StormC3.0

COLLABORATORS			
	TITLE : StormC3.0		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		July 19, 2024	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

StormC3.0

1.1 StormC.guide

StormC V3.0 Enhancements

Software and Documentation
(c) 1997 by HAAGE & PARTNER Computer GmbH

Schlossborner Weg 7
61479 Glashuetten
Germany

Phone: +49 6174/966100
Fax: +49 6174/966101
Email: storm-support@haage-partner.com
Web: www.haage-partner.com

Table of contents

1.2 StormC.guide/STC_Sort

StormShell

A small symbol will be displayed in the project window when a

```
make file is added to a file item.

The item "Edit Make script..." of the menu "Compile" loads
the makescript of the selected item into the text editor.

Arexx commands (containing file of the type #?.(rexx|srx)) and
Shell script (containing files of the type #?.(sh|script|scp))
will now run the script when the "Alt" key is pressed during
double click or <return> is processed on the item. Before
Multiview was used to display the files as it is does with all
the other types moreover.

When scripts are called manually they will receive the same
parameters as a Make script will get by Make while called
automatically: first the name of the project (between ''), the an
1 in the case Use-Single-Object-Directory is the (otherwise a 0)
and third the name of this Single-Object-Directory (between '').
```

1.3 StormC.guide/STC_Sort

```
StormEd
-------

Bookmarks:

Shift-Control-F1 to F10 sets a bookmark. A jump to these
bookmarks is done by Control-F1 to F10. The bookmarks are
stored in the Tooltypes, so they are available after loading
the text again.

Extensions of the dictionary:

Control-0 to Control-8 adds the word under the cursor to the
dictionary. The number of the dictionary corresponds to the
sequence in the settings editor.
Control-DEL will delete the word under the cursor from the
dictionary. These changes will be stored in the dictionaries
at the end of the program or at the next "use" in the editor
settings window.

Comments can now be added to every item of a dictionary. For
that an '|' must be placed behind the item, followed by the
comment. This comment will be displayed in the help line of
the editor window when an item is coloured by this command.
This comment can be used for example for the output of
prototypes of function names.

The cursor can now be placed direct behind a word on a space
character. Then the word left of the space character will
be used. This is very practical for the help function: After
entering a word for which a help is needed, you do not have
to move the cursor to the left, but you can now hit HELP
directly.
```

The Find function will now have set "Ignore Case" as default.

1.4 StormC.guide/STC_Sort

StormRun

The list of functions which are patched for resource-tracking
has been extended. Now it will be recognised automatically when
StdIO files were handed over to CreateNewProc or SystemTagList.
These files are normally freed by the newly generated processes
and therefor deleted from resource-tracking.

Now more functions were checked for locks when they generate
new locks or free existing ones.

The Disassembler is now much faster, better and more Motorola
compatible. Now it supports the complete command set of the 68K CPU
with all kinds of addressing. This makes it much easier to debug
handmade assembler routines and in the ROM.

When the program reaches an area which does not have a debug file, the
debug file "StormC:StormSYS/internal.debug" will be used to display
the register in the variable window. This debug file can be created
easily, e.g. by including all AmigaOS header files. Now the display
of the registers can be transformed to any required type. Without
this debug file registers can not be displayed outside a C program.

1.5 StormC.guide/STC_Sort

Own "Makescripts" with the StormC Project Management

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 e.g. "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, i.e. 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.

1.6 StormC.guide/STC_Sections

THE PROFILER

A profiler is an indispensable tool when optimising a program. Compiler optimisations 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 optimising 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, i.e. 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 optimise 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. Optimising 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 optimise 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 optimisation 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 optimising them any further would be near-impossible anyway.

Inline functions generally cannot be measured.

1.7 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 recognised 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 recognised 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, e.g. 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.