARM_2
     mov dpl,a
     mov dph,#0x00
     ret
END
```

The parameter naming convention is _<*function_name*>_*PARM*_<*n*>, where n is the parameter number starting from 1, and counting from the left. The first parameter is passed in *DPH*, *DPL*, *B* and *ACC* according to the description above. The variable name for the second parameter will be _<*function_name*>_*PARM_2*.

Assemble the assembly routine with the following command:

A51 –c –i asmfunc.asm

Then compile and link the assembler routine to the C source file with the following command:

C51 cfunc.c asmfunc.obj

Assembly Routine (reentrant)

In this case the second parameter onwards will be passed on the stack, the parameters are pushed from right to left i.e. before the call the second leftmost parameter will be on the top of the stack (the leftmost parameter is passed in registers). Here is an example:

```
extern int asm_func(unsigned char, unsigned char,
                    unsigned char) reentrant;

int c_func (unsigned char i, unsigned char j,
            unsigned char k) reentrant
{
```

```
   return asm_func(i,j,k);
}

int main()
{
   return c_func(10,9,8);
}
```

The corresponding assembler routine is:

```
$optc51 --model-small
R_CSEG segment code
public _asm_func
rseg R_CSEG
_asm_func:
    push _bp
    mov _bp,sp ;stack contains: _bp, return address,
               ;second parameter, third parameter
    mov r2,dpl
    mov a,_bp
    add a,#0xfd ;calculate pointer to the second parameter
    mov r0,a
    mov a,_bp
    add a,#0xfc ;calculate pointer to the rightmost parameter
    mov r1,a
    mov a,@r0
    add a,@r1
    add a,r2 ;calculate the result (= sum
             ;of all three parameters)
    mov dpl,a ;return value goes into dptr (cast into int)
    mov dph,#0x00
    mov sp,_bp
    pop _bp
    ret
END
```

The compiling and linking procedure remains the same as for the non-re-entrant case. However note the extra entry and exit linkage required for the assembler code: *_bp* is the stack frame pointer and is used to compute the offset into the stack for parameters and local variables.

int (16 bit) and long (32 bit) Support

For signed and unsigned *int* (16 bit) and *long* (32 bit) variables, division, multiplication and modulus operations are implemented by support routines. These support routines are listed in the table below.

Function	**Description**
mulint.c	16 bit multiplication
divsint.c	signed 16 bit division (calls _divuint)
divuint.c	unsigned 16 bit division
modsint.c	signed 16 bit modulus (calls _moduint)
moduint.c	unsigned 16 bit modulus
mullong.c	32 bit multiplication
divslong.c	signed 32 division (calls _divulong)
divulong.c	unsigned 32 division
modslong.c	signed 32 bit modulus (calls _modulong)
modulong.c	unsigned 32 bit modulus

Since these routines are compiled as *non-reentrant*, interrupt service routines should not do any of the above operations. If this is unavoidable then the above routines will need to be compiled with the *--stack-auto* option, after which the source program will have to be compiled with *--int-long-reent* option. Notice that you don't have to call these routines directly. The compiler will use them automatically every time an integer operation is required.

Floating Point Support

C51 supports IEEE (single precision - 4 bytes) floating point numbers. The floating point support routines consist of the following routines:

Function	Description
fsadd.c	add floating point numbers
fssub.c	subtract floating point numbers
fsdiv.c	divide floating point numbers
fsmul.c	multiply floating point numbers
fs2uchar.c	convert floating point to unsigned char
fs2char.c	convert floating point to signed char
fs2uint.c	convert floating point to unsigned int
fs2int.c	convert floating point to signed int
fs2ulong.c	convert floating point to unsigned long
fs2long.c	convert floating point to signed long

uchar2fs.c	convert unsigned char to floating point
char2fs.c	convert char to floating point number
uint2fs.c	convert unsigned int to floating point
int2fs.c	convert int to floating point numbers
ulong2fs.c	convert unsigned long to floating point number
long2fs.c	convert long to floating point number

Some of these support routines are written in C while others are written in assembly in order to improve space usage and execution speed. Note that if all these routines are used simultaneously, the *data* space might be exhausted. For heavy floating point usage, the large model might be needed. Also notice that you don't have to call these routines directly. The compiler will use them automatically every time a floating point operation is required.

Library Routines

C51 includes many (but not all) of the standard C library functions. The best way to find out if a function is supported is to check the corresponding include file and/or the libraries sources. Some functions that require special attention are described below.

putchar() and getchar()

Both putchar() and getchar() are included as part of the library functions. They both used the standard serial port of the 8051/8052 as the input/output source/destination. Other library functions such as printf(), gets(), puts(), scanf(), etc. call putchar() or getchar(). If the serial port is not the source/destination of your design input/output, you must write new putchar() and getchar() functions tailored to your needs. For example, a putchar() function can be written to use an LCD as the output device.

If you are planning on using the default putchar() and getchar() functions from the libraries, you must initialize the serial port and the baud rate generator of the 8051/8052 microcontroller. C51 includes two additional functions to perform this task. Their prototypes are in <*stdio.h*> as

```
extern void setbaud_timer1 (unsigned char reload);
```

```
extern void setbaud_timer2 (unsigned int reload);
```

In both cases the reload value corresponds to the timer being used to generate the baud rate. In most cases you can use the pre-processor to compute the reload value based in your hardware. For example for a P89V51RD2 processor from NXP using a crystal of 22.1184MHz, this is how the baud rate of the serial port is set to 115200 baud using timer 2:

```
#define CLK 22118400L
#define BAUD 115200L
#define TIMER_2_RELOAD (0x10000L-(CLK/(32L*BAUD)))
// Initialize serial port using timer 2
setbaud_timer2(TIMER_2_RELOAD);
```

If the processor you are using includes a dedicated baud rate generator you will need to write its initialization yourself. For example, for a P89LPC9351 processor from NXP:

```
#define XTAL 7373000L
#define BAUD 115200L
#define BRGR ((XTAL/BAUD)-16)

char InitSerialPort (void)
{
    P1M1&=0xfc;
    P1M2&=0xfc;
    BRGCON=0;
    BRGR1=BRGR/0x100;
    BRGR0=BRGR%0x100;
    BRGCON=3;
    SCON=0x50;
    return 0;
}
```

printf() and scanf()

C51 also includes printf() and scanf() within its libraries. Their options and behaviour are fairly standard, so most internet references will be valid for both these functions as well as for the string version of them, sprintf() and sscanf(). For example:

http://www.cplusplus.com/reference/clibrary/cstdio/printf/

http://www.cplusplus.com/reference/clibrary/cstdio/scanf/

For single file projects, C51 will try to detect if a call to the printf() or scanf() functions requires the use of floating point. If that is the case, the linker will be instructed to link a version of the run time library with floating point support enabled. This behaviour can be forced by using the option *–float-printf*. For multiple file projects, option *–float-printf* must be specified at link time in order to include support for floating point numbers. If the run time library WITHOUT floating point support is linked, and a call to printf() is required to print a floating point number, the message <NO FLOAT> will be displayed. If that is the case, enable floating point support by using *–float-printf* when linking.

Dynamic Memory Allocation

Before using the standard C heap functions (malloc(), free(), etc.) with C51 you'll need to declare/reserve the memory to be used as heap. To do so, include into your code the following lines to reserve heap space:

```
#define HEAP_SIZE 2048 // Set your heap size here!
__xdata char _c51_heap[HEAP_SIZE];
const unsigned int _c51_heap_size = HEAP_SIZE;
```

Memory Models

C51 allows three memory models: small, medium and large. Modules compiled with different memory models WILL NOT be combined together by the linker L51. The library routines supplied with the compiler are compiled as small, medium, and large. The compiled library modules are contained in separate directories as small, medium and large so that you can link to the appropriate set.

When the medium or large model is used, all variables declared without a storage class will be allocated into the external ram, this includes all parameters and local variables (for non-reentrant functions). When the small model is used variables without storage class are allocated in the internal ram.

Judicious usage of the processor specific storage classes and the 'reentrant' function type will yield much more efficient code, than using the large model. Several optimizations are disabled when the program is compiled using the large model, it is therefore recommended that the small model be used unless absolutely required.

External Stack

The external stack (--*xstack* option) is located in *pdata* memory (usually at the start of the external ram segment) and uses all unused space in *pdata* (up to 256 bytes). When the --*xstack* option is used to compile the program, the parameters and local variables of all reentrant functions are allocated in this area. This option is provided for programs with large stack space requirements. When used with the --*stack-auto* option, all parameters and local variables are allocated on the external stack (note: support libraries will need to be recompiled with the same options).

The compiler outputs the higher order address byte of the external ram segment into port P2. Therefore when using the External Stack option, this I/O port *may not* be used by the application program.

Pragmas

Pragmas are used to turn on and/or off certain compiler options. Some of them are closely related to corresponding command-line options. Pragmas should be placed before and/or after a function, placing pragmas inside a function body could have unpredictable results.

C51 supports the following #pragma directives:

save

This pragma saves the most current options to the save/restore stack. See #pragma restore.

restore

This pragma restores saved options from the last save. saves & restores can be nested. C51 uses a save/restore stack: save pushes current options to the stack, restore pulls current options from the stack.

callee_saves function1[,function2[,function3...]]

The compiler, by default, uses a ‘caller saves’ convention for register preservation across function calls. However this can cause unnecessary register pushing and popping when calling small functions from larger functions. This option can be used to switch off the register saving convention for the function names specified. The compiler will not save

registers when calling these functions. The programmer must add extra code manually at the entry and exit points for these functions to save and restore the used registers. Nevertheless, this can significantly reduce code size and improve run time speed.

exclude none | {acc[,b[,dpl[,dph]]]

The exclude pragma disables the generation of pairs of push/pop instructions in *I*nterrupt *S*ervice *R*outines. The directive should be placed immediately before the ISR function definition and it affects ALL ISR functions following it. To enable the normal register saving for ISR functions use *#pragma exclude none*. See also the related keyword *naked*.

less_pedantic

The compiler will not warn you anymore for obvious mistakes. See also the command line option --less-pedantic on page 30. More specifically, the following warnings will be disabled:

- *comparison is always [true/false] due to limited range of data type* (94);
- *overflow in implicit constant conversion* (158);
- *conditional flow changed by optimizer* (110);
- *function '[function name]' must return value* (59).

Furthermore, warnings of less importance (of PEDANTIC and INFO warning level) are disabled, too, namely:

- *constant value '[]', out of range (81);*
- *[left/right] shifting more than size of object changed to zero (116);*
- *unreachable code (126);*
- *integer overflow in expression (165);*
- *unmatched #pragma save and #pragma restore (170);*
- *comparison of 'signed char' with 'unsigned char' requires promotion to int (185);*
- *ISO C90 does not support flexible array members (187);*
- *extended stack by [number] bytes for compiler temp(s) :in function '[function name]': [] (114);*
- *function '[function name]', # edges [number] , # nodes [number], cyclomatic complexity [number] (121).*

disable_warning <nnnn>

The compiler will not warn anymore about warning number <nnnn>.

nogcse

The compiler will stop global common sub expression elimination.

noinduction

The compiler will stop loop induction optimizations.

noinvariant

The compiler will not do loop invariant optimizations.

nojtbound

The compiler will not generate code for boundary value checking, when switch statements are turned into jump-tables (dangerous!).

noloopreverse

The compiler will not do loop reversal optimization

nooverlay

The compiler will not overlay the parameters and local variables of a function.

overlayto

The compiler will overlay parameters and variables into the overlayable data segment <segment_name> for all functions and variables in the current file. Make sure that none of the functions in the file calls a function that uses the same overlayable segment name. This option is intended to be used with libraries or projects where files contain a single function.

stackauto

This pragma behaves similarly to option *--stack-auto*.

opt_code_speed

The compiler will optimize code generation towards fast code, possibly at the expense of code size. Currently this has little effect.

opt_code_size

The compiler will optimize code generation towards compact code, possibly at the expense of code speed. Currently this has little effect.

opt_code_balanced

The compiler will attempt to generate code that is both compact and fast, as long as meeting one goal is not a detriment to the other (this is the default).

std_C51_89

Generally follow the C89 standard, but allow C51 features that conflict with the standard (default).

std_c89

Follow the C89 standard and disable C51 features that conflict with the standard.

std_C51_99

Generally follow the C99 standard, but allow C51 features that conflict with the standard (incomplete support).

std_c99

Follow the C99 standard and disable C51 features that conflict with the standard (incomplete support).

The preprocessor C51PP supports the following #pragma directives:

preproc_asm (+ | -)

Switch the *_asm … _endasm;* block preprocessing on / off. Default is on. Below is an example on how to use this pragma.

```
#pragma preproc_asm
#define NOP /* this is a c code nop */
void foo (void)
{
    _asm
    NOP ; this is an assembler nop instruction
        ; it is not preprocessed to ';' since
        ; the asm preprocessing is disabled
    _endasm;
}
```

The *pragma preproc_asm* should not be used to define multilines of assembly code (even if it supports it), since this behaviour is only a side effect of *c51pp __asm __endasm* implementation in combination with pragma *preproc_asm* and is not in conformance with the C standard. To define multilines of assembly code you have to include each assembly line into it's own __asm __endasm block. Below is an example for multiline assembly defines.

```
#define Nop __asm \
nop \
__endasm
#define ThreeNops Nop; \
Nop; \
Nop
void foo (void) {
...
ThreeNops;
...
}
```

C51_hash (+ | -)

Allow "naked" hash in macro definition, for example:

```
#define DIR_LO(x) #(x & 0xff)
```

Default is off. Below is an example on how to use this pragma.

```
#pragma preproc_asm +
#pragma C51_hash +
#define ROMCALL(x) \
mov R6_B3, #(x & 0xff) \
mov R7_B3, #((x >> 8) & 0xff) \
lcall __romcall
```

```
...
__asm
ROMCALL(72)
__endasm;
```

Some of the pragmas are intended to be used to turn-on or off certain optimizations which might cause the compiler to generate extra stack and/or data space to store compiler generated temporary variables. This usually happens in large functions. Pragma directives should be used as shown in the following example. They are used to control options and optimizations for a given function.

```
#pragma save /* save the current settings */
#pragma nogcse /* turn off global sub expression elimination
*/
#pragma noinduction /* turn off induction optimizations */
int foo ()
{
...
/* large code */
...
}
#pragma restore /* turn the optimizations back on */
```

Defines Created by the Compiler

The compiler creates the following #defines:

#define	**Description**
C51	Always defined. The version number as an int.
C51_HAS_MAIN_FUNCTION	Defined if a main() function is detected
C51_STACK_AUTO	when --*stack-auto* option is used
C51_MODEL_SMALL	when --*model-small* is used
C51_MODEL_MEDIUM	when --*model-medium* is used
C51_MODEL_LARGE	when --*model-large* is used
C51_USE_XSTACK	when --*xstack* option is used
C51_CHAR_UNSIGNED	when --*funsigned-char* option is used
C51_REVISION	Always defined. Current C51 revision #
C51_PARMS_IN_BANK1	when --*parms-in-bank1* is used
C51_FLOAT_REENT	when --*float-reent* is used
C51_INT_LONG_REENT	when --*int-long-reent* is used

pdata access by SFR

With the upcoming of devices with internal xdata, flash memory devices using port P2 as dedicated I/O port are becoming more prevalent. The setting of the high byte for pdata access, which was formerly done by port P2, is then achieved by a Special Function Register. The address and name of this *SFR* is different from manufacturer to manufacturer, but its function remains the same. In order for the startup code to correctly initialize pdata variables, you should define a *SFR* with the name _XPAGE at the appropriate location if the default, port P2, is not used for this purpose. Some examples include:

```
sfr at (0x85) _XPAGE; /* Ramtron VRS51 MPAGE */
sfr at (0x92) _XPAGE; /* Cypress EZ-USB MPAGE */
sfr at (0x91) _XPAGE; /* Infineon C500 family XPAGE */
sfr at (0xaf) _XPAGE; /* some Silabs ICs EMI0CN */
sfr at (0xaa) _XPAGE; /* some Silabs ICs EMI0CN */
```

Other Features available by SFR

Some 8051/8052 variants offer features like double data pointers, multiple data pointers, the ability to decrement register *DPTR*, 16x16 Multiply, and so on. These features are not used by C51. If you absolutely need them you can you always use by means of inline assembly code.