StormC

COLLABORATORS			
	TITLE : StormC		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		April 16, 2022	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

StormC

1.1 StormC.guide

```
                StormC Demo V2.0

Software and documentation
(c) 1996/97 by HAAGE & PARTNER COMPUTER GmbH

Table of contents


             License agreement

             Chapter  1
                Welcome to a new era

             Chapter  2
                Requirements

             Chapter  3
                Installation

             Chapter  4
                What to do in the case of "insoluble" problems

             Chapter  5
                Tutorial

             Chapter  6
                Your first program

             Chapter  7
                Generating a new project

             Chapter  8
                Make and dependency of modules

             Chapter  9
                Editing the source

             Chapter 10
```

1.2 StormC.guide/ST_Order

Please print the enclosed form on your printer.

Please check the desired products and fax or send us the completely filled-out form.

Our address:

HAAGE & PARTNER COMPUTER GmbH
PO box 80

61188 Rosbach v.d.H.

Fax: +49 6007 / 7543

Order(please mark the desired item(s))

```
*  Yes, send me the complete version of StormC
   at a price of 598,- DM


*  I want to order the Cross-Upgrade from my old

   Compiler system: _____________________________________

at a price of 498,- DM

If you do not live in Germany you have to pay in advance plus
20,- DM for shipping.

(please mark)

* Per enclosed cash or advance-check

* I'll pay with Creditcard

* VISA         * Eurocard/Mastercard

Name on the card:_____________________________________________

Cardnumber:      |_|_|_|_| |_|_|_|_| |_|_|_|_| |_|_|_|_|

Expiry date:_____________________________

Date:_________________ Signature:_____________________________


First name: ___________________________________________

Name: _________________________________________

Street: _______________________________________

Zip code: ___________ Town: ___________________________________

Country: ___________________________________

Telephone: _____________________________________________

E-mail: ________________________________________
```

1.3 StormC.guide/ST_CRIGHT

```
Copyrights and trademarks:
Commodore and Amiga are registered trademarks of ESCOM Inc.
SAS and SAS / C are registered trademarks of the SAS institute.
Amiga, AmigaDOS, Kickstart and Workbench are trademarks of ESCOM Inc.
The designation of products which are not from the HAAGE & PARTNER
COMPUTER Ltd. serves information purposes exclusively and presents no
```

trademark abuse.

1.4 StormC.guide/ST_Lizenz

Licensee agreements

1 In general
(1) Object of this contract is the use of computer programs from
the HAAGE & PARTNER COMPUTER GmbH, including the manual as well as
other pertinent, written material, subsequently summed up as the product.
(2) The HAAGE & PARTNER COMPUTER GmbH. and/or the licensee indicated
in the product are owners of all rights of the products and the trademarks.

2 Right of usufruct
(1) The buyer does receive a non-transferable, non-exclusive right, to use the
acquired product on a single computer.
(2) In addition the user can produce one copy for security only.
(3) The buyer is not legitimated, to expel the acquired product, to rent,
to offer sublicenses or to put these in other ways at the disposal of other
persons.
(4) It is forbidden to change the product, to modify or to
re-assemble it. This prohibition counts also for the translating, changing,
re-engineering and re-use of parts.

3 Warranty
(1) The HAAGE & PARTNER COMPUTER GmbH guarantees, that up to the point in
time of delivery the data carriers are physically free of material
and manufacturing defects and the product can be used as described in the
documentation.
(2) Defects of the delivered product are removed by the supplier within a
warranty period of six months from delivery. This happens through free
replacement or in the form of an update, at the discretion of the supplier.
(3) The HAAGE & PARTNER COMPUTER GmbH does not guarantee that the
product is suitable for the task anticipated by the customer.
The HAAGE & PARTNER COMPUTER GmbH does not take any responsibility for any
damage that may be caused.

(4) The user is aware that under the present state of technology it is not
possible to manufacture faultless software.

4 Other
(1) In this contract all rights and responsibilities of the contracting parties
are regulated. Other agreements do not exist. Changes are only accepted in
written form and in reference to this contract and have to be signed by both
parties.
(2) The jurisdiction for all quarrels over this contract is the court
responsible at the seat of HAAGE & PARTNER COMPUTER GmbH
(3) If any single clause of these conditions should be at odds with the law
or lose its lawfulness through a later circumstance, or should a gap in these
conditions appear, the unaffected terms will remain in effect.
In lieu of an ineffective term of the contract or for the completion of the gap
an appropriate agreement should be formulated which best approximates within
the bounds of the law the one that the contracting parties had in mind as they
agreed on this contract.
(4) Any violation of this license agreement or of copyright and trademark rights

will be prosecuted under civil law.
(5) The installation of the product constitutes an agreement with these license conditions.
(6) If you should not agree with this license agreement you have to return the product to the supplier immediately.

September 1995

1.5 StormC.guide/ST_Welcome

Welcome to a new era of Amiga programming.

With the enclosed preview of our brand-new compiler system you will get to know the abilities of a progressive programming language.

In a so-called integrated environment you will find everything you require for programming. Heart and center is the project management facility, from which all other components are invoked and are provided with data. The project manager is not simply a better MAKE, but an administration for all your program modules. Among them e.g. sources, object libraries, documentation, ARexx scripts, pictures and resources are managed. All compiler, editor and project options are managed from here too. If you are under the impression now that controlling all this must be much too complicated , I can set your fears at rest. Please look at the next pages, where the first example is described and you will realize very quickly, that everything can be done very easily and intuitively.

A further component of the system is the editor with its particular ability to emphasise keywords and syntax characteristics colourfully. With this text colouring you can read your program much easier, because you will be better able to see its structure. Apart from this it helps you avoid errors while editing your sources. As soon as a keyword or an Amiga function is entered, the word is marked colourfully and you know you did it right.

Next, allow me to introduce you to the extraordinary debugger. Extraordinary because it makes no difference whether the editor or the debugger is running. The debugger uses the abilities of the editor, this means that the debugger uses the editor window for its output. So you can watch the source, set breakpoints, look for functions and variables and so one with the ease of using the editor. The structuring and the colouring of the source are helping you to do your debugging job.

When we planned the features of StormC, we naturally had some ideas in mind that we still plan to add in the future. In future versions we intend to let you integrate changes to the source file that you make from the debugger directly into the running program. You won’t need to leave the debugger, nor to compile and re-start the program. As you see, this will make software development much easier and much more efficient.

Another big help for the debugger is our "RunShell". With it it is possible to locate typical errors in OS programming very quickly. One example of the errors that are made again and again, can be best illustrated with the functions AllocMem() and FreeMem(). Allocating memory is easy enough to do, but giving it back to the OS seems to be a big problem for many programmers. Either they forget to free all memory or they get the size of the block wrong, usually

resulting in a CPU exception. The RunShell remembers all important data relating to system resources. Thus it knows exactly when these functions are called too often, not often enough or with incorrect arguments.

Another big advantage of the RunShell is the possibility to start the debugger at any time while running the program. You do not have to decide this before you start the program. If you want to debug the running program, just start the debugger from the RunShell. It is that easy.

Now, let us talk about the most important part of our development system - the compiler. Object-oriented programming is all the rage. Hardly any software developer programs in ANSI C anymore, at least that is the impression one gets. The truth, however, is quite the opposite. While many programmers use C++ compilers, these are suited just as well for translating ANSI C code.

That is why we decided to make a compiler for both parties. The traditional programmers will use our very fast and compatible ANSI C compiler. They can switch to object oriented programming with C++ at any time, completely or partially. StormC is their tool for the future. The others will use our outstanding C++ compiler. StormC implements C++ according to the design by Bjarne Stroustrup and it supports the extended AT&T 3.0 standard. The compiler generates code for all Motorola 680x0 CPUs including the 68060. The fundamentally outstanding speed of the compiler is accelerated by a large factor through the use of precompiled header files. The integrated linker processes all current library formats (SAS/C, MaxonC++, ...) and is one of the fastest linkers for the Amiga to boot.

StormC is suitable for all programming projects, be they administrative, graphics, music or game programs. For all these projects StormC should be your first choice. The existing preview version of StormC helps you with the decision for your future compiler system. Therefore we do not offer a self-running demo version; after all you will certainly want to test it with your own sources and compare the system to your old compiler.

The preview version of StormC has no limits in source length or things like that. But you should always keep in mind that it is a preview, not the final version, which will be released in January 96. There will be some changes to the GUI and the functionality and of course to the stability of the whole system.

If you find a function, that does not work or does not work the way you would like it, please feel free to send us a message.

Our address:

HAAGE & PARTNER COMPUTER GmbH
PO Box 80
61191 Rosbach
Germany

Phone : ++49 - 6007 - 93 00 50
Fax : ++49 - 6007 - 75 43

Compuserve: 100654,3133
Internet: 100654.3133@compuserve.com
WWW: http://home.pages.de/~haage
WWW: http://ourworld.compuserve.com/homepages/~haage_partner

1.6 StormC.guide/ST_Maschine

Requirements

In the following list you will find the minimal configuration for using StormC:

- Amiga with MC68020 CPU and a hard disk
- Kickstart and Workbench 3.0 (v39)
- 6 MB RAM
- 10 MB hard disk space

With this computer system you can start programming with StormC, but the project size is limited. Furthermore not all debugger features can be used. For this you need at least a 68030 CPU with MMU and more RAM.

A really good configuration for StormC is the following:
- Amiga with 68030 including MMU
- Kickstart and Workbench 3.1
- 18 MB RAM
- 60 MB hard disk space

As a rule always remember: THE MORE THE BETTER!

1.7 StormC.guide/ST_Install

Installation

The standard AT/Commodore "Installer" is used for installing the package on your hard disk. As most Amiga software uses this program you are probably familiar with its operation.

Please insert the first disk of the preview in your disk drive and double-click on the disk icon. Before you start the installation, please read the "Readme" file in the root directory of the disk. Herein are important additions and hints that could not be included in the manual any more.

After this, double click the icon "Install StormC to HD" and wait while the Installer utility and the installation script are loaded. Now follow the instructions of the installation program. If you are not sure what to do, simply click on the "Help" button to get further information.

If the installation was successful you will receive a corresponding message.

If the installation was not successful, please repeat it while writing a LOG FILE. The option "Log all actions to: Log File" can be selected in the option window, right at the beginning of the installation procedure. After the (unsuccessful) installation you can read this log file to find out what went wrong. Remove the cause of the problem and start the installation again.

1.8 StormC.guide/ST_Problem

What do in the case of "insoluble" problems

If you have problems with StormC during installation or afterwards, please feel free to contact us.

You can reach us at:

HAAGE & PARTNER COMPUTER GmbH
PO Box 80
61191 Rosbach
Germany

Phone: ++49 - 6007 - 93 00 50

(on workdays between 9:00 and 17:00 (MEZ))

Fax : ++49 - 6007 - 75 43

Compuserve: 100654,3133

Internet: 100654.3133@compuserve.com

WWW: http://ourworld.compuserve.com/homepages/haage_partner

1.9 StormC.guide/ST_Tutorial

Tutorial

In the following part of the manual you will learn everything about the operation of StormC. The Features of the compiler system will be shown using convincing examples. You'll learn the basics of the project manager, how to edit source files and how to start the compiler. A step-by-step example will show you how to work with the debugger.

With this tutorial you will get an impression of the power of the StormC development system, and you will never want to work with another one.

1.10 StormC.guide/ST_Start

Your first program

Now you will see how to start a new project, how to edit the source and how to compile and start it.

Please start StormC through a double click on the program icon. You will find the program in the drawer you installed StormC into.

During startup you will see a welcome message, which remains visible on the screen while components of StormC are loaded. After this you will see a horizontal toolbar near the top of your screen. It contains the most important

functions of StormC, close at hand for quick access.

The icon bar provides the following functions:

New Text
Load Text
Store Text

New Project
Load Project
Store Project

Start Compiler (Make)
Execute Program (Make and Run)
Start Debugger (Make, Run and Debug)

We will start - of course - with the one and only typical example: "HELLO WORLD".

Hello World-source code

1.11 StormC.guide/ST_Project

Generating a new project

Please click the icon "new project". A new project window is created.

What is a project?

A project is the summary of all related parts of your program, e.g. C, C++ and assembler sources, headers, object files, link libraries, documentation, graphics, pictures and other resources. Through the separation in various sections you will always keep an overview. The project manager is also a graphically-oriented Make.

1.12 StormC.guide/ST_Make

Make and dependency of modules

On each compiler pass the dependence between ".o", ".h", ".ass", ".asm", ".i", ".c" and of course ".cc" and ".cpp" files are checked by the project manager. So the project manager knows by itself, that a C source must be recompiled if a ".h" header which was included in the ".c" source has been altered.

At the click on the icons "Make" or "Run" all dependencies are examined. The Make routine now decides which module of the program must be compiled again. "Run" only differs from "Make" in that after successful compilation, the program is started automatically.

Project store and the making of a new directory

Next you should store your project into a new directory. Click on the "Save Project" icon. Select the directory "StormC:" and enter the path and file name "Hello World/Hello World". The suffix ".¶" is appended automatically and indicates that this is a StormC project. You can enter this extension manually by typing <ALT> + <P>.

You may wonder why an empty project should be stored, even when the project has no contents yet. We recommend doing this because now the paths to the sources and resources can be written relative to the project path, and it is easier to handle the whole project. Moreover this path will be the default in the file requester.

1.13 StormC.guide/ST_Source

Editing the source

Now we can start with our project. Please open a new editor window with a click on the icon "New Source".

Before you begin to enter the source, I will introduce the main elements of the editor which are in the upper tool bar.

E = toggles hide/show end-of-line marks
T = toggles tab marking on/off
S = toggles space marking on/off
I = toggles auto-indentation on/off
O = toggles overwrite mode on/off
R = toggles write protection on/off

The next field indicates whether the text has been modified.

At the right side of the toolbar is the rows and columns display.

Please type the following text now:

```
/ * Hello World
    demo of the preview tutorial * /

# include <stdio.h>

void main (void)

printf (  "Hello World\n");
}
```

Store this program with the name "Hello World.c" in the directory "Hello World".

Choose the menu item "Project / Add Window". Now the file name appears in the source section of the project window. The project manager looks at the file-name extensions to recognize the different file types.

Before you start the compiler you should indicate a file name for your program.

Otherwise the default filename "a.out" is chosen.

Select the menu "Project" and the item "Select name of the program". Make sure that the project window is the active window.

1.14 StormC.guide / ST_Compile

Compiling

Click on the project icon "Make". The compiler's error window is opened and the translation of the source starts. During translation some messages may be printed in the error window.

If an error occurs during compilation, a description will be printed here which contains the (possible) cause of the error and the number of the line in the source file where it occurred.

Double-clicking on an error message will get you back to the editor and to the source line that caused it. Make your corrections and compile again.

1.15 StormC.guide / ST_PRGStart

Starting a program

After successfully compiling and linking your program, the button "Start" becomes accessible. Click on it or the "RUN" icon in the tool bar to start your program.

If you choose "Run" instead of "Make" the project management first verifies that all modules have been compiled. If this is not the case, it now starts the compiler. After successful translation, the program is started automatically.

The RunShell

Starting a program from the project manager is something more special, so I am going to tell you about it.

You will have noticed that when you start your program, another window is opened. This one is called the "RunShell".

Naturally it is also possible to simply start the program, but if your program is under development you often wish to have more control over the running program. With StormC's RunShell it is possible to debug the program after running it. If an error occurs that normally causes a CPU exception, the RunShell takes control (in most cases) and gives you the opportunity to look at the error in the source.

A further important characteristic of the RunShell is its "Resource Tracking" capability. To allow this all system functions pertaining to resource handling (AllocMem(), OpenWindow(), Open(), ...) are recorded. When the program is finished, the Resource Tracker checks if there are resources which have not been freed, or that have been freed more than once. You can now go directly to the

source and make the necessary changes immediately.

Furthermore the RunShell offers the possibility to send the signals Ctrl-C, Ctrl-D, Ctrl-E, Ctrl-F, which can normally only be sent by the Shell (CLI) from which the program was started.

The priority of the program can also be set to any value between -128 and +127.

The "Pause" button stops the active program. "Pause" is a toggle button. A further click and the program runs again.

With a click on the "Kill" button the program will terminate. All allocated resources are released (storage and signals are released; screens, windows, requester and files are closed). As a result no "dead" tasks will remain in your system, which can normally emerge quite easily from inadvertently programmed endless loops. This feature will save you a lot of time, because it is a much faster way of cleaning up your Amiga than rebooting it.

In our example the program is processed so quickly that the RunShell window will be closed again almost immediately. The next example will show you more of the RunShell and how to work with the debugger.

The output window ("Hello World") that is still open on your desktop is a normal console window. This type of window is opened automatically if your program uses standard input/output routines such as the "printf" function. To close it simply click on the window’s close gadget.

1.16 StormC.guide/ST_Debug

The debugger

The debugger is required for fast detection of any bugs in your program. It allows you to define breakpoints in your programs and to observe changes of variables, structures and classes at these places. In this way errors can be encircled and you will be able to remove them quickly.

To demonstrate the operation of the debugger, you should load an existing project and compile it.

In the directory "examples" you will find the file "Colorwheel". Open the file on the Workbench with a double click on the icon. The project will be loaded and indicated. Choose the menu "Settings" and the item "Project". Click on the upper cycle gadget until "C/C++ options" appears. As you see, "large debug files" are activated.

Start the compiler with a click on the debugger icon. The system now checks whether there are debugger files for all modules or if they must be compiled. This is the same procedure as if you hit the RUN icon. After linking the program will be started in debug mode.

Depending on the preset preferences the module window, the active-variables window and the monitored-variable window will be opened.

Furthermore the module which contains the main function is opened and its source indicated, starting with the main() function.

You will note that the contents of the editor window have changed a bit. The first column of the text is placed further to the right, so that the freed column can be used to show breakpoints. The breakpoint fields are only in the lines at which the program can be stopped. If you click on one of these points they marked with 'X'. This indicates an active breakpoint.

Set a breakpoint directly after the "OpenScreen" call. If the "current variables" window is not active, open it by selecting the menu item "Windows / Current variables".

Click on the icon "Go up to the next Breakpoint" on the RunShell button bar.

Since you have set the breakpoint directly after the function call to "OpenScreen", the program is executed up to here. A new screen is opened and the program is halted.

The next function "GetRGB32" provides an array of unsigned long characters with data. We want to give this array a closer look.

Here are some explanations of the buttons in the "inspect" window:

Q = show corresponding place in the source code
F = show member function of a class type
I = inspect
P = previous
H = show HEX editor
W = take over the variable to the inspect window
Cast = temporarily change type of variable
Low/High = borders of the array display

First we need to put the array variable colortable into the inspect window. In the window "current variables", select the variable "colortable" with the mouse and click on the symbol "I" at the upper edge of the window.

Now execute three single steps and observe how the contents of the inspect window change. Click on the "go one step" symbol in the RunShell window twice. The function "GetRGB32" loads the values of the screen view structure into the array "colortable". Note how simple it is to monitor variables with the inspect window!

You may want set further breakpoints and play with the debugger and its functions. You will become familiar with the controls very quickly.

To exit the debugger, terminate the program started in the debugger mode. All debug windows will be closed automatically as well as the RunShell.

1.17 StormC.guide / ST_Sektion

Sections of a project

The files in a project are categorized into sections automatically by filename extension and, in the case of documents, by entire name as well. If you add a ".c" file to a new project, the project manager creates a new section called "sources" containing this file. When adding further sources to the project, the

```
existing "sources" section is simply expanded.

The following sections are currently recognized:

Sources
        ".c"
        ".cc"
        ".ccp"
        ".c++"
        ".cpp"

Headers
        ".h"
        ".hh"
        ".hhp"
        ".h++"
        ".hpp"

ASM sources
        ".asm"
        ".ate"
        ".s"

ASM headers
        ".i"

Documentation
        ".dok"
        ".doc"
        " .txt"
        ".readme"
        "read me"
        "liesmich"
        "readme"
        "read me"
        "read.me"
        "lies.mich"

ARexx
        ".rexx"

Others
        "*"

Projects
        ".¶"

Amiga Guide
        ".guide"
Locale files
        ".ct"

Program
```

1.18 StormC.guide / ST_Owns

```
Peculiarities of StormC

Despite the standard ANSI C specifications each compiler has its own
peculiarities. These specialities are introduced with "#pragma". As with
#include, #pragma lines are interpreted and executed by the preprocessor.

Modes of the compiler

#pragma -

is a non-standard feature that shifts the compiler to ANSI-C mode.

#pragma +

causes the compiler to translate the source in C++ mode.

Chip and Fast RAM

The architecture of the Amiga is a bit unorthodox in some respects; for instance
there are different classes of RAM.

Normally the programmer is only interested in the answer to "Chip or Fast RAM?",
because eg. graphical data needs to be allocated in Chip memory. StormC
therefore offers the pragmas "chip" and "fast" .

All static data declared after the line

#pragma chip

are loaded into Chip RAM.

#pragma fast

switches into normal mode, in which your data is placed in whatever memory is
available (preferably Fast RAM).

OS calls

The Amiga OS (operating system) functions are called with #pragma amicall.

Such a declaration basically consists of four parts:
        - name of the library-base variable.
        - function offset as a positive integer.
        - name of the function (must have already been declared); to avoid
          ambiguities these names may not be overloaded.
        - list of parameters (represented by a corresponding list of register
          names between parentheses)


An example:

#pragma amicall(Sysbase, 0x11a, AddTask (a1,a2,a3))
#pragma amicall(Sysbase, 0x120, RemTask (a1))
#pragma amicall (Sysbase, 0x126, FindTask (a1))
```

Normally you will never write such declarations yourself since everything is included with the Amiga libraries.

Joining lines

Line breaks are of little consequence in C and C++, but they are significant to the preprocessor as each instruction must fit in exactly one line. Sooner or later you may come to the point, e.g. in an extensive macro definition, where you need to write a very long line. For this case there is the backslash. Any line ending in "\" is joined with the following line; both the backslash and the line break are completely ignored. Example:

```
# define SOMETHING \
47081115
```

This is a valid macro definition, for "47081115" is pulled into the preceding line.

Predefined symbols

The preprocessor knows many predefined macros. Some are defined by the ANSI C standard, others are part of C++ or particular peculiarities of StormC. These macros can not be redefined.

__COMPMODE__

is defined in StormC as the integer constant "0" in C mode and as "1" in C++ mode.

__cplusplus

In StormC the macro "__STDC__" is defined in C as well as in C++ mode. If you want to check whether your source is being compiled in C++ mode, this must be done with the macro "__cplusplus" .

 __DATE__

The macro __DATE__ is expanded to the date of the compilation. This is very useful if you want to furnish a program with a unique version number:

```
#include <stream.h>
void main ()
{  cout  « "version 1.1 from " __DATE__",  "__TIME__" clock\n;  }
```

The date is delivered in the form month - day - year , e.g. "Feb 08 1996"; the time is in the standard "hours:minutes:seconds" format.

__FILE__

This macro contains the name of the current source file as a string variable, e.g.:

```
# include <stream.h>
void main ()
{  cout «  "This is line " « __LINE__ « " in the file " __FILE__ ".\n";  }
```

The value of the __FILE__ macro is a constant character string and can be joined with leading or following strings.

__LINE__

The macro __LINE__ delivers the line number in which it is used as a decimal "int" constant.

__STDC__

This macro delivers the numerical value 1 if the compiler is compatible to the ANSI C standard. Otherwise it is not defined.

__STORM__

This macro gives you the name of the compiler and the version number.

__TIME__

(see __DATE__)

1.19 StormC.guide / ST_Referenz

Menu commands

Project

New A-N
When selected from the icon or project window, a new project is opened; this corresponds to a click on the icon "new project".

If the active window is an editor window, a new editor window is opened; this is the same as clicking on the "new text" icon.

Open... A-O
If the icon or project window is active a project will be opened. When selected from the editor window a text file is loaded. In either case, a standard ASL file requester will pop up, asking for an input file. You can choose Open... from the icon bar as well.

Save A-S
If the project window is active, the project is saved. This corresponds to a click on the icon "Save Project".

If an editor window is active, its text is saved. This corresponds to a click on the icon "Save Text".

Save as...
The Save file requester is opened; here you can select a file name for your project or your text.

Depending on whether the project window or an editor window is active, you can save either the project or the text. The icon bar offers two icons for saving sources and projects respectively.

Save as project pattern
This menu item is only accessible from the project window. The project pattern is a file containing preset project preferences which is loaded whenever you set up a new project. You can set the default options for future projects here.

This includes all options of a project (C/C++ environment, C/C++ pre-processor, C/C++ options, C/C++ warnings, linker options and program start) and of course all sections of the Project Manager.

Save all
With this menu item all source and projects which have not been written do disk yet will be saved. If no file names have been selected for some sources or projects the Save file requester is opened for each missing filename.

Add files...
This menu item is only accessible from a project window. A file can be selected from the file requester, which is then imported into the Project Manager. Depending on filename extensions, different sections are created and the file is placed at the corresponding position in the project.

Add window
This menu item is accessible when there is a project and the editor window is active. With a click the file of the active editor window is placed in the corresponding project section. In contrast to "Add files" the file requester does not appear.

Choose program name...
Of course each program requires a name. As a program may consist of many modules, this can not be done automatically. With the help of the file requester you can enter the program name and choose an appropriate location on your disk.

Close A-K
Depending on whether a project or editor window is active, either the project or the text window is closed. If the project or the text has not been stored, a safety request appears and offers you the possibility to do this now.

The same will happen if you click on the window’s close gadget.

About
This will show a requester in which you will find information about the product, copyright, our current telephone and fax number, and our email address.

Quit A-Q
Exit StormC. If any sources or projects have not been stored yet, you will see a message reminding you of this.

Edit

Mark A-B
This menu item is only available from editor windows. It switches the marking mode of the editor.

Cut A-X
This menu item is only available from editor windows. The marked text area is copied to the clipboard and deleted from the source text.

Copy A-C

This menu item is only available from editor windows. Like "Cut", this copies text into the clipboard; the difference is that with "Copy" the block is not deleted from your source text.

Paste A-V
This menu item is only available from editor windows. It inserts the contents of the clipboard into your text at the current cursor position.

Delete
When selected from an editor window, the marked area is erased from your text. The erased text is not copied into the clipboard, but if necessary you can still restore your text to its previous state using the "Undo" option.

When selected from a project window, the module marked in a section is removed from the project. "Undo" is not possible here!

Undo A-Z
This menu item is only available from editor windows. "Undo" reverts the most recently invoked editor function.

Redo A-R
This menu item is only available from editor windows. With "Redo" you can take back the most recent "Undo".

Find & Replace... A-F
Find & replace can of course only be selected from the active editor window. In the dialogue box that appears after selecting this option, you may enter the search key and any text you wish to replace its occurrences in the text with.

This function can be used for searching as well. Just omit the replacement string and click on the "Find" gadget. The search direction can also be selected.

Using the cycle gadget you can set the options in more detail, in particular you may choose to ignore letter case and accents.

With the three gadgets at the lower edge of the dialogue you may execute the search commands in different ways.

"Find" simply searches for the ‘find’ string
"Replace" replaces the ‘find’ string with the ‘replace’ string once
"Replace all" replaces all occurences of the ‘find’ string found with the ‘replace’ string

Find next A - " . "
This menu can only be selected from the editor window.

"Find next" repeats the most recent find command without opening the dialogue box again.

Replace next A - "-"
This menu can only be selected from the editor window.

"Replace next" repeats the last-made replace command without opening the dialogue box first.

Compile

Compile...
The menu item "compile. .. " is available if an entry is marked in the sources section of the project. It may be also be selected if the source indicated in the active editor window is found in the active project.

With this function you may compile individual modules.

Make... A-M
This menu item is available only from the project window or the error window. It may also be selected if the source indicated in the active editor window is found in the active project.

"Make" compiles all modules that have been altered since the last compilation, taking dependencies between program and header files into account.

Clicking on the "Make" gadget has the same effect.

Run...
This menu item is available from the project window or the error window only. It may also be selected if the source indicated in the active editor window is found in the active project.

If an up-to-date version of the program already exists, it will be executed. Otherwise a "Make" will be performed first, so that all changed modules are compiled. Afterwards the new program is executed.

Clicking on the "Run" gadget has the same effect.

Debug... A-D
This menu item is available only from the project window or the error window. It may also be selected if the source indicated in the active editor window is found in the active project.

"Debug..." does the same as "Run" plus starting the debugger. The program counter is set to the beginning of the first function (main) and execution of the program is halted at this position.

Clicking on the "Debug" gadget has the same effect.

Touch
This menu item is available from the project window only.

If you "Touch" an entry in the source section of the Project Manager it will be marked as "changed". Next time a "Make" is performed, this source is certain to be recompiled.

Touch all
This menu item is available from the project window only.

When selecting this item, all modules of the source section are marked "changed". Next time a "Make" is performed, all sources will be recompiled.

Save program as...
This menu item is only available from the project and error windows.

After succesfully linking a program, it is saved automatically. If you did not

give a name to the program it will be saved under the default name "a.out" in the project directory. With "Save program as..." you may store a copy of the created program with another name at another place on your hard disk.

Windows

Error window...
This menu item is available from the project window only.

It opens the error window.

Modules...
This menu item is only available if the debugger is active. It opens the windows of the modules for which information can be printed. This normally includes all entries of the "sources" category in the project window for which a debug file has been generated. Instead of the source names you will find the file names of the object modules.

Current variables...
This menu item is only available if the debugger is active. It opens the window showing the current variables.

Monitored variables...
This menu item is only available if the debugger is active. It opens the window showing the monitored variables.

Options

Project...
These options are only available when the corresponding project or error window is active.

Include paths and pre-compiled headers

The preprocessor is a software module which processes the source files before the compiler gets to look at them. The preprocessor fetches definitions for e.g. the standard library functions from include files. Use the preprocessor instruction "#include" for this. As you certainly know, there are two possibilities to do this:

If one includes the file names after the "#include" "like this" (ie. between double quotes), the preprocessor looks for them in the current working directory. If you enclose them between angled brackets <like this>, they are assumed to indicate names of standard include files and they are loaded from the appropriate directories. In the "Include Path" Listview you can choose one or more directories to search for the standard include files.

"Copy to:" gives you the option to speed up compilation by caching include files on the RAM DISK; but of course this will use up some more memory.

Another way to speed up compilation dramatically and even to save RAM is to use pre-compiled header files. To use this feature, tell the compiler where the header file ends and your program starts with "#pragma header". A simple way is to put the "#includes" of all header files that are not yours or are never changed (e.g. OS includes) right at the top of each module. Insert a line "#pragma header" directly below that. Below this point you may add your own header files and the rest of your program.

This pragma instruction has no effect unless you select the "Write header file" option and recompile the changed headers. As soon as the compiler reaches the pragma instruction the pre-compiled header files are written into the corresponding directory under the indicated name.

Before you start the compiler the next time you simply switch to "read header files". The compiler reads the pre-compiled header file, searches the instruction "#pragma header" and starts its translation.

You will reach the next option dialogue by cycling through the cycle gadget just below the upper edge of the window.

Preprocessor

Here you can select what warnings the preprocessor should generate, and predefine any preprocessor symbols.

The preprocessor warnings are easy to configure, but the definitions need some explanation. Each entry you make in this Listview will have the same effect as writing it at the top of each source file with a "#define" in front of it. This lets you make global definitions such as "DEBUG" or define tokens such as "TRUE" and "FALSE".

Generating code and debugging

Here you may select the source translation mode (ANSI C or C++) and, in the case of C++, the processing of templates and the use of exception handling.

The next cycle gadget switches debug output generation on or off. If you want debug output the compiler generates additional files with the suffix ".debug" and ".link". These files are required by the debugger; they describe the relation between sources and program code.

If you want to work with a symbolic debugger/disassembler you have the option to add a symbol hunk to the program.

You can even produce assembler sources. The compiler creates additional ".o" and ".s" files. They contain assembler source interspersed with the corresponding C statements in your program.

If you enable "interrupt code generation" the compiler inserts a check for <CTRL> + <C> in every loop.

When creating code for the 68000, you should enable "32 bit multiplication"; this will cause library calls to be used for long word division and multiplication. This switch is ignored when generating code for higher processors.

"Optimise code" makes the program more compact and usually faster.

The next cycle gadget selects the processor type. Please keep downward compatibility in mind: if you generate code for the 68060 the resulting program will not run on a normal Amiga 2000.

Next you can choose between generating FPU code or calling the system libraries for mathematical calculations.

The last cycle gadget toggles between large and small data model.

Warnings

StormC distinguishes eight warnings which you can enable or disable according to your individual needs.

Linker settings

The path for link libraries may be set here; this is similar to the preprocessor's include path.

The next cycle gadget has three options:

"Link program" links the compiled program with the libraries.
"Do not link" does not start the linker.
"Link without startup code" is used for shared libraries or device drivers. Such programs can not be run from CLI or Workbench.

You may also select whether linking should be done if an error occurs during compilation.

Running

If you run a program from StormC's integrated environment you may specify command-line arguments and set the program's stack size.

Load settings...
You can load a debugger settings file with the suffix "RUN". This item is only available if the debugger is active.

Save settings
You can store the complete debugger configuration (which windows are opened and their respective positions). The "StormSettings.Run" file is saved to the home StormC directory. This item is only available if the debugger is active.

Save settings as...
Stores the complete debugger configuration as above, but you are given the opportunity to choose a different name and location. This item is only available if the debugger is active.

1.20 StormC.guide/STC_Sort

Own "Makescripts" with the StormC-Projectmanagement

The rules behind a "Make" are essentially very simple. First of all, all files in the project are checked to see if they need to be recompiled.

In the case of a C source file this means that the file dates of its object and debug files are compared to that of the source text and of any header files that it may #include. If any of these is newer than either the object or debug file, the source file needs to be recompiled.

The source file also needs to recompiled if one of the header files has been changed by some other action by the compiler. This may be the case for instance when the "catcomp" program is used to generate a header file from a Locale file.

Once it has been determined which files are to be recompiled or re-linked, each of them is handled by sending the corresponding ARexx commands to the StormC compiler and the StormLink linker. These commands are then executed in turn.

Makescripts are used when other files than just C and assembler sources need to be translated:

The "Select translation script..." menu option lets you enter an ARexx script for the active project or - if a section title has been selected - for all files in a section. These scripts make it possible to invoke external compilers such as eg. "catcomp" to compile Locale files automatically.

They are called by the project manager whenever the project is to be recompiled. Makescripts should have filenames ending in ".srx". Files with this extension to their names are also included in the ARexx section.

Selecting the "Remove translation script" menu option will remove the makescript from a project entry or from all entries in a project section.

The rules for determining whether a project file that has a makescript attached should be recompiled, are essentially the same as they are for C source files.

A file is always recompiled during the first "Make" after a makescript has been added to it.

As an example of what a makescript looks like, the "catcomp.srx" script is explained below:

```
/*

The script's arguments are the file name (ie. the path to the project entry) and
the base project path.  Both are enclosed in quotes to allow the use of spaces.

The argument list must be terminated by a full stop, so that any additional
arguments that may be passed by future versions of the compiler will be skipped.

*/

PARSE ARG '"' filename '"' '"' projectname '"' .

/*

The object filename is constructed from the filename argument.  This isn't
necessarily a file that is going to be linked and whose filename ends in ".o",
but simply the file that is to be created.  Catcomp happens to create a header
file.

*/

objectname = LEFT(filename,LASTPOS('.cd',filename)-1)||".h"

/*
```

```
All output is sent to a console window.

*/

SAY ""
SAY "Catcomp Script c1996 HAAGE & PARTNER GmbH"
SAY "Compile "||filename||" to header "||objectname||"."

/*

In order to allow the Project Manager to determine when the file should be
recompiled, the object filename must be coupled to the project entry.  If this
statement were to be omitted, the makescript would be called for every "Make".

A maximum of two object filenames may be given as follows:

OBJECTS filename objectname1 objectname2

These names are then attached to the entry and the files are checked when
recompiling.

See also the script "StormC:rexx/phxass.srx".

The OBJECTS statement should not be used if the makescript is used for calling
an assembler in the section "Asm Sources".  For this section the object names
are derived automatically.

*/

OBJECTS filename objectname

/*

This is where the translating program is called.  Error messages are printed in
the console window.

*/

ADDRESS COMMAND "catcomp "||filename||" CFILE "||objectname

/*

As "catcomp" creates a header file, it is advisable to enter this header file
into the project.  The QUIET parameter represses any error messages in case the
header file should already be included in the project.

*/

ADDFILE objectname QUIET

/* End of makescript */

Almost any makescript can be built along these lines.  Another statement may be
useful in some cases:

DEPENDENCIES filename file1 file2 file3 ...
```

This statement connects the project entry to further files whose dates will be checked to see whether or not the makescript should be called. The file named in the project entry itself will always be checked and need not be specified using this statement. Using this statement makes sense in cases where the script involves any extraneous files (the StormC compiler for instance uses it to declare any header files that a source file includes with #include "abc.h"; note that this is not done for headers included with #include <abc.h>).

Makescript settings are ignored for the project section that contains C sources; these files are always run through the StormC compiler. The section containing assembler source files on the other hand allows the use of makescripts - although it will use the built-in default rule for StormASM (which in turn invokes the PhxAss assembler) if no makescript is set.

Passing arguments to makescripts

The script receives the filename (that is, the path to the project entry) and the project path as arguments. Both paths are enclosed in quotes to allow the use of whitespace in file or directory names.

Next comes a numeric argument whose value indicates whether the object files should all be written into a single directory.

0 means that the object file should be stored in the same directory as the source file;

1 means that the object file is to be stored in the object-file directory.

The name of the object-file directory - quoted like the other paths - is passed as the next argument (regardless of the value of the previous argument, ie. even when the preceding numeric argument is 0).

The object-file directory is only interesting to programs that generate code. Source-generating makescripts (eg. "catcomp.srx") will always write their object files to the same directory that the file in the project entry resides in. Thus only assemblers and other compilers really need to care about the object-file directory.

Makescripts for assembly source files are an exception in that they take an additional third argument: The name of the object file. This name must be used when creating the assembler object file. The path to the object-file directory is already included in this name, if necessary.

The argument list must be terminated by a full stop so that any additional arguments that may be passed by future versions of the compiler will be skipped.

A complete PARSE statement for makescripts (other than one for assembler sources, as explained above) is composed as follows:

```
PARSE ARG '"' filename '"' '"' projectname '"' useobjectdir '"' objectdir '"' .
```

For an assembler makescript this would be:

```
PARSE ARG '"' filename '"' '"' projectname '"' '"' objectname  '"' useobjectdir
'"' objectdir '"' .
```

Ready-made makescripts

The directory "StormC:Rexx" contains several ready-to-use makescripts. You may want to adapt them to different uses and situations:

Assembler scripts

Makescripts for assemblers differ from other makescripts in that they may not contain the OBJECTS statement.

"phxass.srx"

This script translates an assembler file using the PhxAss assembler. This script is really superfluous because the assembler is supported by the StormShell directly, but may be useful if you want to use different assembler options.

"oma.srx"

This script translates an assembler source file using the OMA assembler.

"masm.srx"

This script translates an assembler source file using the MASM assembler.

Other scripts

"catcomp.srx"

This script translates a Locale catalogue file by invoking the program catcomp.

"librarian.srx"

The StormLibrarian can also be controlled through makescripts. A project entry in the "Librarian" section can be loaded directly into StormLibrarian by double-clicking it with the mouse, or the linker library can be created simply by double-clicking it while keeping the Alt key pressed. But if a project should always create a link library, the use of makescripts is recommended. The list of object files is created in StormLibrarian as usual. The makescript then invokes the StormLibrarian, which not only automatically generates the library, but also declares the linker library as an object (using OBJECTS) and all object files in the list as dependant files (using DEPENDENCIES). After the first Make this will cause the linker library to be created anew whenever any of its C or assembler source files has been recompiled.

The library will also be recreated if its list of object files has been modified using the StormLibrarian.

"fd2pragma.srx"

This makescript translates an FD file into a header file containing the necessary "#pragma amicall" directives for a shared library. This script shouldn’t normally be necessary as the linker writes this header file automatically whenever a shared library is linked.

1.21 StormC.guide/STC_Sections

THE PROFILER

A profiler is an indispensable tool when optimizing a program. Compiler optimizations can only improve program performance by so much, whereas a profiler can provide the necessary information for identifying the most time-intensive functions in a program. These functions can then be rewritten to use better algorithms if possible, or at least sped up by carefully optimizing the source code by hand.

The StormC profiler is especially powerful; it allows precise timing and provides many valuable statistics about the program.

As always, we have stuck with the our maxim in that no special version of the program needs to be generated for using the profiler. Having the normal debug information generated will suffice. This - like the ability to start the profiler while debugging - is probably unique for compilers on the Amiga.

If you wish to use the profiler, make sure your project is compiled with the Debug option set to either "small debug files" or "fat debug files". Select the "Use profiler" option in the Start Program window. The program can then be started as normal. Simultaneous debugging is possible, but may lead to minor deviations in the profiler's timing measurements.

After starting the program, the profiler window can be opened.

The upper-left corner updates the profiler display, changing all indicated percentage and timing values to reflect the latest results.

The help line shows the cumulated CPU time. This value is the amount of real CPU time used, ie. it does not include time that the program spends waiting (for signals, messages, or I/O) or time used by other programs that are running in the background.

The list below shows the functions including the following information:

1. Function name.

Member functions of a class are displayed using the "scope operator" syntax (class name and member name separated by two colons).

2. Relative running time.

This counts only the time that the program spends in the function itself, or in OS functions called directly from it. Any time this function spends calling other functions in the program is omitted.

This value provides the best hint as to which function uses up the most time. The sum of all values in this column will be 99 - 100% (the missing percent is lost in startup code and minute inaccuracies).

3. Relative recursive running time.

Here all subroutine calls from a function are included in its running time. For this reason the main() function will normally show a value of 99%.

4. Absolute running time.
5. Longest running time.
6. Shortest running time.

These three lines give you a quick overview over the invocations of each function. Just how a function can be made faster often depends on whether the invocations generally take roughly the same amount of time to finish (small difference between longest and shortest running time), or some invocations take noticeably longer to complete than the others (great difference). In the latter case it may be profitable to optimize those special cases.

7. Number of invocations.

Sometimes a function makes up a large chunk of the program's running time only because it is called very often, but each individual invocation takes up very little time. Optimizing such a function is usually a tough nut to crack. However it may be very beneficial in such a case to declare the function inline ("__inline" in C, "inline" in C++).

Above this list are several controls related to the profiler display:

The uppermost line is the help line which shows brief descriptions of the controls.

Directly below that, to the left, you see three buttons. The first of these updates the list of functions.

The second button lets you save the profiler display as an ASCII text file. A requester will appear to let you select a file name.

The third button dumps the information to the printer (using the "PRT:" device).

On the right-hand side of this line are the sorting controls for the function list. The first entry "Relative" sorts the list by the values in the second column, "Recursive" sorts by the third column, and "Alphabetic" sorts alphanumerically by the function name shown in the first column. And finally "Calls" sorts the list by the contents of the last column.

The line below this holds a text entry field for a DOS pattern string. Only those functions are shown whose names match the pattern; this can help reduce excessively long function lists to a manageable size. This can be used for instance to only show member functions of a particular class by entering the class name followed by "#?".

To the right of this sits a numeric entry field where you may enter the minimum percentage of running time that a function must take up in order to be shown in the list. Functions that make up less than 5 or 10% are often difficult to optimize and even doubling the speed of such a function is hardly worthwhile as the program would not become noticeably faster (a mere 2.5 or 5% in this case).

These optional restrictions aside, only those functions are ever shown that are invoked at least once while the program is running.

Double-clicking on a function entry will take you directly to its location in the source text.

The profiler display is also opened and updated automatically when the program terminates. The control window will also remain open. Closing the control window will also cause the profiler display to close and the list is forgotten. Should you want to have access to this information afterwards, make sure you have saved it to file or printed a hardcopy before closing the window.

Profiler technical information

The LINE-$A instructions $A123 and $A124 are used to mark function calls.

These machine language instructions are unused on all members of the Motorola 68K processor family and trigger an exception. This exception is used to update the timing and call statistics.

The use of exceptions has the relative drawback of reducing the effective CPU speed, ie. the program will take longer to execute when the profiler is running than it does when the profiler is not activated. The difference will be particularly noticeable if the program invokes a lot of short functions. However the profiler will still be faster and more accurate than most existing profilers for the Amiga OS. The technique also buys the advantage of not having to recompile your code especially for profiling.

Handling of recursion is limited: The longest and shortest execution times will usually be unreliable, the total execution time (and therefore the relative values also) may be incorrect. A simple case of recursion (where f() calls f()) shows the correct relative values, but in the case of nested recursion (where eg. f() calls g() which calls f()) cumulates all times onto one of the two functions.

Function calls leading out of the recursion will still be shown correctly.

The effect on long jumps is not predictable, but in most cases this should only lead to minor distortions of the statistics for the called function.

Theoretically speaking, not all functions can be measured: Only functions whose machine code starts with a link or movem instruction are available to the profiler. One of these instructions will however be necessary in almost all cases, even at the highest optimization levels. And fortunately any functions that do not need these instructions will be so small (no variables on the stack, only the registers d0, d1, a0, and a1 are altered) that optimizing them any further would be near-impossible anyway.

Inline functions generally cannot be measured.

1.22 StormC.guide/STC_Project

PORTING FROM SAS/C TO STORMC

We have made it a point to equip the StormC compiler with many important properties of the SAS/C compiler, ie. support for various SAS-specific keywords and #pragmas. Nevertheless there may - depending on your programming style - be large or small amounts of trouble when porting software from the SAS/C compiler to the StormC compiler.

Please keep in mind that StormC is an ANSI-C and C++ compiler. SAS/C on the

other hand is a C and ANSI-C compiler (the C++ precompiler is not likely to have found much serious use), meaning that it understands a lot of older syntax that StormC will not accept. This is likely to cause trouble when migrating your sources to StormC, unless you are used to compiling your SAS/C programs strictly in ANSI mode (using SAS/C's ANSI option).

Project settings

First of all make sure that the project you build around your SAS/C sources is in ANSI-C compiler mode.

Try enabling as many warnings as possible, and then adapt your programs until no more warnings are given when compiling. This will give you the best chance that your program will do exactly what you intend it to.

Even for pure ANSI-C projects, switching to C++ later is recommendable. This will have several advantages: Prototypes are expected for all functions, and implicitly converting a void * to another pointer type is no longer legal.

Although this may necessitate a relatively tiresome rework of your programs (especially the latter change which affects a great deal of statements that call malloc() or AllocMem()), but can give you a great deal more confidence in the correctness of the program.

The long symbol names in C++ provide additional security while linking: If a function definition is in any way inconsistent with its prototype declaration, the linker will abort with reports of an undefined symbol.

Switching to C++ will also give you the possibility to extend your program with modern object-oriented concepts, as well as the use of several smaller C++ features (such as the ability to declare variables anywhere in a statement block).

Syntax

Some SAS/C keywords are not recognized by StormC, others are supported well, but the more "picky" StormC compiler only allows them in the typical ANSI-C syntax.

StormC does accept anonymous unions, but not implicit structs. Equivalent structures are not considered identical. If you have made use of this feature, you will need to insert casts in some places.

If this feature is important to you, you may want to consider moving your project over to C++: Equivalent structs are nothing but (an aspect of) inheritance in a different guise.

Type matching is much more strict in StormC. This is especially the case for the const qualifier used on function parameters. An example:

```
typedef int (*ftype)(const int *);

int f(int *);

ftype p = f; // Error
```

For such errors you should either insert the necessary casts, or (and this is

always preferable) write the appropriate declarations for your functions. After all the const qualifier is an important aid in assuring the correctness of your program.

Rest-of-line comments as in C++ ("//") are accepted even in ANSI-C mode, but nested C-style comments are not. In any case you can enable the warning option that detects these dangerous cases.

Accents in variable names are not accepted, nor is the dollar sign.

Keywords

The use of non-standard keywords is generally best avoided - at least for programs that you may want to port to another operating system or a completely different compiler someday.

StormC makes more use of the #pragma directives provided by ANSI-C for adapting software to the special requirements of AmigaOS (eg. #pragma chip and #pragma fast).

For keywords that may not exist in other compiler systems but are not absolutely necessary, the use of special macros is recommended:

```
#define INLINE __inline

#define REG(x) register __##x

#define CHIP __chip
```

These macros can then be easily modified to suit a different compiler environment.

Some optional keywords not recognized by StormC can also be defined as macros:

```
#define __asm

#define __stdarg
```

Here's a list of SAS/C keywords and how StormC interprets them:

__aligned

is not supported. There is no simple way to replace this keyword, but fortunately it is rarely needed.

__chip

This keyword forces a data item into the ChipMem hunk of the object file. Note that this keyword, like all other memory-class specifiers and other qualifiers must precede the type in the declaration:

```
__chip UWORD NormalImage[] = { 0x0000, .... }; // correct

UWORD __chip NormalImage[] = { 0x0000, .... }; // error
```

The latter syntax is not accepted as it is not consistent with ANSI-C syntax.

In StormC the use of "#pragma chip" and "#pragma fast" is preferred. Take notice however of the fact that "__chip" affects only a single declaration whereas "#pragma chip" remains in effect until a "#pragma fast" is found.

__far and __near

are not supported. There is no easy way to replace these keywords, but they are rarely needed.

__interrupt

This keyword is supported like within the SAS/C-Compiler.

__asm, __regargs, __stdarg

are not supported and not needed. If you wish to have function arguments passed in registers, declare the function with the ANSI keyword "register" or modify the individual parameter declarations with the "register" keyword or a precise register specification (eg. "register __a0"). Otherwise the arguments will be passed on the stack.

__saveds

Has an effect similar to SAS/C's "__saveds". This keyword has no effect when using the large data model; in the small data model relative to a4 it saves a4 on the stack and loads it with the symbol "__LinkerDB", in the small data model relative to a6 it does the same for a6.

Do not use "__saveds" lightly. It should be used exclusively for functions that will be called from outside your program, eg. Dispatcher functions of BOOPSI classes.

In the current compiler version it is recommended to use only the large data model when creating shared libraries. Remember that the small data model makes yet another register unavailable to the optimizer leaving only a0 to a3 - this can quickly nullify the advantage of using the small data model if you're not using a great deal of global variables.

__inline

Like the others, this keyword is accepted as a function specifier.

This means that their usage in a function definition must match the prototype.

If an "__inline" function is to be called from several modules, its definition (not just its prototype) should be placed in a header file.

__stackext

is not supported. Stack checking or automatic stack extension is not available at this time.

1.23 StormC.guide/STC_Project

Some Frequently asked Questions and Answers !
****
Q 1:
****

Why has StormC no Global Optimiser" ?

Answer:
The compiler of StormC offers already a good optimisations. Some of the used techniques are also part of a global optimiser. Especially the optimised organisation of the global CPU and FPU registers is such a one.

StormC is already prepared to do more of the global optimisations in further versions. They can be implemented step by step. But as said before the optimisation of StormC and StormLINK.

Q 2:
****

Why does a small "Hello World" gets so long (some Kbytes) ?

Answer:
"Storm.lib" is a highly compatible ANSI C library that offers buffered I/O.

The program "Hello World" uses "printf" of the "Storm.lib", but it does not differ between the output of integer or floating point, so there are unused parts of the library in your code that makes small programs relatively large.

If only ANSI C is used the library "StormAmiga.lib" can be used. This is a highly optimised assembler library that generates very small and fast programs. It only supports "far code" and "far data" model.

Q 3:
****

How can I get a short "Hello World" using Storm.Lib ?

Answer:
If floating point is needed and buffered I/O of AmigaDOS is enough AmigaDOS functions can be used for this. "Vprintf" and "VFPrintf" will directly output to AmigaDOS files, just like "printf". But these functions are not 100% ANSI compatible.

Another possibility is to resign on automatically open and close of libraries and its comfortable error handling which differs between Workbench and CLI start while paying attention to OS 1.3 and older. This comfort is not necessary for all programs.

So a minimal startup code, written in assembler can be used, that only does the essential jobs, e.g. only supporting the small data model, not supporting resident programs ...

Q 4:
****

Why is the library "storm.lib" such a big one and why is there
only a single one (in contrast to SAS/C) ?

Answer:
StormC support a further developed object format that is used for
linker libraries as well. It is 100% compatible to the old one. Its
advantage is that the Linker and the Compiler of StormC can accommodate
more data models in one object file. So the programmer must not decide
which is the right library that fits the used data model (far data,
near data(a4), near data(a6)). This caused many errors. Now the linker
takes the needed parts out of "Storm.lib". For these reasons "Storm.lib"
is nearly as big as these three libraries of SAS/C.

In the future StormC will support even more code modells and CPU and
FPU models, so "Storm.lib" will allow the optimised program generation
automatically.

Q 5:
****

Why does the linker displays the error message "Symbol _exit
not defined", when linking as Shared Library ?

Answer:
The Shared Library calls the ANSI function exit(). This can be done
directly by the programmer or indirectly by the linker library which
uses this function. "Storm.lib" uses this function to automatically
open the used Shared Library, e.g. the "utility.library".

Basically a Shared Library is not allowed to use exit(), cause it
can not be finished simply.

How to avoid this call ?

You should not use the automatic opening of used Shared Libraries.
Instead you must open and close the library as it is described in
the manual.

To get a list of all used libraries you should include

```
void exit() {}
```

into the Shared Library. Now you can link it.

You should use the linker option "Write Map File". The linker will
generate a file with the extension ".map". Now watch all INIT
functions which contain the basis name of the Shared Library, e.g.
INIT_1_UtilityBase.

Now open all these libraries with your own routines. Pay attention
to declare the corresponding basis variable (e.g. UtilityBase). Do not
forget to remove your own exit() function from the source.