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 Erweiterungen

  Software und Dokumentation

  Postfach 80
  61188 Rosbach

  Tel: 06007/930050
  Fax: 06007/7543
  Email: storm-support@haage-partner.com
  Web: www.haage-partner.com

Inhaltsverzeichnis

Desweiteren wird mit der Version 3.0 von StormC der Profi-Texteditor
GoldED 4 zusätzlich zum StormED 3 geliefert. Sie können nun zwischen den beiden
Editoren auswählen. Die Anbindung von GoldED geschieht dabei genauso harmonisch
wie bereits von StormED gewohnt.

 ältere Kapitel zur Version 2.0:

```
Projektverwaltung       Eigene Makescripts

Der Profiler            Neue Kontrollmöglichkeiten mit dem Profiler

Portierung              Anmerkungen zur Portierung von SAS/C nach StormC
```

1.2 StormC.guide/STC_Neu

Alles Neue zusammengefaßt

Die Version 3.0 enthält eine neue StormShell. Diese ist darauf vorbereitet, ihre Entwicklung für das Betriebssystem p.OS und für den PowerPC Prozessor zu unterstützen. Wenn Sie das PPC Modul installiert haben, so erhalten Sie die Möglichkeit, für AmigaOS 68K und PPC Programme zu entwicklen; wenn Sie das p.OS Modul installiert haben, so erhalten Sie die Möglichkeit, Projekte für AmigaOS und für p.OS zu erstellen. Das Assemblermodul unterstützt die Entwicklung von 68K und PPC Assemblerprogrammen. Natürlich lassen sich auch die 3 Module gleichzeitig installieren und kombinieren.

Neu ist der Assistent für neue Projekte. Bislang wurde für ein neues Projekt nur ein Muster geladen, daß z.B. die gewöhnlich benutzten Bibliotheken enthält. Wenn Sie mit der Version 3.0 ein neues Projekt anlegen, öffnet sich ein neues Fenster in dem sie mit einigen wenigen Einstellungen alle wesentlichen Projektarten (Programme, Bibliotheken) für die verschiedenen Konfigurationen (AmigaOS, pOS) erzeugen können.

Die Bibliotheksstruktur hat sich mit Version 3.0 geändert. Die eigentlichen Storm Bibliotheken im Verzeichnis StormC:lib sind deutlich kürzer geworden, denn einiges ist aus diesen Bibliotheken entfernt worden und in speziellen Bibliotheken wiederzufinden. Diese Bibliotheken liegen im Verzeichnis "StormC:StormSYS/lib". Gleichzeitig wurde die automatische Bibliotheksauswahl weiter verfeinert um mehr Bibliotheksvarianten zu unterstützen und noch einfachere Programmierung der shared libraries zu gewährleisten.

Neue Sektionen in einem Projekt erlauben die Aufnahme weiterer Dateitypen: Dateien mit dem Suffix ".p", ".psm" und ".pasm" sind für den PowerPC Assembler (das PowerPC Assembler Modul wird dazu benötigt) reserviert und werden schon jetzt in eine entsprechende Sektion einsortiert. Dateien mit dem Suffix ".sh", ".scp" und ".script" werden in eine Sektion für Shell Skripte einsortiert.

Der Compiler StormC teilt der Projektverwaltung die durch einen Quelltext benutzten Dateien mit, dabei handelt es sich normalerweise um Headerdateien. Jetzt werden includierte Quelltexte (also Dateien mit den Endungen ".c", ".cpp" usw. in eine neue Sektion aufgenommen, in der diese Quelltexte zwar editiert werden können, aber beim Make nicht kompiliert werden.

In den Projekteinstellungen können die Includepfade nun auch in der Reihenfolge verändert werden, gleiches gilt für die Preprozessorsymbole in den Compilereinstellungen.

Der Compilereinsteller enthält eine neue Compileroption: "Debugger freundlich". Damit wird ein etwas geänderter Code erzeugt, der mehr Möglichkeiten beim Debuggen zuläßt, z.B. ein vollständigeres Stackfenster. Die Codeänderungen sind jedoch so harmlos (es werden keine speziellen Debugstatements einkompiliert), daß die Geschwindigkeitseinbußen bzw. Codevergrößerungen kaum meßbar sind.

Dem Debuggen kommt ebenso die erweitere Map Datei des Linkers zugute: Die große Map Datei zeigt nicht nur alle öffentlichen Symbole, sondern auch alle Benutzungen der Symbole. Hartnäckinge Linkprobleme können dadurch besser verfolgt werden. Gleichzeitig wurde die Schreibgeschwindigkeit der Map Datei drastisch erhöht, es werden jetzt auch sehr große Map Dateien (> 500 KB) in wenigen Sekunden erzeugt.

Desweiteren wurden viele Kleinigkeiten geändert und verbessert, die die Arbeit mit der Umgebung zusätzlich erleichtern: Beim hierarchischen Make werden geänderte Projekte (z.B. durch zusätzliche Abhängigkeiten) ohne Rückfrage automatisch gespeichert, die Sektionstitel erhalten immer die korrekte Summe der Dateigrößen.

1.3 StormC.guide/STC_Assistent

Der Projektassistent

Wenn ein neues Projekt erzeugt wird, öffnet zuerst der Projektassistent ein Fenster mit folgenden Möglichkeiten:

1. Projektverzeichnis

Geben Sie darin ein Verzeichnis an, in dem das neue Projekt liegen soll. Der Knopf rechts neben dem Stringfeld öffnet den Fontrequester zur Auswahl eines Verzeichnisses. Mit dem ASL Requester können Sie durch Eingabe eines neuen Verzeichnisnamens auch ein neues Verzeichnis erzeugen.

Verzeichnisse wie StormC: oder StormC:StormSYS sollten Sie vermeiden, der Projektassistent fragt in diesem Fall auch nochmal nach, ob das Projekt wirklich dort angelegt werden soll.

2. Projektname

Dieser Name dient nicht nur als Dateiname für das Projekt (das Suffix ".¶" brauchen Sie nicht anzugeben), sondern auch als Mustername für einige weitere Dateien, die je nach Projektart neu erzeugt werden.

3. Projekttyp

Der Assistent unterscheidet zwischen dem Vorgabeprojekt, ANSI C Programmen, C++ Programmen, Shared Libraries und Linker Bibliotheken. Das Vorgabeprojekt lädt das Musterprojekt ("StormC:StormSYS/template.¶") und nimmt keine weiteren Veränderungen daran vor (außer die Einstellungsänderungen für die gewünschte Konfiguration, siehe unten), die anderen Projektarten erzeugen automatisch einige neue Dateien und beeinflussen auch die Einstellungen generell.

C und C++ Programme erzeugen einen leeren Quelltext. Shared Libraries erzeugen leere Dateien für Quelltext, Prototypen, Headerdatei und FD Datei. Linker Bibliotheken erzeugen ein leeres Dokument für den StormLibrarian.

4. Konfigurationsauswahl

4 Möglichkeiten stehen in diesem Cyclegadget zur Wahl: AmigaOS mit M68K CPU, AmigaOS mit PPC CPU, AmigaOS mit MixedBinary und p.OS mit M68K CPU. Je nachdem, welche Storm Module Sie nicht installiert haben, können einige dieser Möglichkeiten fehlen.

Diese Auswahl setzt viele Einstellungen der Projektumgebung, des Compilers und des Linkers auf passende Werte um ein Projekt für die gewünschte CPU und das gewünschte Betriebssystem zu erstellen. Für p.OS muß das Assign pOS: auf das Root Verzeichnis der p.OS Installation zeigen (z.B. assign pOS: Work:, wenn das Verzeichnis pOS auf dem Volume Work: liegt).

Die Erzeugung eines MixedBinary führt sogar zum Anlegen zweier Projekte: ein 68K Projekt mit dem gewünschten Projekttyp und ein PPC Unterprojekt. Wie bei allen hierarchischen Projekten muß auch bei MixedBinaries zuerst das Unterprojekt kompiliert und dann die so erzeugten Objektdateien mit dem Menüpunkt "Projekt/Dateien Hinzufügen" in das 68K Projekt aufgenommen werden. Nachfolgende Hierarchische Makes arbeiten dann vollatuomatisch.

5. Debuggerversion

Mit diesem Schalter werden viele Einstellungen zur Unterscheidung einer Debuggerversion und einer entgültigen Version eines Projekts erzeugt. Dazu gehören unterschiedliche Unterverzeichnisse für die Objekte ("objects" oder "objects_debug"), Erzeugung der Debugdateien mit Debugger freundlichem Code, Erzeugung der Linker Map Datei usw.

Wenn Sie die Einstellungen gesetzt haben, drücken Sie nun den Knopf "OK". Es wird ein neues Projekt in dem ausgewählten Verzeichnis erzeugt und je nach Projektart werden passende Quelltexte und Headerdateien hinzugefügt. Bevor das Projekt gesichert wird, wird wie gewöhnlich vor dem Überschreiben eines schon existierenden Projekts gewarnt, ebenso vor Erzeugung der neuen Dateien.

Das Projekt steht dann fix und fertig bereit, mit dem Öffnen des Quelltextes kann dann sofort mit der Eingabe des Programmtextes begonnen werden.

1.4 StormC.guide/STC_Shell

StormShell

Projektfenster

In den Projektfenstern wird angezeigt, ob zu einem Dateieintrag ein Makescript hinzugefügt wurde. Dazu wird rechts neben der Textgrößenanzeige in einer neuen Spalte ein kleines Symbol angezeigt.

Projektsektionen

Neu ist die Sektion für PPC Assemblerprogramme. Diese Programme sollen die Endung .p haben (es werden auch noch die Endungen .psm und .pasm akzeptiert). Der PowerPC Assembler PowerASM ist als Assembler Modul für StormC separat erhältlich. Mit diesem Assembler Modul wird auch die Unterstützung des 68K Assemblers PhxAss stark verbessert.

Die Sektionen für ARexx Befehle (enthalten Dateien der Art #?.(rexx|srx)) und für Shell Skripte (enthalten Dateien der Art #?.(sh|script|scp)) führen das Skript aus, wenn man beim Doppelklick oder Return auf dem Dateieintrag zusätzlich die Alt- Taste drückt. Bislang wurde dort (wie bei allen anderen Dateitypen auch) Multiview zur Anzeige der Datei aufgerufen.

Diese Skripte erhalten beim manuellen Aufruf ähnliche Parameter wie die Makescripte beim automatischen Aufruf durch Make: zuerst den Namen des Projekts (in Hochkommata), als nächstes eine 1, falls Use-Single-Object-Directory gesetzt ist (ansosnsten eine 0) und als dritten Namen der Name dieses Single-Object- Directories (in Hochkommata).

Makeskripte

Der Menüpunkt "Edit Makescript..." aus dem Menü "Compile" lädt das Makescript des ausgewählten Dateieintrags in den Texteditor. Es kann dort verändert und wieder gesichert werden, das neue Makeskript wird beim nächsten Kompilieren benutzt.

Einstellungen

Die Voreinsteller "Project Environment" und "Compiler" erlauben, die Reihenfolge der Einträge in den Listen der Includepfade und der Preprozessorsymbole zu ändern.

Jeder selektierte Eintrag kann nach oben oder unten geschoben werden.

Preprozessorsymbole, die im Projekt und nicht in einem Quelltext definiert werden, sind meistens von der Reihenfolge unabhängig, hier dient die Sortierungsmöglichkeit eher der Übersichtlichkeit.

Die Reihenfolge der Includepfade kann jedoch sehr wichtig sein, da die Includeverzeichnisse streng in der Reihenfolge der Pfade durchsucht werden.

Wenn eine Includedatei mit ihrem Dateinamen und eventuellen Directoryangaben doppelt im Pfad vorkommt ist die Reihenfolge der Pfade entscheidend, welches Includedatei geladen wird.

Beachten Sie dabei, daß Includeanweisungen mit Hochkommata (#include "test.h") zuerst das Verzeichnis, in dem der Quelltext steht, durchsucht, bevor auch die Includepfade benutzt, beim Includeanweisungen mit spitzen Klammern (#include <test.h>) zuerst die Includepfade und zuletzt das Quelltextverzeichnis durchsucht werden.

Eine Headerdatei wird nur dann in das Projekt aufgenommen und als Abhängigkeit der Quelltextdatei definiert, wenn sie über das Verzeichnis des Quelltextes statt eines Includepfads gefunden wurde.

Die neue Option "Debugger freundlich" im Compiler Einsteller zwingt den Compiler zu einer Debugger freundlicheren Codegenerierung. Im Falle der 680x0 Codegenerierung betrifft das vor allen Dingen die Einrichtung eines Stackframes für Funktionen. Dadurch kann der Stack eines Programms besser diagnostiziert werden und das Stackfenster wird vollständiger angezeigt.

Bei der PPC Codegenerierung hat diese Option weitreichenden Einfluß, da vollständig optimierter Code in einem Disassembler kaum wiederzuerkennen ist. Auch im PPC Code wird dann für jede Funktion ein Stackframe errichtet, aber insbesondere der Scheduler, der die Reihenfolge des Codes beeinflußt, wird stark gebremst. (Beachten Sie jedoch, daß in der Version 3 das PPC Modul keinen Debugger enthält, dieser wird in der nächsten Version folgen).

1.5 StormC.guide/STC_Ed

StormEd

Bookmarks

Mit Shift-Control-F1 bis F10 wird ein Bookmark gesetzt. Mit Control-F1 bis F10 wird zum jeweiligen Bookmark gesprungen. Die Bookmarks werden (per Tooltype BOOKMARK0 bis BOOKMARK9) gespeichert, stehen also nach dem Laden eines Textes wieder zur Verfügung.

Lexikonerweiterungen

Mit Control-0 bis Control-8 wird das Wort unter dem Cursor dem Lexikon hinzugefügt. Dabei entspricht die Lexikonnummer der Reihenfolge im Settingseditor.

Ctrl-1: Preprozessor

Ctrl-2: C/C++ Symbole

Ctrl-3: C/C++ Bibliotheksfunktionen

Ctrl-4: AmigaOS Typen

Ctrl-5: AmigaOS Funktionen

Ctrl-6: Benutzer Lexikon 1

Ctrl-7: Benutzer Lexikon 2

Ctrl-8: Benutzer Lexikon 3

Mit Control-Del wird das Wort unter dem Cursor wieder aus seinem Lexikon entfernt. Die so veränderten Lexika werden am Ende des Programms oder beim nächsten "Use" im Editor- Einstellungsfenster gespeichert.

Die Lexika können jetzt auch Kommentare zu jedem Eintrag enthalten. Dazu muß hinter dem Eintrag ein senkrechter Strich 'l' folgen und dann der Kommentar. Dieser Kommentar wird in der Hilfezeile des Editortextfensters ausgegeben, sobald das Wort nach der Eingabe eines Textzeichens eingefärbt wird. Dieser Kommentar kann beispielsweise für die Ausgabe von Prototypen zu Funktionsnamen genutzt werden.

Lexika können im Texteditor verändert werden. Lassen Sie die ersten 4 Zeilen unverändert, dort werden unter anderem die Farbinformationen gespeichert. Ab der 5. Zeile stehen die Wörter des Lexikons.

Hilfefunktion

Soll das Wort unter dem Cursor ermittelt werden, so kann der Cursor jetzt auch unmittelbar hinter einem Wort auf einem Leerzeichen stehen, dann wird als Wort unter dem Cursor das links vom Cursor stehende Wort benutzt.

Diese Erweiterung ist besonders nützlich für die Hilfefunktion: Nach dem Eintippen eines Wortes, für das Hilfe benötigt wird, muß man nicht mehr einen Schritt mit dem Cursor nach links gehen, sondern kann direkt Help drücken.

Finden&Ersetzen

Im Findfenster ist jetzt standardmäßig "Ignore Case" gesetzt.

1.6 StormC.guide/STC_Run

StormRun

Resourcetracking

Die Liste der gepatchten Funktionen für das Ressource-Tracking wurde erweitert. So wird jetzt automatisch erkannt, wenn StdIO Dateien an CreateNewProc oder SystemTagList übergeben werden. Diese Dateien werden im Normalfall von den neu erzeugten Prozessen selbst freigegeben und daher beim Aufruf dieser Funktionen aus dem Ressource-Tracking entfernt.

Außerdem werden mehr Funktionen für Locks überprüft, die entweder neue Locks erzeugen oder existierende Locks freigeben.

Disassembler

Der Disassembler ist jetzt schneller, besser und Motorola-konformer. Es wird der komplette Befehlssatz aller 68K CPUs unterstützt, mit allen Adressierungsarten. Dadurch wird auch das Debuggen in handgeschriebenen Assemblerroutinen und im ROM viel einfacher.

Registeranzeige

Befindet sich das Programm in einem Bereich für den keine Debugdatei existiert, so wird für die Anzeige der Register im Variablenfenster die Debugdatei "StormC:StormSYS/internal.debug" benutzt. Diese Debugdatei kann man sich einfach selbst erzeugen und dazu zum Beispiel sämtliche AmigaOS Headerdateien includen - dadurch kann man die Registeranzeige in jeden benötigten Datentyp wandeln. Ohne diese Debugdatei können die Register außerhalb eines C Programms nicht angezeigt werden.

Datentypen, die in den Standardinclude-Dateien nicht vorkommen (zum Beispiel auch viele Pointer auf Datenstrukturen) kann man durch Definition selbst erzeugen:

#include <exec/execbase.h>

typedef ExecBase *h1;

Am Ende der Quelltextes sollte wenigsten eine leere Funktion definiert werden, damit der Quelltext nicht völlig leer ist und vom Compiler akzeptiert wird:

void foobar() { };

1.7 StormC.guide/STC_INITEXIT

INIT- und EXIT-Funktionen

Bis zur Version 2.0 wurden die Funktionen InitModules() und CleanupModules() durch den Linker erzeugt. Diese Funktionen rufen alle INIT bzw. EXIT Funktionen eines Programms auf. Nachteil diese Methode war, daß ein Abbruch des Programms aus einer INIT Funktion (z.B. mit exit()) alle EXIT Funktionen aufgerufen hat. Die EXIT Funktionen mußten deshalb spezielle Vorkehrungen treffen, auch dann korrekt zu arbeiten, wenn "ihre" INIT Funktion nicht abgearbeitet war.

Dieses Problem stellte sich z.B. durch das automatische Öffnen der shared libraries. Wenn eine wichtige shared library nicht geöffnet werden kann, wird das Programm mit exit() verlassen.

Die INIT bzw. EXIT Funktionen werden nach wie vor durch die Funktionen InitModules() bzw. CleanupModules() aufgerufen, doch sind diese Funktionen nun Teil der stormcstartup.lib (siehe auch Linker Bibliotheken. Der Linker stellt die INIT und EXIT Funktionsadressen dazu als Tabelle zur Verfügung.

Neu ist dabei die Zuordnung der INIT und EXIT Funktionen untereinander. Dabei werden eine INIT Funktion und eine EXIT Funktion als Paar aufgefasst, wenn die Priorität beider Funktionen gleich ist und der Namensteil übereinstimmt, z.B.:

INIT_5_Dateien und EXIT_5_Dateien bilden ein Paar,

INIT_5_Dateien und EXIT_4_Dateien bilden kein Paar, wegen unterschiedlicher Priorität,

INIT_5_Dateien und EXIT_5_Datei bilden kein Paar, wegen unterschiedlichem Namensteil.

Die Funktion CleanupModules() (die zum Beispiel auch durch exit() oder das Ende des Programms aufgerufen wird) ruft nur die EXIT Funktionen auf, deren INIT Funktion vorher vollständig abgearbeitet wurde.

Verläßt die Funktion INIT_5_Dateien das Programm mit exit(), so wird die Funktion EXIT_5_Dateien nicht aufgerufen, jedoch alle EXIT Funktionen, deren INIT Funktion abgearbeitet wurden, also typischerweise alle Funktionen mit kleinerer Priorität.

Natürlich muß nicht jede INIT Funktion eine passende EXIT Funktion haben oder umgekehrt. Dann wird als Partner eine Leerfunktion angenommen.

In shared libraries erhalten INIT und EXIT Funktionen den Basiszeiger der shared library als Argument. Dabei entspricht der Prototyp der INIT und EXIT Funktionen dem einer argumentenlosen öffentlichen shared library Funktion, also:

Für 68K AmigaOS shared libraries:

```
void INIT_0_test(register __a6 struct MySharedLibBase *Base) { }
```

Für PPC AmigaOS shared libraries und p.OS shared libraries:

```
void INIT_0_test(struct MySharedLibBase *Base) { }
```

Sollen shared library Funktionen auch aus der shared library heraus über die externe Schnittstelle (also der Sprungtabelle der shared library) statt direkt als C Funktionen aufgerufen werden, so empfiehlt sich folgende INIT Funktion:

```
struct Library *MyBase;

void INIT_0_MyBase(register __a6 struct Library *Base) { MyBase = Base; }
```

Mit dieser Funktion wird die Basisvariable initialisiert, die beim Aufruf der shared library Funktionen über die pragma Anweisungen benötigt wird.

1.8 StormC.guide/STC_LinkLibs

Linker Bibliotheken

Bis einschließlich Version 2.0 lagen im Verzeichnis "StormC:StormSYS/lib" nur die Mathematikbibliotheken. StormC wählt die für die Projekteinstellungen günstigste Bibliothek automatisch aus. In Version 3.0 unterscheidet die Projektverwaltung Mathebibliotheken für jede denkbare CPU/FPU Kombination: 68000, 68020/030 ohne FPU, 68020/030 mit FPU, 68040 mit FPU, 68060 mit FPU. Die Mathematikbibliotheken enthalten nicht nur typische mathematische Funktionen (sin, cos, exp etc.), sondern auch alle anderen Funktionen der ANSI und C++ Bibliotheken, die mit Fließkommazahlen umgehen. Dadurch wird z.B. auch die Ausgabe der Fließkommazahlen mit printf() der jeweiligen FPU angepaßt.

Das Prinzip der automatischen Bibliotheksauswahl wurde auf 2 weitere Bibliotheken ausgeweitet.

Die "Stormcsupport.lib" ist eine Bibliothek, die Funktionen enthält, die der Compiler zusätzlich benötigt, z.B. für fehlende Integeroperationen der CPU (32 Bit Multiplikation der 68000 CPU, 64 Bit Arithmetik auf allen CPUs). Diese Bibliothek ist völlig betriebssystemunabhängig und wird sowohl für ihre AmigaOS Projekte, wie auch für die p.OS Projekte verwendet.

Die "Stormcstartup.lib" enthält alles, was der Programmstart benötigt, z.B. das Argumentenparsing für die ANSI main Funktion, automatisches Öffnen und Schließen von Bibliotheken etc. Diese Bibliothek liegt in verschiedenen Varianten für AmigaOS und p.OS vor. Sie unterscheidet außerdem das benutzte Datenmodell, die benutzte CPU Variante und ob das Projekt als normales Programm oder shared library gelinkt werden soll.

Fügen Sie keine dieser Bibliotheken Ihrem Projekt manuell hinzu. Die Entwicklungsumgebung wählt vor dem Linken automatisch die günstigste aus.

Die eigentliche ANSI und C++ Bibliothek liegt ebenfalls in allen nötigen Varianten für die verschiedenen Prozessoren vor, wenn Sie alle Module installiert haben sind das:

storm.lib: CPU 68000 und höher

storm020.lib: CPU 68020 und höher

storm603.lib: CPU PPC 603e und 604e

stormMixedbin.lib: CPU 68020 und höher, sowie CPU 603e oder 604e

pStorm020.lib: CPU 68020 und höher für p.OS.

Diese Bibliotheken fassen jeweils alle Datenmodelle zu einer Bibliothek zusammen, der Linker wählt die nötigen Teile automatisch aus.

Bei der 680x0 Codegenerierung wird das große Datenmodell (far) mit 32 bit Adressierung und unbeschränkter Datengröße, das kleine Datenmodell (near) mit 16 bit Adressierung und damit einem Datenbereich von maximal 64 KB, sowie das kleine Datenmodell bzgl. Register a6 mit 16 bit Adressierung unterschieden.

Von der storm.lib und storm020.lib gibt es die drei Datenmodelle far, near und near a6 auch jeweils als eigene Bibliothek, damit kann die Größe der Bibliothek eingeschränkt werden, ein (geringfügig) schnelleres Linken und weniger Speicherverbrauch beim Linken sind die Vorteile. Allerdings muß dann beim Wechsel des Datenmodells auch die Bibliothek im Projekt ausgetauscht werden.

Das Prinzip der automatischen Bibliotheksauswahl wurde auch auf die storm.lib ausgeweitet. Wenn im Projekt die Bibliothek storm.lib aufgenommen wurde (und nicht storm020.lib oder storm603.lib usw.), dann wählt die Projektverwaltung statt der storm.lib die für CPU und Betriebssystem geeignete Bibliothek aus.

Wenn also als CPU eine 68020 in den Compilereinstellungen ausgewählt wurde, so wird die storm020.lib gebunden, soll ein Mixed Binary für 68K und PPC erzeugt werden, so wird automatisch die stormMixedbin.lib benutzt.

Weitere Bibliotheken

Die amiga.lib ist die Standardbibliothek des AmigaOS. Sie enthält Stub Funktionen für die AmigaOS Aufrufe (wenn z.B. aus Debugging Gründen auf die Benutzung der pragma amicall Anweisungen verzichtet werden soll) und weitere Hilfsfunktionen.

Die ppcamiga.lib wird für PPC Programme benötigt. Sie enthält Funktionen für den Aufruf der AmigaOS shared library Funktionen (siehe AmigaOS Callbacks).

Die debug.lib und ddebug.lib sind zwei kleine Bibliotheken, die in der Debugphase einer Programmentwicklung helfen können. Sie enthalten insbesondere die Funktion

extern "C" void kprintf(char *,...);

Diese Funktion entspricht einem printf (ohne Fließkommaausgabe) bzw. der Funktion RawDoFmt der exec.library, allerdings werden die Ausgaben auf die serielle Schnittstelle geleitet.

Die cybergl.lib wird für die Benutzung der cybergl.library benötigt. Zusätzlich zu dieser Bibliothek sind auch die nötigen Includes für StormC (Prototypendatei, pragma Datei) beigelegt.

1.9 StormC.guide/STC_SharedLibs

Shared Library Generierung

Die Stormcstartup Bibliotheken für shared libraries (z.B. "stormclibstartup000.lib") enthalten andere Algorithmen für das automatische Öffnen weiterer shared libraries als die Bibliotheken für Programme. Dadurch entfällt das Problem der fehlenden exit() Funktion in shared libraries und die Initialisierung der shared library wird abgebrochen, falls eine nötige andere shared library nicht geöffnet werden kann.

Wenn Sie selbst aus einer INIT Funktion die Initialisierung der shared library abbrechen wollen, so brauchen Sie nur die Funktion abortLibInit() aufzurufen. Diese Funktion bricht den Initialisierungsvorgang entsprechend exit() ab, es werden also die nötigen EXIT Funktionen noch aufgerufen (siehe auch INIT/EXIT Funktionen).

Die Funktion LibOpen() wird beim Öffnen einer shared library aufgerufen (wird die shared library neu geladen läuft zuvor die Initialisierung mit Aufruf der INIT Funktionen). Diese Funktion ruft nun eine zusätzliche Funktion auf:

struct Library *__LibOpen(register __a6 struct Library *);

Diese Funktion erhält den Basiszeiger der shared library und muß einen Basiszeiger zurückgeben, wenn das Öffnen erfolgreich war, sonst NULL. Die Startup Bibliotheken enthalten eine solche Funktion, die nur den übergebenen Basiszeiger als Ergebnis zurückgibt. Sie können aber diese Funktion durch eine neue ersetzen (Funktionen im Programmtext haben Vorrang vor Funktionen in Bibliotheken), z.B. um zu verhindern, daß die Funktion mehr als einmal aufgerufen wird:

struct Library *__LibOpen(register __a6 struct Library *Base) { if (Base->lib_OpenCnt > 0) return NULL; return Base; }

Die Funktion LibClose() wird beim Schließen der Library aufgerufen. Diese Funktion ruft jetzt die folgende Funktion auf:

void __LibClose(register __a6 struct Library *);

Diese Funktion ist in den Startup Bibliotheken als Leerfunktion definiert, kann aber durch eine neue Funktion überladen werden, die z.B. Allokationen freigibt, die für jedes Öffnen der shared library in __LibOpen gemacht wurden.

1.10 StormC.guide/STC_Shell2

```
Eigene "Makescripts" in der StormC-Projektverwaltung

Ein Make verläuft prinzipiell nach einigen einfachen Regeln. Zuerst werden
alle, im Projekt eingetragenen Dateien daraufhin überprüft, ob sie neu
kompiliert werden müssen.

Bei einem C Quelltext sieht das so aus, daß das Datum der Objektdatei und der
Debugdatei mit dem Datum des Quelltextes und aller Headerdateien, die von dem
Quelltext eingebunden werden, verglichen werden. Ist einer dieser Texte jünger
als die Objekt- oder die Debugdatei, muß der Quelltext neu kompiliert werden.

Der Quelltext muß ebenfalls neu kompiliert werden, wenn eine der Headerdateien
durch eine andere Compileranweisung neu erzeugt wird. Das ist z.B. der Fall,
wenn das Programm "catcomp" aus einer Lokaledatei eine Headerdatei erzeugt.

Stehen die neu zu kompilierenden Dateien fest, werden dann für alle Dateien,
die neu kompiliert werden müssen und für das Programm, das gelinkt werden muß
ARexxanweisungen erzeugt, die den Compiler StormC und den Linker StormLink
steuern. Diese Anweisungen werden nacheinander abgearbeitet.

Um auch andere Dateien als nur C Quelltexte und Assemblerquelltexte übersetzen
zu können, werden Makescripts benutzt:

Der Menüpunkt "Select Makescript..." erlaubt die Angabe eines ARexxskripts
für die ausgewählte Projektdatei oder - falls ein Sektionstitel ausgewählt ist -
```

für alle Dateien der Sektion. Diese Skripte erlauben den Einsatz externer Compiler wie z.B. "catcomp", um die Lokaledateien automatisch zu kompilieren. Sie werden automatisch aufgerufen, wenn die Projektdatei neu kompiliert werden muß. Diese Makescripts sollten das Suffix ".srx" besitzen. Dateien mit dieser Endung werden auch in die Sektion ARexx aufgenommen.

Mit dem Menüpunkt "Remove Makescript" wird das Makescript von einem Projekteintrag oder allen Projekteinträgen einer Sektion entfernt.

Die Regel, ob eine Projektdatei, für die ein Makescript gesetzt ist, neu kompiliert werden muß, ist prinzipiell mit der Regel der C Quelltexte identisch.

Beim ersten Make nach der Angabe eines Makescripts wird die Datei immer neu kompiliert.

Als Beispiel für ein Makescript wird hier das Makescript "catcomp.srx" kommentiert ↩
:

```
/*

Als Argumente werden der Dateiname (das ist der Pfad des Projekteintrags) und der
Pfad des Projekts übergeben. Die beiden Pfade sind in Hochkommata eingeschlossen, ↩
   um
auch Leerzeichen zu ermöglichen.

Am Ende der Argumentenliste muß unbedingt ein Punkt stehen, um zusätzliche ↩
   Argumente, die
in zukünftigen Versionen übergeben werden, zu überlesen.

*/
PARSE ARG '"' filename '"' '"' projectname '"' .

/*

Der Objektname muß vernünftig aus dem Dateinamen entwickelt werden. Objektname ist
dabei nicht nur eine Datei, die auf ".o" endet und gelinkt wird, sondern jeweils ↩
   die
Datei, die erzeugt wird. Catcomp erzeugt eine Headerdatei.

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

/*

Alle Ausgaben werden auf einem Konsolefenster ausgegeben.

*/
SAY ""
SAY "Catcomp Script ©1996 HAAGE & PARTNER GmbH"
SAY "Compile "||filename||" to header "||objectname||"."

/*

Damit die Notwendigkeit einer Neukompilierung ermittelt werden kann, muß der
Objektdateiname mit dem Projekteintrag verknüpft werden. Fehlt diese Anweisung,
wird das Makescript bei jedem Make aufgerufen.
```

```
Es können maximal zwei Objektdateinamen angegeben werden, also etwa so:

OBJECTS filename objectname1 objectname2

Diese werden beide der Datei zugeordnet und beim Neukompilieren getestet.

Der Befehl OBJECTS darf nicht verwendet werden, wenn das Makescript für den
Aufruf eines Assemblers der Sektion "Asm Quelltexte" benutzt wird. Diese Sektion
ermittelt den Objektnamen immer selbst. Siehe auch das Makescript
"StormC:rexx/phxass.srx"

*/
OBJECTS filename objectname

/*

Hier erfolgt der Aufruf des Übersetzerprogramms. Fehlermeldungen werden auf der
Konsole ausgegeben.

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

/*

Da catcomp eine Headerdatei erzeugt, ist es sinnvoll, diese Headerdatei im Projekt
anzumelden. Der Parameter QUIET unterdrückt dabei die Ausgabe einer Fehlermeldung,
falls die Headerdatei schon Teil des Projekts ist.

*/
ADDFILE objectname QUIET

/* Ende des Makescript */

Fast jedes Makescript kann diesem Schema folgen. Unter Umständen ist noch eine
Anweisung praktisch:

DEPENDENCIES filename file1 file2 file3 ...

Mit dieser Anweisung verknüpft man den Projekteintrag mit weiteren Dateien, deren
Datum daraufhin geprüft wird, ob das Makescript aufgerufen werden muß oder nicht.  ↩
   Der
eigentliche Projekteintrag wird dabei immer getestet und muß nicht extra mit  ↩
   diesem
Befehl gesetzt werden. Aber wenn das Skript auch noch andere Dateien mit  ↩
   einbezieht,
ist diese Anweisung sinnvoll (der StormC Compiler setzt zum Beipiel alle
Headerdateien, die mit einer Anweisung #include "abc.h" - aber nicht
#include <abc.h> - geladen werden).

Die Sektion für C Quelltexte beachtet gesetzte Makescripts nicht. C Quelltexte  ↩
   werden immer
mit dem StormC Compiler compiliert. Die Sektion für Assemblerquelltexte hingegen  ↩
   erlaubt
die Benutzung der Makescripts - obwohl hier die eingebaute Regel für das Programm  ↩
   StormASM
```

```
benutzt wird (welches seinerseits den Assembler PhxAss aufruft), wenn kein  ↩
    Makescript gesetzt
ist.

Argumentenübergabe an Makescripts

Als Argumente werden der Dateiname (das ist der Pfad des Projekteintrags) und der
Pfad des Projekts übergeben. Die beiden Pfade sind in Hochkommata eingeschlossen,  ↩
    um
auch Leerzeichen zu ermöglichen.

Dann folgt ein Zahlargument, dessen Wert angibt, ob die Objektdateien alle in ein
Verzeichnis geschrieben werden sollen. 0 bedeutet, die Objektdatei gehört in das  ↩
    gleiche
Verzeichnis, wie die Quelltextdatei; 1 bedeutet, die Objektdatei gehört in das  ↩
    Objektverzeichnis.
Der Name des Objektverzeichnisses wird - ebenfalls in Hochkommata eingeschlossen - ↩
     als nächstes
Argument übergeben (der Name wird immer übergeben, also auch, wenn die vorherige  ↩
    Zahl 0 ist).

Das Objektverzeichnis ist nur interessant für Programme, die Code erzeugen.  ↩
    Makescripts,
die Quelltext erzeugen (z.B. "catcomp.srx"), schreiben ihr Objekt immer in das  ↩
    Verzeichnis,
in dem auch der Projekteintrag steht. Das Objektverzeichnis ist also nur für  ↩
    Assembler und
andere Compiler interessant.

Makescripts für Assembler Quelltexte erhalten als einzige ein zusätzliches  ↩
    Argument an
dritter Stelle: der Name der Objektdatei. Dieser Name sollte unbedingt benutzt  ↩
    werden, um
die Assemblerobjektdatei zu erzeugen. Im Pfad dieser Objektdatei ist dabei das  ↩
    mögliche
Objektverzeichnis schon berücksichtigt.

Am Ende der Argumentenliste muß unbedingt ein Punkt stehen, um zusätzliche  ↩
    Argumente, die
in zukünftigen Versionen übergeben werden, zu überlesen.

Eine vollständige PARSE Anweisung für Makeskripts sieht also so aus (kein  ↩
    Assemblerskript):

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

Und die Anweisung für Assemblermakescripts:

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


Die fertigen Makescripts

Im Verzeichnis "StormC:rexx" liegen einige fertige Makescripts. Diese können auch
an die jeweilige Situation angepaßt werden:
```

Die Assemblerskripte:

Makescripts für Assembler sind insofern eine Besonderheit, daß die Makescripts nicht
die Anweisung OBJECTS enthalten dürfen.

"phxass.srx"

Dieses Skript übersetzt eine Assemblerdatei mit dem Assembler PhxAss. Eigentlich ein
unnötiges Skript, da dieser Assembler direkt von der StormShell unterstützt wird,
aber vielleicht sollen ja spezielle Assembleroptionen benutzt werden.

"oma.srx"

Dieses Skript übersetzt eine Assemblerdatei mit dem Assembler OMA.

"masm.srx"

Dieses Skript übersetzt eine Assemblerdatei mit dem Assembler MASM.

Weitere Skripte:

"catcomp.srx"

Dieses Skript übersetzt eine Lokale Katalogdatei mit dem Programm catcomp.

"librarian.srx"

Auch der StormLibrarian kann über Makeskripts aufgerufen werden. Ein Projekteintrag
der Sektion "Librarian" kann mit Doppelklick in den StormLibrarian geladen
oder mit Alt-Doppelklick kann direkt die Linkerbibliothek erzeugt werden. Wenn jedoch
ein Projekt immer eine Linkerbibliothek erzeugen soll, empfiehlt sich der Einsatz
eines Makescripts. Die Liste der Objektdateien muß wie gewöhnlich im StormLibrarian
erzeugt werden. Das Makescript ruft dann den StormLibrarian auf. Dieser erzeugt nicht
nur die Bibliothek automatisch, sondern setzt auch die Linkerbibliothek als
Objekt (mit OBJECTS) und alle Objektdateien der Liste als abhängige Dateien (mit
DEPENDENCIES). Dadurch wird nach dem ersten Make automatisch die Linkerbibliothek
neu erzeugt, wenn einer der C oder Assemblerquelltexte neu kompiliert wurde.
Ebenso wird die Bibliothek neu erzeugt, wenn an der Liste der Objektdateien mit Hilfe
des StormLibrarian Änderungen gemacht wurden.

"fd2pragma.srx"

Dieses Makescript erzeugt aus einer FD Datei eine Headerdatei mit den notwendigen
"#pragma amicall" Anweisungen für eine shared library. Normalerweise ist dieses
Skript nicht notwendig, denn der Linker erzeugt ebenfalls diese Headerdatei
automatisch, wenn eine shared library gelinkt wird.

1.11 StormC.guide/STC_Profiler2

Der Profiler

Ein Profiler ist ein unverzichtbares Hilfsmittel, wenn es darum geht, ein Programm zu optimieren. Alle Optimierungsmöglichkeiten des Compilers können ein Programm immer nur graduell beschleunigen. Ein Profiler kann jedoch Hinweise geben, welche Funktionen eines Programms besonders viel Zeit benötigen. Diese Funktionen sollten dann, falls möglich, mit einem besseren Algorithmus neu geschrieben oder zumindest mit der Hand so weit wie möglich optimiert werden.

Der StormC Profiler ist dabei besonders leistungsfähig, erlaubt eine exakte Zeitmessung und gibt viele, statistisch wertvollen Information über das Programm.

Wie immer haben wir uns an unsere Maxime gehalten, daß auch für den Profiler keine spezielle Programmversion erzeugt werden muß. Die Erzeugung der Debuginformationen reicht aus, um das Profilerprotokoll zu erstellen. Dies - wie auch die Möglichkeit den Profiler während des Debuggens laufen lassen zu können - ist wohl einmalig auf dem Amiga.

Um den Profiler zu aktivieren, müssen im Compilereinsteller die Debugoption auf "Große Debugdateien" oder "Kleine Debugdateien" stehen. Im Starteinsteller muß die Option "Profiler benutzen" angewählt sein. Dann kann das Programm wie gewohnt gestartet werden. Das gleichzeitige Debuggen des Programms ist möglich, kann jedoch geringfügige Fehler in der Zeitmessung zur Folge haben.

Nach dem Start des Programms kann das Profilerprotokollfenster geöffnet werden. Der Knopf links oben bringt das Protokoll auf den aktuellen Stand, d.h. alle Prozent- und Zeitwerte werden auf den aktuellen Stand gebracht.

In der Hilfezeile wird die aktuelle CPU Zeit angezeigt. Diese Zeit ist die echte CPU Zeit, d.h. Zeitspannen, in welchen das Programm wartet (auf Signale, ↩
Nachrichten
oder I/O), werden nicht mit eingerechnet, ebensowenig Zeitspannen, in denen das Programm nicht läuft, weil andere Programme im Multitasking an der Reihe sind.

In der Liste unten werden die Funktionen mit folgenden Informationen angezeigt:

1. Der Funktionsname.

Memberfunktionen einer Klasse wird der Klassenname mit zwei Doppelpunkten vorangestellt.

2. Der prozentuale Zeitbedarf.

Dabei wird nur die Zeit berechnet, die das Programm
in der jeweiligen Funktion oder einer tiefer verschachtelten ↩
Betriebssystemfunktion
verbracht hat. Nicht berücksichtigt werden Funktionsaufrufe im eigenen Programm. Dies ist der wichtigste Wert, um herauszufinden, welche Funktion besonders viel Zeit benötigt. Die Summe dieser Spalte aller Funktionen beträgt 99 - 100 % (das fehlende Prozent geht im Startupcode und kleinen Ungenauigkeiten verloren).

3. Der prozentuale rekursive Zeitbedarf.

Hierbei werden alle Unterprogrammaufrufe

mit aufsummiert. Deshalb hat die Funktion main() fast immer 99 %.

4. Der absolute Zeitbedarf.

5. Der maximale Zeitbedarf.

6. Der minimale Zeitbedarf.

Diese drei Spalten erlauben einen Überblick über die Aufrufe dieser Funktion. Wie eine Funktion optimiert werden kann, hängt oft auch davon ab, ob die Funktion im allgemeinen für jeden Aufruf ähnlich viel Zeit benötigt (geringer Unterschied zwischen maximaler und minimaler Zeit) oder für einige Aufrufe deutlich länger als für andere Aufrufe braucht (hoher Zeitunterschied). Dann lohnt es sich eventuell, diese Spezialfälle zu optimieren.

7. Die Anzahl der Aufrufe.

Eine Funktion, die relativ viel Zeit benötigt, aber nur dadurch, daß sie sehr ↩ häufig
aufgerufen wird, pro Aufruf nur sehr kurze Zeit braucht, ist ein "schwerer Kandidat" zur Optimierung. Natürlich kann es hier besonders helfen, die Funktion als Inlinefunktion zu deklarieren ("__inline" in ANSI-C, "inline" in C++).

Über dieser Liste befinden sich einige Elemente zur Arbeit mit dem Protokoll:

Die oberste Zeile ist die Hilfezeile mit kurzen Hilfetexten über die Bedienungselemente.

Darunter befinden sich links drei Knöpfe. Der erste bringt die Funktionsliste auf den neusten Stand.

Mit dem zweiten Knopf kann man das Profilerprotokoll als ASCII Datei abspeichern, dazu öffnet sich ein Dateirequester.

Der dritte Knopf druckt das Protokoll auf dem Drucker (verwendet wird das Gerät "PRT:") aus.

Rechts in dieser Zeile kann man die Sortierung der Funktionsliste festlegen. Der erste Eintrag "Prozentual" sortiert die Liste nach der 2. Spalte, der Eintrag "Rekursiv" sortiert die Liste nach der 3. Spalte, der Eintrag "Alphabetisch" sortiert die Liste alphabetisch nach den Funktionsnamen der 1. Spalte. Und der letzte Eintrag "Aufrufe" sortiert die Liste nach der letzten Spalte.

Eine Zeile tiefer befindet sich ein Texteingabefeld für ein DOS Pattern. Dieses DOS Pattern wird auf die Funktionsnamen angewendet und erlaubt so die Beschränkung der häufig sehr langen Liste auf ein überschaubares Maß. Zum Beispiel kann man die Liste sehr einfach auf alle Methoden einer Klasse beschränken, indem man den Namen der Klasse gefolgt von "#?" eingibt.

Rechts daneben befindet sich ein Zahleingabefeld zur Angabe einer ↩ Mindestprozentzahl,
die die Funktionen belegt haben müssen, um angezeigt zu werden. Funktionen mit ↩ weniger
als 5 oder 10 % sind häufig nur sehr schwer zu optimieren und selbst eine ↩ Halbierung
der Laufzeit würde das Programm nur unwesentlich (eben um 2.5 bis 5 %) ↩ beschleunigen.

Neben diesen beiden wählbaren Einschränkungen kommen grundsätzlich nur Funktionen zur Anzeige, die mindestens einmal aufgerufen wurden.

Eine Funktion kann über einen Doppelklick auch direkt im Quelltext angesprungen werden.

Das Profilerprotokoll wird am Ende des Programmlaufs auch automatisch geöffnet und die Anzeige auf den neuesten Stand gebracht. Das Kontrollfenster bleibt dann ebenfalls geöffnet. Wird das Kontrollfenster geschlossen, so wird auch das Protokollfenster geschlossen und die Liste gelöscht. Werden die Profilerinformationen noch benötigt, so müssen sie vorher gespeichert oder gedruckt werden.

Technische Informationen

Beim Funktionseinsprung werden die LINE $A Befehle $A123 und $A124 benutzt. Diese beiden Maschinenbefehle sind bei allen Motorola 68K CPUs unbenutzt und lösen eine Exception aus. Diese Expcetion führt die Protokollierung der Zeiten und Aufrufe durch.

Durch die Nutzung der Exceptions hat man den kleinen Nachteil, daß diese die maximale Programmgeschwindigkeit dämpfen, d.h. das Programm langsamer abläuft, als ohne Profilerbetrieb. Besonders bei sehr vielen Aufrufen kurzer Funktionen kann sich das bemerkbar machen. Allerdings sollte der Profiler immer noch schneller und vor allen Dingen genauer arbeiten als die meisten bisher unter Amiga OS bestehenden. Zudem wird dieser Nachteil sicher durch den Vorteil aufgewogen, das Programm für den Profiler nicht speziell kompilieren zu müssen.

Die Benutzung von Rekursionen ist eingeschränkt: die Minimum- und Maximumwerte sind meistens nicht korrekt, die Gesamtzeit (und damit auch die prozentualen Werte) können falsch sein. Eine einfache Rekursion (f() ruft f() auf) zeigt die richtigen prozentualen Werte an, eine verschachtelte Rekursion (f() ruft g() ruft f() auf) akkumuliert alle Zeiten auf einer der beiden Funktionen. Funktionsaufrufe, die aus der Rekursion herausführen, werden allerdings korrekt behandelt.

Die Wirkung auf Long Jumps ist nicht vorhersehbar, meistens sollte es jedoch nur zu einer kleinen Ungenauigkeit bei der verlassenen und eingesprungenen Funktion kommen.

Theoretisch können nicht alle Funktionen gemessen werden: Nur Funktionen, deren Maschinencode mit einem link oder einem movem Befehl beginnen, stehen dem Profiler zur Verfügung. Allerdings ist selbst bei hohen Optimierungsstufen des Compilers fast immer einer der beiden Befehle notwendig. Und Funktionen, die ohne diese Befehle auskommen, müssen so kurz sein (keine Variablen auf dem Stack, nur die Register d0, d1, a0 und a1 verändert), daß sie kaum noch optimiert werden können oder dafür auch nur interessant sind.

Inlinefunktionen können im allgemeinen nicht gemessen werden.

1.12 StormC.guide/STC_SASPort

Anmerkungen zur Portierung von SAS/C nach StormC

Wir haben uns bemüht, dem StormC Compiler viele wichtige Eigenschaften des SAS/C Compilers mitzugeben, d.h. diverse Schlüsselwörter und #pragmas, die SAS/C spezifisch sind. Dennoch wird es - abhängig von Ihrem Programmierstil - zu mehr oder weniger großen Problemen kommen, wenn Sie Software vom SAS/C Compiler auf den StormC Compiler portieren.

Bedenken Sie, daß StormC ein ANSI-C und C++ Compiler ist. SAS/C hingegen ist ein C und ANSI-C Compiler (den C++ Precompiler hat wohl kaum jemand ernsthaft benutzt), d.h er versteht viele alte Syntax, die StormC nicht akzeptiert. Nur wenn Sie Ihre Programme mit SAS/C grundsätzlich strikt ANSI übersetzt haben (SAS/C Option ansi), sollten Ihre Programm auch problemlos mit dem StormC übersetzbar sein.

Projekteinstellungen

Zuerst sollten Sie darauf achten, daß in dem Projekt, das Sie aus ihren SAS/C Quelltexten aufgebaut haben, ANSI-C als Compilermodus eingestellt ist. Schalten Sie möglichst viele Warnungen an, und versuchen Sie später, Ihre Programme so zu verändern, daß auch keine Warnungen kommen. Dadurch haben Sie die größtmögliche Sicherheit, daß das Programm auch das tut, was Sie wollen.

Selbst für reine ANSI-C Projekte empfiehlt sich die spätere Umstellung auf C++. Dadurch erhalten Sie einige Vorteile: Prototypen werden immer erwartet und der implizite Cast eines void * in einen anderen Pointer ist verboten. Dies bedeutet zwar zuerst eine relativ aufwendge Überarbeitung Ihres Programms (besonders der zweite Fall kann durch malloc() oder AllocMem() sehr häufig vorkommen), aber dann können Sie viel sicherer sein, daß Ihr Programm auch wirklich das tut, was es soll.

Ebenso bieten die langen C++ Symbole eine zusätzliche Sicherheit beim Linken: Wenn Prototyp und Funktionsdefinition aus irgendeinem Grund nicht übereinstimmen, so wird der Linker einen entsprechenden Fehler ausgeben, daß ein bestimmtes Symbol nicht gefunden werden konnte.

Außerdem steht Ihnen dann die Möglichkeit offen, Ihr Programm um die modernen objektorientierten Konzepte zu erweitern und weitere kleinere C++ Features zu benutzen (z.B. die Variablendeklaration an jeder Stelle im Code und nicht nur am Anfang eines Klammerblocks).

Syntax

Manche SAS/C Keywords versteht der StormC Compiler nicht, andere sehr wohl, aber nur in der typischen ANSI-C Syntax und nicht so frei wie der SAS/C Compiler.

Unbenannte unions werden auch von StormC akzeptiert, aber implizite Strukturen nicht. Ebenso werden equivalente Strukturen unterschieden. Wenn Sie dieses Feature verwendet haben, müssen Sie an den jeweiligen Stellen Casts einfügen. Falls Ihnen dieses Feature wichtig ist, so sollten Sie über eine Umstellung Ihres Projekts auf C++ nachdenken: Äquivalente Strukturen sind nichts anderes als ein Aspekt der Vererbung.

Typübereinstimmung wird in StormC viel strenger gehandhabt. Das gilt insbesondere auch für den Funktionsparameterqualifizierer const. Ein Beispiel:

```
typedef int (*ftyp)(const int *);
int f(int *);
ftyp p = f; // Fehler
```

Entweder sollten Sie entsprechende Casts einführen, oder aber (was auf alle Fälle zu bevorzugen ist) Ihre Funktionen passend deklarieren, schließlich ist der const Qualifizierer ein wichtiges Hilfsmittel zur Sicherung der Korrektheit Ihres Programms.

Die C++ Kommentare ("//") werden von StormC auch im ANSI Modus akzeptiert, verschachtelte C Kommentare werden hingegen nicht akzeptiert. Allerdings können Sie eine entsprechende Warnung aktivieren, die auf diese Gefahr hinweist.

Umlaute werden in Variablennamen nicht toleriert, ebenso nicht das Dollarzeichen.

Keywords

Grundsätzlich sollten Sie auf den Einsatz der speziellen Keywords möglichst verzichten - zumindest für Programme, die Sie vielleicht auch einmal auf ein anderes Betriebssystem oder einen völlig anderen Compiler portieren wollen.

StormC benutzt stärker die in ANSI-C bevorzugten #pragma Anweisungen zur Anpassung der Software an die speziellen AmigaOS Bedingungen (z.B. #pragma chip und #pragma fast).

Für Keywords, die auf anderen Compilersystemen vielleicht nicht existieren, aber auch nicht notwendigerweise benutzt werden müssen, empfiehlt sich die Verwendung geeigneter Makros, z.B.:

```
#define INLINE __inline

#define REG(x) register __##x

#define CHIP __chip
```

Diese Makros können dann einfach den Compilergegebenheiten angepasst werden.

Manche Keywords, die der StormC Compiler nicht kennt, die aber auch nicht notwendig sind, können auch als Makro definiert werden:

```
#define __asm
#define __stdarg
```

Hier folgen die Keywords des SAS/C Compilers und Ihre Interpretation durch StormC:

__aligned: ist unbekannt. Es gibt keine einfache Möglichkeit, dieses Keyword zu ersetzen, aber es wird auch nur sehr selten benötigt.

__chip: Dieses Keyword legt ein Datenelement in die Chip-Memory Sektion des Objektfiles. Beachten Sie jedoch, daß dieses Keyword wie jede Speicherklasse und andere Qualifier nur vor dem Typ der Deklaration und nicht dahinter stehen kann:

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

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

Die 2. Syntax wird nicht akzeptiert, da sie nicht ANSI-C typisch ist.

In StormC werden "#pragma chip" und "#pragma fast" bevorzugt. Beachten Sie dabei jedoch, daß __chip sich nur auf eine Deklaration bezieht, "#pragma chip" jedoch alle folgenden Datenelemente in die Chip Sektion legt, bis zum nächsten "#pragma fast"

__far und __near: sind unbekannt. Es gibt keine einfache Möglichkeit, diese Keywords zu ersetzen, aber sie werden nur sehr selten benötigt.

__interrupt: Ist ebenso unter StormC nutzbar, wie Sie es vom SAS/C her gewohnt sind.

__asm, __regargs, __stdarg: sind unbekannt und werden nicht benötigt. Wenn Sie Funktionsparameter in Registern übergeben wollen, so sollten Sie die Funktion mit dem ANSI Keyword "register" deklarieren, oder die Parametern einzeln mit dem Keyword "register" oder einer genauen Registerspezifikation (z.B. "register __a0") versehen. Ansonsten werden die Register auf dem Stack übergeben.

__saveds: Verhält sich ähnlich wie das SAS/C __saveds. Im großen Datenmodell erzeugt dieses Keyword keinen Code, im kleinen Datenmodell bzgl a4 wird a4 auf den Stack gesichert und mit dem Symbol __LinkerDB geladen, im kleinen Datenmodell bzgl a6 wird a6 entsprechend behandelt.

Verwenden Sie __saveds nicht leichtfertig. Es sollte ausschließlich für Funktionen benutzt werden, die von außerhalb Ihres Programms aufgerufen werden, z.B. Dispatcher von BOOPSI Klassen.

In der jetzigen Compiler Version empfiehlt sich nur die Verwendung des großen Datenmodells für die Erstellung von shared libraries. Bedenken Sie dabei auch, daß im kleinen Datenmodell ein weiteres Register für den Optimierer verloren geht und somit nur noch die Register a0 bis a3 zur Verfügung stehen - der Vorteil des kleinen Datenmodells kann so sehr schnell verloren gehen, wenn Sie nicht sehr viele globale Variablen benutzen.

__inline: Dieses Keyword ist ein Funktionsspezifizierer wie die anderen auch. D.h. Prototypen und Funktionsdefinitionen müssen darin übereinstimmen.

Wenn mehrere Module eine __inline Funktion benutzen wollen, so muß diese Funktion mit der Definition (also nicht nur der Prototyp) in einer Headerdatei stehen.

__stackext: ist unbekannt. Stackcheck oder automatische Vergrößerung des Stacks wird derzeit nicht unterstützt.

Pragmas

#pragma libcall, #pragma tagcall, #pragma flibcall: Obwohl diese SAS/C pragmas unterstützt sind, werden die StormC Pragmas #pragma amicall und #pragma tagcall bevorzugt. Deren Syntax ist flexibler und leichter lesbar.

Der Linker erzeugt aus der FD Datei, die die Funktionen einer shared library beschreibt, automatisch eine passende Headerdatei mit den benötigten #pragma amicall und #pragma tagcall Anweisungen.