C/C++ Compiler

User Guide

C/C++ Compiler: User Guide

Table of Contents

Annotation

Described in this document, referring to program “C Language For ISO/IEC 9899:1990(E) Standard And C++ Language For ISO/IEC 14882:1998(E) Standard Compiler” are the following:

- Use of the Command Line;
- Implementation Peculiarities.

Chapter 1. Introduction

C Language is a general-purpose programming language, which allows to effectively use almost all peculiarities of architecture of target platform. C++ Language is object-oriented general-purpose programming language, almost completely compatible with C, which allows to write in C++ as effective programs as in C.

Publications on C/C++ Programming Languages

This list of books will help to study C/C++ Languages:

The C Programming Language, 3-d edition.
Brian Kernighan, Dennis Ritchie.

The C++ Programming Language, 3-d edition.
Bjarne Stroustrup.

How To Program In C++
Harvy Datle, Paul Datle.

The Design And Evolution of C++
Bjarne Stroustrup.

Compiler Overview

Full support of C Language ISO/IEC 9899:1990 (E) Standard (see Implementation Peculiarities).

Full implementation of C Language Library, according to ISO/IEC 9899:1990 (E) Standard.

Full support of C++ Language ISO/IEC 14882:1998 (E) Standard (see Implementation Peculiarities).

Full implementation of C++ Language Library, according to ISO/IEC 14882:1998 (E) Standard.

Support of the features of UMC 320 microprocessor.

Support of inline assembling and interaction of inline assembler code with C/C++ programs.

Support for operation on system and general-purpose registers from C/C++ programs.

Data Types

Fundamental types – built-in C or C++ language types. Species of these types are usually called “variables”.

Fundamental types

Table 1.1. Data Types

Data type	Size	Range
char	1 byte	-127 ... +128
unsigned char	1 byte	0 ... 255
bool	1 byte	true, false
short	2 bytes	-32768 ... +32767
unsigned short	2 bytes	0 ... 65535
int	4 bytes	-2147483648... +2147483647
unsigned int	4 bytes	0 ... 4294967295
long	4 bytes	-2147483648... +2147483647
unsigned long	4 bytes	0 ... 4294967295
float	4 bytes	±1,176E-38 ... ±3,40E+38
double	8 bytes	±1,7E-308 ... ±1,7E+308
long double	8 bytes	±1,7E-308 ... ±1,7E+308
wchar_t	2 bytes	-32768 ... +32767
long long	8 bytes	-9223372036854775808 ... +9223372036854775807
unsigned long long	8 bytes	0 ... 18446744073709551615
__int64	8 bytes	-9223372036854775808 ... +9223372036854775807
unsigned __int64	8 bytes	0 ... 18446744073709551615

Optimizations

Optimizing algorithms are realized in compiler, to switch them use `-OPT` options. By default all optimizing algorithms, which may lead to incorrect compiler work or complicate debugging are switched off. Algorithms, that may not have such consequences, are always switched on.

Compiler Launching. Command Line

For meaningful compiler operation, it is necessary to give additional information to the compiler – to declare the source file name, to specify the operating mode and so on. This information is given to compiler in command line. The general format of command line is the following:

<name of the compiler executable file> <options-line>

Typical example:

isc.exe -I Include test.c

Program Example

A classic program example. It may be compiled with the following command line: **isc.exe -I In-**

clude hello.c. As the result of successful translation, an assembler file (`hello.asm`) will be created, which it is necessary to translate with the help of assembler (`asm.exe`) and link by a linker (`lnk.exe`) with libraries (`rtl.lib` and `clib.lib`).

```
#include <stdio.h>
int main()
{
        printf("Hello, world!\n");
        return 0;
}
```

Chapter 2. C/C++ Compiler

Environment Variables

Compiler uses an `INCLUDE` environment variable to search for include files. Additional paths can be specified in command line with the help [-I] option.

Compiler Launching

Command Line

Compiler is supplied as an executable file `isc.exe`. Depending on source file extension it works as C (.c) or C++ (.cpp, .cxx or .cc) compiler.

Source File And Output File

Compiler proceeds one source file and generates one output file during a single launch. Source file name should be specified with the command line option `-S` (the option itself may be omitted).

Compiler output file may contain either intermediate representation or preprocessor output file, or assembler program text. Type of output file is specified by the options `-p` and `-P`. Output file name may be declared with `-O` option.

Preprocessor output file contains C/C++ program text, with preprocessor directives removed (with the exception, possibly, of `#line` directives) and with all the macro substitutions performed.

By default, an assembler file with source file name and `.asm` extension is created.

General Option Syntax

Option Types

Compiler options are divided into 2 categories: options with arguments and options without arguments (keys).

Options with arguments are denoted by upper-case letters of English alphabet. Keys are denoted by lower-case letters of English alphabet.

An option should start with one or several blank characters, following by the “-“ (minus) character and letter, specifying the option, for example:

-E10 –p –c -W10

You may group keys right after the “minus”:

-E10 –pcl -W10

Such “group” of keys may be followed by an option with argument (but only one):

-E10 –pclW10

Option arguments may (sometimes should – see below) be separated from option letters by one or several blanks:

-E 10 –pclW 10

Default Option Values

Many options have their default values. If the option is present in the command line the value given there overrides the default one.

The default value for keys is “false”. This means that if a key is not specified, the corresponding internal compiler parameter has “false” value. If the key is specified, the corresponding internal compiler parameter is set to have value “true”.

Options with arguments do not have default values, if not otherwise specified in the description of the corresponding option.

Multiple Specification Of An Option In The Command Line

Command line is parsed from left to right.

If there is more than one instance of an option in the command line specified, than the more right option instance we’ll call repeated instance (in relation to the left on). Depending on option semantics, the repeated instance either overrides the value specified by the initial instance, or adds value to it, or is treated as error.

General Syntax Of Option Arguments

A letter, denoting option with argument, should be followed with an argument. Argument is a sequence of letters (substring), not containing blanks. Right before argument (after letter, denoting an option) one or several blanks are allowed. According to option semantics, argument is interpreted as an integer or as a text line.

Arguments-numbers are to be specified in decimal format. The value should have non-negative values from 0 to 2147483647. For a specific option additional limitations may be specified for a number argument.

It is possible to take string arguments in double quotation marks; this gives a possibility to enter string arguments, containing blank characters, in this case there should be at least one blank character between option letter and an argument. To pass string argument, containing “double quotation mark” symbol, it is necessary to insert backslash (\) before each double quotation mark. For a specific option additional limitations or different syntax may be specified for string arguments – see options description.

Options Description

-S `src_file`	*Source File Name* This option specifies for the compiler the source file for translation. The `src_file` parameter should be given according to rules, accepted in the operating system in use. Compiler does not have any assumptions about the source files extension. As a result, it should always be specified explicitly. In this and only this option the option letter **-S** may be omitted.

If the path to the source file is not specified, compiler searches it in the current directory. The path may be specified in relative format; in this case it is interpreted as relative to the current folder.

Example:

```
-S file.cpp
-S "..\my sources\file.c"
c:\myproj\file.cpp
```

-O `out_file`

Output File Name

This option specifies the name and placement of the output file to the compiler. The `out_file` parameter is specified according to rules, accepted by operating system in use. It is possible to omit path, output file name and extension, in this case compiler will use default settings. By default, output file name gets extension `.asm` (if `-P2` is set) or `.ir` (if `-P0` is set).

If the path to the output file is not specified, compiler places it in the current directory.The path may be specified in relative format; in this case it is interpreted as relative to the current folder.

```
-O ofile1.ttt
-O ..\OBJ\ofile2.t3
-O c:\myproj\ofile3
```

-I `ipath`

Include Files Search Paths

This option sets a list of paths , which compiler will use to search for include files.

`ipath` is a list of folders, separated by a semicolon; their syntax should correspond to one of the operation system in use. If the path to the folder is not specified, compiler searches it in the current directory.The path may be specified in relative format; in this case it is interpreted as relative to the current folder.

Paths to include files are included to the internal compiler list of such paths starting from the most left entry in the ipath string and then consequently from left to right. If the **-I** option is repeated, new folders will be added to the beginning of the list. For example, command line, containing **-I** 1;2 **-I** 3;4 will add following to the include directories list: 3; 4; 1; 2.

Notes

Search in folders, specified by `-I` option, is performed before search in folders, specified by the `INCLUDE` environment variable.

The only case when compiler does not search for the include file using the folders list is when the corresponding `#include` directive contains absolute path for the include file.

If the system include file (specified in `#include` directive with angle brackets) is specified without its absolute path, a file will be searched only in folders, enumerated in **-I** options and `INCLUDE` environment variable. User include file (specified in `#include` directive with double quotes) is searched in the following order:

- In the same folder, where the source file is placed;
- In current folder;
- In folders, enumerated in **-I** options and `INCLUDE` environment variable.

If a relative path is used in `#include`directive, the search is performed the same way as if only file name with no path is specified, with the difference that specified relative path is added to each folder in the folder list.

Example:

Compiler launching:

```
C:\Prj\Src\Obj > isc.exe -IC:\Prj\Inc;INC -S..\a.cpp
```

File `C:\Prj\Src\a.cpp` :

```
#include "Sub/inc.hpp"
```

Search for the `inc.hpp` file is performed in following folders and in following order:

```
C:\Prj\Src\Sub
C:\Prj\Src\Obj\Sub
C:\Prj\Inc\Sub
C:\Prj\Src\Obj\INC\Sub
```

-A `opt_file`	*Include options from file to command line*
	
	This option includes the contents of a file (we will call it further an *option file*) to the command line. This allows to avoid command line size limitations of many operating systems. `opt_file` parameter is specified in accordance with the rules, accepted by operating system in use.
	
	If the path to the option file is not specified, compiler searches it in the current directory. The path may be specified in relative format; in this case it is interpreted as relative to the current folder.
	
	Compiler ignores “carriage return” and “line feed” symbols in the option file (with the exception of the fact that these symbols are not to be met *inside* options), thus an option file may consist of an arbitrary number of lines. Besides, comments are supported. Comment may be placed in any line of the option file; it must start with a ‘#’ symbol and end with the end of this line. There are no size limitations for an option file.
	
	Example:
	
	`-A options.opc`
	`-A "..\My Sources\options.opc"`
	`-A c:\myproj\opt1.opc`

-D `name[=value]`	*Macro Definition*
	
	Instructs compiler to define a macro with specified name and value. Allows user to define one or several macros without modification of the source texts of the program. The effect of this option is analogous to adding directive `#define` to the beginning of the module (before all lines of the source file, as if in "zero" line).
	
	Two forms of the option usage correspond to following directives, added in the beginning of the source file:
	
	-D `macro_name=macro_text` corresponds to the directive:
	
	`#define macro_name macro_text`
	
	-D `macro_name` corresponds to the directive:
	
	`#define macro_name`
	
	Semantics completely corresponds to semantics of `#define` directives for object-like macros. Namely, if the value of the macro is not specified, it is set to be empty, multiple entries of this option for macros with the same name are processed as multiple `#define` directives for the same macro name, etc. Situation, when this option is used in command line together with **-U** option, is treated, as corresponding sequence of `#define` and

	`#undef` directives.
	
	This option doesn't allow to declare function-like macros.
	
	Example:
	
	`-D FOO=1`
	`-D MOO`
	`-D "ZOO=(FOO + MOO)"`
	`-D "POO=\"string, containing spaces\""`

-p	*Perform preprocessing only* Instructs compiler to generate file, containing processor output. In this case intermediate representation or assembler text file are not generated. The output file name for the preprocessor output should be specified with the help of **-O** option (see "Output File Name").
-c	*Include comments to the preprocessor output file* Instructs compiler to leave comments untouched while preprocessing; they will be contained in the preprocessor outputfile. Using it makes sense with **-p** option specified only.
-E `max_err`	*Maximum number of error messages* Instructs compiler to interrupt translation, if the number of errors reaches `max_err`. Such situation is interpreted by compiler as fatal error, with a corresponding message given. Default value: 25.
-W `max_warn`	*Maximum number of warnings* Instructs compiler to interrupt translation, if the number of warnings output reaches `max_warn`. Such situation is interpreted by compiler as a fatal error, with a corresponding message given. Default value: 100000.
-w	*Suppress warnings* By default, a compiler displays all warnings (see also description of option Maximum number of warnings). This option instructs compiler to give no warnings. If the option **-w** is specified, as an indirect effect, **-W** `max_warn` option has no effect.
-wn`XXXX`	*Disable warnings number XXXX* Instructs compiler not to display warnings number `XXXX`.

-q	*Form a complex return code(contrary to simply zero or nonzero)* This option allows to get a return code from which more detailed information can be obtained.
-P`phase`	*Compilation phases* This option instructs compiler which of the compilation phases to perform. One of two values can be specified: • `0` – syntax of the source program is checked, output file contains intermediate representation (unless option `-p` is specified); • `2` – complete compilation to the assembler file is performed (default). No spaces should be there between option letter and number of phase.
-OPT`level`	*Optimization level* Specifies optimization level of the resulting program. Option may take one of the following values: • `0` – optimization is switched off(default); • `1` – simple optimizations; • `2` – complex optimizations. No spaces should be there between option letter and degree of optimization.
-i	*Switching off generation of debugging information* By default, debugging information is generated and included in the resulting program. This option switches off debugging informartion generation.
-except	*Switching off the exception handling.* C++ only. By default, support for exception handling is enabled. This option disables it, which results in a faster code.
-rtti	*Switching off RTTI support.* C++ only. By default, support for RTTI (dynamic_cast and typeid) is enabled. This option instructs compiler to switch off RTTI support, resulting in smaller code.
-Ia`file`	*Including a file after source text*

	Adds all contents of file `file` after the end of the source text.
-Ib`file`	*Including file before source text* Adds the contents of file `file` before the source text.

-U `name`	*Macro Definition Removal*
	
	You may undefine macros defined earlier in the command line (and only them) by using option '-U'. This option is analogous to using `#undef` directive at the very beginning of the module (before all lines of the source file; as if in "zero" line).
	
	Option **-U** `macro_name` corresponds to the directive:
	
	`#undef macro_name`
	
	Semantics coinsides with one of the `#undef` directive.
	
	This directive removes only macros, defined earlier in the command line with option `-D`.
	
	Example:
	
	`-U FOO`
	`-U "ZOO"`

-C`codepage`	*Source file codepage setting*
	
	Orders to the compiler to interpret the source text as written in the set coding. The codepage available volumes:
	
	"en_US.850" - english lenguage, code page 850;
	
	"ru_RU.1251" - russian language, code page 1251 (Windows);
	
	"ru_RU.866" - russian language, code page 866 (DOS).
	
	If option **-C**"ru_RU.1251" or **-C**"ru_RU.866" specified output file will be in the 1251 coding.
	
	If option **-C**"en_US.850" specified output file will be in the 850 coding.
	
	The default setting is "en_US.850" (english lenguage, code page 850).
	
	Example:
	
	`-C "ru_RU.1251"`
	`-Cen_US.850`

-M`codepage`	*Compiler messages language and codepage setting*
	
	Orders to the compiler to output all messages in the set language and coding. The codepage available volumes:
	
	"en_US.850" - english lenguage, codepage 850;
	
	"ru_RU.1251" - russian language, code page 1251 (Windows);
	
	"ru_RU.866" - russian language, code page 866 (DOS).
	
	The default setting is "en_US.850" (english lenguage, code page 850).
	
	Example:
	
	`-M "ru_RU.1251"`
	`-Men_US.850`

-T`notation`	*Notation* Specifies suorce file notation • `c` – C file (ISO/IEC 9899:1990); • `c99` – C file (ISO/IEC 9899:1999); • `p` – C++ file.
-?	*Display short list of options* This option instructs compiler to display the list of supported options with their short descriptions in English on the console.

Return Codes

Return code may be used to determine the results of compiler work. By default, 0 indicates success, and nonzero indicates that there were errors. If the option **-q** is specified, result code contains information, from which one may obtain the result of work in general (successful compilation or there were errors etc.), component, that produced an error, and the number of the last diagnostic message.

Return code is passed to opearating system according to rules, accepted by it.

Table 2.1. Compiler return code with option -q specified

Compiler work results	Return Code
No errors, no warnings	0
There were warnings, but no errors	1 + (`message number` * 256)
There were non-fatal errors, and possibly warn-	2 + (`message number` * 256)

Compiler work results	Return Code
ings	
There were fatal errors and possibly other errors and warnings	3 + (`message number` * 256)

`message number` – a number of the last diagnostic message, produced by compiler.

Chapter 3. Language Extensions

Call Agreements And Stack

The compiler by Interstron, ltd. compiler supports `stdcall` call agreement for UMC 320 platform at the present moment, that is, all parameters of called function are passed to it by calling function through program stack, the first parameter being passed at the top of the stack. Program stack grows towards zero address. Further a more detailed description of actions of calling and called parties is given.

1. Every function, generated by a compiler, has register window size, equal to 7. “Calling function registers” and “Called function registers” are mentioned further. Register numbers used with these notions are mentioned as the corresponding function sees them. So, the called function registers with a specified number from 0 to 8 are at the same time calling function registers with number larger by 7 (from 7 to 15 correspondingly).

2. To pass parameters to a function a program stack is used. The first parameter is placed at the top of the stack (at address of the top of the stack), next parameter – at the next address (towards greater addresses) and so on. All parameters are aligned in stack at addresses, multiple by 4. Stack is increasing towards zero address. Address of the top of stack (which is also the frame pointer at the moment of function call) is passed in zero register of the called function. Pointer to structure is always passed instead of structure, that is the structure itself is placed in stack, and its address is put within other parameters . 8-byte values are always placed like that: the lower 4 bytes are located in lower addresses including the cases when they are parameters.

3. Stack cleanup is not performed, as address of the top of stack address is passed to called function and therefore calling function has "its own" address of the top of the stack.

4. Return value, if its type is not a structural type, is passed in zero (in case, its size is not more than 4 bytes) or in zero and in first (in case its size is equal to 8 bytes) called function registers. If the returned value of the called function has a structural type, it is passed through stack as follows: calling function allocates the structure and transfers its address as a first parameter (i.e. before explicit parameters), and the calling function fills it.

Note

If you use C++ compiler, not C, functions written in assembler and functions, called from assembler code, should have extern “C” linkage. In this case, their labels consist of their names preceding with an underscore symbol. Extern “C++” functions get more complicated labels.

Example 3.1. An example of calling assembler function from C function and vice versa

—

file `sample.c`

```
int asmfunc (char c, int i);

int cfunc (char c, int i)
```

```
{
        return i;
}

int main ()
{
        char c = '!';
        int i = 10;
        return asmfunc (c, i); // usual call
}
```

file `sample.asm`

```
global _cfunc
global _asmfunc

        section code
_asmfunc:
        entry 3
        ldbs r1, [r0]         ;loading parameter c to r1
        lid  r2, [r0], 1 ;loading parameter i to r2
        ; some actions
        ld   r3, r0         ;copying stack pointer
        std  [--r3], r2 ;writing parameter i to stack
        std  [--r3], r1 ;writing parameter c to stack
        clf  _cfunc         ;cfunc function call
        ld   r1, r3         ;using return code of cfunc
        ;  some more actions
        ld   r0, r1         ;return value
        ret
        ends
```

—

Directives

Non-Standard Macro Definitions

Compiler contains a set of predefined macros (without parameters), which may be used, for example, to increase informativity of user-defined diagnostic messages, to increase program mobility, to enable conditional translation depending on specified options etc.

Predefined macro names should not be used in `define`, `undef` directives.

Worth mentioning, that any other C++ implementations (compilers of other companies) may contain non-standard predefined macros with the same names, their semantic and values may be different from ones in our compiler. Though this is not likely, one should take this possibility into account. So, to write portable code and use non-standard macros, we recommend such the following method:

```
#ifdef __ISCPP__
```

```
/*
   code, using non-standard macros
*/
#endif
```

```
__ISCPP__
```

Expands to the decimal integer literal, specifying the compiler version (current value is 300). This macro allows to determine, whether the compiler, which processes the source file, is a C/C++ compiler made by our company. In this case this macro is defined and expands to compiler version number (decimal integer literal; for example, 1234 for version 12.34). Most probably other compilers will not define this macro. So, by using #ifdef/#ifndef you may specify code for translation by our compiler along with code for other compilers.

```
__ORCHID__
```

Expands to a decimal integer literal equal to 1.

This macro allows to determine, that UMC 320 microprocessor is a target platform, for which the source text is compiled.

#pragma fastcall

Functions that get parameters throw general purpose registers gan be specified to the compiler. These functions have to be pointed in tth directive

```
#pragma fastcall(functions)
```

, where *functions* - these functions names devided by spaces.

If pointed in the `#pragma fastcall` function has more then five parameters only first five of them will be gotten by function throw general purpose registers.

`_asm` Directive (Inline Assembler)

Sometimes it is necessary to use assembler instructions inside a program in C/C++. The syntax of inline assembler is thfollowing:

```
_asm {
        // assembler code
}
```

Assembler code is just transferred to resulting (after compiler translaton) assembler file. The following substitutions are performed (only in case, if the corresponding identifiers do not coincide with the assembler keywords):

- global variable name is substituted with a corresponding label
- function name is substituted with a corresponding label
- local variable name of the current function is substituted with its offset from the frame pointer

Opening and closing sections is prohibited in inline assembler. Label definition and usage is allowed, with the exception, that the first symbol in label name should not be a latin upper-case letter ‘L’. Also it is not recommended to define labels, which name starts with underscore (‘_’), as this name may coincide with the label, corresponding to any of the functions, written in C/C++ (user-defined or library). All macro facilities and directives are allowed.

Use of inline assembler allowed only inside function bodies.

Keywords

Several keywords are added to this compiler implementation. They are all included into the following table:

Table 3.1. Keywords

__int64	_asm	__asm

`long long` And `unsigned long long` Types

`long long` and `unsigned long long` types are 64-bit integer types, signed and unsigned correspondingly. They may be used in all language constructions, where all other integer types are allowed. There is library support for such types, similar to the support for other integer types.

For example, header file `limits.h` contains macros `LLONG_MIN`, `LLONG_MAX` and `ULLONG_MAX`, denoting minimum and maximum values of `long long` type and maximum value of `unsigned long long` correspondingly.

You may see Chapter the section called “Numbers To Strings And Strings To Numbers Conversion Functions” for additional information about the support for these types.

Code Support According To Platform Limitations

Generated code confines to the platform limitations for instructions order. It is achieved by inserting `nop` instruction where subsequent instructions are not allowed.

Chapter 4. Libraries

Standard header files for C library and C++ language support library are located in `include` subfolder of an installation folder. Header files of full C++ library are located in `STL` subfolder, if one uses them it is not necessary to add `include` subfolder to list of search paths for header files.

Library files are located in `lib` subfolder of an installation folder. Library rtl.lib contains C language support functions, it is necessary for any program, written in C/C++. Library clib.lib contains C standard library functions. Library cpplib.lib contains C++ language support functions. Library stllib.lib contains C++ standard library functions.

`_alloca` macro

`_alloca` macro serves for memory accolation in stack, in stack frame of the current function, for temporary information storage. With `_alloca` macro memory is allocated much faster, than with the help of `malloc()` function, and does not require any special releasing (it is released automatically when the function exits).

The parameter of the macro is the memory size to be allocated in bytes. The result of the macro is a `void*` pointer to allocated memory. Memory, allocated with the help of `_alloca`, can be used before exit of the function, in which it was allocated (including in functions called from it).

To use macros `_alloca` it is necessary to header file `malloc.h`.

Registers Operation Via Pseudovariables

To ease use of registers directly from C/C++ program, register pseudovariables are supported. The value of register may be obtained as value of the corresponding variable. Analogously, assigning value to the corresponding variable will result in writing this value to the register. Variables, corresponding to registers, writing to which is not allowed, have const-qualified type.

There are following pseudovariables (declared in header file `orchid.h`):

Registers	Pseudovariables	Correspondence
`acc`	`__ACC` of the type `volatile acc_t`	correspond to accumulator
`r0 ... r15`	`__R0 ... __R15` of the type `volatile genr_t`	correspond to general purpose registers
`s0, s1`	`__S0` and `__S1` of the type `volatile const sysr_t`	correspond to system registers
`s4 ... s127`	`__S4 ... __S127` of the type `volatile sysr_t`	correspond to system registers

Also for some system registers there are symbols with convenient symbol names, corresponding to their names on platform and in assembler: `__PSW`, `__PC`, `__RSL`, `__RSH`, `__TBA`, `__TDP`, `__DFSTC`, `__CCP`, `__SRP`, `__RFI`, `__RMI`, `__RCI`, `__CWP`, `__SCBP`, `__RSP`, `__SRSP`. About these registers purpose see information about UMC 320 platform architecture.

Types `acc_t`, `genr_t` and `sysr_t` are defined in header file `orchid.h` as follows:

```
typedef unsigned genr_t;            // general register type
typedef unsigned sysr_t;            // system register type
typedef unsigned long long acc_t;   // accumulator type
```

Warning

Registers are used in executed program (explicitly general-purpose registers and accumulator, implicitly some system registers). Incorrect register pseudovariables use may lead to unpredictable results.

Non-Standard C Library Functions Description

Constants

File Constants

Worth mentioning, that following flags use is possible with low-level functions only, operating on environment. Flags `_O_SEQUENTIAL` and `_O_RANDOM` are used as “hints” to library for hard-disc access optimization.

`_O_RDONLY`
: Read-only file.

`_O_WRONLY`
: Write-only file.

`_O_RDWR`
: File is opened for both reading and writing.

`_O_APPEND`
: File is opened for writing to the end of the file.

`_O_CREAT`
: Create and open new file.

`_O_TRUNC`
: Open a file and remove its contents. A permission to write to a file is required. If used together with `_O_CREAT` opens an existing file or makes a new file.

`_O_EXCL`
: Opens file only in case, if file does not exist.

`_O_TEXT`
: Opens file in text mode.

`_O_BINARY`
: Opens file in binary mode.

`_O_NOINHERIT`
Child process does not inherit the file descriptor.

`_O_TEMPORARY`
Temporary file. File is deleted, when the last file descriptor is deleted. Used together with `_O_CREAT`.

`_O_SHORT_LIVED`
A temporary file is made, from which nothing, if possible, is written to disc. Used together with `_O_CREAT`.

`_O_SEQUENTIAL`
Sequential access is used operating on a file.

`_O_RANDOM`
Contrary to previous flag, specifies random access.

_fstat structure fields

Table 4.1. _fstat fields description

Field	Description
st_atime	The time of the last access to file.
st_ctime	The time of file creation.
st_dev	Value 0 corresponds to the disc file. If the file is a device, the variable has the value of this file descriptor.
st_mode	Bitmask.
st_mtime	The time of the last file modification.
st_nlink	Always 1 in non-NTFS file systems.
st_rdev	Value 0 corresponds to disc file. If the file is a device, the variable has the value of this file descriptor.
st_size	File size in bytes.

Special Floating Point Values

Some functions and operations on floating point types may return the following special values (outside parethesis the name for double type is specified, in parenthesis – for float type; corresponding values are different, but have the same meaning for these types). More about functions, operating on these constants you may see Chapter the section called “Functions, operating on special values of floating point numbers”.

`_dqnan` (`_fqnan`)
`NaN` (Not a Number) – not a number. For example, 0/0 or ∞/∞.

`_dsignan` (`_fsignan`)
`signaling NaN` – signaling not a number. Raises corresponding signal.

`_dposinf` (`_fposinf`)

INF (INFinity) – positive infinity. For example, 1/0

`_dneginf` (`_fneginf`)
INF (INFinity) – negative infinity. For example, -1/0

Functions, Operating On Environment

All functions are low-level analogues of standard library functions operating on files. For example, function `_open()` is analogous to function `fopen()` from standard library, and function `_close()` is analogous to `fclose()` and so on.

_open

Open file .

Function	Header file
_open	__replace.h

Synopsis:

```
_HANDLE _open (const char* fileName , unsigned
oflags );
```

Description: Function `_open` opens the file, indicated in *`fileName`*. The way of file opening is indicated in parameter *`oflags`*. Possible flags are enumerated in Chapter the section called “File Constants”.

Return value: File descriptor, zero upon error.

Example:

```
#include <stdlib.h>
#include <stdio.h>
#include <__replace.h>
#include <malloc.h>

char buf[120];

int main()
{
  _HANDLE fh;
  size_t nbytes = 100, bytesread;

/* File read.c is open for reading*/
  if ((fh = _open("read.c", _O_RDONLY|_O_TEXT)) == -1)
  {
    perror( "open failed on input file" );
    exit( 1 );
  }

/* If the file was successfully open, bytesread bytes is read from it */
  if (( bytesread = _read( fh, buf, nbytes ) ) <= 0 )
    perror( "Problem reading file" );
  else  // quantity of bytes actually read from file is displayed
    printf( "Read %u bytes from file\n", bytesread );

  _close( fh );
  }
```

See also: the section called “_close” the section called “_read”

_close

Close file.

Function	Header file
_close	__replace.h

Synopsis: `void _close (_HANDLE file );`

Description: Function `_close` closes the stream. Before closing all buffers connected with this stream are flushed. Receives file descriptor as argument.

Return value: Zero in case of failure, **`errno`** being set to `EBADF`. Non-zero value in case of success.

Example: See example for the section called “_open”

See also: the section called “_open”

_read

Read from file.

Function	Header file
_read	__replace.h

Synopsis: `size_t **_read** (_HANDLE file , void* buf , size_t size );`

Description: Function `_read` reads from file *`file`* to buffer *`buf`* no more than *`size`* byte. Reading starts from the current position of stream, connected with the file. After reading the current position is the next byte after the last byte read.

Return value: Number of bytes actually read, which may be less than indicated in *`size`*. -1 in case of error.

Example: See example for the section called “_open”

See also: the section called “_open” the section called “_write”

_write

Write to file.

Function	Header file
_write	__replace.h

Synopsis:

```
size_t _write (_HANDLE file , void* buf , size_t
size );
```

Description: Function `_write` writes data to file *`file`* read from buffer *`buf`*. No more then *`size`* byte are written to file.

Return value: Number of bytes actually written, which may be less than specified in *`size`*. -1 in case of error.

Example:

```
#include <stdio.h>
#include <stdlib.h>
#include <__replace.h>

char buffer[] = "This is a test of '_write' function";

int main(){
  int fh;
  unsigned byteswritten;

  if((fh = _open("write.o", _O_RDWR | _O_CREAT)) != -1)
  {
    if(( byteswritten = _write(fh,
                          buffer, sizeof(buffer))) == -1)
        perror( "Write failed" );
    else
        printf( "Wrote %u bytes to file\n", byteswritten );

    _flush(fh);
    _close( fh );
  }
}
```

Result:

```
Wrote 36 bytes to file
```

See also: the section called “_open” the section called “_read”

_flush

Flush the buffer, connected with file.

Function	Header file
_flush	__replace.h

Synopsis: `int _flush ( _HANDLE file );`

Description: Function `_flush`cleans the file buffer contents, writing information to disc. Receives file descriptor *`file`* as argument.

Return value: Non-zero value in case of success.

Example: See example for the section called “_write”

See also: the section called “_close”

_getpos

Obtain the value of pointer to the current file position.

Function	Header file
_getpos	__replace.h

Synopsis: `int _getpos (_HANDLE file , fpos_t* pos );`

Description: Function `_getpos` stores the value of *`file`* current file position, in object, to which the second argument *`pos`* points. Obtained value can be used by function `_setpos` for positioning to the saved position.

Return value: Zero in case of error, and any positive number in case of success.

See also: the section called "_setpos"

_setpos

Set the value of current file position.

Function	Header file
_setpos	__replace.h

Synopsis: `int _setpos (_HANDLE file , const fpos_t* pos );`

Description: Function `_setpos` sets the value of *`file`* current file position, according to the object, to which the second argument *`pos`* points. Value *`pos`* should be obtained by previous call of the function `_getpos`. If the function call is successful, indicator of the end of the file is cleaned.

Return value: Zero in case of error, and any positive number in case of success.

See also: the section called “_getpos”

_rename

Rename file or folder.

Function	Header file
_rename	__replace.h

Synopsis:

```
int _rename (const char* oldName , const char* new-
Name );
```

Description: Function `_rename` renames file or folder. Arguments are old and new names correspondingly.

Return value: Zero in case of error, any positive number in case of success.

Example:

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

void main( void )
{
  int  result;
  char old[] = "main1.c";
  char nnew[] = "_main1.c";

  /* Attempt to rename file: */
  result = _rename( old, nnew );
  if( result != 0 )
    printf( "Could not rename '%s'\n", old );
  else
    printf( "File '%s' renamed to '%s'\n", old, nnew );
}
```

Result:

```
File 'main1.c' renamed to '_main1.c'
```

See also: the section called “_remove”

_remove

Remove file.

Function	Header file
_remove	__replace.h

Synopsis: `int _remove (const char* path );`

Description: Function `_remove` removes file.

Return value: 0 in case of success, 1 in case of attempt to remove read-only file, and 2 if file does not exist.

Example:

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

void main( void )
{
  int result = _remove( "_main1.c" );
  if ( result == 1 )
    perror( "Could not delete '_main1.c'" );
  else if ( result == 2 )
    perror( "Error: '_main1.c' does not exits" );
  else
    printf( "Deleted '_main1.c'\n" );
}
```

Result:

```
Deleted '_main1.c'
```

See also: the section called “_rename”

_time

Obtain system time.

Function	Header file
_time	__replace.h

Synopsis: `time_t _time (time* timer );`

Description: Function `_time` determines the calendar time. The result is returned in seconds, since 00:00:00 01.01.1970. If *`timer`* is not zero, a return value is placed in object, specified in *`timer`*. Data should fit in the interval starting with 00:00:00, 01 January 1970 and ending with 19:14:07, 18 January 2038.

Return value: Time in seconds since 00:00:00 01.01.1970.

See also: the section called "_gmtoffset"

_getenv

Obtain current value of the environment variable.

Function	Header file
_getenv	__replace.h

Synopsis: `size_t` **`_getenv`** `(const char* name , char* buf , size_t bufsize );`

Description: Function `_getenv` obtains the current value of the environment variable. The arguments are the name of the environment variable *`name`*, buffer *`buf`*, in which the value is returned, and buffer size *`bufsize`*.

Return value: Number of symbols written into buffer, 0 is environment variable was not found and -1 in case of error.

_terminate

To terminate the program.

Function	Header file
_terminate	__replace.h

Synopsis: `void _terminate (void);`

Description: Function `_terminate` performs low-level actions to terminate the program.

_putchar

To put character to stdout.

Function	Header file
_putchar	__replace.h

Synopsis: `int _putchar (int ch );`

Description: Function `_putchar` puts the character to stdout.

Return value: 0 in case of error, any positive number in case of success.

See also: the section called “_getchar”

_getchar

To get character from file.

Function	Header file
_getchar	__replace.h

Synopsis: `int _getchar (void);`

Description: Function `_getchar` returns character from file. If the file end is reached, function `_getchar` returns the file end indicator.

Return value: 0 in case of error, any positive number in case of success.

See also: the section called “_putchar”

_fstat

Obtain information about open file.

Function	Header file
_fstat	__replace.h

Synopsis: `int _fstat ( _HANDLE handle , struct _stat* buffer );`

Description: Function `_fstat` obtains information about open file with *`handle`* descriptor. Information is saved in the second argument *`buffer`*. Fields of _fstat structure are described in the section called “ _fstat structure fields ”

Return value: 0 in case of success, -1 in case of error (**`errno`** is set to `EBADF`)

_gmtoffset

Obtain information about time zone.

Function	Header file
_gmtoffset	__replace.h

Synopsis: `time_t` **`_gmtoffset`** `(void);`

Return value: Time offset from Greenwich in seconds. For example, for Moscow this offset is equal to 3 hours. Function does not accept arguments.

See also: the section called “_time”

Additional functions.

_fopen_userbuf

Opens file (like `fopen`) not using dynamic memory. Designed for use with debugging versions of functions operating on dynamic memory.

Function	Header file
_fopen_userbuf	stdio.h

Synopsis:

```
FILE *_fopen_userbuf (const char *fileName , const
char *mode , const void *buf , size_t bufsize );
```

Description: `_fopen_userbuf` function opens file (like `fopen`) not using dynamic memory. First two parameters are the same as of `fopen`, *`buf`* is the buffer to place file data, and *`bufsize`* is its size. *`bufsize`* should be greater than or equal to `sizeof(`*`FILE`*`)`

Return: The same as of `fopen` function, except that if *`bufsize`* is less than `sizeof(`*`FILE`*`)`, file is not open and `NULL` is returned.

Example:

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

extern FILE* _debug_malloc_out;

int main ()
{
    FILE buf;
    _debug_malloc_out = _fopen_userbuf ("debug.out",
                                        "wt",
                                        (const void*)(&buf),
                                        sizeof (FILE)); // unbuffered out
    // operations on memory
    _debug_malloc_out = stderr;
    fclose (&buf);
}
```

See also: the section called “_open”

_sys_info

Obtain system information.

Function	Header file
_sys_info	__replace.h

Synopsis: `size_t **_sys_info** (int operation , void *buf , size_t bufsize );`

Description: Function `_sys_info` allows to get system information. *`operation`* is numeric code of operation performed, *`buf`* is user buffer, in which input data is passed and into which output data is written (both depend on operation), *`bufsize`* is buffer size.

Table 4.2. Operations, performed by _sys_info

Code of the operation	Meaning of the data in buffer	
	On function entry	On return from the function
`_SYS_INFO_GET_STACK`	None	Call stack (addresses array), starting from the function, that called `_sys_info`, to the program entry point
`_SYS_INFO_GET_FUNNAME`	Address in code segment	Non-qualified name of the function, containing this address
`_SYS_INFO_GET_COORDS`	Address in code segment	Coordinates in source code (a string of format: fi-

Code of the operation	Meaning of the data in buffer	
	On function entry	On return from the function
		le-name:line_number)

Return value: If buffer size is enough for placing output data in it, the size of really written data is returned, if not – required buffer size (nothing is written to buffer). In case of error, (for example, *buf* == NULL for operation code, implying input data in buffer) (size_t)(-1) is returned.

_msize

Determine the size of dynamic memory block.

Function	Header file
_msize	malloc.h

Synopsis: `size_t` **`_msize`** `(const void* pH );`

Description: Function `_msize` returns size of memory block, allocated by functions `calloc`, `malloc` or `realloc`. A pointer to memory block (*pH*) is passed as an argument.

Return value: Size of memory block in bytes. In case of error (memory block was not allocated or was released) -1 is returned.

_toupperstr

Change letters to upper-case.

Function	Header file
_toupperstr	ctype.h

Synopsis: `char * _toupperstr (char* s );`

Description: Every string letter is changed from lower-case to upper-case, if it's possible.

Return value: Transformed string.

See also: the section called "_tolowerstr"

_tolowerstr

Change letters to lower-case.

Function	Header file
_tolowerstr	ctype.h

Synopsis: `char * _tolowerstr (char* s );`

Description: Every string letter is changed from upper-case to lower-case, if its possible.

Return value: Tranformed string.

See also: the section called “_toupperstr”

Numbers To Strings And Strings To Numbers Conversion Functions

Number, buffer and in some cases base of scale of notation (2 <= base <= 36) are passed as arguments.

String to number conversions are performed with standard functions (see Chapter 7.20.1.4 of ISO/IEC 1899:1999 C language Standard).

_ulltostr, _ultostr, _lltostr и _ltostr

Convert integer to string.

Function	Header file
_ulltostr	stdlib.h
_ultostr	stdlib.h
_lltostr	stdlib.h
_ltostr	stdlib.h

Synopsis:

```
char* _ulltostr (char* s , unsigned long long value , int base );

char* _ultostr (char* s , unsigned long value , int base );

char* _lltostr (char* s , long long value , int base );

char* _ltostr (char* s , long value , int base );
```

Description: Conversion of integer of corresponding type (*value*) into string placed in buffer *s*. After function execution *s* contains a sequence of characters (digits and latin letters), representing a number *value* in scale of notation *base*. Argument *base* is to be more or equal to 2 and less or equal to 36.

Return value: *s*.String, pointed by *s* is unchanged if argument *base* goes behind limits (from 2 to 36 inclusively). NULL, if NULL was passed as the first argument.

See also: the section called “_lltoa, _ltoa and _itoa” the section called “_dtoa”

_lltoa, _ltoa and _itoa

Convert integer into string, using decimal scale of notation.

Function	Header file
_lltoa	stdlib.h
_ltoa	stdlib.h
_itoa	stdlib.h

Synopsis:

```
char* _lltoa (char* s , long long value );

char* _ltoa (char* s , long value );

char* _itoa (char* s , int value );
```

Description: Conversion of integer of corresponding type (*value*) into string placed in buffer *s*. After function execution *s* will contain a representation of *value* in decimal scale of notation.

Return value: *s*.

See also: the section called “_ulltostr, _ultostr, _lltostr и _ltostr” the section called “_dtoa”

_dtoa

Convert floating-point number into string.

Function	Header file
_dtoa	stdlib.h

Synopsis:

```
char* _dtoa (char* s , double value , int precision
, char eE );
```

Description: Conversion of the floating number *value* into string, placed in buffer *s*. The string will contain *precision* digits after decimal point. If *precision* == 0, 6 digits will be kept by default, if *precision* is equal to -1, string will represent either integer part of the number, i.e. largest integer, less than or equal to *value*, without decimal point, or in case of scientific notation it will have a mantissa of only one digit. Argument *eE* defines which notation is to be used of conversion of the number — scientific ('e' or 'E'), or usual representation of the number, in the latter case 0 should be passed. If passed *eE* parameter is different from specified values, default actions take place, namely, the number is converted in scientific notation.

Return value: *s*

Example:

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

int main()
{
  char buf[20];
  double x = 1.23456e+123;

  // Convert a number to string. String should contain the value
  // in scientific notation with 2 digits after decimal point
  _dtoa(buf, x, 2, 'e');

  printf("buf = %s\nx   = %.2e\n", buf, x);

  return(0);
}
```

Result:

```
buf = 1.23e+123
x   = 1.23e+123
```

See also: the section called "_ulltostr, _ultostr, _lltostr и _ltostr" the section called "_lltoa, _ltoa and _itoa"

_fcvt and _ecvt

Convert a floating-point number into string.

Function	Header file
_fcvt	stdlib.h
_ecvt	stdlib.h

Synopsis:

```
char* _fcvt (double value , int precision , int* dec
, int* sign , char* buf );

char* _ecvt (double value , int precision , int* dec
, int* sign , char* buf );
```

Description: Also convert double floating-point number to string (`_fcvt` in usual format, `_ecvt` – in scientific notation). Essential distinction from `_dtoa` is the fact, that in buffer *`buf`* the value without decimal point and sign is stored. Functions accept the following arguments: *`value`*, *`buf`* and *`precision`* – value to convert, buffer and precision indicator (number of digits after decimal point), they have the same meaning, as corresponding arguments of `_dtoa` function; following parameters will be placed by addresses *`dec`* and *`sign`*: decimal point position from the beginning of string (can be positive, zero or negative) and sign (0 means '+', any other number – '-') correspondingly.

Return value: *`buf`*

See also: the section called “_dtoa”

Mathematical Functions

_dapproxcmp

Approximate comparison of floating-point numbers.

Function	Header file
_dapproxcmp	math.h

Synopsis: `int` **`_dapproxcmp`** `(double x1 , double x0 , double epsilon );`

Description: Comparison of two arguments (*x0* and *x1*) for equality with *epsilon* precision, which means: absolute value of their difference is less than *epsilon*.

Return value: 1 in case of success (arguments are equal with specified precision), 0 otherwise.

_powExactly

Fast raise to power.

Function	Header file
_powExactly	math.h

Synopsis: `double _powExactly (double base , unsigned exponent );`

Description: This function is analogous to the `pow` function, except that due to second argument is integral it is faster and more precise.

Return value: Analogous to `pow` standard function (see Chapter 7.5.5.1 of C Language ISO/IEC 9899:1990(E) Standard)

_round

Rounding of number.

Function	Header file
_round	math.h

Synopsis: `double _round (double x , int precision );`

Description: Rounding of *x* to *precision* digits after decimal point. Rounding is performed according to rules of rounding.

Return value: Rounded number value. In case of error **errno** is set to EDOM.

hypot

To calculate hypotenuse.

Function	Header file
hypot	math.h

Synopsis: `double hypot (double x , double y );`

Description: Finds hypotenuse, supposing arguments *x* and *y* to be cathetuses of right triangle. This function call is analogous to calculation of square root of sum of squares of *x* and *y*.

Return value: Hypotenuse length of right triangle.

Functions, operating on special values of floating point numbers

Additional functions for operation on floating point numbers. Allows to determine, if the number is a positive or negative infinity, not-a-number. For more info on special floating-point values see Chapter the section called “Special Floating Point Values”.

_disqnan, _fisqnan

Determine, whether the argument is not-a-number.

Function	Header file
_disqnan	_dconst.h
_fisqnan	_fconst.h

Synopsis:

```
int _disqnan (double d );

int _fisqnan (float f );
```

Description: Checks if the argument value `d` (for function `_disqnan`) or `f` (for function `_fisqnan`) is quiet NaN. Functions `_disqnan` and `_fsiqnan` determine the so-called quiet NaN ("quiet" not-a-number).

Return value: 1, if a parameter is a quiet NaN, 0 otherwise.

Example: See example for the section called “_disinf, _fisinf”

See also: the section called “_disinf, _fisinf” the section called “_dissignan, _fissignan” the section called “_disspecial, _fisspecial”

_dissignan, _fissignan

Determine, if the argument is signalling not-a-number.

Function	Header file
_dissignan	_dconst.h
_fissignan	_fconst.h

Synopsis:

```
int _dissignan (double d );

int _fissignan (float f );
```

Description: Checks, if the argument value `d` (for function `_dissignan`) or `f` (for function `_fissignan`) is a signaling NaN. Functions `_dissignan` and `_fissignan` determine the so-called signaling NaN.

Return value: 1, if the parameter is a signaling NaN, 0 otherwise.

Example: See example for the section called "_disinf, _fisinf"

See also: the section called "_disinf, _fisinf" the section called "_disqnan, _fisqnan" the section called "_disspecial, _fisspecial"

_disinf, _fisinf

Determine if the argument is infinity (positive or negative).

Function	Header file
_disinf	_dconst.h
_fisinf	_fconst.h

Synopsis:

```
int _disinf (double d );

int _fisinf (float f );
```

Description: Checks, if *d* (for function `_disinf`) or *f* (for function `_fisinf`) is a positive or negative infinity.

Return value: 1, if the parameter is infinity, 0 otherwise.

Example:

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main()
{
  double x = -HUGE_VAL;  // the value of HUGE_VAL is _dposinf.

  if(_disinf(x))
  {
    if(x > 0)
      printf("POSINF\n");
    else
      printf("NEGINF\n");
  } else
       printf("x = %e is not INF\n");

  return(0);
}
```

Result:

```
-HUGE_VAL
```

See also: the section called "_disqnan, _fisqnan" the section called "_dissignan, _fissignan" the section called "_disspecial, _fisspecial"

_disspecial, _fisspecial

Determine if the argument is one of the special values.

Function	Header file
_disspecial	_dconst.h
_fisspecial	_fconst.h

Synopsis:

```
int _disspecial (double d );

int _fisspecial (float f );
```

Description: Functions `_disspecial` and `_fisspecial` determine, if the argument is one of special values : infinity (positive or negative) or not-a-number (NaN) (signalling or quiet).

Return value: 1, if the parameter is a special value, 0 otherwise.

Example: See example for the section called “_disinf, _fisinf”

See also: the section called “_disinf, _fisinf” the section called “_dissignan, _fissignan” the section called “_disqnan, _fisqnan”

Peculiarities Of Locale Realisation

In Interstron Ltd. Libraries (both C and C++) only "C" locale is supported, as demanded according to standards of these languages. Also only the so-called basic character set is supported, that is, characters with ASCII-codes from 0 to 127 (their character representation also corresponds to ASCII). For wide characters analogous extended character set is supported.

Chapter 5. Optimizations

During assembler code generation architecture limitations are taken into account.

Optimizations, Independent Of The Target Platform

Raise of local variables to registers. This may significantly decrease the time of program execution, as interaction with register is much faster than interaction with memory. Besides, this allows saving memory.

Removal of excess copying between registers. In process of code-generation a compiler creates excess copyings between registers. This optimization removes such copyings. At present only part of such copyings is removed, however, in nearest future algorithms will become better and all or the dominant majority of such copyings will be removed.

Removal of unreachable code. Code, which can never be executed is removed.

Removal of "dead" code. Dead code is the code, which does not influence the function return value and does not produce side effects. Dead code is removed. This may considerably decrease the program size and the time of its execution.

Removal of common subexpressions. If several expressions have one subexpression in common or one of them is a subexpression of others, this optimization leads to the following: this subexpression is calculated only once, not for each of the containing expressions. This optimization is local, that is, it operates within the block.

Note

Block is a sequence of assembler instructions, containing no jumps. The program consists of blocks and jumps between them. *Local optimization* is an optimization, operating within the boundaries of one separate block.

Optimization of jumps and branchings. Within this optimization branching concatenation is performed, if they lead to the same place, and conversion of conditional jump into unconditional; removal of exceed jumps (if a block has only one descendant and it is its only ancestor, they are concatenated, the corresponding jump instruction being removed; if a block consists of jumps only, they are added to every ancestor, the block being removed).

Simplification of operations on specific constants. The following things are simplified: addition to zero, substraction of zero (are transformed to copying and can be removed), multiplication by zero, division of zero, remainder calculation with module 1 or -1 (is transformed to loading zero), multiplication and division by the power of 2 with sign (are transformed to the corresponding shift by exponent value and negation, if the sign is negative), shifts by zero (are transformed to copyings and can be removed).

Complete global constant propagation. Within this optimization not only the preparation of a more complete constant substitution into instructions as immediate arguments will be done, but all constant expression calculations, and also optimization of jumps by condition, which is a constant expression will be performed in the compilation phase.

Global register allocation with register coalescing. If lifetimes of the values in registers do not intersect, these values may be placed in one register. Global register allocation considerably decreases the number of register loads/stores to memory (the so-called spill code). Besides, within this optimization the removal of all (or the dominant majority) of excess value copyings between registers will be performed.

Optimizations, Based On UMC 320 Platform Features

Constants substitution in instructions (elementary constant propagation). Constants in generated code are loaded into registers and then operations on these registers are performed. However, execution time and code size may be decreased by using instructions with immediate operands. This is done with this optimization. If the constant is numeric (that is known at the phase of compilation), an optimal instruction is chosen depending on the size, sign and some other conditions of the constant (a choice between `ldi`, `ldis` and pair `ldi`/`ldhi`is made; between `adt`, `adi` and `add` with preliminary loading of the constant; between `li_(s)`/`si_` and `ld_(s)`/`st_` with preliminary loading of the offset ('_' means one of the suffixes b, w, d)).

Use of instructions of loading/saving with address decrement/increment. Load/store and addition/substraction of address of a number equal to the element size are combined into instructions of the kind

```
ld_(s) r1, [r2++]
ld_(s) r1, [--r2]
st_ [r1++], r2
st_ [--r1], r2
```

('_' denotes one of the suffixes b, w, d).

Use of load/store instructions with address, equal to the sum of two registers. Generated code (for example, addressing to the array element) often have the following form: index register multiplication by element size, its addition to the address register, load/store at address, represented with their sum. This optimization converts this code to instructions `lr_(s)`/`sr_` ('_' denotes one of suffixes b, w, d).

Comparisons/transitions in corresponding to platform architecture form. In generated code compiler creates comparison and jumps of the following kind: comparison with the result recorded in the register and jump depending on the value in register. This optimization (it is obligatory, that is always performed) converts this into comparison setting flags and jump depending on flags.

Appendix A. Release Notes

Unsupported features

1. The version of obvious specialization of tamplate classes and functions described in item 14.7.3#18 of the Standard is not supported when one or severaled of covering template classes remain not specialized. Example:

```
template<class T> struct A
        {
                template<class U> struct B
                {
                        static int m;
                };
        };
        template<> template<class U> static int A<int>::B<U>::m = 1;
```

   Let's notice, that this property is not supported by any known for us compilers.

2. The temlpate export is not supported. Only EDG compiler supports the template export of all known compilers.