CREST

C–Programmierung
unter RTOS–UH

©Stephan Litsch

IEP
Am Pferdemarkt 9c
30853 Langenhagen
Tel.: 0511–70832–0
Fax.: 0511–70832–99

26. Januar 2000

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Urheberrecht und Haftung

In Verbindung mit dem Kauf von Software erwirbt der Käufer einfaches, nicht übertragbares Nutzungsrecht. Dieses Recht zur Nutzung bezieht sich ausschliesslich darauf, dass dieses Produkt auf oder in Zusammenhang mit jeweils **einem** Computer zu benutzen ist. Das Erstellen einer Kopie ist ausschliesslich zu Archivierungszwecken unter Aufsicht des Käufers oder seines Beauftragten zulässig. Der Käufer haftet für Schäden, die sich aus der Verletzung seiner Sorgfaltspflicht ergeben, z. B. bei unautorisiertem Kopieren, unberechtigter Weitergabe der Software etc. . .

Der Käufer gibt mit dem Erwerb der Software seine Zustimmung zu den genannten Bedingungen. Bei unlizensiertem Kopieren muss vorbehaltlich einer endgültigen juristischen Klärung von Diebstahl ausgegangen werden. Dies gilt ebenso für Dokumentation und Software, die durch Modifikation aus Unterlagen und Programmen dieser Distribution hervorgegangen ist, gleichgültig, ob die Änderungen als geringfügig oder erheblich anzusehen sind.

Eine Haftung des Autors für Schäden, die auf den Gebrauch von Software, Hardware oder Benutzung dieses Manuskriptes zurückzuführen sind, wird ausdrücklich ausgeschlossen, auch für den Fall fehlerhafter Software oder irrtümlicher Angaben.

Das Einverständnis des Käufers oder Nutzers für den Haftungsausschluss gilt mit dem Kauf und der Nutzung der Software und dieser Unterlagen als erteilt.

Kapitel 1

Einleitung

1.1 Vorwort zur Release 2

Zunächst muss ich mit einer grossen Entschuldigung beginnen: die Arbeit an anderen Projekten hat sich übel auf die Aktualität der gedruckten Dokumentation zum CREST–C-Projekt ausgewirkt. Ausser der HISTORY-Datei existierte auch zwei Jahre nach den ersten Beta-Auslieferungen praktisch keinerlei gültige Dokumentation zu den Produkten dieser Generation.

Ich habe aus dem Feedback der letzten fünf Werke aus meiner Hand zweierlei gelernt: mehr als 300 Seiten wirken abschreckend und werden schon aus Prinzip nicht mehr gelesen. Als Resultat dieser Erkenntnis sind einige Kapitel ersatzlos gestrichen oder gnadenlos ausgedünnt worden, um trotz lesbarbarer Schriftgrösse das angepeilte Seitenlimit nicht zu überschreiten.

Als weitere Neuerung ist eine Online-Version der Dokumentation in Form von HTML-Seiten hinzugekommen, um bei Änderungen nicht stets neues Papier drucken zu müssen. Weiterhin ist das Handbuch in einer DVI-, PostScript- und PDF-Version verfügbar.

Mit der Release 2.xxx existieren zudem nur noch drei offizielle Ausführungen des CREST–C–Paketes: die Crossentwicklungsplattformen unter WINDOWS–95/NT4.0 und LINUX sowie eine unter RTOS-UH laufende Variante! Den Abschied von DOS als Entwicklungsumgebung halte ich persönlich vier Jahre nach Einführung von WINDOWS 95 für durchaus vertretbar.

1.2 Literatur zu C

Dieses Manual wendet sich immer noch an C–Programmierer, die die Besonderheiten der CREST–C–Implementierung unter RTOS–UH kennenlernen wollen. So krass es auch klingen mag: **Das hier ist kein Lehrbuch für C!** Und ich habe nicht das zeitliche Potential, ein gutes Mannjahr in Literatur für Anfänger zu stecken, solange es für ein paar Mark im Buchhandel Bücher kompletter und fehlerfreier im Doppelzentner zu erwerben gibt. Es ist auch in Zukunft eine Frage der Prioritäten, dass ich meine Zeit nicht damit verbringe, C–Bücher abzuschreiben und lieber Programme schreibe oder Dinge dokumentiere, die es sonst nicht zu kaufen gibt.

Es wurde besonderer Wert darauf gelegt, gerade an den Stellen Erklärungen zu liefern, wo das RTOS–UH–Handbuch sich eisern ausschweigt. Es handelt sich dabei nicht um Geheiminformationen aus der Hexenküche des Instituts für Regelungstechnik der Universität Hannover, sondern um den Versuch, in verständlicher Form die Fähigkeiten und Grenzen dieses Betriebssystems und dessen Programmierung unter C zu erläutern.

Als grundlegende Einführung empfiehlt sich — sofern Sie nicht gerade Ihre allererste Programmiersprache zu erlernen versuchen — die Bibel der C–Programmierer: Kernighan, Brian W., The C Programming Language, Second Edition, Prentice Hall Software Series! Dieses Buch enthält zwar ein paar Ungereimtheiten; so sind z.B. die C–Bibliotheken etwas zu kurz gekommen, aber es sind zu diesem Thema wiederum spezielle Nachschlagewerke verfügbar: Plauger, P.J., The Standard C Library, Prentice Hall. Auch dieses sehr zu empfehlende Buch ist nicht völlig fehlerfrei — wie mir inzwischen bewiesen wurde —, aber als genereller Leitfaden ist es dennoch bestens geeignet.

1.3 Allgemeines zu C unter RTOS-UH

Wenn Sie ANSI–C–Quelltexte aus der DOS– oder UNIX–Welt portieren wollen, sind — bis auf das leidige Thema der Bibliotheken — nur wenige Klippen vorhanden, die es zu umschiffen gilt. Im Laufe der Jahre ist aus CREST–C in dieser Hinsicht ein echter Allesfresser geworden, der sich darum bemüht, aus jedem halbwegs sinnvollen ANSI–C–Quelltext noch lauffähigen Code zu erzeugen. Die Gefahr besteht allerdings darin, dass CREST–C auch grobe Programmierfehler mit lapidaren Warnungen abtut — gemäss dem Motto: der Anwender wird schon wissen, was er da tut!

Der weitere Vorteil von CREST–C besteht darin, dass sich nun nahezu alles auf Hochsprachenebene abhandeln lässt, was vormalig den Einsatz eines Assemblers erforderlich gemacht hätte. Wenn es nicht gerade darum geht, an speziellen Registern der CPU (SR, SSP, FPSR, etc...) herumzuspielen oder der Einsatz von Sonder–Adressierungsarten (MOVEP, ADDX, etc...) unumgänglich ist, reicht der Sprachumfang üblicherweise aus, um Probleme bequem und lesbar zu kodieren. Beim Zugriff auf Peripheriebausteine wirkt der C–Code dann oft wie der eines Makro–Assemblers — und steht diesem im Bezug auf Effizienz auch kaum nach. Wenn eingefleischte Assemblerprogrammierer einen fünfseitigen Assembleroutput von CREST–C mit den Worten: *Also, die drei Befehle sind ja wohl Mist!* kommentieren, bin ich es durchaus zufrieden. Bei der Wartbarkeit und der Portierbarkeit der Quelltexte sammelt eine Hochsprache in jedem Falle die entscheidenden Pluspunkte.

In vielen Fällen liegt es nicht im Sinne des Entwicklers, portabel zu programmieren. Stattdessen soll das einmal ausgewählte Betriebssystem bis an die Grenzen der Leistungsfähigkeit ausgereizt werden. Wer der Hardware grundsätzlich das Maximum an Leistung entlocken möchte, kommt um die Programmierung auf Assemblerebene nicht herum. Ein Compiler kann prinzipiell keinen besseren Code erzeugen, als es ein ausgebuffter Programmierer mit mehrjähriger Prozessorerfahrung und viel überflüssiger Zeit mittels eines Assemblers tun könnte. Der Code eines Compilers ist stets nur ein Kompromiss aus vertretbarem Aufwand bei der Übersetzung und der Qualität des erzeugten Codes. Mit einer Tabelle zum Auszählen der Taktzyklen und viel Zeit an der Tastatur, sieht Compilercode bei Laufzeitvergleichen immer blass aus. Die Frage ist dabei nur, welcher Aufwand wirtschaftlich noch vertretbar ist. Die Entwicklung auf Maschinenebene verursacht erheblich höhere Kosten, als die Entwicklung in C. Dieser Punkt dürfte unstrittig sein — wird aber immer wieder gerne von den sogenannten *Entscheidungsträgern* ignoriert. Bei der Auswahl der Programmiersprache scheiden sich dann endgültig die Geister.

PEARL wurde so konzipiert, dass selbst unerfahrene Programmierer einigermassen schnell zu vertretbaren Ergebnissen gelangen können. Bei PEARL wurde besonderer Wert auf minimale Anforderungen an den Entwicklungsrechner gelegt — und auch an den Entwickler. Es ist ein grosser Vorteil von RTOS–UH, dass man selbst auf Embedded–Controllern — und ich meine nicht die VME–Bus–Schlachtschiffe mit MPC604 — noch Software entwickeln kann. Bereits mit kleinen Entwicklungsrechnern lassen sich mittels PEARL grosse modulare Echtzeitanwendungen schreiben, compilieren und testen.

Bei CREST–C hängt der Brotkorb schon deutlich höher. Die Einarbeitungszeit ist erheblich länger, da die Möglichkeiten **weit** über den Sprachumfang von PEARL hinausgehen. C ist eine Sprache, die sich an ernsthafte Programmierer wendet und sei keinem Freizeitprogrammierer empfohlen, der ohne Handbuch den Unterschied zwischen Prä– und Postinkrement nicht im Gedächtnis behalten kann.

Der Unterschied zu einer Schrotflinte besteht lediglich darin, dass jeder sich einen C–Compiler kaufen darf und niemand dabei nach einem Waffenschein gefragt wird. Wenn Sie Maschinensteuerungen entwickeln, dann können Sie mit einem schlampig geschriebenen Programm ebensogut einen Menschen töten, als ob Sie ihm eine Pistole an den Kopf halten und abdrücken...

Diese vom Konzept her sehr gelungene Sprache ist durch Leute in Verruf gekommen, die der Ansicht sind, man könne mal so eben ein Programm schreiben. Bestätigt durch schnelle Anfangserfolge — man glaubt zu schnell, die Sprache zu beherrschen, obwohl diese Annahme definitiv **falsch** ist — werden viele Anfänger leichtsinnig und verwenden Features, die der Compiler zwar als korrekt anerkennt — und das ist erschreckend viel —, über deren Folgen dann allerdings alle Menschen staunen, die davon später unfreiwillig betroffen sind.

C steht im Ruf, eine System–Programmiersprache zu sein und ermöglicht dem Programmierer nahezu unbegrenzten Zugang zu allen Möglichkeiten, die Rechner und Betriebssystem bieten. Die von vielen Gurus verteufelten Pointer erlauben eine freie Manipulation des gesamten Systems. Bewusst eingesetzt, lassen sich wunderbare Dinge damit anstellen, von denen FORTRAN– und PEARL–Programmierer nur träumen können. Gerade für den Anfänger werden die Freiheiten, die die Sprache ihm einräumt, sehr schnell zum Alptraum — und auch für mich, wenn ich feststellen muss, dass mal wieder ein Anwender am Telefon hängt, der meint, RTOS–UH mittels CREST–C vergewaltigen zu müssen, ohne auch nur von rudimentärsten Sprach– oder Betriebssystemkenntnissen belastet zu sein. Wenn ich in ein unbekanntes Auto einsteige, dann schaue ich in der Bedienungsanleitung nach, wie die Gangschaltung funktioniert und wie ich das Licht anzuschalten habe — eben so als Minimum meines Interesses. Viele Neu–C–Programmierer sind da anders: wenn Sie am Baum kleben, beschimpfen Sie die Sprache, weil Sie sich nie die Mühe gemacht haben, sich über die Existenz einer funktionierenden Bremse zu informieren...

Die Sprache C verlangt viel freiwillige Disziplin vom Anwender. Manche Leute nutzen diese Möglichkeiten von C dickfällig aus — und fallen damit regelmässig auf die Nase. Der Erfolg solcher Unternehmungen ist dann in der Regel übelster Müll, der besser gleich nach der Fertigstellung in der Tonne verschwinden sollte. Unwartbare Programme sind vergeudete Zeit. Was Sie bei der Programmerstellung an Minuten sparen, wird beim Austesten und Debuggen in Stunden und Tagen auf Sie zurückkommen. Vor ein paar Jahren äusserte sich Brian Kernighan zum Thema C sehr treffend: *Die beste Methode, um eine Sprache zu lernen, besteht darin, einen Compiler dafür zu schreiben!* Ich habe es so gehalten und bin heute noch manchmal überrascht, was man alles mit C anstellen kann, wenn man die Sprache und das zugrundeliegende Konzept durchblickt hat. Betrachten Sie die Tatsache, dass selbst ich als Compilerbauer gelegentlich sehr scharf nachdenken muss, um herauszubekommen, was mein eigener C–Compiler jetzt wohl aus einem bestimmten Konstrukt zu machen gedenkt, als Warnung!

Gerade beim Einsatz in sicherheitsrelevanten Bereichen obliegt es dem Programmierer, mit besonderer Sorgfalt zu Werke zu gehen. Aber warum erzähle ich das: gerade die Leute, die sich getroffen fühlen sollten, sind die notorischen Nicht–Handbuchleser.

Anhänger von anderen Sprachen führen gerade in Hinsicht auf den Sicherheitsaspekt erbittert Krieg gegen C. Es werden die dümmsten Argumente mit seltsam anmutendem Pathos vorgetragen. C–Programmierer sind schreibfaul, schlampig, verantwortungslos und verfassen nur kryptische Programme. Ich persönlich kann mit jeder Programmiersprache Schaden anrichten — und wenn ich mir fremde Quelltexte anschaue, drängt sich mir der Verdacht auf, dass ich damit keinesfalls alleine bin. Gute und lesbare Programme sind nicht sprachgebunden. Schlechte Programme durch die falsche Wahl der Programmiersprache gibt es dagegen zu Hauf. Bitmanipulationen in PEARL sind dafür ein prima Beispiel.

Moderne Programmiersprachen geben dem Anwender in der Regel ausreichend Möglichkeiten in die Hand, seine Probleme zu lösen. Es gibt für nahezu alle Einsatzgebiete Spezialsprachen, die den jeweiligen Aufgabenstellungen mehr oder weniger gut angepasst sind. Es gibt keine wirklich guten oder schlechten Sprachen. Die Auswahl des Werkzeugs muss nur dem zu lösenden Problem angemessen sein — oder würden Sie einen Schlagbohrer zum Deckenstreichen verwenden? Jeder würde einen solchen

Versuch als Schildbürgerstreich abtun. In der Programmierer–Szene geht es in diesem Punkt schon etwas emotionaler zu. Da wird fleissig der Bohrer in den Farbkübel getunkt und bei jedem Pinselstrich mit Inbrunst der arme Bohrhammer verflucht.

C besitzt im Vergleich zu PEARL in einer Echtzeitumgebung einen gravierenden Nachteil: es existiert keinerlei Normung der Echtzeitfeatures! Unter RTOS–UH kommt für CREST–C noch ein Manko hinzu: das Betriebssystem ist quasi als Laufzeitumgebung für `UH-PEARL`–Programme entwickelt und optimiert worden! Dieser Nachteil dürfte durch die inzwischen recht umfangreiche und vollständige RTOS–UH–Funktionsbibliothek von CREST–C hinreichend kompensiert sein. Durch die Integration diverser `#pragma`–Kommandos ist die Systemunterstützung von CREST–C letztlich sogar viel umfassender.

Die ANSI–C–Norm in Bezug auf die Sprache C wurde weitestgehend implementiert. Es existieren folgende Abweichungen:

- Trigraph–Ersatzzeichen werden vom Präprozessor nicht unterstützt.
- Der Gebrauch von MC68xxx–Registernamen als Bezeichner für C–Objekte ist durch die Dummheit des eingebauten Assemblers leider nicht möglich.

Was die ANSI–C–Bibliotheken betrifft, sieht es nicht gar so rosig aus. C ist eine Sprache die für und unter UNIX aufgewachsen ist — und RTOS–UH ist nun mal kein UNIX! Funktionen aus den Bereichen der Includedateien <`locale.h`> und <`signal.h`> sind nicht vorhanden.

Der Status der implementierten ANSI–C–Bibliotheken ist ansonsten als gut zu bezeichnen. Eine Reihe von kostenlosen Bibliotheks–Testpaketen ist mit den Funktionen einverstanden — was nicht viel zu bedeuten hat, aber wenigstens eine gewisse Beruhigung meinerseits sicherstellt. An eine offizielle Validierung des Compilers ist aus Kostengründen wohl weder jetzt noch in Zukunft zu denken!

Wenn Sie CREST–C frisch in Händen halten, dann entspricht das Paket weitestgehend meiner eigenen Version. Das impliziert allerdings auch, dass eine Reihe von undokumentierten Eigenschaften und Funktionen enthalten sind, die erst noch gründlich ausgetestet werden müssen, bevor sie festgeschrieben werden. Wenn Sie über Funktionen in den Bibliotheken oder Parameter in den Usage–Meldungen stolpern, die nicht im Handbuch beschrieben sind, dann sollten Sie nicht blindwütig ausprobieren, was sich dahinter wohl verbergen mag. In der Regel handelt es sich um Dinge, die noch nicht *wasserdicht* sind, sich in der Experimentalphase befinden oder nur zum Debuggen des Compilers selbst notwendig sind.

Es handelt sich keineswegs um den Versuch, Ihnen Dinge vorzuenthalten, die prinzipiell wichtig sind — dem miesen Stil von manchen grossen Softwarehäusern möchte ich nicht unbedingt nacheifern. Manchmal handelt es sich auch nur um Features, die für den von mir benötigten Anwendungszweck zwar korrekt funktionieren, aber eben nicht allgemeingültig sind. Das `history`–File liefert in diesem Fall meist wertvolle Hinweise. Dennoch sollten Sie unbedingt anrufen, wenn noch Fragen offen geblieben sind.

Kapitel 2

Der Einstieg

Das folgende Kapitel soll einen Überblick bezüglich Lieferumfang und Tools des CREST–C–Paketes liefern. Weiterhin soll das Zusammenspiel der verschiedenen Programme erläutert und an Hand von einigen Beispiel erklärt werden, wann man unter Einsatz welcher Waffen zu dem gewünschten Ergebnis kommen kann.

2.1 Installation

Die Auslieferung des CREST–C–Paketes erfolgt inzwischen, verursacht durch die Grösse der für den Debugger benötigten Bibliotheken, auf einer CD.

Unter WINDOWS liegt das CREST–C–Paket unkomprimiert auf der CD vor. Kopieren Sie den kompletten Inhalt der Scheibe in ein beliebiges Verzeichnis auf ihrer Festplatte.

Unter LINUX liegt das CREST–C–Paket als komprimiertes `tar`-Archiv vor und kann nur komplett in einem beliebigen Verzeichnis entpackt werden.

Das CREST–C–Paket besteht im wesentlichen aus vier Komponenten:

- den ausführbaren Programmen
- den C-Standard-Bibliotheken
- den zugehörigen Headerdateien
- der Dokumentaion

Unterhalb des angegebenen CREST–C–Installationspfades liegen folgende Ordner:

- cbin: die ausführbaren Programme
- clib–68k: die Startup's und Bibliotheken für 68k-CPU's
- clib-ppc: die Startup's und Bibliotheken für PowerPC-CPU's
- h: die zu den Bibliotheken gehörigen Headerdateien

Diese Pfade sollten dem Betriebssystem über zugehörige Environment–Variablen bekannt gemacht werden, um den weiteren Umgang mit den Programmen zu vereinfachen. In den folgenden Beispielen repräsentiert der Ausdruck [INSTALL] den kompletten Pfad der CREST–C–Installation.

2.1.1 WINDOWS-95/98

Folgende Einträge sind in der Datei `AUTOEXEC.BAT` vorzunehmen:

```
PATH=%PATH;[INSTALL]\cbin
SET CCC_INCLUDE=[INSTALL]\h
SET CCC_68K_LIBS=[INSTALL]\clib_68k
SET CCC_PPC_LIBS=[INSTALL]\clib_ppc
```

2.1.2 WINDOWS-NT4.0

Die folgenden Einstellungen sind über das `Control Panel` unter `System/Environment` vorzunehmen:

```
Path: {alter Kram};[INSTALL]\cbin
CCC_INCLUDE: [INSTALL]\h
CCC_68K_LIBS: [INSTALL]\clib_68k
CCC_PPC_LIBS: [INSTALL]\clib_ppc
```

2.1.3 Linux

Folgende Einträge sind in der Datei `/etc/profile` vorzunehmen:

```
export PATH=$PATH;[INSTALL]/cbin
export CCC_INCLUDE=[INSTALL]/h
export CCC_68K_LIBS=[INSTALL]/clib_68k
export CCC_PPC_LIBS=[INSTALL]/clib_ppc
```

Damit sollten die vorbereitenden Massnahmen bereits abgeschlossen sein.

2.2 Lieferumfang im Überblick

2.2.1 cbin

Der Ordner `cbin` enthält die ausführbaren Programme des Paketes.

Unter WINDOWS sind dies:

Bei Installation für 68k-Zielsysteme:

- ccc68k.exe : C-Compiler
- cln68k.exe : Programm-Linker
- lnk68k.exe : Library-Linker
- ssl68k.exe : Shared-Library-Linker
- clm68k.exe : Library-Manager

- cop68k.exe : Object-Inspector

Bei Installation für PowerPC-Zielsysteme:

- cccppc.exe
- clnppc.exe
- lnkppc.exe
- sslppc.exe
- clmppc.exe
- copppc.exe

In beiden Paketen sind zudem die folgenden plattformunabhängigen Tools enthalten:

- cmake.exe
- ced.exe

Unter LINUX fehlt den korrespondierenden Tools die Extension `.exe`.

2.2.2 clib–68k

Der Ordner `clib-68k` enthält die Startupdateien und Standardbibliotheken für 68k-CPU's. Nach CPU-Gruppen sortiert sind dies:

- Zielsystem 68000:

fast-Libs	long-Libs	stack-Libs	debug-Libs	stackdebug-Libs
std0fast.lib	std0long.lib	std0stack.lib	std0debug.lib	std0stackdebug.lib
flt0fast.lib	flt0long.lib	flt0stack.lib	flt0debug.lib	flt0stackdebug.lib
ffp0fast.lib	ffp0long.lib	ffp0stack.lib	ffp0debug.lib	ffp0stackdebug.lib

- Zielsystem CPU32:

fast-Libs	long-Libs	stack-Libs	debug-Libs	stackdebug-Libs
std3fast.lib	std3long.lib	std3stack.lib	std3debug.lib	std3stackdebug.lib
flt3fast.lib	flt3long.lib	flt3stack.lib	flt3debug.lib	flt3stackdebug.lib
ffp3fast.lib	ffp3long.lib	ffp3stack.lib	ffp3debug.lib	ffp3stackdebug.lib

- Zielsystem 68020:

fast-Libs	long-Libs	stack-Libs	debug-Libs	stackdebug-Libs
std2fast.lib	std2long.lib	std2stack.lib	std3debug.lib	std3stackdebug.lib
flt2fast.lib	flt2long.lib	flt2stack.lib	flt2debug.lib	flt2stackdebug.lib
ffp2fast.lib	ffp2long.lib	ffp2stack.lib	ffp2debug.lib	ffp2stackdebug.lib
fpu2fast.lib	fpu2long.lib	fpu2stack.lib	fpu2debug.lib	fpu2stackdebug.lib

2.2.3 clib-ppc

Der Ordner `clib-ppc` enthält die Startupdateien und Standardbibliotheken für PowerPC-CPU's. Da bislang erst die Prozessoren MPC603/MPC604 unterstützt werden und auf Compilerebene die

Controller-Famile MPC8xx wie ein MPC604 ohne FPU behandelt werden kann, existiert nur ein Satz von Bibliotheken.

fast-Libs	long-Libs	stack-Libs	debug-Libs	stackdebug-Libs
stdpfast.lib	stdplong.lib	stdpstack.lib	stdpdebug.lib	stdpstackdebug.lib
fltpfast.lib	fltplong.lib	fltpstack.lib	fltpdebug.lib	fltpstackdebug.lib
fpupfast.lib	fpuplong.lib	fpupstack.lib	fpupdebug.lib	fpupstackdebug.lib

2.3 Der Schnell–Einstieg

Das erste Programm, das mit einem neuen C–Compiler ausprobiert wird, ist in der Regel `hello.c`. An Hand dieses Musterbeispiels mit hohen Wiedererkennungswert soll die Funktionalität von CREST–C demonstriert werden. Im Ordner `hello/` befindet sich der zugehörige Quelltext.

Die Datei `hello.c` soll compiliert werden. Die Ausgabedatei wird im gleichen Ordner angelegt und soll `hello.obj` heissen.

```
ccc hello/hello.c hello/hello.obj
```

Ich setze ausreichend Abstraktionsvermögen voraus, dass Sie statt `hello/` den Zugriffspfad ihres `hello.c` eingeben werden und gegebenenfalls die verwendeten Slashes durch Backslashes ersetzen und die korrekte Variante des Compilers verwenden (*ccc68k* bzw. *cccppc*). Nach kurzem Grübeln sollte sich der Compiler wieder melden und verkünden, er habe `hello/hello.obj` ohne Fehler erzeugen können. Jetzt wird das Objektfile im nächsten Schritt zu einem kompletten Programm zusammengebunden:

```
cln hello/hello.lnk hello/hello.sr -N=HELLO
```

Dazu muss die Datei `hello.lnk` entsprechend erstellt worden sein. Sie enthält die Liste sämtlicher Objekt– und Libraryfiles, die zum Endprodukt gehören. Es soll das Startupfile mit dem Objektcode unseres `hello.c` zusammengebunden werden. Fehlende Funktionen soll der Linker aus der Standardbibliothek für den MC68000 entnehmen. Als Beispiel soll ein C–Shellmodul entstehen. Der Name, unter dem die Shellextension später dem System bekannt sein soll, wird mit der Option `-N=HELLO` angegeben. Das Resultat heisst `hello.sr` und kann auf dem Zielsystem wie üblich in den Speicher geladen werden:

```
LOAD hello.sr
```

Mittels `?` können Sie nun feststellen, dass ein neuer Bedienbefehl `HELLO` im System vorhanden ist. Jetzt kann das Programm gestartet werden:

```
HELLO
```

Sie haben jetzt also einen kompletten Entwicklungszyklus per Hand durchgeführt. Nun können Sie den Zyklus nochmals mittels CMAKE durchspielen.

```
cmake hello/hello.mak
```

Sie werden sehen, dass Sie nichts sehen. Da Ihr Projekt bereits in aktueller Form vorliegt, gelangt CMAKE auch zu der Überzeugung, es gäbe nichts zu tun. Nach einer beliebigen Änderung in `hello.c` und einem erneuten Aufruf von CMAKE wird jedoch der gesamte Zyklus, den Sie gerade manuell eingegeben haben, vom CMAKE abgespult und Sie erhalten ein neues `hello.sr`.

Kapitel 3

Der Compiler ccc

Bei Fehlbedienungen oder Aufruf ohne Parameter gibt der CCC einen kurzen Überblick über die unterstützten Optionen aus. Sie können diese Usage–Meldungen im Anhang (ccc68k: Abschnitt C.1; cccppc: Abschnitt C.2) nachschlagen.

Das hier behandelte Beispiel kann sich von der Anzeige des von Ihnen erworbenen Compilers unterscheiden, da Software in der Regel schneller als die zugehörige Dokumentation wächst. Die Kombination aus *read_me*– und *history*–File auf der ausgelieferten CD sollte dann den jeweils aktuellen Stand beschreiben.

Der Compiler gibt Release und Erstellungsdatum aus. In diesem Fall handelt es sich um eine Crossversion unter LINUX, die Code für 68k-Prozessoren erzeugen kann.

Als erster Parameter wird der Name des zu übersetzenden C-Programmes gefordert. Es wird ein gültiger Dateiname erwartet. Wenn Ihre Datei nicht mit `.c` endet, so wird diese Extension automatisch ergänzt. Der Aufruf `ccc test` ist also identisch zu `ccc test.c`.

Der zweite Parameter ist optional. Hier kann der Name der zu generierenden Objektdatei angegeben werden. CCC generiert sich bei dessen Fehlen aus dem Namen der Eingabedatei eine Ausgabedatei mit der Endung `.obj`. Statt `ccc test` hätte man also auch `ccc test.c test.obj` mit identischer Wirkung eingeben können.

3.1 Die Optionen des ccc

Die Optionen des CCC erlauben die Ansteuerung unterschiedlichster Übersetzungsparameter des Compilers. Ohne Angabe von weiteren Optionen wird Code für den MC68000 generiert. Die weiteren Parameter stehen ebenfalls auf den Standardwerten für den kleinsten unterstützten Prozessor. Die Standard–Optionen für den `ccc68k` lauten: `-0 -A=1 -C=0 -D=0 -E=0 -G=0 -R=0 -W=5 -Y=0`.

Die Angabe der Optionen des CCC erfolgt hinter dem bzw. den Dateinamen und wird durch einen Bindestrich „–“ eingeleitet. Optionen ohne zusätzliche Parameter können aneinandergereiht werden. So ist die Schreibweise `-2V` identisch mit `-2 -V`; die Eingabe `-R=1V` ist dagegen unzulässig. Die mehrfache Angabe von identischen Optionen führt in der Regel nicht zu Fehlermeldungen.

In den folgenden Kapiteln werden die Optionen des Compilers nach Themengebieten sortiert beschrieben. In den Tabellen ist — sofern vorhanden — die Standardoption in Fettdruck hervorgehoben.

3.1.1 Zusätzliche Argumente

Sind mehr Argumente an den CCC zu übergeben, als in der Kommandozeile untergebracht werden können, so lassen sich durch eine Argumentdatei weitere Parameter an den Compiler durchreichen. Die Anweisung

```
ccc test.c -2 @test.opt -A=2 @/h0/args
```

wird so abgearbeitet, dass zunächst alle direkten Optionen, die in der Kommandozeile aufgetaucht sind — hier `-2` und `-A=2` — ausgewertet und danach hintereinander die Datei `test.opt` auf dem aktuellen Workingdirectory und die Datei `/h0/args` gelesen und ausgewertet werden.

Innerhalb der Argumentdateien sind die Optionen zeilenweise anzugeben. Das folgende Beispiel stellt eine korrekt aufgebaute Argumentdatei dar.

```
-2U
# Ich bin ein Kommentar
-#NO_MEMORY_H
-#NO_FCNTL_H
-V
```

Der Ersatztext bei Makrodefinitionen umfasst den gesamten Text der Zeile und wird nicht hinter eventuell enthaltenen Blanks abgebrochen. Lediglich abschliessende Blanks werden geskippt.Über die Kommandozeile hätte der Ersatzstring des Makros in doppelte Hochkommata gesetzt werden müssen. In der Argumentdatei wird jede Zeile quasi als ein Kommandozeilenargument aufgefasst. Unzulässig sind demnach Angaben der Form `-C=1 -R=2` in einer Zeile der Argumentdatei.

Leerzeilen in Argumentdateien sind zulässig und werden ignoriert. Die Angabe von Kommentaren kann (wie bei Linkfiles und Makefiles) hinter dem Doppelkreuz # erfolgen. Die Verwendung der `-@`–Option innerhalb einer Argumentdatei ist nicht erlaubt. Die Argumentdateien können nur Optionen und keine Dateinamen aufnehmen.

3.1.2 Einstellung des 68K–Zielprozessors

Entsprechend dem vorgesehenen Zielsystem sind Optionen verfügbar, die die Auswahl des jeweiligen CPU/FPU–Befehlssatzes steuern. Der CCC generiert dann die entsprechenden Assembler–Sequenzen für die ausgewählte CPU/FPU–Kombination. Dabei sind vier Wahlmöglichkeiten verfügbar. Die Option `-0` ist die Standardeinstellung und erzeugt Code für die Prozessoren MC68008, MC68000, MC68010, MC68301 etc...

Die nächste Prozessorgruppe wird unter der Option `-2` zusammengefasst und betrifft die CPU's MC68020 bis MC68060. Es werden doppeltindirekte Addressierungen generiert, Multiplikationen und Divisionen von Langworten von der CPU direkt unterstützt, Tests auf Adressregister sowie lange relative Sprünge und lange Adressregister– und PC–relative Adressierungsarten zugelassen.

Die Option `-3` bedient die `CPU32`–Controller-Familie von Motorola. Diese Familie besitzt einen kastrierten Befehlssatz des MC68020, bei dem z.B. die eher selten benutzten Bitfeld–Operationen und die doppeltindirekten Adressierungsarten gestrichen wurden. Aus Sicht des Hardwareentwicklers stellt die `CPU32`–Reihe jedoch eine leistungsfähige und preisgünstige Alternative zum MC68020 dar.

Die Optionen -0, -2 und -3 schliessen sich gegenseitig aus.

Die Prozessoren MC68040 und MC68060 sind die HighEnd–Prozessoren in Motorolas 68k–Serie. Sie enthalten eine abgespeckte Version des Floatingpoint–Prozessors MC68881/MC68882. Um die FPU

ansprechen zu können, existiert die Option `--fpu`. Der CCC generiert dann die entsprechenden FPU–Befehle und verzichtet auf die Emulation der Fliesskommaarithmetik. Bei Verwendung dieser Option legt der CCC intern das Makro `__FPU__` an

Durch die Verwendung dieser Optionen werden automatisch Makros vom Compiler definiert, die die Compilationsparameter wiederspiegeln. Der Tabelle 3.1 können Sie die definierten Namen und deren Ersatztexte entnehmen.

Option	Makro	Ersatztext
`-0`	__MC68000__	(1)
`-2`	__MC68020__	(1)
`-3`	__CPU32__	(1)
`--fpu`	__FPU__	(1)

Tabelle 3.1: CPU/FPU – Optionen 68K

3.1.3 Prozessorabhängige Adressierungsarten für 68K

Der MC68000 und seine Artverwandten sind durch eingeschränkte Adressierungsarten sind Vergleich zu ihren Nachfolgern im Nachteil. Er ist z.B. nicht in der Lage, Adressregister– und PC–relative Adressierungen oder relative Sprünge von mehr als 32kB auszuführen. Das bedeutet für den CCC bei der Option `-0`, dass

- ab 32kB–Sprungdistanz absolute Sprünge verwendet werden müssen
- der Umfang globaler Variablen (ohne Sonderaufwand) auf 32kB beschränkt ist
- als konstant abgelegte Daten nie weiter als 32kB vom Ort ihrer Referenzierung abgelegt werden dürfen
- lokale Variablen innerhalb einer Funktion 32kB nicht überschreiten können

Die Nachfolgechips bieten neue Möglichkeiten. Die Optionen `-C=digit`, `-D=digit`, `-E=digit` und `-R=digit` sind dazu gedacht, um Einfluss auf die Adressierungsarten der gewählten Ziel–CPU zu nehmen.

`-R` steuert die Generierung von Unterprogrammaufrufen. Der MC68000 besitzt keine Möglichkeit, Unterprogrammaufrufe mit Sprungdistanzen grösser 32kB mittels *relativer* Adressierung auszuführen. Erstmal ist das kein grosser Verlust, aber in Hinsicht auf die Erstellung von verschieblichem Code schmerzt diese Tatsache dann doch. Nähere Informationen über verschieblichen Code können dem Abschnitt 3.11 entnommen werden. Die neueren Prozessoren sind glücklicherweise in der Lage, dieses Problem sauber zu lösen, und so lässt sich verschieblicher Code produzieren, bei dem die 32kB–Grenze nicht mehr begrenzend wirkt. `-R` ist mit vier Parametern zulässig:

Option	Funktionsaufruf	verschieblich	Zielprozessor
–R=0	JSR function	nein	alle
–R=1	JSR (function.W,PC)	ja	alle
–R=2	BSR.L function	ja	ab MC68020–Kern
–R=3	BSR.W function	ja	alle

Tabelle 3.2: Funktionsaufrufe

Programme für CPU´s mit 68000er–Kern dürfen also nie mit der Option `-R=2` compiliert werden. Standardmässig ist die Option `-R=0` eingestellt, aber wenn die Gesamtgrösse des Codes unterhalb der magischen 32kB–Grenze liegt, so lässt sich das Projekt auch mit der Option `-R=1` oder `-R=3`

übersetzen. Dabei spart die CPU bei jedem Sprung das Lesen eines Maschinenwortes aus dem RAM — der Code wird kürzer und schneller.

Option	Variablenzugriff	verschieblich	Zielprozessor
–C=0	MOVE.L D0,(var.W,A5)	ja	alle
–C=1	MOVE.L D0,(var.L,A5)	ja	ab MC68020–Kern

Tabelle 3.3: Zugriff auf globale Variablen im .data– und .bss–Section

Die Optionen `-C=digit`, `-D=digit` und `-E=digit` tragen ebenfalls den eingeschränkten Fähigkeiten des MC68000 Rechnung. Der vom CCC generierte Code ist üblicherweise — und wenn nicht explizit anders verlangt — wiedereintrittsfest. Um das zu erreichen, werden die globalen und lokalen Variablen eines erzeugten C–Programmes nicht auf festen Adressen abgelegt — das ergäbe ein übles Gehaue um Variablen und sicherlich nicht das erwünschte Ergebnis. Vielmehr findet die Adressierung dieser Speicherstellen über Adressregister statt, deren Inhalt sich von C–Subtask zu C–Subtask unterscheidet. Lediglich die Offsets relativ zum Basisregister sind für alle erzeugten C–Subtasks stets identisch.

Beim MC68000 tritt bei diesen Offsets die 32kB–Grenze wieder auf. Für den C–Programmierer bedeutet dies, dass der Gesamtspeicherbereich der globalen Variablen, die bereits **beim Start** des Programmes zur Verfügung stehen, 32kB nicht überschreiten darf und der Prozessor immer mit WORD–Offsets adressregister–relativ auf die Variablen zugreifen muss. Das stellt allerdings — was den verfügbaren Speicher angeht — keine grosse Einschränkung dar, da sich z.B. über die Bibliotheks–Routine `malloc()` beliebig Nachschlag zur Laufzeit anfordern lässt.

Die Option `-C=digit` bezieht sich auf Variablen, die in der `.data`– bzw. in der `.bss`–Section abgelegt sind. Die Option `-E=digit` steuert den Zugriff auf die task–lokalen Variablen in der `.local`–Section. Bedenken Sie bitte bei der Verwendung von Variablen aus der `.local`–Section, dass **jede** Task — also alle Sohnprozesse, die von einem `main()` abgespalten werden — eigenen Speicher für diese Variablen anlegt. Dort mehr als 32kB abzulegen, ist zwar prinzipiell mittels `-E=1` machbar, sollte aber vermieden werden.

Option	Variablenzugriff	verschieblich	Zielprozessor
–E=0	MOVE.L D0,(var.W,A4)	ja	ab MC68000
–E=1	MOVE.L D0,(var.L,A4)	ja	ab MC68020

Tabelle 3.4: Zugriff auf task–lokale Variablen in der .local-Section

Gerade für Konstanten — z.B. Tabellen — besteht zudem keinerlei Notwendigkeit, jeder Subtask eine eigene Kopie zur Verfügung zu stellen. Deshalb werden derartige Blöcke in der `.text`–Section des erzeugten Programmes untergebracht und `PC`–relativ adressiert. Auch die `PC`–relative Adressierung derartiger Konstanten unterliegt der Beschränkung auf 32kB. Deshalb gibt es auch hier eine Einstellmöglichkeit mittels der `-D=digit`–Option. Liegt eine Konstante ausserhalb der 32kB–Reichweite, so muss das auf die Konstante zugreifende Modul mit `-D=1` compiliert werden, um den Linkerlauf fehlerfrei zu überstehen.

Option	Konstantenzugriff	verschieblich	Zielprozessor
–D=0	MOVE.L (var.W,PC),D0	ja	ab MC68000
–D=1	MOVE.L var.L,D0	nein	bei MC68000
–D=1	MOVE.L (var.L,PC),D0	ja	bei MC68020

Tabelle 3.5: Zugriff auf globale Konstanten

Besitzer grösserer Prozessoren können bei Benutzung der 16–Bit–Varianten der gerade aufgeführten Optionen viele unnütze Befehls– und Offset–Worte im Programmcode einsparen — sie sind jedoch im Gegensatz zu MC68000–Programmierern nicht dazu gezwungen. Die gleiche Begrenzung auf 32kB

gilt auch für die lokale Speicherplatzanforderung beim Start einer Funktion. Da der Compiler hier Überschreitungen der 32kB –Grenze schon zur Übersetzungszeit feststellen kann, generiert er Fehlermeldungen bei einem derartigen Ansinnen eines mit `-0` übersetzten Programmes. Die Überschreitung des globalen Speicherbedarfes kann jedoch meist erst beim Linken der einzelnen Module festgestellt werden. Der Linker gibt dann Relokationsfehler aus.

3.1.4 Vorzeichen bei Charactern

Da CREST–C üblicherweise davon ausgeht, dass der Datentyp `char` als `unsigned char` aufzufassen ist, besteht mittels der `-S`–Option die Möglichkeit, den Datentyp `signed char` als Standard aufzufassen.

3.1.5 Übergabe von Makros

Mit der Option `-#macnam[=repl]` lassen sich von der Kommandoshell aus argumentfreie Makros definieren. Der Aufruf `-#HALLO=HELLO` entspricht einem `#define HALLO HELLO` zu Beginn der ersten zu compilierenden Datei. Es lassen sich (nur durch die Länge der Kommandozeile begrenzt beliebig viele Makros auf diese Weise importieren. Zusätzliche Makros lassen sich — wie im Abschnitt 3.1.1 erläutert — über Argumentdateien einlesen.

3.1.6 Zeilenkommentare

CREST–C erlaubt die von `C++` eingeführten Zeilenkommentare. Zwei Schrägstriche (Slashes) „//“ leiten (solange sie sich nicht in einem String auftreten) einen Kommentar ein, der bis zum Zeilenende geht:

```
// Dies ist ein Kommentar, der bis zum Zeilenende geht
```

3.1.7 Boolscher Datentyp

Mittels der Compileroption `-+` lässt sich der von `C++` bekannte Datentyp `bool` im CCC zuschalten. Dazu gehören die vordefinierten Literale `true` und `false` . Wird diese Spracherweiterung verwendet, so definiert der CCC automatisch das Makro `__CPLUSPLUS__` .

3.1.8 Makroverarbeitung in Pragma–Kommandos

Üblicherweise werden `#pragma`–Kommandos vom CCC nicht durch den Makro–Präprozessor geschickt, sondern in der Form interpretiert, wie sie im Quelltext vorzufinden sind. In manchen Fällen ist es jedoch sehr nützlich, auch Kommandozeilen mittels Makros abzufassen oder gar den Compiler für sich rechnen zu lassen. Mittels der Compileroption `-q` wird die Quelltextzeile hinter der Steueranweisung `#pragma` zunächst durch den Makroverarbeitungsteil des Compilers geschickt.

Die `#pragma`–Kommandos, die numerische Optionsparameter entgegennehmen können, wurden in der Syntax dahingehend erweitert, dass zusätzlich zu den bislang unterstützten Zahlenangaben gemäss C–Nomenklatur nun auch *konstante* Ausdrücke ausgewertet werden können. Um den CCC syntaktisch davon zu überzeugen, dass ein auszuwertender Ausdruck folgt, **muss** dieser grundsätzlich geklammert werden.

```
#define MY_IR_LEVEL  4
typedef struct
{  Task   *tid          ;
   char   *reader_ptr   ;
   char   *writer_ptr   ;
   char    buffer[ 256 ] ;
}  DataSpace ;
#pragma ALLOCATE_INTERRUPT_BUFFER   \
        LEVEL MY_IR_LEVEL           \
        SIZE ( sizeof( DataSpace ) )

#define MY_STACK     ( 1024 + 256 )
#pragma TASK STACKSIZE MY_STACK
```

Abbildung 3.1: Makroverarbeitung und Expressionauswertung in `#pragma`–Zeilen

Wie der Darstellung 3.1 zu entnehmen ist, ist nicht nur die Expressionauswertung von numerischen Konstanten möglich; vielmehr sind alle Expressions gültig, die bereits der Compiler evaluieren kann und die ein Integerresultat liefern.

3.1.9 Includedateien

Ein C–Programm besteht üblicherweise aus mehreren Modulen (Übersetzungseinheiten), die die Dateiendung `.c` besitzen. Globale Informationen, die in mehreren Modulen verfügbar sein müssen, werden in der Regel in gesonderten Dateien gesammelt. Bei diesen gemeinsamen Informationen handelt es sich in der Regel um eine Zusammenstellung von Makros, applikationsspezifische Datentypen, Variablendeklarationen und Funktionsprototypen. Der Aufbau und Inhalt dieser Dateien ist weitestgehend eine Erfahrungssache und Geschmacksfrage. Im Prinzip kann eine solche Datei einen beliebigen Namen erhalten und beliebigste gültige C–Syntax enthalten — also auch Funktionen oder Bruchstücke von Funktionen. In der Praxis haben sich jedoch einige Regeln in Bezug auf derartige Dateien weitestgehend durchgesetzt. Die Dateiendung lautet normalerweise `.h` — wobei diese Extension für `header` steht, da diese Dateien üblicherweise ganz oben — also im Kopf — eines C–Modules mittels der Präprozessoranweisung `#include` eingefügt werden. Im Sprachgebrauch haben sich diverse Bezeichnungen wie *Headerfile* und *Includedatei* festgesetzt.

Die Aufgabe des Präprozessors besteht nun unter anderem darin, die `#include`–Anweisungen korrekt aufzulösen und die verschiedenen Dateien für den späteren Compilerlauf zu einer Einheit zu verschmelzen. Im Umgang mit Headerdateien treten mehrere interessante (und lästige) Probleme auf:

1. Der Präprozessor muss die angegebene Includedatei finden.
2. Ein mehrmaliges Einfügen der gleichen Includedatei innerhalb einer einzelnen Übersetzungseinheit sollte vermieden oder wenigstens so gehandhabt werden, dass der Compiler durch das wiederholte Auftreten identischen Quelltexte nicht in syntaktische Probleme verwickelt wird.
3. Wenn eine Headerdatei in mehr als einem Modul eines C–Programmes eingefügt werden soll, so darf sie keine Definitionen von globalen Symbolen enthalten, da sonst später der Linker mit einer Fehlermeldung abbricht.

3.1.9.1 Kontrollausgaben während der Compilation

Die Optionen `-V` und `-Q` dienen lediglich der optischen Überprüfung, mit welcher Datei der CCC zur Zeit beschäftigt ist. Im Normalfall sind diese Optionen obsolet und bremsen nur die Compilation durch

unnötige Bildschirmausgaben.

Option	Wirkung
–V	Anzeige der includierten Dateien und übersetzten Funktionen
–r	Anzeige fehlgeschlagener Zugriffe auf Includedateien
–Q=digit	Anzeige des Quelltextes bis zur angegeben Includetiefe

Tabelle 3.6: Kontrollausgabe bei der Compilierung

Die `-V`–Option zeigt die gerade in Arbeit befindlichen Dateien namentlich an und und bringt auch die jeweilige Funktion auf den Schirm.

Nützlich ist in diesem Zusammenhang die Option `-r`, die während des Übersetzungslaufes die fehlgeschlagenen Versuche des Compilers ausgibt, Includedateien zu lokalisieren.

Die Option `-Q` veranlasst den Compiler, die gerade gelesene Quelltextzeile auf dem Schirm auszugeben. Mit `-Q=hexdigit` lässt sich die Tiefe der auszugebenden Includeschachtelung bestimmen. Bei `-Q=0` wird nur noch der Quelltext der Hauptdatei angezeigt. Die Anzeige der includierten Dateien unterbleibt. Will man auch noch den ersten Includelevel betrachten, so lässt sich dies durch `-Q=1` erreichen, etc...

Mit der Option `-H=path` lassen sich zusätzliche Includepfade angeben. Es sind bis zu 16 zusätzliche Pfade möglich, die entsprechend der Reihenfolge der Definition abgearbeitet werden. Wenn in den zusätzlichen Pfaden kein Treffer erzielt wurde, so wird zuletzt der Standardpfad `H/` bzw. `h/` nach der angegebenen Includedatei durchsucht.

Wenn Sie oft fremde Quelltexte portieren, kann es zuweilen lästig werden, stets die Schreibweise der Includedateien an die gewünschte Gross– oder Kleinschreibung des Compilers anzupassen. Mit der `-J=0`–Option können Quelltexte, die Includedateien in der Form `#include <stdlib.h>` vereinbart haben, trotzdem erfolgreich auf eine Datei `STDLIB.H` zugreifen. Umgekehrt findet der Compiler unter Verwendung der Option `-J=1` bei Angabe des Dateinamens in der Form `#include <STDLIB.H>` auch die kleingeschriebene Variante der Includedatei `stdlib.h`.

Option	Wirkung
–J=0	File in Grossschreibung suchen
–J=1	File in Kleinschreibung suchen
–J=2	File in Gross– und Kleinschreibung suchen

Tabelle 3.7: Suche nach Include–Dateien

Die `-J`–Option greift erst, wenn die Datei nicht in der Originalschreibweise gefunden wurde. Die Suche nach alternativen Schreibweisen erfolgt stets nach der Umwandlung des *kompletten* Suchpfades in Gross– bzw. Kleinbuchstaben. Bei der Angabe der Option `-J=2` wird zunächst die Umwandlung in Grossbuchstaben durchgeführt. Erst danach wird nach der Datei in kleiner Schreibweise gesucht.

Die Option `-J=digit` ist für nachfolgende Compilerversionen abgekündigt!

3.1.9.2 Einmaliges Includieren

Bei der Erstellung von Quelltexten tritt oftmals das Problem auf, dass bestimmte Headerdateien zwingend Vereinbarungen aus anderen Headerdateien benötigen, also selbst `#include`–Anweisungen enthalten. Um Rekursionen oder mehrfaches Includieren von Headerfiles zu vermeiden, wird in der Regel innerhalb der Includedateien eine Konstruktion der folgenden Art verwendet:

```
#ifndef __NAME_DER_DATEI_H__
#define __NAME_DER_DATEI_H__
/*
```

```
 *    Inhalt der Datei
 */
#endif
```

Diese Konstruktion besitzt jedoch den Nachteil, dass der Compiler diese Datei trotzdem öffnen und parsen muss, wenn diese zum wiederholten Male includiert werden soll. Der CCC besitzt deshalb die Möglichkeit, mittels des Kommandos `#pragma INCLUDE_ONCE` beim ersten Einlesen einer Includedatei diese entsprechend zu markieren. Findet der Compiler diese Steueranweisung innerhalb einer Includedatei, so erweitert er intern eine Includeliste um deren *kompletten* Namen; dabei werden auch enthaltene Links aufgelöst.

```
#ifdef __CRESTC__
#pragma INCLUDE_ONCE
#endif
```

Taucht der Dateiname abermals innerhalb einer `#include`–Anweisung auf, so wird dieser Lesebefehl nicht nochmals ausgeführt. Bei aktiver Compileroption `-V` lässt sich an Hand der Ausgaben des CCC erkennen, welche Dateien gelesen (`*** INCLUDE datei.c`) bzw. ignoriert `*** SKIP datei.c` werden.

3.1.9.3 Ausgabe von Datei–Abhängigkeiten

Die Option `-G=digit` ist bei der Arbeit mit dem CMAKE–Tool interessant, das im Kapitel 10 beschrieben wird. Um die Abhängigkeiten von Includedateien und C–Dateien verwalten zu können, ist der CCC in der Lage, die includierten Files in einem gesonderten File, dessen Name sich aus dem Namen der Eingabedatei und der Extension `.dpc` ergibt, abzuspeichern. Im Normalfall werden keine Dependency–Dateien (`-G=0`) erzeugt.

Option	Wirkung
–G=0	kein `dpc`–File erzeugen
–G=1	nur Basis– und Zielfile
–G=2	zusätzlich “include “–Dateien
–G=3	zusätzlich <include>–Dateien
–G=4	nur Liste der Includefiles erzeugen

Tabelle 3.8: Erzeugung von Dependency–Files

3.1.10 Anschalten der Stacküberwachung

Mit der Option `-U` wird eine Stacküberwachung fest in den generierten Maschinencode eingebaut. Es erfolgt dabei eine Überprüfung, ob beim Eintritt in die jeweilige Funktion der aktuelle Stackpointer in den Sicherheitsbereich (256 Bytes vom physikalischen Stackende) der verfügbaren Stacks eingedrungen ist oder gar den verfügbaren Bereich schon verlassen hat. In diesem Fall erfolgt eine sofortige Unterbrechung der verursachenden Task.

```
Stack_overflow_(SUSPENDED)
```

Bei einer Fortsetzung dieser Task mittels `CONTINUE` terminiert diese sich unverzüglich. Sie sollten dabei immer im Hinterkopf haben, dass sich zuvor schon ein `GAU` ereignet hat und die Task nur gestoppt wurde, weil sie entweder schon schlimme Verwüstungen angerichtet hat oder im nächsten Augenblick eine Breitseite in Ihre Variablen oder gar auf die Systemverzeigerungen abgefeuert hätte. Es ist **immer**

angebracht, nach einem solchen Vorfall alle offenen Dateien zu schliessen und RTOS–UH neu zu starten.

Was es mit dem Stack genau auf sich hat, wird im Abschnitt 3.7 explizit beschrieben.

3.1.11 Generierung von C–Zeilennummern im Assemblercode

Die Option `-L` führt dazu, dass der Compiler zu jedem Label in der Assemblerausgabe die Zeilennummer des C–Quelltextes angibt. Diese Option gestaltet die Assembler– bzw. Listingdateien lesbarer.

3.1.12 Generierung von C–Zeilennummern im Programmcode

RTOS–UH bietet die Möglichkeit, Tasks auf Ebene von Zeilennummern zu tracen. Zu diesem Zweck hält jede Task eine 16–Bit–Zelle mit der aktuellen Zeilennummer in ihrem Taskworkspace. CREST–C bietet dem Anwender zwei unterschiedliche Möglichkeiten, die Line–Zelle mit der jeweiligen Zeilennummer des C–Quelltextes zu versorgen.

Die erste Methode besteht in der Verwendung des dafür vorgesehenen Systemtraps `.LITRA`. Die Sache hat allerdings einen bösen Haken, da dieser Trap eine ganze Anzahl von Prozessorregistern verändert und somit ein aufwendiges Retten und Restaurieren dieser Register unumgänglich ist. Dieser Vorgang kostet deutlich Rechenzeit und kann — je nach Anforderung an die Echtzeitfähigkeit — indiskutabel sein.

Option	Line–Zelle
–Y=0	keine Aktion
–Y=1	über `MOVE`
–Y=2	über `.LITRA`

Tabelle 3.9: Besetzen der Line–Zelle

Die zweite Methode kann verwendet werden, wenn Sie nicht beabsichtigen, auf Zeilennummern zu tracen. In diesem Fall trägt der CCC ohne Verwendung des Systemaufrufes die Zeilennummer selbstständig ein.

Folgendes ist zu beachten:

- Sie können nur auf Zeilennummern tracen, für die der CCC auch `.LITRA`–Traps in die Assemblerausgabe geschrieben hat. Das klingt zunächst trivial, ist es im praktischen Umgang aber nicht, da der CCC keineswegs für jede Zeilennummer eine Markierung setzt.
- Aus Sicht des RTOS–UH ist ein erzeugtes CREST–C–Programm ein einziges Modul. Wenn mehrere Dateien zusammengelinkt werden, die mit `-Y` übersetzt wurden, kann (und wird) es zu mehrdeutigen Zeilennummern aus den unterschiedlichen Quelltextdateien kommen. Es bietet sich deshalb an, nur jeweils eine *verdächtige* Datei mit `-Y` zu compilieren.
- Die Line–Zelle wird vom `DL`–Kommando der Shell etwas arg seltsam ausgegeben. Um `lesbare dezimale` Angaben zu erhalten, muss die Zeilennummer in BCD–Darstellung angegeben werden — d.h. dass die Zeilennummer `137` als `$0137` in der Linezelle eingetragen ist. Damit beschränkt sich der Wertebereich auf maximal 9999 Zeilen.

An folgendem Beispielprogramm in Abbildung 3.2 wird das Verhalten des CCC bei der Vergabe von Zeilennummern demonstriert.

Die Pfeile hinter den Zeilennummern markieren die vom CCC erzeugten Line–Markierungen. Die Vergabe der Zeilennummern orientiert sich an den erkannten *expression*'s bzw. *statement*'s im Quelltext.

```
/*  1     */    short  a, b, c, d, e, f, g ;
/*  2     */
/*  3     */    void test( void )
/*  4     */    {
/*  5 --> */       a = ( b+c*d )
/*  6     */         - ( e*f/g )
/*  7     */         * ( b-c*e ) ;
/*  8     */
/*  9 --> */       b = a * c ;
/* 10 --> */    }
```

Abbildung 3.2: Beispielprogramm für die Y–Option

Für jede *expression* wird exakt eine Zeilennummer gespeichert — auch wenn diese über mehrere Zeilen hinwegreicht. Bei der mitgeführten Nummer handelt es sich um die Zeile, in der die *expression* begonnen hat. Die Zeilenmarkierung wird im Code immer vor dem Maschinencode eingebaut, der zu der entsprechenden *expression* erzeugt wird.

Die Abbildung 3.3 zeigt den erzeugten Code bei Verwendung der `-Y`–Option.

Achtung: Benutzen Sie **nie** die `-Y`–Option aus Programmcode heraus, bei dem `A4` nicht korrekt gesetzt ist! So besitzen z.B. weder Interruptroutinen, Kalt– noch Warmstartscheiben einen Taskworkspace und folglich auch kein gültiges `A4`. Der Compiler unterdrückt zwar selbstständig innerhalb solcher *Sonderfunktionen* die Generierung des Linetracer–Codes. Dies gilt jedoch nicht für Funktionen, die aus einem derartigen Rumpf heraus aufgerufen werden. Findet z.B. innerhalb der Interruptroutine `Interrupt()` der Funktionsaufruf `TesteBitteDieHardware()` statt und enthält eben diese Funktion Linetracer–Code, dann geht der Rechner nach relativ kurzer Zeit in die ewigen Jagdgründe ein. Der Compiler hat wenig Chancen, Konstellationen dieser Art abzufangen. Hier ist Umsicht auf der Programmiererseite verlangt...

Option	Code
`-Y=0`	
`-Y=1`	`MOVE.W  #$0007,(_line_cell_.W,A4)`

`-Y=2`	`MOVEM.L  D1/D6/D7/A1,-(SP)`
	`.LITRA`
	`.DC.W  7`
	`MOVEM.L  (SP)+,D1/D6/D7/A1`

Abbildung 3.3: Erzeugter Code bei `-Y`–Option für Zeile 7

Die Line–Zelle lässt sich auch in eigenen Programmen setzen. Derartige Aktionen machen dort Sinn, wo man z.B. den Benutzer mit kleinen Informationen über den Fortschritt oder Status eines Programmes bei Laune halten möchte. Die Funktionen `rt_set_LINENO()` und `rt_get_LINENO()` erwarten 16–Bit–Eingabewerte. Wenn Sie die Zeilennummer in BCD–Kodierung übergeben, ist die Ausgabe später dezimal ablesbar — soll heissen: `rt_set_LINENO( 0x1234 )` liefert später die Ausgabe `1234`.

```
void    rt_set_LINENO( LineNo  line ) ;
LineNo  rt_get_LINENO( void ) ;
```

3.1.13 Assemblercode in C–Quelltexten

In Hinsicht auf etwas mehr Bedienerfreundlichkeit bei der Erstellung von kleinen Einschüben in Assemblersprache, wurde der CCC um die Kommandos `#asm` und `#endasm` erweitert. Das Kommando `#asm` bewirkt, dass der CCC alle nachfolgenden Zeilen bis zu abschliessenden `#endasm` direkt in sei-

nen Assembleroutput übernimmt. Da der C–Präprozessor auch innerhalb dieser Einschübe noch aktiv ist — zumindest, was die Interpretation von Kommandozeilen betrifft —, sind so bedingte Assemblierung und `#include`-Anweisungen innerhalb des Assemblercodes möglich. Um kein Missverständniss aufkommen zu lassen: es handelt sich bei diesem Feature **nicht** um einen Inline–Assembler. Die Verwendung der Kommandos ist nur ausserhalb von C–Funktionen erlaubt und kann folglich nur verwendet werden, um **komplette** Funktionen in Assembler zu kodieren. Einschübe innerhalb des C–Quelltextes sind weder möglich noch geplant.

3.1.14 Ausgabe einer Assemblerdatei

Der CCC ermöglicht mittels der Option `-s` die Ausgabe des erzeugten Assemblercodes. Der Name dieser Datei ergibt sich aus dem Namen der C–Datei durch Ersetzung der Extension mit der Endung `.s`. Wenn die Option `-s` selektiert wurde, so erscheint der Name der Ausgabedatei in der Abschlussmeldung des CCC.

3.1.15 Ausgabe eines Assemblerlistings

Der CCC ermöglicht mittels der Option `-x` die Ausgabe eines Listfiles. Der Name dieser Datei ergibt sich aus dem Namen der C–Datei durch Ersetzung der Extension mit der Endung `.lst`. Wenn die Option `-x` selektiert wurde, so erscheint der Name der Ausgabedatei in der Abschlussmeldung des CCC.

3.1.16 Unterdrückung von BRA–Optimierungen

Der Assembler übernimmt einige Aufgaben des CCC, wenn es um die Verbesserung des Zielcodes geht. Da der Codegenerator des CCC bei der Ausgabe lediglich Textverarbeitung betreibt, bleibt es dem nachgeschalteten (internen) Assembler überlassen, alle relativen Sprünge innerhalb einer Eingabedatei auf möglichst kurze Adressierungsarten zurückzuführen. Dabei werden die Extensions `.B`, `.W` und `.L` hinter den Sprunganweisungen `Bcc` im Quelltext schlicht ignoriert und durch *optimal kurze* Sprungbefehle ersetzt — was auch bedeuten kann, dass redundante Sprunganweisungen wegoptimiert werden können.

Mit der Option `-t` können diese Eigenmächtigkeiten des Assemblers bei Bedarf unterdrückt werden. Im Schnitt bringt die standardmässige Optimierung jedoch in durchschnittlichen C–Quelltexten etwa 10%–Codeeinsparungen und auch Laufzeitgewinne in dieser Grössenordnung und sollte nicht ohne besonderen Grund abgeschaltet werden.

3.1.17 Unterdrückung von MOVEM–Optimierungen

Der interne Assembler versucht, die vom Compiler generierten `MOVEM`–Befehle zum Retten und Restaurieren der Register bei Unterprogrammaufrufen zu verbessern. Es werden die `MOVEM`–Kommandos komplett gestrichen, wenn die angegebene symbolische Registerliste leer ist. Bei Operationen mit bis zu zwei Registern wandelt der Assembler den `MOVEM`–Befehl in ein oder zwei einzelne `MOVE`'s um, was nach Aussage von Taktzyklentabellen noch minimale Laufzeitgewinne bringt und beim Retten eines Einzelregisters auch noch jeweils ein Wort im Code einspart.

Da bei `MOVE`'s mit Datenregistern das Statusregister modifiziert wird, lässt sich die Optimierung des Assembler mit der `-m`–Option abschalten. Für vom CCC erzeugte Assemblerdateien besteht in dieser Hinsicht keinerlei Gefahr, da der CCC das Statusregister nach einem Funktionsaufruf stets als undefiniert betrachtet und bei Bedarf selbst Testbefehle für das Funktionsergebnis generiert, die das Statusregister der CPU in einen definierten Zustand versetzen.

3.1.18 Unterdrückung von 68K–Optimierungen

Die Resultate der Optimierungsmassnahmen von Compilern decken sich keineswegs immer mit den Absichten des Programmierers. Gerade beim Zugriff auf Peripheriebausteine ergibt sich oftmals eine gewisse Diskrepanz zwischen *optimiertem* und *funktionsfähigen* Code.

3.1.18.1 Der `CLR`–Befehl bei 68K–CPU's

Die `-N`–Option wurde implementiert, um einer gewöhnungsbedürftigen Eigenschaft mancher Motorola–68K–CPU´s zu begegnen. Der `CLR`–Befehl eignet sich auf Grund kompakteren Maschinencodes besser zum Löschen vom Speicher, als ein korrespondierender `MOVE`–Befehl mit Null. Der Codegenerator des CCC verwendet deshalb standardmässig den `CLR`–Befehl bei Schreiboperationen mit Nullmustern.

Seltsamerweise führt der `CLR`–Befehl vor der Schreiboperation einen Lesezyklus aus. Aus diesem Grunde eignet er sich nicht für dem Zugriff auf I/O-Bereiche, die als *write–only*–Register ausgelegt sind oder bei denen Leseoperationen interne Statusänderungen auslösen.

Mittels der Compileroption `-N` wird der CCC angewiesen, bei Zugriffen über Pointer oder auf absolute Speicherpositionen das `CLR`–Kommando zu vermeiden. Die Option gilt wie üblich für die gesamte Datei. Um auch lokal innerhalb der Datei Einfluss auf das Verhalten des Compilers nehmen zu können, wurden zwei `#pragma`–Kommandos implementiert.

```
#pragma DISABLE_CLEAR_MNEMONICS
#pragma ENABLE_CLEAR_MNEMONICS
```

Mittels `#pragma DISABLE_CLEAR_MNEMONICS` wird für jede nachfolgende Funktion innerhalb der Datei bei der beschriebenen Klasse von Zugriffen der `CLR`–Befehl unterdrückt. `#pragma ENABLE_CLEAR_MNEMONICS` stellt das übliche Verhalten des Compilers für den Rest der Datei wieder her und übersteuert auch die Compileroption `-N`.

3.1.18.2 68K–Bitzugriffe auf I/O–Bereiche

Die Motorola–68K–CPU's besitzen die Fähigkeit, Operationen, die sich auf Einzelbits auswirken, mit speziellen Befehlen auszuführen. So ist z.B. das Anschalten eines einzelnen Bits innerhalb einer 16–Bit–Speicherzelle sowohl mittels der CPU–Anweisung `ANDI.W` als auch mittels des speziellen Befehl `BSET`. Da der Befehl `BSET` allerdings nicht in der Lage ist, mit 16–Bit–Werten zu hantieren, sondern nur auf 8–Bit–Werte im Speicher bzw. auf 32–Bit–Werte innerhalb von Datenregistern arbeiten kann, sind Zugriffe auf Hardwareadressen, die explizit nur Wort– oder Langzugriffe ermöglichen, zum Untergang verdammt.

Die Option `-n` wurde im Compiler implementiert, um unerwünschte Optimierungen von Bitoperationen verhindern. Es entstehen keine `BTST`, `BSET`, `BCLR` oder `BCHG`–Befehle mehr im Assembleroutput. Die Unterdrückung dieser Optimierung ist erforderlich, wenn der wort– oder langwortweise Zugriff auf den Speicher zwingend erforderlich ist, da die optimierten Befehle nur byteweise auf den Speicher langen und so in der Regel das beabsichtigte Ziel verfehlen.

Entsprechend der Funktionsweise der `#pragma`–Kommandos `ENABLE_CLEAR_MNEMONICS` bzw. `DISABLE_CLEAR_MNEMONICS` wurden zur Unterstützung dieser Funktionalität die Anweisungen

```
#pragma DISABLE_BIT_MNEMONICS
#pragma ENABLE_BIT_MNEMONICS
```

implementiert. Folge dieser Option ist geringfügig ineffizienterer Code, der jedoch den deutlichen Vorteil besitzt, die im Quelltext angegebenen Zugriffsbreiten auf externen Speicher nicht mehr zu verändern.

3.1.19 Globale Registervariablen

Die ANSI–C–Norm untersagt den Gebrauch des Schlüsselwortes `register` ausserhalb von Funktionen. Mittels der `-Z`–Option wird der CCC in die Lage versetzt, globale Registervariablen zu verwenden.

Die Option wurde nur geschaffen, um bei kleinen Programmen die Geschwindigkeit zu erhöhen. Generell ist von der Verwendung dieser Option dringend abzuraten! Bei fehlerhafter Verwendung globaler Registervariablen stürzt das System mit Sicherheit ab.

Der CCC interpretiert das Schlüsselwort `register` ausserhalb von Funktionen als den Wunsch des Programmieres, den Gültigkeitsbereich auf die Hauptdatei zu beschränken, in der die Definition auftrat. Externe Referenzen auf derartige globale Register sind nicht möglich. Wenn in mehreren Modulen globale Registervariablen verwendet werden sollen — wovon dringend abgeraten wird —, so müssen alle Module eines Projektes mit der identischen Deklarationen ausgestattet werden. Sie sollten dabei auch bedenken, dass einige Bibliotheksroutinen Ärger machen werden, die nichts davon wissen, dass der Gebrauch bestimmter Register unzulässig ist. Diese Einschränkung bezieht sich auf `call-back`–Bibliotheksroutinen wie `qsort()`, die in Nutzerroutinen zurückspringen und deshalb prinzipbedingt nicht funktionieren können.

3.1.20 Fehlerbehandlung

Es sind vier Klassen von Meldungen zu unterscheiden:

- **WARNING**: Warnungen führen lediglich zu einer Meldung auf dem Bildschirm und unterbrechen den Übersetzungsvorgang nicht.
- **ERROR**: Fehlermeldungen dieser Kategorie führen zu einem Abbruch der Codegenerierung. Der Compiler bearbeitet zwar noch den Rest des Quelltextes, versucht aber nicht, die folgenden Funktionen zu optimieren oder Code zu generieren. Nach Abschluss des Übersetzungsvorgangs wird die `.obj`–Datei automatisch gelöscht.
- **FATAL**: Fatale Fehler dieser Art betreffen Betriebszustände, bei denen der Compiler entweder physikalisch nicht in der Lage ist, eine sinnvolle Fortführung der Übersetzung zu betreiben — kein Speicherplatz im RAM oder auf der Platte — oder er in einem Zustand erwischt wurde, wo mir ein erneutes Neuaufsetzen der Compilation als unsinnig erschien.
- **COMPILER–FATAL**: Diese Gruppe von Fehlermeldungen sollten Sie eigentlich nicht so häufig zu sehen bekommen, denn sie deuten darauf hin, dass der CCC sich in einem Zustand befindet, den **ich** nicht erwartet habe. Wenn Sie dennoch von derartigen Meldungen betroffen sind, dann sollten Sie mir die Quelltexte zuschicken, die dazu geführt haben.

Eine Ausnahme von dieser Regel stellen die `FATAL`–Meldungen dar, die Sie auffordern, die Optionen `-a`, `-d` oder `-f` anzuwenden. Lesen Sie dazu bitte die näheren Einzelheiten im Abschnitt 3.1.21.2 nach.

Die Optionen `-A`, `-B` und `-W` bestimmen das Verhalten des Compilers im Fehlerfalle. Ohne Verwendung dieser Parameter versucht der CCC, die Fehlerzeile auf den Schirm auszugeben und die Position des Fehlers zu markieren. Anschliessend wartet der Übersetzer auf eine Reaktion des Anwenders:

```
(C)ontinue (A)bort (E)dit.
```

Bei Eingabe von A (gegebenenfalls noch ENTER eingeben) terminiert der Compiler augenblicklich, verabschiedet sich mit User break und kehrt mit Fehlerstatus zum Aufrufer zurück. Bei C setzt der Compiler nach der Fehlerposition die Compilation fort. Bei E ruft der Compiler den Editor CED auf und springt die Zeile an, in der der Fehler erkannt wurde. Nach der Rückkehr aus dem Editor terminiert sich der CCC.

In jedem Falle stellt der Compiler nach Feststellung des ersten ERROR's die Codegenerierung ein und löscht nach Beendigung des Programms auch das generierte .obj–File. Bei gravierenden Fehlern kann es auch zu einem sofortigen Programmabbruch ohne die Möglichkeit einer Einflussnahme kommen. Fehlender Speicherplatz und viele Deklarationsfehler führen zu einer FATAL–Meldung und sofortigem Abbruch.

WARNING–Meldungen führen nicht zum Abbruch des Compilers und beeinflussen auch nicht seinen Rückgabestatus an den Aufrufer. Trotzdem sollte man gelegentlich einen Gedanken daran verschwenden, ob es nicht besser wäre, die Quelltexte zu überarbeiten. Normalerweise rauschen WARNING's ohne Halt über den Schirm. Hier greift die -A–Option. Als Vorbesetzung startet der Compiler mit -A=2. Im Batch–Betrieb oder bei Verwendung von CMAKE kann ein solchermassen erzwungener Bedienereingriff höchst unerwünscht sein, denn bis zu einer Bestätigung hängt der Compiler gnadenlos auf Ihrem Terminal und lauert auf Eingaben. Mittels -A=0 gibt der Übersetzer Ruhe und lässt alle Meldungen über den Schirm rollen, ohne den Bediener weiter mit Eingabeaufforderungen zu belästigen. Bei -A=3 wartet er dagegen auch nach WARNING–Meldungen auf die Bestätigung des Bedieners.

Option	Stoppt den Compiler
–A=0	NIE
–A=1	bei FATAL
–A=2	bei FATAL und ERROR
–A=3	bei FATAL, ERROR und WARNING

Tabelle 3.10: Steuerung von Fehlermeldungen

Um die aufgelaufenen Fehlermeldungen nach Abschluss des Compilerlaufes analysieren zu können, wurde die -B–Option eingeführt. Sie erzeugt eine Errordatei, deren Name sich aus dem Namen der Eingabedatei und der Extension .err ergibt. Alle auflaufenden Fehlermeldungen und Warnungen werden in dieser Datei gesammelt. Wird der Compilerlauf ohne Fehler und Warnings beendet, so wird keine Errordatei erzeugt und eine etwaige alte Errordatei automatisch gelöscht.

Die -W=digit–Option dient zum Abschalten der Ausgabe einiger WARNING's. Die WARNING's sind in der aktuellen Implementierung grob nach ihrer Wichtigkeit gestaffelt. Deuten die meisten W=6–Ausgaben auf Unsauberheiten und kleinere Sünden hin, so steigt die Wahrscheinlichkeit, dass ein Programm, das nur mit W=0 ohne Meldungen compiliert werden konnte, Ihnen zur Laufzeit ins Gesicht springt, doch ganz erheblich.

Option	Wirkung
–W=6	Alle Warnings ausgeben
–W=5	Fast alle Warnings ausgeben
–W=4	Haarspaltereien unterdrücken
–W=3	Fehlende Argumentprototypen ignorieren
–W=2	Fehlende Funktionsprototypen ignorieren
–W=1	Wesentliche Warnings ausgeben
–W=0	Alle Warning unterdrücken

Tabelle 3.11: Unterdrückung von Compiler–Warnings

3.1.21 Bekannte Compilerfehler

Hier müssen bedauerlicherweise zwei bekannte Fehler des CCC dokumentiert werden.

3.1.21.1 Fehler bei der Syntaxanalyse von Symbolen

Der CCC besitzt aus historischen Gründen einen etwas unzulänglichen Parser in Bezug auf die korrekte Erkennung von Deklarationen und Expressions — dieser stammt noch aus seiner K&R–Vergangenheit, als die `namespaces` von Typen und Variablen noch nicht getrennt waren. Ein paar Beispiele, wo sich das zu einem Problem entwickeln kann:

1. Der CCC bricht auf der Position `'+='` mit der Fehlermeldung ab, nun sei ein Name oder eine Deklaration zu erwarten.

   Grund: nach dem sich öffnenden Compound `'{'` versucht er zunächst, Deklarationen zu bearbeiten und findet tatsächlich einen Bezeichner `'s'`, der als Typ akzeptiert wird. In diesem Betriebszustand erwartet er nun zwingend (und fehlerhafterweise) eine Variablenvereinbarung.

   ```
   typedef int  s ;
   void f( int  s )
   {
      s += 17 ;
   }
   ```

   Abhilfe: Ein Semikolon, das als leere Expression verstanden wird und den CCC davon überzeugt, dass der Deklarationsteil des Compounds nunmehr beendet ist.

   ```
   typedef int  s ;
   void f( int  s )
   {
      ;
      s += 17 ;
   }
   ```

2. Auch hier wieder ein Fall, wo der CCC sich intern verläuft:

   ```
   typedef struct { int hi ; } s ;
   s *f(s *s)
   {
      return((s*)(s->hi));
   }
   ```

   In der Return-Expression sind zwei unterschiedliche Verwendungen des Bezeichners `'s'` auseinanderzuhalten — was er leider nicht sauber geregelt bekommt. In beiden Fällen geht er bei einer geöffneten Klammer innerhalb einer Expression zunächst auf die Suche nach einer abstrakten Deklaration — oder im Klartext: er schaut nach, ob es sich um einen Cast handelt. Im ersten Fall `'(s*)'` wird er dabei korrekt fündig. Im zweiten Fall `'(s->hi)'` geht er ebenfalls hinter der Klammer davon aus, dass es sich beim Bezeichner `'s'` um einen Datentyp handelt und scheitert beim Auftreten des Strukturselektors an dieser fehlerhaften Beurteilung der Lage.

   Abhilfe: Die Klammern um die Expression weglassen, um den CCC von dem Gedanken abzubringen, dass nun ein Cast folgen könnte.

   ```
   typedef struct { int hi ; } s ;
   ```

```
s *f(s *s)
{
   return((s*)s->hi);
}
```

Da es sich bei dem Quellcode des Compilers, der Deklarationen behandelt, um einen Programmteil handelt, den ich vorsichtig als *nicht–trivial* bezeichnen würde, ist mit einer schnellen Behebung dieses Fehlers z.Z. nicht zu rechnen...

Als Abhilfe ist angeraten, die Namen von Datentypen und Variablen möglichst unterschiedlich zu wählen oder mittels der beiden gerade beschriebenen Verfahren dem CCC bei Zweideutigkeiten in der Syntax auf den rechten Weg zu helfen.

3.1.21.2 Fehlerhafte Registerverteilung

Durch einen konzeptionellen Fehler im CCC kann es unter bestimmten Konstellationen bei der Codegenerierung dazu kommen, dass der Compiler mehr Variablen automatisch als implizite Registervariablen behandelt, als die CPU oder FPU überhaupt besitzt. Um Panikreaktionen vorzubeugen: wenn Sie diese Fehlermeldung nicht sehen, dann sind auch keine Fehler dieser Art aufgetreten! Der CCC generiert auf Grund dieser Macke **definitiv keinen** fehlerhaften Zielcode, sondern bricht mit einer Fehlermeldung ab.

Entgegen der ersten Prognose, dieser Fehler sei mit einer schnellen Überarbeitung des Programmteiles im CCC zu beheben, der sich mit der Verteilung von Prozessor–Registern an die einzelnen Variablen Ihrer Programme beschäftigt, können Sie der Existenz dieses Abschnitts entnehmen, dass ich im `history`–File der `1.756`–Release — als dieser Fehler erstmals bekannt wurde — zu optimistisch war. Die realistische Einschätzung ergab vielmehr, dass rund 200 Kilobytes an Quelltexten des Compilers in die Mülltonne wandern und neukodiert werden müssten — Zeit, die ich einfach nicht habe...

Was vormals als temporäre Abhilfe gedacht war, bleibt auch bei dieser zweiten Major–Release des Compilers die amtliche Handlungsanweisung. Folgende `COMPILER-FATAL`–Meldungen sind denkbar:

- `Register allocation error: Please recompile with -a=number`
  Ursache: Es wurden durch den Compilerfehler zuviele Adressregister vergeben. Der interne Fehler wurde erkannt, aber zu spät, um compilerinterne Reparaturmassnahmen zu ergreifen...

  Abhilfe: Neuübersetzen des Quelltextes mit der Option `-a=number`. `number` legt hierbei die Zahl der Adressregister fest, die nicht bei der Optimierung einbezogen werden dürfen. Sinnvoll sind hier die Werte 0 bis 8. 0 ist der Standardwert und führt in 99,9% aller Fälle zum Erfolg!
- `Register allocation error: Please recompile with -d=number`
  Entsprechend bei Daten–Registern...
- `Register allocation error: Please recompile with -f=number`
  Entsprechend bei Fliesskomma–Registern...

Der erzeugte Code für die gesamte zu übersetzende Datei wird durch die Verwendung dieser Optionen möglicherweise geringfügig ineffizienter. Wenn hochgradig zeitkritische Funktionen in der Datei enthalten sind, die diesen Fehlerabbruch gemeldet hat, bei denen es zudem um jede Mikrosekunde gehen könnte — was wohl nur Kenner des generierten Motorola–Assemblercodes beurteilen können —, so sollten diese Funktionen bei Bedarf in eine separate Datei ausgelagert werden, die dann ohne diese *Reparatur*–Optionen zu übersetzen ist.

Eine weitere Abhilfe besteht darin, den Compiler durch Angabe des Schlüsselwortes `auto` bei der

Definition lokaler Variablen dazu zu zwingen, die Zahl potentieller Registervariablen der fehlerhaft compilierten Funktion zu reduzieren. Da bei Argumenten die Angabe der Speicherklasse `auto` nicht zulässig ist, klappt dieser Trick nur, wenn der CCC nicht bereits bei der Zuteilung dieser Parametern ins Trudeln geraten ist.

3.2 Vordefinierte Makros im ccc

ANSI–C–Compiler bieten die Möglichkeit, mittels vordefinierter Makros Informationen in die Quelltexte zu integrieren, die erst zur Übersetzungszeit bekannt sind. Dazu zählen neben dem Dateinamen und der Zeilennummer auch Datum und Uhrzeit der Übersetzung.

Makro	Bedeutung	Ersatztext
`__FILE__`	Dateiname als String	”test.c”
`__LINE__`	Zeilennummer als Zahl	4
`__DATE__`	Datum als String	”Jul 30 1993”
`__TIME__`	Uhrzeit als String	”16:04:22”
`__STDC__`	ANSI? als Zahl	(1)

Tabelle 3.12: ANSI–Einbau–Makros

Die Einbau–Makros lassen sich nicht übersteuern. Der Versuch, eines dieser Makros mittels `#undef` ungültig zu machen, führt zu einer Fehlermeldung.

Neben diesen Makros sind noch die beiden compilerspezifischen Makros `__CRESTC__` und `__RTOSUH__` vordefiniert. Beide enthalten keinen Ersatztext und lassen sich mittels:

```
#if defined( __CRESTC__ )
   // Teile, die nur mit CREST-C uebersetzbar sind
#endif
#if defined( __RTOSUH__ )
   // Teile, die nur unter RTOS-UH sinnvoll laufen
#endif
```

abfragen. Sie erhalten so die einfache Möglichkeit, compiler– und betriebssystemspezifische Programmteile bedingt zu übersetzen.

Zur Unterscheidung der Prozessorserie, für die CREST–C Code generiert, legt der CCC jeweils eines der beiden Makros `__M68K__` bzw. `__MPPC__` an. Beide enthalten keinen Ersatztext und lassen sich mittels:

```
#if defined( __M68K__ )
   // Teile, die 68K-spezifischen Code enthalten
#endif
#if defined( __MPPC__ )
   // Teile, die PowerPC-spezifischen Code enthalten
#endif
```

Bei der Verwendung der Compileroption `-+` wird zudem das Makro `__CPLUSPLUS__` vordefiniert, um etwaige Kompatibilitätsprobleme zwischen ANSI–C und `C++` abfangen zu können.

Weitere Makros, die über CPU–Optionen automatisch definiert werden, sind im Abschnitt 3.1.2 beschrieben.

3.3 Basisdatentypen

CREST–C kennt alle von ANSI–C aufgeführten Basisdatentypen. Die binäre Darstellung im Speicher sieht wie folgt aus:

Datentyp	Auflösung in Bits	default
`char`	8	unsigned
`short`	16	signed
`int`	32	signed
`long`	32	signed
Pointer	32	
`float`	32	
`double`	64	
`long double`	96 (80)	
`enum`	32	
`bool`	32	

Tabelle 3.13: Binäre Darstellung der Basisdatentypen

Beachten Sie dabei folgende Randbedingungen:

- Bei dem Datentyp `bool` handelt es sich nicht um einen unter ANSI–C genormten Datentyp. Vielmehr liegt hier eine `C++`-Erweiterung vor.
- Der Datentyp `long double` ist für den PowerPC lediglich mit 64 Bit Auflösung implementiert und entspricht dem Datentyp `double`.

3.4 Benutzereigene Datentypen

Es gibt in C vier wesentliche Arten, sich benutzereigene Datentypen zu schaffen.

- Felder
- Strukturen
- Unions
- Bitfelder

Alle diese Datentypen stellen eine nahezu beliebige Zusammenfassung von Basisdatentypen und benutzereigenen Datentypen dar. Die Regeln, die es hier einzuhalten gilt, sind in jedem guten C–Lehrbuch nachzulesen. In den folgenden Abschnitten sollen demnach nur Besonderheiten bei der Implementierung in CREST–C angesprochen werden.

3.4.1 Padding innerhalb von Strukturen

Die meisten CPU's besitzen ein paar unschönen Eigenarten, was den Zugriff auf Code und Daten betrifft. So müssen bei den meisten 16–bittigen CPU's die Maschinenbefehler auf 16–Bit–Grenzen liegen. Bei 32–Bit–Maschinen in einigen Fällen auf 32–Bit–Grenzen und so fort. Hält man (bzw. der Compiler) sich nicht an diese Regeln, so ist im positivsten Falle mit leichten bis gravierenden Laufzeiteinbussen zu rechnen. In der Regel führt die falsche Ablage von Code zu bösen Fehlermeldungen des Betriebssystems, das die Abneigung der CPU wenigstens noch dokumentiert.

Beim Zugriff auf Datenbereiche sind die Beschränkungen in der Regel nicht gar so restriktiv wie bei dem Versuch, Code, der *misaligned* ist — sich also nicht auf den maschinenspezifischen Grenzen befindet — zu exekutieren. Dennoch gelten auch hier Regeln, an die sich der Compiler halten muss oder sollte. Für Sie als Programmierer ist es nur dann wichtig, diese Regeln zu kennen, wenn es darum geht, innerhalb von Programmen mit externen Datenstrukturen umgehen zu müssen. Die Abbildung der Datenstrukturen auf den realen Speicher ist **nicht** genormt und hängt von den Vorgaben der Maschine ab, für die der Compiler Code erzeugen soll.

Unter dem Begriff des `Padding` versteht man das automatische Auffüllen von Datenstrukturen mit Leerbytes, um die Anwenderdaten auf Adresslagen zu zwingen, die die Maschine, auf der die Daten verarbeitet werden sollen, vorschreibt. So ist z.B. die CPU `MC68000` nicht in der Lage, 16– bzw. 32–Bit–Daten auf ungeraden Adressen zu lesen oder zu schreiben. Ein 68K–Compiler trägt dem dadurch Rechnung, indem er alle Wort– und Langwort–Daten automatisch auf der nächsten geraden Adresse ablegt. Für den Anwender, der etwas wie `long a = 3 ;` in sein Programm geschrieben hat, verhält sich dieses automatische Padding völlig transparent. Wo und wie der Compiler den Platz für die 32–Bit–Variable `a` allokiert, ist ausschliesslich Sache des Übersetzers.

Interessant wird die Angelegenheit erst bei Datenstrukturen, die Hardware, Kommunikations–Protokolle oder Datensätze beschreiben sollen, die einen vorgegebenen Aufbau besitzen. Es ist oft zu unhandlich, eine Struktur als sequentielle Abfolge von Basisdatentypen zu betrachten und deshalb lohnt sich zumeist der Aufwand, eine Datenstruktur für die Zielmaschine und den zugehörigen Compiler zu stricken, deren Speicherrepräsentation den Vorgaben entspricht.

3.4.1.1 Memberpadding 68K

Die `MC68000`–CPU und `CPU32`–Familie wünscht bei Wort– und Langwortzugriffen im Speicher die Daten auf geraden Adressen. Ab MC68020 ist das kein Zwang mehr, verhilft dem armen Prozessor aber zu deutlich verbessertem Datendurchsatz. CREST–C legt automatisch alle Daten, auf die wort– oder langwortweise zugegriffen werden muss (oder sollte), auf der nächsten geraden Adresse ab, führt also wortweises (16-bittiges) Padding durch. Bei der Definition von Strukturen sollten Sie das stets im Hinterkopf behalten.

```
struct test {
   char   a ;  // Offset   0
   long   b ;  // Offset   2
   char   c ;  // Offset   6
   long   d ;  // Offset   8
   char   e ;  // Offset  12
   long   f ;  // Offset  14
   char   g ;  // Offset  18  } ;
```

Abbildung 3.4: Beispiel für 68K–Padding in Strukturen

Es wurde jedoch eine Möglichkeit vorgesehen, dieses implizite 16–Bit–Padding zu übersteuern. Mittels des `#pragma`–Kommandos `#pragma MEMBER_PADDING_OFF` wird das Padding von Strukturmitgliedern ausgeschaltet. Beim Einsatz dieses Kommandos sollten Sie Vorsicht walten lassen, weil nunmehr auch die Definition von Strukturen möglich ist, die auf MC68000er–Prozessoren zu übler Verwirrung und fehlerhaften Datenzugriffen führen. Das Pendant zu diesem Kommando stellt der Befehl `#pragma MEMBER_PADDING_ON` dar, der das Default-Verhalten der CPU restauriert. Auf 68K–Compilern ist dazu das Kommando `#pragma MEMBER_PADDING_68K` äquivalent zu verwenden, Die Kommandos wirken global auf alle Strukturen, die im weiteren Verlauf der Übersetzungseinheit vereinbart werden. Das Padding der Gesamtstrukturgrösse wird davon jedoch nicht berührt und ist grundsätzlich ein gerader Wert.

```
typedef struct Test1 {
   char  a ; //  0
   short b ; //  2
   char  c ; //  4
   int   d ; //  6
   char  e ; // 10 }  Test1 ; // sizeof( Test1 ) = 12

#pragma MEMBER_PADDING_OFF
typedef struct Test2 {
   char  a ; // 0
   short b ; // 1
   char  c ; // 3
   int   d ; // 4
   char  e ; // 8 }  Test2 ; // sizeof( Test1 ) = 10
#pragma MEMBER_PADDING_ON
```

CREST–C legt die Komponenten einer Struktur stets in der Reihenfolge der Definition ab und versucht nicht, den Speicherbedarf einer Struktur zu optimieren. Die Struktur in Abbildung 3.4 belegt 20 Bytes im Speicher, obwohl die Summe der einzelnen Komponenten nur 16 Bytes beträgt. Bei der Anpassung eigener Strukturen an vorgegebene Datentypen müssen Sie folgendes beachten:

- Alle *nicht–char–Daten* liegen auf geraden Adressen. Wenn Sie nicht an einen bestimmten Aufbau gebunden sind, können Sie durch Umstellung der Komponenten innerhalb solcher Strukturen oft erheblich Speicherplatz sparen.
- Strukturen liegen immer auf 16–Bit–Grenzen
- Felder liegen immer liegen immer auf 16–Bit–Grenzen, sofern der Basisdatentyp nicht 8–Bit breit ist. In diesem Sonderfall werden die Daten auf 8–Bit–Grenzen gelegt.
- Strukturen, Unions und Arrays werden durch Padd–Bytes stets auf gerade Längen aufgefüllt. Der Compiler nutzt dieses Wissen, um bei Zuweisungen von Struktur an Struktur oder von Union an Union mit Wort– oder Langwortbefehlen zu arbeiten. Bei der Programmierung des MC68000 bzw. der `CPU32` müssen Sie also auf `memcpy()` umsteigen, wenn Sie beabsichten, Strukturen/Unions von/zu ungeraden Adressen zu transferieren.

Sie sollten sich also ganz schnell angewöhnen, die Berechnung der Grössen benutzereigener Objekte dem Compiler zu überlassen. Der `sizeof`–Operator ist die **einzige** amtliche Methode, bei solchen Aktionen portabel zu bleiben. Für die Bestimmung des Offsets einer Struktur–Komponente gibt es das `offsetof`–Makro in <`stddef.h`>.

3.4.1.2 Memberpadding PPC

Für den PowerPC existieren im Vergleich zur 68K–Familie leicht abweichende Regeln bezüglich des Paddings innerhalb von Strukturen. Innerhalb der PowerPC–Familie exitiert ein 32– und ein 64–Bit-Zweig. Da bislang nur RTOS–UH–Implementierungen für Vertreter der 32–Bit–Gattung vorliegen (601, 603, 604, etc...), beziehen sich alle weiteren Ausführungen ausschliesslich auf diese CPU's.

Für den den auszuführenden Code gilt die einfache Regel: alles auf 32–Bit–Grenzen ablegen weil die CPU sonst ernsthaft maulig ist! Bei den Datenzugriffen ist die Lage entspannter. Intern rechnen die PowerPC's grundsätzlich mit 32–Bit–Registern. Lediglich wenige Maschinenbefehle sind überhaupt in der Lage, mit anderen Datenformaten umzugehen. Dazu gehört selbstredend ein Satz von Befehlen, der den Transfer von Daten vom und in den Hauptspeicher ermöglicht. Hier sind 8–, 16– und 32–Bit–Transfers möglich. Beim Datenzugriff gilt die einfache Regel, dass ein Zugriff beliebiger Breite auf

beliebigen Adressen möglich ist. Entspricht die Adresse jedoch keinem Vielfachen des zu lesenden oder zu schreibenden Datums, so wird man mit schlechten bis lausigen Zugriffszeiten bestraft. Der PPC–Compiler legt folglich — in Abweichung zu seinem 68K–Kollegen — alle 32–Bit–Variablen tunlichst auf Langwortgrenzen, um die Performance nicht in den Keller gleiten zu lassen.

```
struct test {
   char   a ;  // Offset   0
   long   b ;  // Offset   4
   char   c ;  // Offset   8
   long   d ;  // Offset  12
   char   e ;  // Offset  16
   long   f ;  // Offset  20
   char   g ;  // Offset  24 } ;
```

Abbildung 3.5: Beispiel für PPC–Padding in Strukturen

Es wurde jedoch eine Möglichkeit vorgesehen, dieses implizite 32/16–Bit–Padding zu übersteuern. Mittels des `#pragma`–Kommandos `#pragma MEMBER_PADDING_OFF` wird das Padding von Strukturmitgliedern ausgeschaltet. Das Pendant zu diesem Kommando stellt der Befehl `#pragma MEMBER_PADDING_ON` dar, der das Default-Verhalten der CPU restauriert. Auf PPC–Compilern ist dazu das Kommando `#pragma MEMBER_PADDING_PPC` äquivalent zu verwenden, Die Kommandos wirken global auf alle Strukturen, die im weiteren Verlauf der Übersetzungseinheit vereinbart werden. Das Padding der Gesamtstrukturgrösse wird davon jedoch nicht berührt und ist grundsätzlich ein durch vier teilbarer Wert.

```
typedef struct Test1 {
   char  a ; //  0
   short b ; //  2
   char  c ; //  4
   int   d ; //  8
   char  e ; // 12 }  Test1 ; // sizeof( Test1 ) = 16

#pragma MEMBER_PADDING_OFF
typedef struct Test2 {
   char  a ; // 0
   short b ; // 1
   char  c ; // 3
   int   d ; // 4
   char  e ; // 8 }  Test2 ; // sizeof( Test1 ) = 10
#pragma MEMBER_PADDING_ON
```

CREST–C legt die Komponenten einer Struktur stets in der Reihenfolge der Definition ab und versucht nicht, den Speicherbedarf einer Struktur zu optimieren. Die Struktur in Abbildung 3.5 belegt 28 Bytes im Speicher, obwohl die Summe der einzelnen Komponenten nur 16 Bytes beträgt. Bei der Anpassung eigener Strukturen an vorgegebene Datentypen müssen Sie folgendes beachten:

- Alle 8–Bit–Daten werden nicht gepadded.
- Alle 16–Bit–Daten werden auf 16–Bit–Adressen abgelegt.
- Alle 32–Bit–Daten werden auf 32–Bit–Adressen abgelegt.
- Strukturen liegen immer auf 32–Bit–Grenzen
- Felder liegen immer liegen immer auf 32–Bit–Grenzen, sofern der Basisdatentyp nicht 8– oder 16–Bit breit ist. In diesen Sonderfällen werden die Daten auf 8– bzw. 16–Bit–Grenzen gelegt.

- Wenn Sie nicht an einen bestimmten Aufbau gebunden sind, können Sie durch Umstellung der Komponenten innerhalb solcher Strukturen oft erheblich Speicherplatz sparen.
- Strukturen, Unions und Arrays werden durch Padd–Bytes stets auf eine durch vier teilbare Länge aufgefüllt. Der Compiler nutzt dieses Wissen, um bei Zuweisungen von Struktur an Struktur oder von Union an Union mit Wort– oder Langwortbefehlen zu arbeiten.

3.4.1.3 Strukturzuweisungen

Die bereits angesprochene Einschränkung, Struktur– oder Unionzuweisungen nicht von/zu ungeraden Adressen durchführen zu können, lässt sich bei Bedarf ebenfalls übersteuern. Der CCC besitzt `#pragma`–Kommandos, um auf unterschiedlichste Art mit Strukturen umgehen zu können. Normalerweise findet das Umkopieren von Tags (Strukturen und Unions) stets langwortweise statt — bis auf das eventuell abschliessende Wort. Bis zu einer Grösse von 20 Bytes findet das Umkopieren durch explizite `MOVE.L` bzw. `MOVE.W`–Befehle im Code statt. Bei grösseren Strukturen werden `DBF`–Schleifen generiert, um zum Kopieren nicht kilobyteweise Code zu generieren.

```
struct dummy { int x[ 10 ] ; double y[ 20 ] ; } ;
#pragma TAG_COPY_BYTE
void b( struct dummy a, b ; a = b ; } // b byteweise kopieren
#pragma TAG_COPY_WORD
void w( struct dummy a, b ; a = b ; } // b wortweise kopieren
#pragma TAG_COPY_LONG
void l( struct dummy a, b ; a = b ; } // b langwortweise kopieren
```

Mittels der `#pragma`–Kommandos `TAG_COPY_BYTE`, `TAG_COPY_WORD` und `TAG_COPY_LONG` kann der CCC angewiesen werden, byte–, wort– oder langwortweise zu kopieren. Damit lässt sich auch von/zu ungeraden Adressen kopieren (bei `TAG_COPY_BYTE`) beziehungsweise direkt von/auf Peripherie lesen/schreiben, die nur byte– oder wortweise Zugriffe erlaubt. Default–Einstellung bleibt weiterhin `TAG_COPY_LONG` — soll heissen: die schnellste und kürzeste Methode!

Auch der Schwellenwert, bei dem der CCC beim Kopieren auf Schleifengenerierung umstellt, lässt sich mittels einer `#pragma`–Anweisung steuern. Die Angabe

```
#pragma TAG_COPY_SIZE 100
```

bewirkt, dass je nach eingestellter Kopierbreite bei Strukturen kleiner/gleich 100 Byte Grösse mit einzelnen `MOVE`–Anweisungen gearbeitet wird. Erst ab einer Groesse von 102 Bytes werden Schleifen erzeugt. Default–Einstellung für `TAG_COPY_SIZE` im CCC ist 20 Bytes. Je nach Cache–Grösse des Zielprozessors kann es Sinn machen, in Hinblick auf maximale Geschwindigkeit möglichst viel Code linear hintereinander abzuspulen (beim MC68000, der keinen Cache besitzt, die optimale Methode) oder darauf zu bauen, dass das betreffende Codestückchen schon im Cache landen wird, wenn es kurz genug ist (wovon man in der Regel beim MC68030 und dessen Nachfolgern guten Gewissens ausgehen kann).

3.4.2 Bitfelder

Da Bitfelder zu den Datentypen gehören, die sich erfolgreich einer strikten Normung durch die ANSI–C–Gewaltigen entzogen haben, sei hiermit kurz auf die Implementierung unter CREST–C eingegangen. Der Grund für die fehlende Normung wird klar, wenn man bedenkt, dass es nicht Sinn einer Norm sein kann, eine Sprachfestlegung so zu treffen, dass eine bestimmte Hardware zwingend zur Implementierung erforderlich ist.

Im Prinzip sind Bitfelder keineswegs Felder, sondern sollten eher als **Bitstrukturen** bezeichnet werden. Auch die Syntax entspricht der von Strukturen. Als Basistyp der Strukturmitglieder sind allerdings nur die Datentypen `signed int` und `unsigned int` zulässig, wie in Abbildung 3.6 ersichtlich ist. Manche Compiler erlauben auch andere Datentypen — die Norm ist an diesem Punkt sehr freizügig. Der angegebene Datentyp sagt **nichts** über die Bitanzahl aus, mit der der betreffende Eintrag abgelegt wird. Es wird dadurch lediglich bestimmt, wie das betreffende Strukturmitglied extrahiert werden muss. Strukturmitglieder, die als `signed int` vereinbart wurden, werden vorzeichenbehaftet interpretiert — womit wohl klar sein dürfte, dass beim Datentyp `unsigned int` das höchstwertige Bit des Strukturmembers nicht als Vorzeichenbit aufgefasst wird...

```
struct test
{    signed int   a : 7 ,
                  b : 4 ;
   unsigned int   c : 7 ,
                    : 6 ;
   unsigned int   d : 8 ;
} ;
```

Abbildung 3.6: Deklaration eines Bitfeldes

Die Mitglieder einer solchen Struktur werden mit aufsteigenden Speicheradressen bitweise hintereinander abgelegt. Die maximale Breite einer einzelnen Komponente darf 32 Bit nicht überschreiten — hierbei handelt es sich schlicht um eine sinnvolle Beschränkung auf die Möglichkeiten von 32–Bit–Prozessoren und keineswegs um eine Vorgabe gemäss ANSI–Norm. Namen sind für die Strukturmitglieder nicht zwingend vorgeschrieben. So führt z.B. die 6 zwischen den Einträgen `c` und `d` zur Belegung von 6 Bits. Ein Zugriff kann dann jedoch auf Grund des fehlenden Namens nicht mehr erfolgen. Der Abbildung 3.7 können Sie entnehmen, in welcher Form CREST–C Bitfelder im Speicher ablegt.

Eine Null als Längenangabe führt zu einem Padding auf die nächste Byte–Grenze. Grundsätzlich sollten Sie jedoch das Padding durch explizite Angaben selbst übernehmen, da sich unterschiedliche Compiler auf unterschiedlichen Maschinen keineswegs sonderlich einig sind, ob auf Byte–, Wort– oder Langwortgrenzen gepaddet werden soll.

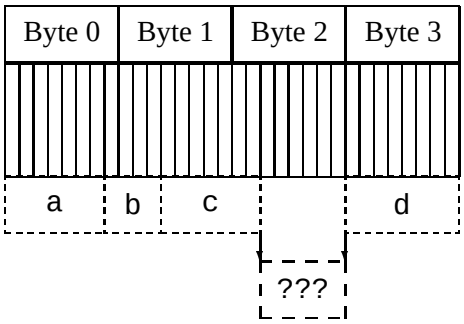


Abbildung 3.7: Speicherbelegung eines Bitfelds

Eine wesentliche Grenze bei der Auslegung von Bitfeldern unter CREST–C basiert auf der Unfähigkeit des MC68000, Datenworte oder –langworte von einer ungerade Adresse zu lesen. Bei Bitfeldkomponenten mit mehr als 17 Bit Länge ist stets darauf zu achten, dass ein solcher Wert mit einem Langwortzugriff — von der vorausgegangenen Wortgrenze beginnend — zu lesen sein muss. In der Regel ist die beschriebene Einschränkung allerdings eher rein akademischer Natur.

Überhaupt ist der praktische Nutzwert von Bitfeldern eher kritisch zu beurteilen. Bitfelder sollten dann eingesetzt werden, wenn es darum geht, Integer–Daten mit beschränktem Wertebereich möglichst speicherplatzsparend zu verwalten und die Laufzeit des Programmes im Vergleich zum Speicherbedarf für die Daten eine untergeordnete Rolle spielt. Der Zugriff auf Bitfeldelemente ist in Hinsicht auf den erzeugten Code (und die damit verbundene hohe Laufzeit) ungleich unökonomischer, als der Zugriff auf

Basisdatentypen. Ab dem MC68020 stehen zwar spezielle Bitfeldbefehle in der CPU–Hardware zur Verfügung, aber selbst diese Befehle sind — mit Verlaub gesagt — noch lausig langsam...

Auch Zugriffe auf externe Bausteine mittels Bitfeldselektoren sind mit Vorsicht zu betrachten, wenn die Hardware spezielle Zugriffsbreiten (8, 16 oder 32 Bit) verlangt. Da der CCC seine Bitfeldzugriffe möglichst schnell durchzuführen trachtet, werden prozessorabhängig sowie nach Breite und Lage des Bitfeldmitglieds im Speicher, unterschiedlichste Befehle und Zugriffsbreiten generiert. Beim Zugriff auf Mimosen–Hardware sollten Sie demnach besser auf die Syntax von Bitfeldern verzichten und den gewünschten Wert mittels normaler Bitbefehle bearbeiten — was in Hinsicht auf die Laufzeit der Programme in neun von zehn Fällen sowieso die bessere Lösung darstellt.

3.5 Vereinigungsdatentypen

Nach der ANSI–C–Norm wird ein Ausdruck grundsätzlich auf Basis des Vereinigungsdatentyps bewertet. Unter dem Vereinigungstyp versteht man bei zwei Operanden mit unterschiedlichem Datentyp den Datentyp, der in der Lage ist, den Wertebereich beider beteiligter Datentypen abzudecken. So steht es als Theorie klar und deutlich in der Norm. Das Ignorieren der Konsequenzen führt gerade bei zeitkritischen Funktionen oft dazu, dass der Compiler Code generiert, der wie ein Schlag ins Gesicht wirkt. Alternativ sind auch Ergebnisse denkbar, die zwar logisch erklärbar sind, aber von vielen Anwender so nicht erwartet wurden. Als beste Beispiele dienen da Vergleiche, die nie oder unerwartet immer greifen:

```
{  unsigned int  a ;
     signed int  b ;

   if ( a >= 0  ) {...}
   if ( b >= 0L ) {...}
}
```

Ebenso unerfreulich sind die Versuche, Variablen unterschiedlichen Datentyps mit identischem Bitmuster zu vergleichen und feststellen zu müssen, dass der Compiler anderer Ansicht über Identität der Objekte ist.

```
{
     signed char a = 0xFF ;
   unsigned char b = 0xFF ;

   if ( a == b )
      ; // Wird nie durchlaufen
   else
      ; // Wird immer durchlaufen
}
```

Wenn Sie derartige Probleme vermeiden wollen, sollten Sie tunlichst darauf verzichten, Variablen und Konstanten unterschiedlicher Datentypen ungecastet aufeinander loszulassen oder sich wenigstens die Tabelle 3.14 einprägen.

Die kursiv dargestellten Einträge *UL* in der Tabelle repräsentieren die Problemfälle, bei denen zwei unterschiedliche Integer–Datentypen nicht auf einen grösseren Datentyp ausweichen können, der beide Wertebereiche komplett überstreicht. Unter CREST–C wird hier mit dem Typ `unsigned long` gerechnet. Es gibt jedoch auch einige C–Compiler, die hier anders verfahren und deshalb ist in Hinblick auf Portierungen mit besonderer Vorsicht zu verfahren.

	SC	**UC**	**SS**	**US**	**SI**	**UI**	**SL**	**UL**	**FL**	**DB**	**EX**
SC	SC	SS	SS	SI	SI	UL	SL	UL	FL	DB	EX
UC	SS	UC	SS	US	SI	UI	SL	UL	FL	DB	EX
SS	SS	SS	SS	SI	SI	UL	SL	UL	FL	DB	EX
US	SI	US	SI	US	SI	UI	SL	UL	FL	DB	EX
SI	SI	SI	SI	SI	SI	*UL*	SL	*UL*	*FL*	DB	EX
UI	UL	UI	UL	UI	*UL*	UI	*UL*	UL	*FL*	DB	EX
SL	SL	SL	SL	SL	SL	*UL*	SL	*UL*	*FL*	DB	EX
UL	UL	UL	UL	UL	*UL*	UL	*UL*	UL	*FL*	DB	EX
FL	FL	FL	FL	FL	*FL*	*FL*	*FL*	*FL*	FL	DB	EX
DB	DB	DB	DB	DB	DB	DB	DB	DB	DB	DB	EX
EX	EX	EX	EX	EX	EX	EX	EX	EX	EX	EX	EX

Tabelle 3.14: Cast–Tabelle

Abkürzung	**Bedeutung**
SC	`signed char`
UC	`unsigned char`
SS	`signed short`
US	`unsigned short`
SI	`signed int`
UI	`unsigned int`
SL	`signed long`
UL	`unsigned long`
FL	`float`
DB	`double`
EX	`long double`

Tabelle 3.15: Bedeutung der Abkürzungen

Die Einträge *FL* sind bei Fliesskommaberechnungen mit besonderer Vorsicht zu geniessen. Der Wertebereich einer `float`–Variable deckt nicht den gesamten Wertebereich einer 32–Bit–Integervariable ab. Bei automatischen Casts können deshalb signifikante Bits verlorengehen.

Bitfelder verhalten sich identisch zu den korrespondierenden `signed`– bzw. `unsigned int`–Basisdatentypen. Der Typ `enum` ist in Hinsicht auf implizites Casting identisch zum Datentyp `signed int`.

Um vernünftigen Maschinencode zu erhalten, ist es in der Regel auch angebracht, Zahlen–Konstanten und Variablen entsprechend zu casten. In vielen Fällen kann ein Compiler nicht sinnvoll entscheiden, ob Ihr Quelltext eine genormte Eigenschaft von C ausnutzt oder ob es sich um schlampige Schreibweisen handelt, die er gefahrlos optimieren könnte.

3.6 Die Prototypen

Gegner der Sprache C kritisierten in erster Linie den laxen Umgang der C–Compiler beim Aufruf von Funktionen. Glücklicherweise hat sich mit der ANSI–C–Norm hier viel geändert. Das Zauberwort heisst *Prototyp*. Beim Compilieren alter UNIX–Sourcen ist es immer wieder erschreckend, welcher Schrott da sein Unwesen getrieben haben mag. Der Umgang mit den Parametern ist — selbst bei unkritischer Betrachtung — einfach grausam. Gut, für variable Parameterlisten gab es früher keinen allgemeingültigen Leitfaden für Programmierer. Die Ergebnisse sind entsprechend. Da werden Integer–

Datentypen locker mit Pointern kombiniert, die übelsten Zugriffe auf Stacks und in lokale Parameterlisten veranstaltet und hinterher läuft der Kram dann auch noch — oder wenigstens meistens — oder nur manchmal — oder eben überhaupt nicht!

Die Schöpfer der ANSI–C–Norm haben hier Abhilfe geschaffen. Leider ist auch hier wieder nur eine Halbheit genormt worden — schliesslich will man ja keinem Besitzer alter Sourcen wehtun. Die K&R–Quasi–Norm bleibt noch gültig. Zwar wurde eine neue Syntax für die Definition von Funktionsparametern geschaffen — `PASCAL` und `C++` lassen herzlich grüssen — aber die alten Quelltexte sollen selbstverständlich noch korrekt übersetzt werden.

Und gerade diesen Punkt halte **ich** nicht für selbstverständlich. CREST–C wurde als K&R–Compiler geboren und wurde auf ANSI–C–Standard aufgebohrt. Der K&R–Standard ist also im Grunde genommen der Heimatspielplatz dieses Compilers gewesen. Trotzdem habe ich mich in der Mitte der Entwicklungsphase dazu entschlossen, mich von der alten Norm endgültig zu verabschieden. Nahezu zwei Jahre lang verstand CREST–C keine K&R–Definitionen mehr und in dieser Zeit habe ich mich — gezwungenermassen — daran gewöhnt, die ANSI–C–Norm als Mass aller Dinge zu betrachten. Ich möchte behaupten, dass mir das nicht geschadet hat. Der Zwang, sämtliche Quelltexte erst ANSI–C–konform umzuhacken, bevor ich sie CREST–C vorwerfen konnte, führte zu interessanten Einsichten.

- ANSI–C–Compiler führen definitiv nicht zu betriebssichereren Programmen, solange man sie nicht mit ANSI–C–Quellen füttert.
- Ein Compiler, der auch die alte Norm akzeptiert, führt bei Portierungen meist zu einem wunderbaren Mischmasch, da man sich meist nicht der Mühe unterzieht, Abhilfe zu schaffen.
- Warnings führen nicht dazu, dass Quelltexte überarbeitet werden. Wenn der Compiler dem Programmierer nicht sofort an die Kehle springt, ändert sich in der Regel auch nichts am Quelltext.

Das Resultat meiner Überlegungen sehen Sie nun vor sich. CREST–C liebt Prototypen. Sie sollten sich ihm schnell anschliessen, wenn Sie nicht in einem Hagel von `BUS-ERROR`'s untergehen möchten. CREST–C ist zwar im Gegensatz zu den ersten Version unendlich viel nachsichtiger mit schlampig geschriebenen Quelltexten geworden, aber wenn Sie nicht genau wissen, was Sie tun, sollten Sie bei `WARNING`'s bezüglich fehlender Prototypen stets davon ausgehen, dass das erzeugte Programm eher nicht lauffähig sein wird. Funktionsaufrufe ohne Prototypen sollten Sie tunlichst vermeiden. Wenn der Rückgabetyp einer Funktion nicht bekannt ist, wird standardmässig `int` vorausgesetzt. Deckt sich diese Annahme nicht mit den Realitäten, entsteht entweder schlechterer Code oder das Programm läuft irgendwann aus schwer nachvollziehbaren Gründen gar heftig gegen die nächste deutsche Eiche.

Eine Meldung bezüglich fehlender Funktionsprototypen sollte ebenfalls ernst genommen werden. Fehlende Argumentprototypen sind unschön, führen aber in der Regel zu korrektem Code.

3.6.1 Prototypen für Rückgabewerte

Jede Funktion hat standardmässig den Datentyp `int` als Rückgabewert. Leider wurde auch in der ANSI–C–Norm die unselige Angewohnheit von FORTRAN nicht abgelegt, einen Datentyp als Standard zu betrachten. Funktionen können sämtliche Basisdatentypen als Ergebnis liefern. Seit Einführung des ANSI–C–Standards können auch `struct`'s und `union`'s als Funktionsergebnis zurückgeliefert werden. Ausserdem kann eine Funktion ohne Rückgabe nun mit dem Typ `void` vereinbart werden. Es empfiehlt sich dringend, alte Quellen zu überarbeiten, die ständig mit dem `default`–Datentyp `int` hantieren, da Sie sich sonst kaum vor der Masse der Fehlermeldungen von CREST–C retten können, wenn er fehlende `return`–Werte moniert.

Der minimale Prototyp, der von CREST–C akzeptiert wird, besteht aus dem Funktionsnamen und seinem Rückgabewert. Fehlt dieser und kommt es zu einer Verwendung der Funktion vor ihrer Definition, so gibt der Compiler eine Warnung aus. Um es mal so auszudrücken: starten können Sie derartige Pro-

gramme schon, aber Sie sollten sich diese Experimente besser verkneifen. Der Aufruf einer Funktion, die mit dem `default`–Datentyp `int` als Rückgabe rechnet, erwartet ein Funktionsergebnis im Datenregister `D0` — nachzulesen im Abschnitt über den Registergebrauch des CCC (3.10). Sieht die Definition der Funktion anders aus, als die jeweiligen Referenzen, dann sind Crashs zur Laufzeit vorprogrammiert.

3.6.2 Prototypen für Argumente

Schöner ist es natürlich, dem Compiler auch die Argumentprototypen zu liefern. Gerade dort kracht es schliesslich mit Vorliebe, wenn falsche Argumente auf den Stack geschoben werden, bei denen der Zugriff oder die spätere Verwendung den Rechner ins Nirvana befördert. Fehlt ein solcher Argumentprototyp, so generiert CREST–C eine `WARNING`–Meldung und bittet Sie um Abhilfe. Fruchtet dies nicht, so werden die Werte eben entsprechend den ANSI–C–Konventionen gecastet — Sie werden ja sehen, was Sie sich damit einhandeln. Parameter, die Sie an Funktionen ohne Prototyp oder mit offener Parameterliste übergeben, werden von CREST–C automatisch gecastet. `char`– und `short`–Parameter werden auf `int` gezogen und auf den Stack geschoben. `float`–Parameter und `double`–Parameter werden vor der Argumentübergabe auf `long double` gecastet. Parameter vom Typ `long` oder `long double` werden nicht gecastet. Es kommt also **definitiv nicht** zu impliziten Genauigkeitsverlusten, wenn Sie z. B. einen `long double`–Parameter mit `printf()` ausgeben wollen.

Dadurch ergeben sich bei der Verwendung offener Parameterlisten komische Effekte. Bei der Funktion `printf()` sind die Grössenangaben vor Integer– und Floatingpoint–Datentypen redundant — sollten aber in Hinblick auf portable Programme nie vergessen werden. Wenn Sie selbst Funktionen schreiben, die derartige *offene* Aufrufe handhaben sollen, so berücksichtigen Sie bitte, dass alle Integerdatentypen im `int`–Format und die Floatingpoint–Datentypen im `long double`–Format auf dem Stack liegen.

Selbstverständlich erstreckt sich die Typenüberprüfung auch auf die möglichen Argumente der Argumente. Die Kontrolle erfolgt rekursiv solange, bis CREST–C überzeugt ist, dass Sie wissen, was Sie tun. Zwei kleine Beispiele sollen das verdeutlichen:

```
extern int  f() ;
e( void ) { f( 1 ) ; f( 1, 2 ) ; f( 1, 2, 3 ) ; }
f( void ) { ... }
g( void ) { f( 1 ) ; f( 1, 2 ) ; f( 1, 2, 3 ) ; }
```

Die Verwendung der Funktion `f()` in der Funktion `e()` führt zu Warnungen. Ist jedoch die Definition erst erfolgt oder war der Prototyp von `f()` komplett, so ist CREST–C erst wirklich verärgert. So zu sehen bei der erneuten Anwendung von `f()` in der Funktion `g()`.

Das zweite Beispiel zeigt die Verwendung komplexerer Prototypen.

```
extern char *f( int         a,
                double  *(**b)(),
                struct z  (*c)(int v,float w,...) ) ;
```

Das entspräche dem Prototyp der Funktion `f()`, die drei Argumente erwartet und einen Pointer auf `char` liefert. Das Argument `a` ist ein trivialer Integerwert. Als zweiten Parameter erwartet der Compiler einen Pointer auf einen Pointer einer Funktion, die einen Pointer auf `double` liefert. Bei der Verwendung dieses Funktionspointers `b` wird CREST–C darüber meckern, dass die Argumentprototypen fehlen. Das dritte Argument ist jedoch komplett beschrieben. Der Compiler wird angewiesen, bei der Verwendung von `c` die Argumente `v` und `w` zu überprüfen, gegebenenfalls zu casten und alle weiteren Argumente — sofern vorhanden — kommentarlos zu akzeptieren und lediglich dem üblichen ANSI–C–Casting zu unterziehen.

Dummerweise akzeptiert der ANSI–C–Standard auch die Angabe von Prototypen, bei denen die Na-

men der Argumente fehlen. Das eben angeführte Beispiel sähe dann etwa so aus:

```
char *f(int,double*(**)(),struct z(*)(int,float,...));
```

Der riesige Vorteil dieser Schreibweise besteht darin, dass Sie nie wieder Angst vor dem Diebstahl Ihrer Quelltexte zu haben brauchen. Diesen Unfug versteht keiner — und Sie nach spätestens einer Woche ebenfalls nicht mehr! Sie sollten den Zwang des Prototypeings als eine Chance sehen, verständlichere und sicherere Quelltexte zu erstellen und von dem Ausweg der Kurzschreibweise nach Möglichkeit keinen Gebrauch machen. Weiterhin sei dringend angeraten, die Lesbarkeit durch sinnvolle und sprechende `typedef`'s zu erhöhen. Funktionelle Änderungen ergeben sich schliesslich durch derartige Typenvereinbarungen in der Sprache C nicht.

3.6.3 Abweichende Funktionsaufrufe

Ihren schlechten Ruf hat die Sprache C sich nicht zuletzt deswegen eingehandelt, weil sich bei fehlerhaften Funktionsaufrufen nahezu beliebiger Unfug im Rechner abspielen kann. Wenn man einen Stapel UNIX–Quelltexte zu portieren hat, ist unweigerlich immer wieder festzustellen, dass derartige Konventionen sehr langlebig sind und echte Freaks nicht sonderlich begeistert von dem Gedanken sind, sich von einem Compiler vorschreiben zu lassen, wie man vernünftige Programme zu schreiben hat. Auf erwachsenen Betriebssystemen mag eine solche Einstellung ja noch angehen, da diese mittels MMU–Einsatz dazu erzogen wurden, binäre Amokläufer ohne Schaden für andere Prozesse zu erschlagen. Unter RTOS–UH sind die Folgen in der Regel um Grössenordnungen fataler.

Mit einem reinrassigen ANSI–C–Compiler haben Sie nahezu keine Chancen, ältere K&R–Quelltexte zu läuffähigen Programmen zu übersetzen. Eine Funktion, die mit drei Argumenten definiert wurde, sollte eben nicht mit zwei oder vier Parametern aufgerufen werden. Soviel zu meiner felsenfestenÜberzeugung — und nun zur Implementierung!

CREST–C insistiert inzwischen nicht mehr darauf, dass derart grober Unfug wirklich ein Fehler sein muss und warnt nur noch eindringlich. Sie bekommen dennoch ladbaren Code heraus. Ob die Programme dann auch wirklich lauffähig sind, bleibt nunmehr Ihrem Forscherdrang überlassen. Auf alle Fälle sollten Sie die folgenden Warnungen **niemals** unbeachtet lassen.

```
Too few arguments passed to this function
Too many arguments passed to this function
Expected xxx argument(s) in call of function <yyy>
```

Um Ihnen wenigstens die Chancen auf ein funktionsfähiges Programm zu erhalten, werden*überschüssige* Parameter beim Funktionsaufruf so gecastet, wie es im Abschnitt 3.6.2 beschrieben wurde. Nutzen Sie bitte die Chance, Programme jetzt auf die ANSI–C–Schreibweisen umzurüsten. Ich bin langsam etwas verärgert über Leute, die mir meterlange Faxe mit gravierenden `WARNING`'s dieser Kategorie zuschicken und dann auch noch ganz unschuldig fragen, weshalb das Programm gelegentlich instabil läuft.

Beim Entwurf der ANSI–C–Norm ist leider eine Unart nicht abgestellt worden, die gerade bei der Portierung fremder Quelltexte stets zu rätselhaften Abstürzen führt. Die Angabe von Prototypen — respektive extern–Deklarationen — ist immer noch innerhalb von Funktionen zulässig. Vom Prinzip her ist daran nichts auszusetzen. Nur: innere Prototypen verdecken kommentarlos alle äusseren Vereinbarungen. Diese Möglichkeit bereitet in neun von zehn Fällen Ärger. CREST–C folgt bei Prototypen folgenden Regeln:

- Prototypen sind nach ihrer Vereinbarung grundsätzlich global bis zum Ende der Datei bekannt.
- Die Verdeckungsregeln sind bei Prototypen nicht gültig. Die Angabe eines Prototypen innerhalb einer Funktion darf **nie** im Widerspruch zu bereits getroffenen Vereinbarungen stehen, die dem

Compiler bis dahin bekannt geworden ist. Die zuletzt getroffene Vereinbarung ist stets für den Rest des Moduls und nicht nur für den gerade aktiven Compound–Level gültig.

- Die Übersteuerung eines Prototypen durch allgemeinere Prototypen oder speziellere Vereinbarungen ist zulässig.
- Wurde die Funktionsdefinition gelesen, ist keine weitere Veränderung durch abweichende Prototypen mehr möglich.

Der Grund für diese Regelung sollte eigentlich einleuchten. Nach Einführung der ANSI–C–Norm gibt es mittels der Prototypen und der klaren Regelung von Funktionen mit offenen Parameterlisten endlich die Möglichkeit, den Schmuddelkram vergangener Tage zu vermeiden. Eine Übersteuerung von Prototypen macht schlicht keinen Sinn und führt z.B. bei Veränderung der Rückgabedatentypen mit nahezu absoluter Sicherheit zum Exitus des übersetzten Programmes zur Laufzeit. Ich liebe zwar die Freiheiten, die mir die Sprache C bietet, lehne aber gerade in diesem Punkt die Freizügigkeit strikt ab, da diese Regelung allenfalls Katastrophen begünstigt und keinerlei Vorteile mehr bietet.

3.7 Der Stack

Die flüchtigen Daten werden — wie die C–Syntax bereits aussagt — *automatisch* bei jedem Eintritt in eine Funktion neu angelegt. Ihre Lebensdauer beschränkt sich auf die Laufzeit der jeweiligen Funktion, in der sie deklariert wurden. Da C–Funktionen rekursiv aufgerufen werden können, wäre es unzulässig, für diese Datenbestände einen festen Datenbereich im Speicher zu allokieren. Vielmehr wird dazu eine dynamische Form der Speicherplatzanforderung benutzt: der Stack oder Stapel!

Dummerweise ist vielen rekursiven Problemen nicht direkt anzusehen, wie tief die Rekursion zu ihrer Lösung reichen wird. Und damit ist man dann an einem kritischen Punkt: *Wie gross muss der Stack gewählt werden, um einen Überlauf zu vermeiden?* Eine eindeutige Antwort darauf kann es in den meisten Fällen nicht geben. Im Prinzip sind Sie selbst dafür verantwortlich, welche Grösse Sie als sinnvoll ansehen. Sie müssen sich beim Linken Ihrer Programme darauf festlegen, wieviel Speicherplatz Sie für den Stack verschwenden wollen. Wenn Sie zu grosszügig sind, bleibt viel Speicher unbenutzt im Besitz Ihrer Task. Sind Sie zu kleinlich, kommt es zu einem `Stackoverflow` und damit zu einigen üblen Verwirrungen in Ihrem System.

Um die variablen Daten zu verwalten, werden von CREST–C drei Adressregister benutzt. Der Stackpointer `SP` bzw. `A7` zeigt jeweils auf die aktuelle Position in Ihrem Stack. Die globalen Daten werden über das Adressregister `A5` angesprochen. Die task–lokalen Variablen werden über Adressregister `A4` verwaltet.

Globale Variablen und Stack toben beide im gleichen Speicherblock herum. Die globalen Variablen belegen einen fest vorgegebenen Speicherbereich im oberen Teil — mit kleineren Adressen — dieses Blockes. Der Stack wächst mit jedem Unterprogrammaufruf ein Stück in Richtung auf die globalen Variablen zu. Es soll Ihrer Phantasie überlassen bleiben, was wohl passiert, wenn der Stack durch Ihre globalen Daten hindurchwandert. Die globalen Variablen wirken dabei zwar als eine Art Sicherheitspuffer, den der Stack zuerst durchwandern muss, bevor er Datenbestände anderer Tasks vernichtet oder gar die interne Speicherverzeigerung von RTOS–UH breitmacht und den Rechner in die ewigen Jagdgründe schickt. In der Regel läuft so nur die Task, die sich die eigenen Datenbestände zerstört hat, voll gegen die Wand und erhält Ihnen wenigstens die Freundschaft der anderen Benutzer an Ihrem Rechner.

Um derart unschöne Effekte zu vermeiden, gibt es im Compiler eine Option, um die grausamen Folgen eines `Stackoverflow`'s des zu erzeugenden Programmes zu verringern. Mittels der Compileroption `-U` wird der CCC angewiesen, bei der Generierung jeder Funktion des Moduls einen Unterprogrammaufruf einzustreuen. Diese Funktion testet dann ab, ob unser Stackpointer den globalen Variablen schon gefährlich nahe gekommen ist oder gar schon in fremden Revieren wildert.

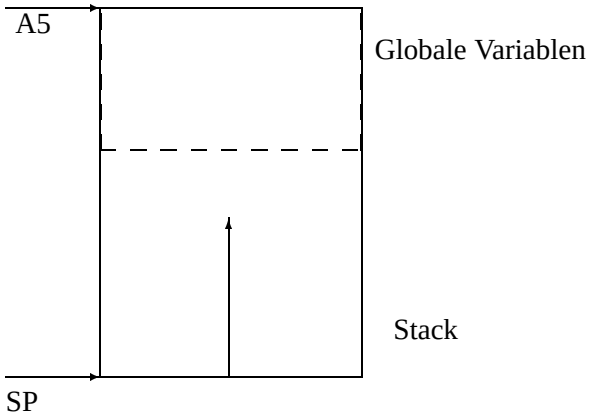


Abbildung 3.8: Lage des CREST–C–Stacks

Sicherlich ist eine solche Situation unschön, aber zumindest gelingt es in den meisten Fällen, den sonst unvermeidlichen Crash abzufangen. Die Testsequenz droppt Ihre Task bei `Stackoverflow`, setzt den Stackpointer wieder auf den Wert, den er beim Start der Task hatte und spult ein wenig Epilog–Code ab. So werden z.B. noch alle offenen Dateien geschlossen, um das Filesystem vor Schaden zu bewahren. Wenn Sie allerdings mit dem Stack so sparsam umgegangen sind, dass selbst bei diesen trivialen Aktionen erneut ein Overflow auftritt, so haben Sie endgültig verspielt — und es wohl auch nicht besser verdient.

Als Anhaltswert für eine sinnvolle Stackgrösse sollten Sie etwa 2 bis 8kB im Hinterkopf behalten. Durch Rekursion oder Verwendung grosser `auto`–Felder innerhalb der Funktionen, können sich jedoch rasch deutlich höhere Werte ergeben.

3.8 Lokale Variablen und Argumentübergabe

Bei der Auslegung der Grösse des Stacks ist ein wesentlicher Punkt zu berücksichtigen. Der Stack dient auch zur Speicherung lokaler Variablen. Diese lokalen Zellen werden beim Durchlaufen Ihres Programmes stets dynamisch angefordert. Jeder Funktionaufruf belastet den Stack durch

- die an die Funktion zu transferierenden Argumente.
- die dynamischen Variablen der Funktion.
- die temporären Zellen, die der Compiler zur Bearbeitung des Codes benötigt.
- die Rücksprungadresse aus der Funktion.
- einen möglichen Pointer auf die Zelle, in der die Funktions–Rückgabe erfolgen soll.
- die zu rettenden Register.

Um es an einem Beispiel zu demonstrieren. Die Funktion `f()` habe drei Übergabeparameter, drei lokale `int`–Variablen und nutze ein Array von Charactern. Zudem ist ersichtlich, dass die Funktion sich rekursiv selbst aufruft.

Die minimale Belastung des Stacks bei einem Aufruf von `f()` ergibt sich aus der Summe des Speicherplatzbedarfes von Argumenten, dynamischen Variablen, Rücksprungadresse zur aufrufenden Funktion und dem unvermeidlichen Verwaltungskram, den ein Funktionsaufruf leider nach sich zieht. In diesem

```
void f( int  x, int  y, int  z )
{
   char  testfeld[ 1000 ] ;
   int   a, b, c          ;
       ......
   if ( ! abbruchbedingung )
      f( x+1, y+1, z+1 ) ;
       ......
}
```

Abbildung 3.9: Beispielfunktion zur Speicherplatzabschätzung

Falle ergäben sich demnach:

```
Stackbelastung =    3 * 4 Byte (Argumente        )
               + 1000 * 1 Byte (char-Feld        )
               +    3 * 4 Byte (lokale Variablen )
               +        4 Byte (Ruecksprungadresse)
               = 1028 Byte + ?
```

Abbildung 3.10: Abschätzung des lokalen Speicherbedarfs einer Funktion

In dem Fragezeichen sind die temporären Zellen zusammengefasst, die der Compiler nur intern nutzt und die nach aussen nicht in Erscheinung treten. Weiterhin versteckt sich dort der Speicherplatzbedarf für die Register, die die Funktion verändert und die beim Verlassen wieder restauriert werden müssen.

Wenn Sie sichergestellt haben, dass die Funktion `f()` nie tiefer als in die zehnte Rekursion absteigen kann, genügt also eine Stackanforderung von etwa 12 kB für diesen Programmteil. Sollte eine klare Aussage über die zu erwartende Rekursionstiefe nicht zu machen sein, so hilft nur eine grosszügige Auslegung des Stacks und eine vernünftig gewählte Abbruchbedingung. Anschliessend können Sie einen Gott Ihrer Wahl um Erleuchtung und Beistand anflehen und das Programm starten.

Die Angelegenheit klingt gefährlich — und sie ist es auch. Mit etwas Übung hat man jedoch sehr rasch heraus, wie gross der Stack bei bestimmten Problemen gewählt werden muss, um sich beruhigt und sicher zu fühlen. Es empfiehlt sich dringend, in der Testphase den Stack reichlich zu bemessen und die Programme mit der `-U`–Option zu übersetzen.

Da ich Sie nun genug erschreckt habe, ist es an der Zeit, Ihnen zu verraten, wie es auch einfacher geht. In der Includedatei <`rtos.h`> ist der Prototype der Funktion `rt_used_stack()` zu finden. Wenn Sie Programme mit `-U`–Option compiliert haben, führt das laufende Programm eine Sicherheitsüberprüfung und zusätzlich eine kleine Statistik über den maximal benutzten Stack durch. Sie können also am Programmende die Funktion `rt_used_stack()` aufrufen und sich die maximale Zahl der allokierten Bytes auf dem Stack anschauen. Es empfiehlt sich bei derartigen empirischen Versuchen, den zu erwartenden *worst case* zu simulieren, einen Sicherheitsaufschlag beim Linken zu spendieren und rekursive Funktionsaufrufe zusätzlich zu den programmtechnischen Massnahmen mit der Stackcheck–Option abzudichten. Bei ausgetesteten Programmen sollte die `-U`–Option allenfalls in kritischen Bereichen aktiviert sein, da die ständigen Funktionsaufrufe doch ganz erhebliche Rechenzeit kosten. Je nach Komplexität der Funktionen können da schon mal 5 bis 30% der Rechenleistung kommentarlos den Bach runtergehen.

Achtung: Benutzen Sie **nie** die `-U`–Option aus Programmcode heraus, bei dem `A4` nicht korrekt gesetzt ist! So besitzen z.B. weder Interruptroutinen, Kalt– noch Warmstartscheiben einen Taskworkspace und folglich auch kein gültiges `A4`. Der Compiler unterdrückt zwar selbstständig innerhalb solcher *Sonderfunktionen* die Generierung des Stackcheckcodes. Dies gilt jedoch nicht für Funktionen, die aus einem derartigen Rumpf heraus aufgerufen werden. Findet z.B. innerhalb der Interruptroutine `Interrupt()` der Funktionsaufruf `TesteBitteDieHardware()` statt und enthält eben diese Funktion

Stackcheckcode, dann geht der Rechner nach relativ kurzer Zeit in die ewigen Jagdgründe ein. Der Compiler hat keine Chance, derartige Konstellationen abzufangen. Hier ist Umsicht auf der Programmiererseite verlangt...

Achten Sie auch peinlich genau darauf, dass die Verwendung der mit `-U` übersetzten Bibliotheken tödlich ist, wenn Funktionen dieser Libraries von Code aufgerufen werden, bei denen `A4` nicht gesetzt ist. Selbst ein unschuldiger `strcmp()` hat auf Interruptebene schon das Potential, den Rechner zu himmeln, wenn er aus den *Test*–Bibliotheken hinzugelinkt wurde.

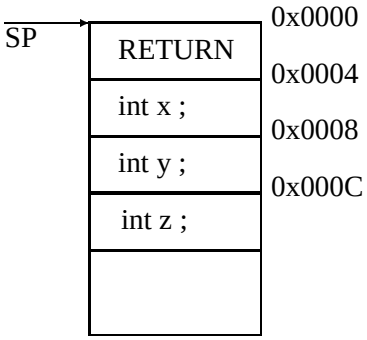
Kommen wir nun zum Parameter–Transfer an Funktionen. Argumente werden **grundsätzlich über den Stack** an die aufgerufene Funktion übergeben. Für reine C–Programmierer ist der Parameter–Transfer kein Thema. Normalerweise werden die Parameter schliesslich namentlich von der Funktion angesprochen und es kann dem Programmierer egal sein, wo diese Werte physikalisch herkommen. Bei offenen Parameterlisten, wie sie z.B. bei `printf()` Verwendung finden, gibt es nach ANSI–C–Norm endlich eine saubere Möglichkeit, an die Übergabewerte heranzukommen. In der Includedatei <`stdarg.h`> sind die folgenden Makros vereinbart, die eine portable Programmierung ermöglichen.

```
typedef void    *va_list ;
#define va_start(ap,parmN) ((ap)=((char*)&(parmN))+sizeof(parmN))
#define va_end(ap)
#define va_arg(ap,type)    (*((type*)(ap))++)
```

Abbildung 3.11: Auszug aus <stdarg.h>

Mit `va_start()` holen Sie sich den Pointer auf das erste Element der Parameter, die sich hinter den drei Punkten verstecken. Jeder Aufruf von `va_arg()` stellt Ihnen anschliessend den nächsten Wert auf dem Stack zur Verfügung. Sie müssen darauf achten, dass bei der Verwendung dieser Routinen auf der Aufruferseite die Argumente entsprechend der ANSI–C–Norm gecastet werden. Funktionen mit offener Parameterliste legen alle Integer–Datentypen als 32–Bit–`int's` auf den Stack. Fliesskomma–Datentypen werden als 96–Bit–`long double's` transferiert — der ANSI–C–Standard möchte da zwar gerne `double`–Typen sehen, aber manchmal bin ich eben etwas dickköpfig, wenn ich den Sinn einer unsinnigen Anweisung nicht erklärt bekomme...

Für Assembler–Programmierer stellt sich natürlich die Frage, an welcher physikalischen Position die Parameter zu finden sind. Wie bereits erläutert wächst der Stack zu kleineren Adressen hin. Nach einem Funktionsaufruf liegt ganz oben auf dem Stack die Rücksprungadresse zur aufrufenden Funktion.



Die Parameter schliessen sich *nach üblicher C–Konvention* dahinter an, werden also in umgekehrter Reihenfolge ihrer Deklaration auf den Stack geschoben.

Der Programmierer darf die Übergabe–Parameter jetzt beliebig modifizieren, hat jedoch dafür Sorge zu tragen, dass der Stack beim Verlassen der Funktion an der alten Position steht und die Rücksprungadresse niemals unbeabsichtigt verändert wird.

Innerhalb der Funktion darf beliebiger Assemblercode auftreten. Der Programmierer ist allerdings angehalten, veränderte Registerinhalte beim Verlassen der Funktion zu restaurieren. Die Ausnahme bestätigt hier mal wieder die Regel. Funktionen mit Rückgabewert transferieren diesen in einem Register an die aufrufende Funktion. Dieses Register sollte einleuchtenderweise hinterher nicht auf den ursprünglichen Wert gezwungen werden.

Für Funktionen mit Rückgabewert gilt folgende einfache Regel:

- Integerrückgaben wie `char`, `short` und `long` werden in `D0` erwartet.
- Pointerrückgaben wie `char*` oder `void*` werden über `A0` transferiert.
- Fliesskommawerte wie `float` oder `double` können von der aufrufenden Funktion dem Register `FP0` entnommen werden. Wenn keine FPU verhanden ist wird es etwas komplexer.
- Die Rückgabe von Strukturen oder Unions erfolgt über einen Pointer auf die Variable, die das Ergebnis aufnehmen soll.

Bei Datentypen, die nicht in Registern untergebracht werden können, wird die Angelegenheit für Assemblerprogrammierer schwieriger. Das Ergebnis wird über einen Pointer auf die Ergebniszelle weggespeichert. Dieser Pointer wird von der aufrufenden Funktion als letztes Argument auf den Stack gepackt, liegt also direkt vor der Rücksprungadresse auf dem Stack. Wenn der Aufrufer nicht am Rückgabewert einer Funktion interessiert ist, die gerne eine Rückgabe liefern würde, ist ein `NULL`–Pointer zu übergeben. Zur Laufzeit wird dann stets geprüft, ob der Aufrufer den Rückgabewert wirklich haben möchte.

3.9 Dynamische Stackverwaltung

Über Stacks haben Sie ja bislang gelernt, dass die Auswahl einer sinnvollen Stackgrösse eine echte Fleissarbeit darstellt. In der Regel ist die Angelegenheit nicht so gemein, wie sie bislang erschienen sein mag. Fehler enden zwar **garantiert** mit einem Crash, aber die Stellen, an denen Rekursionen auftreten, sind durch einen Blick in die Linker–Map eben leicht auszumachen. Solche Funktionen können und müssen gesondert behandelt werden.

Mittels des `#pragma`–Kommandos `DYNAMIC_STACK` lässt sich für ausgewählte Funktionen Code generieren, der selbsttätig kontrolliert, ob der aktuelle Stack verbraucht wurde und es gilt, sich Nachschlag anzufordern. Sie haben dabei keinerlei besondere Vorkehrungen auf Code–Ebene zu treffen. Sie teilen dem Compiler lediglich mit, wie gross der Stack sein soll, mit dem beim Überlauf weitergearbeitet werden soll und wie gross der Sicherheitsabstand ist, bei dem sich das Programm einen neuen Stapel besorgen muss. Die Syntax des Kommandos lautet wie folgt:

```
#pragma DYNAMIC_STACK NEWSTACK  16384L RANGE 2048L
#pragma DYNAMIC_STACK NEWSTACK  0x6400
#pragma DYNAMIC_STACK RANGE     512
#pragma DYNAMIC_STACK
```

Die `default`–Werte für `NEWSTACK` (16384) und `RANGE` (2048) sind sehr reichhaltig ausgefallen.

Es mag zwar trivial klingen, aber selbstverständlich können Sie nur Funktionen mit dynamischen Stackcheck versehen, die keine offenen Parameterlisten besitzen — das sind die Dinger mit den Pünktchen. Dem Compiler **muss** die Zahl der Parameter bekannt sein, die eine derartige Funktion erwartet. Auch CREST–C benutzt diese Option intern. Es gab eine Zeit, in der die Abschätzung des Stackbedarfs bei dem Compiler eine echte Geduldsprobe darstellte. Ein durchschnittliches Programm brauchte etwa ein halbes Megabyte und etwas komplexere Probleme konnten auch schon mal zwei oder drei MB's verbraten. Inzwischen hat sich diese Situation deutlich entspannt. Mit zunehmend besserem Code sackte der

Stackbedarf zum Compilieren eines durchschnittlichen Programmes auf zwei bis drei Kilobytes. Damit liegt der CREST–C–Compiler beim Stackbedarf durchaus im Bereich von Übersetzern, die ihren Job mit Parametertransfer über Register erledigen. Um auch Ausnahmesituationen halbswegs sicher zu handhaben, wurden die C–Compiler bis zu Release 1.306 mit einem Stack von 150kB übersetzt, wodurch aber in der Regel knappe 150kB pro gestartetem Compiler im Rechner verschwendet wurden.

Wenn Sie sich nicht in Assemblerprogrammierung auskennen, sei hier zunächst kurz und oberflächlich beschrieben, worauf Sie bei der Auslegung dynamischer Stacks zu achten haben. Stellen Sie sich vor, dass Sie ein Programm schreiben wollen, bei dem folgende Funktion auftaucht:

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

#pragma DYNAMIC_STACK NEWSTACK 16384L RANGE 2048L
void f( int  x )
{
   char  testfeld[ 100 ] ;

   printf( "X = %6d   STACK = %5ld\n", x, rt_used_stack() ) ;

   if ( x > 0 )
      f( x-1 ) ;
   else
      ++testfeld[ 0 ] ;
}

void main( void )
{
   f( 1000  ) ;
   f( 10000 ) ;
}
```

Abbildung 3.12: Beispielfunktion für rekursive Stacks

Im Kapitel über die Stackbelastung durch lokale Variablen und Argumente (siehe Abschnitt 3.8) wurde bereits über die Abschätzung des Stackbedarfs rekursiver Funktionen einiges ausgesagt. In dem kleinen Beispiel — einen Sinn werden Sie darin übrigens nicht finden — wird jeweils die maximale Auslastung des aktuellen Stacks angezeigt. Dazu sollte der CCC selbstverständlich mit der Option `-U` aufgerufen werden, da sonst keine Statistik betrieben wird und `rt_used_stack()` nur einen Haufen unbefriedigender Nullen liefert. Weiterhin sollten Sie die Entwicklungsbibliotheken `std`*x*`.lib` oder `flt`*x*`.lib` hinzulinken, da nur diese ebenfalls eine Stacküberwachung eingebaut haben. Mit allen anderen Bibliotheken spielen Sie ohne Netz und doppelten Boden!

Für jede Rekursion werden hier grob 120 Bytes auf dem Stack allokiert — dieser Wert ist jedoch abhängig von der Compilerversion, die Sie gerade benutzen und kann nach oben oder unten variieren. Ohne das `#pragma`–Kommando über der Funktion `f()` würde die Angelegenheit etwa 60 Durchläufe lang gutgehen (solange man auf die Ausgaben über `printf()` verzichtet), wenn man von einer Stackgrösse des Testprogramms von 8kB ausgeht. Danach erwischt es die globalen Variablen (sofern vorhanden) und dann die interne RTOS–UH–Verzeigerung. Jetzt gehen die Lichter an Ihrem Rechner entweder an oder aus — jedenfalls redet das Ding vermutlich hinterher nicht mehr mit Ihnen, da eine derartige Breitseite mitten in den Speicher wohl kaum auf viel Gegenliebe stossen wird! Wenn Sie die `printf()`–Ausgabe wieder einbauen, so kommt es etwas früher zum Crash.

Wenn Sie mit Bestimmtheit wissen, dass `f()` grundsätzlich einmal mit dem Höchstwert 10000 aufgerufen wird, dann brauchen Sie auch keinen dynamischen Stack. Setzen Sie bei Linken die Stackgrösse auf etwa 1.2 MB, laden das erzeugte Programm und gut ist es. Zur Not dürfen Sie dann halt noch ein paar RAM's nachstöpseln, aber um den Speicherbedarf kommen Sie nun wirklich nicht herum.

In der Regel sieht es aber so aus, dass man zur Übersetzungszeit nicht weiss, ob bei 10000 eine Grenze vorliegt oder in wievielen Fällen der angenommene Höchstwert gar deutlich über– oder unterschritten wird. Zeigt die Erfahrung im Umgang mit einem solchen Programm, dass der durchschnittliche Aufrufwert der rekursiven Funktion nicht bei einem klar definierten Maximalwert angesiedelt ist, dann ist es Schwachsinn, den Stackbedarf für einen geschätzten Maximalwert vorzuhalten. Hier — und erst an einem solchem Punkt — greift der dynamische Stack. Wenn Sie das Beispielprogramm übersetzt und gestartet haben, werden Sie feststellen, dass für jeden Aufruf der Funktion `f()` die bereits erwähnten 120 Bytes verbraucht werden (Differenz jeweils zweier Ausgaben). Also wird der ermittelte Wert als `RANGE`–Angabe eingesetzt, das Programm neu compiliert und gestartet.

```
#pragma DYNAMIC_STACK NEWSTACK 16384L RANGE 120
```

Prima, wieder was gelernt! Der Rechner schimpft über einen `Stackoverflow` — das bedeutet nämlich die lustige Meldung

```
Stack_overflow_(SUSPENDED)
```

Die Auswirkungen der `-U`–Option des CCC hat die Task aus dem Rennen geworfen, bevor die Grenze des dynamischen Stacks überhaupt greifen konnte. Die implizite Grenze zur Stacküberwachung der `-U`–Option ist auf 256 Bytes fest eingestellt ist — eine kleinere `RANGE`–Angabe für dynamische Stacks kann folglich nicht greifen. Nächster Versuch:

```
#pragma DYNAMIC_STACK NEWSTACK 16384L RANGE 512
```

Der Rechner mault oder steht schweigend auf dem Tisch — traurig aber wahr. Jedenfalls war die Angelegenheit bestimmt nicht so geplant. Aber schliesslich wird man nur durch Schaden klug. Die `RANGE`–Angabe ist als `Sicherheitsabstand` gedacht. Da die Abfrage zu Beginn der Funktion stattfindet, die mit einem dynamischen Stack bedacht werden soll, muss selbstverständlich auch all das berücksichtigt werden, was sich innerhalb der Funktion selbst abspielt. In unseren Falle steht dort ein unschuldig wirkendes `printf()` und der rekursive Aufruf von `f()`. Da das Programm eine Weile gelaufen ist und uns stets verraten hat, dass die rekursive Funktion `f()` sich selbst mit einer Stackbelastung von besagten 120 Bytes erreichen kann, muss etwas unvollständig sein — und so ist es auch, wenn Sie bei den Tests nicht die Testbibliotheken `std`*x*`.lib` oder `flt`*x*`.lib` verwendet haben.

Die meisten Bibliotheken betreiben selbst keinerlei Statistik. Sie sind grösstenteils ohne `-U`–Option compiliert worden. Der Wert, den unsere Funktion `rt_used_stack()` liefert, basiert ausschliesslich auf den Zahlen, die von den Modulen geliefert wurden, die explizit mit der `-U`–Option übersetzt wurden. Damit ist dann auch der Täter klar, der gerade den Rechner abgeschossen hat. `printf()` hat sich über den eigenen Stackbedarf nicht geäussert und lustig über die Grenzen hinweggemalt, die als Sicherheitsabstand des dynamischen Stack vorgegeben haben.

Also was tun? Nehmen Sie es einfach als eine Wahrheit, die vom Himmel fällt, dass kein *ordnungsgemässer* Aufruf einer Bibliotheksfunktion mehr als 2kB Stack verschlingt oder benutzen Sie zu Testzwecken die Bibliotheksfunktionen `std`*x*`.lib` und flt*x*.lib, die mit aktiver `-U`–Option ausgeliefert werden. Üblicherweise reicht ein Minimalwert von 2kB aus, um eine Task ohne lokale Variablen ablaufen zu lassen.

```
#pragma DYNAMIC_STACK NEWSTACK 16384L RANGE 0x800
```

Beim Verzicht auf den `printf()`–Aufruf ist auch ein Sicherheitsabstand von etwas über 256 Bytes hinreichend. Als Anhaltspunkt sollten Ihnen folgende Regeln dienen:

- **Nie** weniger als 256 Bytes als `RANGE`–Value angeben, solange das Programm mit `-U`–Option compiliert wurde.
- **Nie** den exakten Differenzwert von `rt_used_stack()`–Angaben als `RANGE`–Value angeben.

Die geringste Modifikation am Programm kann dann später zum Überlauf führen.

- **Minimal** 512 Bytes auf den mittels `rt_used_stack()` ermittelten Wert addieren, um etwas Sicherheit zu bekommen. Ich persönlich bevorzuge allerdings Sicherheitszuschläge von 1,2 bis 3 — je nach ermittelter Stackgrösse und der Panik, die der Gedanke an den Absturz einer Anwendung bei mir auslöst...

Mittels dieser wenigen Regeln bekommt man recht schnell stabile Programme zustande. Der Verzicht auf eine Überprüfung des benötigten Stacks bei Programmen, die in Produktionsrechnern eingesetzt werden sollen, ist in jedem Fall fahrlässiges Verhalten!

3.10 Interner Registergebrauch von CREST–C

Für eigene Assemblerprogramme ist es zunächst wichtig, die Verhaltensweise des aufrufenden Compilers und des zugrundeliegenden Betriebssystems zu verstehen. Hier sei zunächst das Verhalten für 68k–Maschinen erläutert.

CREST–C verwendet zur Parameterübergabe an Funktionen grundsätzlich den Stack. Der Stack ist ein Speicherbereich, dessen jeweils aktuelle Position mittels des Stackpointers `SP` verwaltet wird. Der Stackpointer wandert bei zunehmender Belegung von hohen zu niedrigen Adressen. Die aufrufende Funktion belegt den Stack mit den Argumenten, führt den Funktionsaufruf durch und hat den selbstständig f ür die Korrektur des Stackpointers Sorge zu tragen.

Zur Verwaltung der Sections belegt CREST–C zwei Adressregister fest mit deren Startadressen. Die globalen Variablen in der `.data`– und `.bss`–Section erreichen Sie über das Adressregister `A5`. Auch dieses Register sollten Sie nie mutwillig modifizieren und spätestens bei der Rückkehr aus einer assemblercodierten Routine wieder restaurieren.

Um task–lokale Variablen in der `.local`–Section zu verwalten, wird zudem das Adressregister `A4` verwendet. Dieses entspricht dem Zeiger auf den TWSP der Task, denn unter CREST–C wird die `.local`–Section hinter dem von RTOS–UH benötigten TWSP allokiert.

Weiterhin belegt CREST–C bei Generierung von Code für den Debugger das Adressregister `A6`.Über dieses Register wird die Aufrufstruktur der Funktionen verwaltet, um so einen *Callingstack* verwalten zu können.

Zur Rückgabe von Funktionswerten verwendet CREST–C für Integer-Rückgaben das Datenregister `D0` und für Pointer–Rückgaben das Adressregister `A0`.

68k-Register	Sonder-Bedeutung
D0	Integer-Rückgaben
D1	
D2	
D3	
D4	
D5	
D6	
D7	
A0	Pointer–Rückgaben
A1	
A2	
A3	
A4	Zeiger auf die `.local`–Section
A5	Zeiger auf die `.data`–Section
A6	Zeiger auf den Calling–Stack
A7/SP	Stackpointer

Der CCC benutzt alle Register, deren er habhaft werden kann. Er betrachtet A4 und A5 (und im Debugmodus zusätzlich A6) als gesperrt — allenfalls bei der Codegenerierung für Interruptroutinen, Kalt– und Warmstartscheiben vergibt der CCC diese Register. Die Adressierung der C–lokalen Variablen und der Übergabeparameter erfolgt grundsätzlich über den Stackpointer SP.

3.10.1 Besonderheiten des PowerPC

RTOS–UH basiert sowohl für 68K–Targets als auch für den PowerPC grössenteils auf identischen Assembler–Quelltexten. Der ablauffähige Assemblercode für den PowerPC wird durch Cross–Assemblierung dieser Quelltexte erreicht. Die Syntax des RTOS–UH–Assemblercodes basiert auf den Mnemonics für 68K–CPU's. Erst bei der Cross–Assemblierung erfolgt die Umsetzung in reinen PowerPC-Code. RTOS–UH bildet die bekannten 68K–Registernamen auf die GPR's (General Purpose Register) des PowerPC ab. Diese Register sind im Gegensatz zur 68K–Architektur universell einsetzbar. Die Aufteilung in Daten– und Adressregister existiert beim PowerPC nicht mehr. Die im vorausgegangenen Abschnitt getroffenen Aussagen bezüglich der 68K–Maschinen sind unter Verwendung der folgenden Umsetzungstabelle auch für den PowerPC gültig.

PPC-Register	68K-Register	Sonder-Bedeutung
`r0`	*D0*	Integer- und Pointer-Rückgaben
`r1`	*D1*	
`r2`	*D2*	
`r3`	*D3*	
`r4`	*D4*	
`r5`	*D5*	
`r6`	*D6*	
`r7`	*D7*	
`r8`	*A0*	
`r9`	*A1*	
`r10`	*A2*	
`r11`	*A3*	
`r12`	*A4*	Zeiger auf die `.local`-Section
`r13`	*A5*	Zeiger auf die `.data`-Section
`r14`	*A6*	Zeiger auf den *Callingstack*
`r15`	*A7*/*SP*	Stackpointer
`r16`		
`r17`		
`r18`		
`r19`		
`r20`		
`r21`		
`r22`		
`r23`		
`r24`		
`r25`		Temporäres Register RTOS–UH
`r26`		Temporäres Register RTOS–UH
`r27`		Temporäres Register RTOS–UH
`r28`	*G3*	Temporäres Register RTOS–UH /CREST–C
`r29`	*G2*	Temporäres Register RTOS–UH /CREST–C
`r30`	*G1*	Temporäres Register RTOS–UH /CREST–C
`r31`	*G0*	Temporäres Register RTOS–UH /CREST–C

Es existieren jedoch zwei bedeutsame Abweichungen.

1. Der `cccppc` verwendet `r0` sowohl für Integer– und Pointer–Rückgaben!
2. Im Gegensatz zur 68K–Implementierung betrachtet RTOS–UH bei Systemaufrufen eine gewisse Anzahl von Registern *grundsätzlich* als Scratch–Register, die nicht gerettet und restauriert werden. Der `cccppc` verfährt bei der Codegenerierung ähnlich und betrachtet die Register `r28` bis `r31` als temporäre Zellen.

3.11 Verschieblicher Code

Unter verschieblichem Code versteht man ein Programm, dessen S–Record keine speziellen Anweisungen mehr enthält, um den darin kodierten Binärcode an beliebiger Stelle ins EPROM brennen zu können. Ein derartiges Programm könnte eine binäre Kopie von sich selbst erstellen und die Kopie exekutieren. Ein Beispiel dafür ist RTOS–UH selbst. Es muss klar festgestellt werden, dass Verschieblichkeit keine Voraussetzung für lauffähigen Maschinencode darstellt!

RTOS–UH verwendet den Buchstaben `R` innerhalb von S–Records, um das nachfolgend abgelegte Langwort um die Ladeadresse des Moduls zu korrigieren. S–Records, die ein `R`–Symbol enthalten, sind nicht mehr frei verschieblich. Notwendig werden derartige Konstrukte immer dann, wenn der Maschinencode absolute Adressen (absolute Sprünge, Pointer auf die `.text`–Section...) enthält. Bei der Verwendung des `LOAD`–Kommandos werden `R`–Symbole (Text Relokationen) automatisch aufgelöst.

Da viele Tools mit den RTOS–UH–spezifischen Erweiterungen der Motorola–S–Records nicht umgehen können, ist es stellenweise wünschenswert, Programme zu schreiben, deren S–Records keine `R`–Symbole mehr enthalten. Der Linker CLN gibt die Zahl der Relokationen am Ende des Linkerlaufes aus.

```
LOAD Relocs...
Text Relocations :          0
Data Relocations :          0
Runtime Relocs...
Data Relocations :          0
Local Relocations:          0
```

Unterschieden wird dabei zwischen `LOAD Relocs` und `Runtime Relocs`. Die Angaben unterhalb des `LOAD Relocs`–Eintrages beziehen sich auf die Zahl der im S–Record enthaltenen `R`–Symbole, die zur Adressierung von absoluten Zugriffen auf die `.text`–Section benötigt wurden. `Text Relocations` bezieht sich auf Zugriffe, die innerhalb der `.text`–Section auftreten (z.B. Funktionsaufrufe). `Data Relocations` bezieht sich auf Zugriffe, die innerhalb der `.data`–Section gefunden wurden (z.B. initialisierte Funktionspointer oder Zeiger, die auf konstante Tabellen verweisen). Werte ungleich Null sind ein klares Indiz für die Tatsache, dass das gelinkte Programm nicht frei verschieblich ist.

Die Angaben unterhalb des `Runtime Relocs`–Eintrages beziehen sich auf Relokationen, die erst nach dem Start des Programmes ausgeführt werden können, da die notwendigen Adressinformationen weder während des Linkerlaufes noch beim Laden des Programmes zur Verfügung stehen. CREST–C–Programme wickeln diese Adressauflösungen innerhalb des Startup–Codes ab. `Runtime Relocs` führen nicht zu `R`–Symbolen innerhalb der S–Records und haben keinerlei Einfluss auf die Verschieblichkeit von Programmen. Die dort aufgeführten `Data Relocations` geben die Zahl der absoluten Zugriffe auf Symbole in der `.data`–Section an. `Local Relocations` beziehen sich entsprechend auf Zugriffe in der `.local`–Section.

Es gibt eine einfache Möglichkeit, Maschinencode, der nicht frei verschieblich sein kann und muss, von den enthaltenen `R`–Symbolen zu befreien. Man weist den CLN an, das Programm für eine bestimmte Stelle im RAM oder EPROM zu übersetzen. Der CLN bekommt dies über die Optionen `-T=textaddress` und `-C=commonaddress` mitgeteilt. Das Programm ist dann fest gebunden und muss exakt an diese Position `textaddress` gebrannt werden und erwartet etwaige `absolute` Variablen an der Adresse `commonaddress`.

Der Weg zu frei verschieblichem Code — sofern es denn überhaupt einen gibt — ist deutlich steiniger. Zunächst ist der CCC anzuweisen, für sämtliche Funktionsaufrufe relative Adressierungsarten zu verwenden. Für die PowerPC–Familie passiert das bereits implizit. Für die 68K–Familie geschieht das über die Option `-R=3`, die in Abschnitt 3.1.3 erläutert wird. Kommt es danach beim Linken des Programmes zu Fehlermeldungen, so ist bei 68000er–Zielsystemen Schluss, denn es existieren keine relativen Sprunganweisungen für Distanzen grösser als 32kB. Ein solches Programm ist einfach zu gross, um verschieblich kodiert zu werden. Bei Prozessoren ab dem 68020 aufwärts können mittels `-R=2` auch lange Sprünge relativ adressiert werden. Ein derart übersetztes Programm sollte im Vergleich zu einer mittels `-R=0` übersetzten Variante deutlich kleinere Angaben in der `LOAD Reloc`–Statistik des CLN aufweisen.

Der nächste Schritt besteht aus Anpassungen im Linkfile, da die Standard–Startup–Dateien `tstart.obj` und `sstart.obj` nicht frei verschieblich sind.

Startupfile	Prozessor	verschieblich	ladbar
`sstart.obj`	alle 68K	nein	ja
`sstartr0.obj`	MC68000	ja	ja
`sstartr2.obj`	MC68020	ja	ja
`sstartr3.obj`	CPU32	ja	ja
`tstart.obj`	alle 68K	nein	ja
`tstartr0.obj`	MC68000	ja	nein
`tstartr2.obj`	MC68020	ja	nein
`tstartr3.obj`	CPU32	ja	nein

Tabelle 3.16: Verschieblichkeit von 68K–Startupfiles

Zunächst müssen Sie die Startupdatei `sstart.obj` durch deren verschiebliches Pendant `sstartrx.obj` ersetzen bzw. bei Verwendung von `tstart.obj` stattdessen die Datei `tstartrx.obj` einsetzen. Das `x` im Namen gibt die Prozessorkennung an.

Startupfile	verschieblich	ladbar
`sstart.obj`	ja	ja
`sstarta.obj`	nein	ja
`sstartal.obj`	nein	ja
`tstart.obj`	nein	ja
`tstartr.obj`	ja	nein
`tstarta.obj`	nein	ja
`tstartal.obj`	nein	ja

Tabelle 3.17: Verschieblichkeit von PPC–Startupfiles

Bei PowerPC–Programmen ist `tstart.obj` durch `tstartr.obj` zu ersetzen.

Beachten Sie bitte, dass alle Startup's mit dem Prefix `tstartr` sich nicht mehr zum Laden des Programmes mittels `LOAD`–Befehl eignen und der CLN eine entsprechende Warnung ausgibt.

Im nächsten Schritt ist eine geeignete Bibliothek zu wählen. Eine Auflistung der verfügbaren Bibliotheken ist im Abschnitt 12 zu finden. Für Zielsysteme mit 68000er–Kern ist die Verwendung einer `fast`–Bibliothek zwingend erforderlich. Für alle anderen Prozessortypen kann wahlweise eine `fast`– oder eine `long`–Variante verwendet werden.

3.12 Variablen und Konstanten unter CREST–C

RTOS–UH basiert auf dem Prinzip, sämtliche Tasks eines Projektes **gleichzeitig** im Speicher zu halten. Nur so ist es möglich, akzeptable Taskwechselzeiten zu erzielen, die unter Einbeziehung langsamer Peripherie in die Taskwechselzeiten nicht zu gewährleisten wären. In der Regel ist es absolut indiskutabel, mal eben ein paar Kilobytes von der Diskette oder Festplatte nachzuladen, um auf einen Interrupt oder eine Einplanung zu reagieren. Wenn es gilt, Reaktionszeiten von unterhalb einer Millisekunde zu erzielen, erscheinen schon die Spurwechselzeiten einer modernen Festplatte von 9 bis 20 Millisekunden geradezu astronomisch hoch — selbst wenn man die Zeit zum Laden der Daten und den Overhead zur Verwaltung des Paging vernachlässigt.

Da sich unter RTOS–UH alle Programmdaten physikalisch im Speicher befinden müssen, sind die Taskwechselzeiten (weitestgehend) deterministisch. Die Motorola–Prozessoren ab dem MC68030 besitzen eine interne MMU (ab dem MC68020 bereits extern verfügbar und zu Zeiten des MC68000 bereits mit Waschbrettern voll IC's in damaligen Workstations realisiert). RTOS–UH ignoriert diese Tatsache und deshalb ist das Speichermodell sehr simpel zu erklären: es gibt einen gemeinsamen linea-

ren Adressraum, den sich alle Tasks brüderlich zu teilen haben. Wenn eine einzige Task unkumpelig ist und in Bereichen rumschreibt, die ihr nicht gehören, dann himmelt es eben in der Regel das komplette System. Aber Speicherschutz ist ohne Einsatz einer MMU faktisch ein Ding der Unmöglichkeit.

Dem willentlichen Austausch von Daten kommt dieses Konzept jedoch sehr entgegen. Wenn eine Task mit einer anderen Task Daten teilen möchte, muss es lediglich eine Möglichkeit geben, mittels derer die Tasks sich auf einen gemeinsamen Speicherplatz im Adressraum einigen können. Die Daten liegen dann an einer absoluten Position und beide Tasks können unter Verwendung der absoluten Adresse darauf zugreifen.

Wenn Sie in PEARL kodieren und mit globalen Variablen arbeiten, dann ermittelt der `LOAD`–Befehl zur Ladezeit einen freien RAM–Bereich für diese Variablen und ersetzt sämtliche Zugriffe auf diese Speicherzellen, die in den S–Records mittels spezieller Escapesequencen vermerkt sind, durch die nun bekannte absolute Position im Speicher. Diese Methode klappt ganz wunderbar, hat jedoch in einer Reihe von Anwendungen ein entscheidendes Manko.

Eine Task, die ihren ausführbaren Code mit absoluten Variablenbereichen verknüpft hat, ist nicht wiedereintrittsfest!

In vielen Fällen braucht man auch keinen wiedereintrittsfesten Code. Eine in PEARL kodierte Task legt ihre Variablen auf feste Positionen. Demnach muss verhindert werden, dass eine erneute Aktivierung einer bereits gestarteten Task zu einer Doppelnutzung identischer Variablenbereiche führt. Unter RTOS–UH ist dies so realisiert, dass das System derartige Mehrfachaktivierungen verhindert und die Aktivierungen puffert, bis die Task sich terminiert hat oder extern terminiert wird. Für jede PEARL–Task steht exakt ein Verwaltungsblock — der Taskkopf — zur Verfügung, in dem der Zustand der Task vermerkt ist. Gerade im Bereich von Steuer– und Regelungsaufgaben ist die Lösung sehr praktisch, da so sichergestellt ist, dass eine Task ihre Aufgaben abgeschlossen hat bevor sie erneut gestartet werden kann.

Die Alternative besteht darin, wiedereintrittsfesten Code zu generieren, der seinen eigenen Variablenspeicher verwaltet. Unter RTOS–UH wird dieses Konzept der Entkopplung von Programmcode und Variablen mittels Shellmodulen verwirklich, die unabhängige Subtasks aufsetzen. Das grundlegende Prinzip basiert auf der Tatsache, dass der Code hier keinen festen Taskkopf enthält. Stattdessen wird dort ein Shellmodulkopf im Code abgelegt, der nur den Hinweis enthält, dass ein Taskkopf dynamisch beim Start einer derart unfertigen Task angelegt werden muss.

3.13 Syntaxerweiterungen unter CREST–C

ANSI–C bietet keinerlei sprachliche Unterstützung für Betriebssysteme, bei denen gleichzeitig die Mittel für Zugriff auf Variablen mit absoluten und mit relativen Adressen gewünscht ist. Die sinnvollste Methode bei der Implementierung des Compilers bestand darin, den flexibleren Weg zu wählen und alle CREST–C–Programme als Subtasks mit relativer Adressierung der Variablen zu implementieren. Eine dauerhafte Variable einer C–Task, die aus einem Shellkommando erzeugt wurde, ist demnach **niemals** von einer anderen C–Subtask erreichbar, die vom identischen Shellkommando aus erzeugt wird.

Der Begriff der globalen Variable ist unter CREST–C also immer nur so zu verstehen, dass die globalen Variablen einer CREST–C–Subtask nur innerhalb der Funktionen einer solchen Subtask und den ’Sub’–Subtasks (Threads) dieser Subtasks bekannt sind.

3.14 Schlüsselworte für Speicherklassen

Die folgenden Schlüsselworte bestimmen, wie der Compiler Objekte verwaltet, wo diese erreichbar sind und wie sie im Speicher abgelegt werden.

Schlüsselwort	K&R	ANSI	CREST–C
`absolute`	—	—	X
`auto`	X	X	X
`extern`	X	X	X
`local`	—	—	X
`pearl`	—	—	X
`register`	X	X	X
`static`	X	X	X

Tabelle 3.18: Schlüsselworte zur Angabe der Speicherklassen

Schlüsselwort	K&R	ANSI	CREST–C
`const`	—	X	X
`volatile`	—	X	X

Tabelle 3.19: Schlüsselworte zur Modifikation der Speicherklassen

3.14.1 Das Schlüsselwort absolute

Um die Kommunikation von absolut unabhängigen Prozessen miteinander zu ermöglichen, wurde die Speicherklasse `absolute` eingeführt. Das Schlüsselwort wird in folgenden Kombinationen unterstützt:

```
       absolute int  a ;
static absolute int  b ;
extern absolute int  x ;

void f( void )
{
   static absolute int  c ;
   extern absolute int  y ;
}
```

Die absoluten Variablen dienen zur Kommunikation zwischen vollständig unabhängigen Prozessen. Als Beispiel diene ein Programm, das in der Lage sein soll, zwei dauerhafte Subtasks mit Namen `A` und `B` zu erzeugen. Das Programm selbst sei vom Linker mit dem Namen `X` versehen worden. Nach dem Laden des Programmes existiert demnach ein C–Shellmodul `SMDL X` im System.

```
absolute absolute_variable ;

long  x ;
#pragma TASK
void A( void )
{         int  a0 ;
   static int  a1 ;
}
```

```
long  y ;
#pragma TASK
void B( void )
{         int  b0 ;
   static int  b1 ;
}

long  z ;
void main( void )
{ int  w ;

   A() ; B() ; rt_suspend() ;
   A() ; B() ; rt_suspend() ;
}
```

Wenn Sie dieses C–Shellmodul `X` starten, so erzeugt das System einen neuen Taskkopf und im Normalfall liegt danach eine Subtask mit dem Basisnamen `X` und einer vom RTOS–UH verteilten Kennummer vor. Der Einfachheit halber fangen wir im Beispiel bei /00 mit der Verteilung an. Unsere Task liegt also als `X/00` in der Speicherkette vor. Die Aufrufe der *Funktionen* `A()` und `B()` führen zur Erzeugung zweier Sohnprozesse `A/01` und `B/02`. Die exakte Vorgehensweise ist im Abschnitt 13.3.1 beschrieben. Anschliessend wird unserer Vaterprozess suspendiert, um das Zwischenergebnis anzuschauen.

Wenn unsere `main()`–Task `X/00` das Bedürfnis verspürt, sich mit ihren Ablegern zu unterhalten, so kann das über die normalen globalen Variablen von `X/00` geschehen, da bei der Erzeugung der beiden Kinder auch das Wissen um die Lage der globalen Zellen von `X/00` vererbt wurde.

Dabei gelten die bei C üblichen Scope–Regeln. So kommt `X/00` an alle drei globalen Zellen `x, y` und `z` heran, während `A/01` nur die Variable `x` kennt und `B/02` nur Zugriff auf `x` und `y` hat. Die Kinder haben beide je zwei lokale Variable. Bei den Variablen `a0` und `b0` handelt es sich um automatische Variablen, die auf dem Stack der jeweils laufenden Subtask abgelegt sind bzw. nach Gutdünken des Compilers auch in Registern gehalten werden könnten. Bei den statischen Variablen `a1` und `b1` handelt es sich jedoch um dauerhafte Objekte, die zwar nur innerhalb der beiden Tasks bekannt sind, aber im globalen Datenbereich des Vaterprozesses angesiedelt sind.

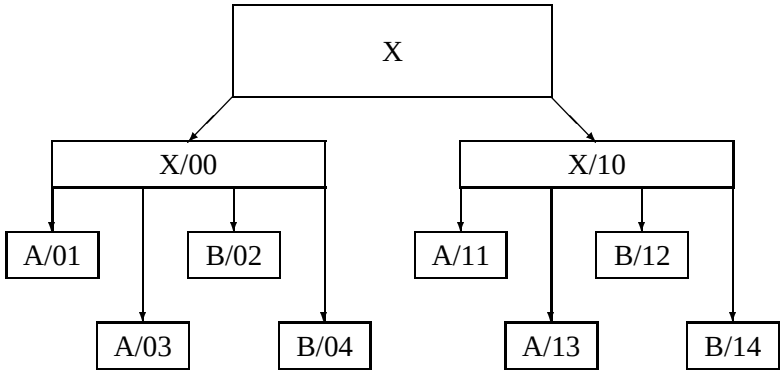
Setzen wir den Vaterprozess `X/00` fort, so entstehen zwei weitere Ableger, die z.B. `A/03` und `B/04` heissen könnten. Auch diese Sohnprozesse sind in der Lage, mit dem Vaterprozess Daten über die bereits erläuterten globalen Variablen zu tauschen. Die beiden neuen Kinder sind aber zudem fähig, sich über die funktions–statischen Variablen mit ihren Zwillingen zu unterhalten. So könnte die Variable `a1` dazu dienen, Daten zwischen den Tasks `A/01` und `A/03` auszutauschen, ohne dass die Tasks `X/00`, `B/02` und `B/04` jemals Zugang zu diesen Speicherzellen hätten. Sie als Programmierer sind dabei selbstverständlich dafür verantwortlich, den Zugriff auf diese gemeinsamen Variablen zwischen Vater und Söhnen so zu synchronisieren, dass die Konsistenz gewahrt bleibt.

Wenn zwei konkurrierende Prozesse auf gleichen Datenbeständen arbeiten, so ist die Absicherung mittels Semaphoren zwingend notwendig!

Wenn Sie sich an diesen gutgemeinten Ratschlag halten, besteht für Sie die Hoffnung, deterministische Programme zu entwickeln — andernfalls eben nicht! Im Beispiel habe ich diesen Vorgang eingespart. Aber Sie wissen ja: Beispiele zeigen immer nur, wie man es besser nicht machen sollte.

Bislang haben sind demnach fünf Prozesse im Speicher, die alle Zugriff auf die globalen Variablen des Vater–Prozesses `X/00` haben. Jede dieser Tasks besitzt ihren eigenen Stack, ihren eigenen Stackpointer und arbeitet mit einem eigenen Satz an lokalen Variablen.

Und jetzt fangen wir an, die Sache unübersichtlich zu gestalten und werfen ein weiteres Mal unser C–Shellmodul `X` an. Wie kaum anders zu erwarten, entsteht nun eine neue Vatertask, die der Einfachheit halber als `X/10` bekannt sein soll — nur, um das Zahlenwirrwarr nicht auf die Spitze zu treiben. Nach einmaliger Fortsetzung dieser neuen Task besitzt nun auch diese vier Söhne `A/11`, `A/13`, `B/12` und `B/14`.



Jetzt bitte nicht verzweifeln, denn viel komplexer wird die Angelegenheit nicht mehr. Bislang haben wir ja schon geklärt, dass:

- Väter mit **ihren Söhnen** über die globalen Variablen verhandeln können
- Gleichnamige Söhne des **gleichen Vaters** untereinander über die jeweilig gemeinsamen funktions–statischen Variablen Daten tauschen oder sich der globalen Variablen des Vaters bedienen können
- Söhne mit **ihrem Vater** über die globalen Variablen reden

Bleibt einzig das Problem zu klären, wie es möglich ist, dass die beiden Vaterprozesse `X/01` und `X/10` miteinander kommunizieren können. Mittels der Speicherklasse `absolute` kann man nun auch Daten zwischen den beiden unabhängigen `main()`–Tasks transferieren. Dazu dient ein Speicherbereich, der nicht den Vaterprozessen, sondern vielmehr direkt dem C–Shellmodul zugeordnet ist. Jetzt ist das C–Shellmodul bezüglich der Variable `absolute_variable` nicht mehr wiedereintrittsfest, da nun die Grundbedingung von völliger Trennung der Datenbereiche nicht mehr gegeben ist.

Im Beispiel war bislang nur die Rede von C–Subtasks innerhalb eines C–Shellmoduls. Im Abschnitt 13.5.3.3 wird auch deutlich, dass selbst innerhalb eines einzigen C–Shellmoduls durchaus die Möglichkeit besteht, vollständig vom Vaterprozess abgekoppelte Prozesse zu erzeugen. Auch hier kann es von grosser Wichtigkeit sein, prozessübergreifend auf gemeinsame Datenbestände zugreifen zu können.

3.15 Lebensdauer von Variablen

C unterscheidet Objekte zunächst nach deren Lebensdauer. Jedes Objekt, das ausserhalb einer Funktion vereinbart wird, existiert vom Start der Task bis zu deren Terminierung. Alle dauerhaften Variablen erhalten beim Start einer Task ein Null–Muster zugewiesen, wenn nicht explizit ein Initialwert angegeben wurde. Dauerhafte Variablen im Sinne von ANSI–C sind z.B. folgendermassen definierte Objekte:

```
          int  a ; // Diese Variablen werden beim Programmstart
   static int  b ; // einmalig vorneweg mit Nullmu-
stern initialisiert.

   void f( void )
   {
```

```
      static int  c ;
   }
```

Alle drei Variablen `a`, `b` und `c` sind dauerhafte Variablen. CREST–C reserviert — da keine keine Initialisierung erfolgt ist — für diese Objekte Speicherplatz in der `.bss`–Section. Dieser Speicherbereich wird beim Start einer Task automatisch mit Nullen initialisiert. Erfolgt jedoch — wie im nächsten Beispiel ersichtlich — eine explizite Initialisierung durch den Programmierer, so legt CREST–C die Daten in der `.data`–Section ab. Dieses unterscheidet sich von der `.bss`–Section lediglich dadurch, dass der Compiler nun auch Startwerte hinter dem Programmcode ablegen muss, die bei jedem Aufsetzen einer Task in die der Task zugeordnete `.data`–Section übernommen werden müssen.

```
         int  a = 1 ; // Diese Variablen werden beim Programmstart
   static int  b = 2 ; // einmalig vorneweg mit den Startwer-
ten versehen.

   void f( void )
   {
      static int  c = 3 ;
   }
```

Um auf einen häufig gemachten Anfängerfehler nochmals hinzuweisen: Die `static`–Variable `c` wird nicht bei jedem Durchlauf durch die Funktion `f` erneut initialisiert. Die Initialisierung erfolgt nur einmalig beim Programmstart.

Variablen innerhalb von Funktionen sind in der Regel flüchtiger Natur. Das bedeutet, dass ihre Lebensdauer nur vom Eintritt in die Funktion bis zum Verlassen der Funktion reicht — oder um korrekt zu sein: sich nur auf den Bereich erstreckt, der dem zugeordneten Compound–Level (den beiden geschweiften Klammern, die zur Definition gehören) entspricht.

```
( 01 ) void f( void )
( 02 ) {
( 03 )             int a     ; // Diese Variablen werden bei
( 04 )             int b = 1 ; // jedem Programmaufruf neu
( 05 )    auto     int c = b ; // initialisiert...
( 06 )    register int d = a ; // "                         "
( 07 )    {                    // "                         "
( 08 )       int e ;           // "                         "
( 09 )       int a ;           // "                         "
( 10 )    }
( 11 ) }
```

Im Beispiel sehen Sie die Definition von flüchtigen, kurzlebigen, automatischen (oder wie immer man sie auch bezeichnen will) Variablen. Innerhalb von Funktionen können die Speicherklassen `auto` und `register` verwendet werden. Entsprechend der ANSI–C–Norm werden alle Variablen innerhalb einer Funktion implizit der Speicherklasse `auto` zugeschlagen, wenn man sich als Programmierer nicht die Mühe gemacht hat, eine Speicherklasse anzugeben.

Nach der ursprünglichen Sprachdefinition von K&R bedeutete die Verwendung des Schlüsselwortes `register` für den Compiler den Hinweis (oder gar Zwang), das betreffende Objekt in einem schnellen Prozessor–Register unterzubringen. Registerzugriffe verlaufen in der Regel um Faktoren schneller ab, als der Zugriff auf den langsamen Hauptspeicher. Mit zunehmenden Qualität der Compiler sank diese Bedeutung des Schlüsselwortes `register`, da die Rechenknechte normalerweise einen besseren Überblick besitzen, wo und wann sich die Verwendung von Prozessor–Registern für bestimmte Objekte lohnt. Auch die Entwicklung der Hardware mit grösseren Cache–Speichern und zunehmend

intelligenteren Algorithmen, um Zugriffe auf den langsamen Hauptspeicher zu vermeiden, verwischt zunehmend die ursprüngliche Bedeutung dieses Schlüsselwortes. CREST–C kümmert sich bei der Registerverteilung nicht um den liebevoll ausformulierten Wunsch des Benutzers, eine Variable einem Register zuzuordnen und verwendet stattdessen eigene Regeln über Häufigkeit des Auftretens und Art der Benutzung. Trotzdem sollten Sie das Schlüsselwort `register` immer dann verwenden, wenn Sie sich davor schätzen wollen, dass innerhalb des Quelltextes die Adresse eines bestimmten Objektes gebildet werden kann. Sie können so sicherstellen, das ein Objekt — egal ob es real in einem Register existiert oder im Hauptspeicher — nicht durch einen unüberlegten Pointerzugriff modifiziert wird. Durch explizite Angabe der Speicherklasse `auto` kann dem Compiler die Wahlfreiheit genommen werden. Explizit als `auto` ausgewiesene Variablen werden von CREST–C grundsätzlich **nie** in Registern untergebracht und immer auf dem Stack abgelegt.

Flüchtige Variablen müssen explizit bei jedem Eintritt in eine Funktion initialisiert werden. Die Werte sind ansonsten undefiniert. Die beiden Zeilen (4) und (5) des Beispiels stellen demnach korrekten Code dar. Beide Variablen `b` und `c` enthalten hinter Zeile (5) dem Wert Eins. Dagegen kostet Sie die Kombination der Zeilen (3) und (6) mit an Sicherheit grenzender Wahrscheinlichkeit ein paar Stunden Arbeit extra, denn die Variablen `a` und `d` enthalten nach Zeile (6) zwar identische Werte — aber halt *irgendwas aus dem Wertebereich* eines `int`'s (-2147483648 bis 2147483647) mit dementsprechend geringer Wahrscheinlichkeit, dass die von Ihnen erhofften Zahlen zufällig getroffen wurden. Gemäss Murphy stehen allerdings während aller Testläufe immer die richtigen Werte in den nichtinitialisierten Variablen. Die Tatsache, dass solche Bugs immer erst beim Kunden und unter Zeitdruck auffallen, ist geläufige Praxis.

Flüchtige Variablen werden keiner Speichersection zugeordnet, sondern befinden sich entweder in Prozessor–Registern oder auf dem Stack. Sie werden automatisch beim Eintritt in eine Funktion dynamisch angelegt bzw. im Fall von Registern zugeteilt und verlieren ihre Gültigkeit beim Verlassen der Funktion.

3.16 Gültigkeitsbereich von Variablen

Unter C gibt es im Prinzip drei unterschiedliche Härtegrade, um die Namen von Objekten nach aussen hin zu verstecken. Zunächst sollten Sie sich stets vor Augen führen, dass jedes Objekt, dessen Name ausserhalb einer Funktion definiert wurde — dazu zählen auch die Funktionen selbst — unter C erstmal global über das gesamte Projekt hinweg namentlich erreicht werden kann. Über das Schlüsselwort `static` lässt sich eine Begrenzung des Gültigkeitsbereiches erzielen. Eine Variable ausserhalb einer Funktion wird durch `static` im Gültigkeitsbereich auf die Datei (bzw. Übersetzungseinheit) beschränkt, in der die Definition stattgefunden hat.

```
        int  a = 1 ;
static int  b = 2 ;

void f( void )
{
   static int  c = 3 ;
}
```

Die Variable `b` im vorausgegangenen Beispiel ist nur innerhalb der Übersetzungseinheit bekannt, in der dieser Code auftritt — und zwar selbstverständlich erst ab dem Ort der Definition. Die dritte Abstufung des Gültigkeitsbereiches kann dadurch erreicht werden, dass `static`–Variablen innerhalb von Funktionsrümpfen untergebracht werden. Die Lebensdauer ändert sich dadurch nicht. Eine `static`–Variable innerhalb eines Funktionsrumpfes ist und bleibt dauerhaft über die Lebenszeit der Task! Sie lässt sich jedoch nicht ausserhalb der Funktion namentlich ansprechen.

Im Beispiel kann demnach die Variable `a` von anderen Dateien aus mittels `extern int a;` referenziert werden. Auch hier sollte nochmals darauf hingewiesen werden, dass es die Sprache C nicht stört, wenn der Datentyp der Referenz nicht mit dem der Definition in einer anderen Datei korrespondiert — Ihr Programm wird dann zwar nicht mehr statbil laufen, aber es handelt sich dann auf der Compilerseite nur um Unwissenheit und auf Ihrer Seite um echte Dummheit. Es hat sich in der C-Welt die Angewohnheit durchgesetzt, eine Includedatei mit sämtlichen Variablenvereinbarungen zu erstellen.

```
#ifndef Extern
#define Extern extern
#endif

Extern int  var1     ;
Extern int  var2 = 1 ;
```

Durch eine derartige Konstruktion ist gewährleistet, dass Definitionen und Referenzen immer zusammenpassen. Lediglich eine einzige Datei — meist die, in der `main()` enthalten ist — muss vor dem Aufruf der Includedatei die Zeile `#define Extern` enthalten und schon entfallen viele Stunden unnützen Debuggens. CREST–C ist im Gegensatz zu anderen Compilern recht kumpelig in Bezug auf Initialisierungen hinter `extern`–Vereinbarungen. Etwa die Hälfte der mir bekannten Compiler insistiert beim Auftreten des Gleichheitszeichens (wie z.B. bei `var2 = 1`) darauf, dass das Schlüsselwort `extern` wohl eher nicht ernst gemeint sein kann. Die Folge besteht dann darin, dass man für jede Datei, die eine solche Konstruktion benutzt hat, beim Linken einen Fehler für eine doppelt aufgetretene Definition eingeschenkt bekommt. Die andere Hälfte der C–Compiler (CREST–C gehört dazu) beschliesst in solchen Fällen, dass dann wohl eher die Initialexpression überflüssig ist und betrachtet die Variable als Referenz.

3.17 Zugriffs–Modifizierer

Die Schlüsselworte `const` und `volatile` sind erst mit dem ANSI–C–Standard in den Sprachumfang eingegliedert worden. Sie bieten dem Anwender die Möglichkeit, die Zugriffsrechte auf Objekte einzugrenzen und den Compiler zu einer bestimmten Form der Objekt–Verwaltung zu zwingen.

Als `const` definierte Objekte können nach der Initialisierung nicht mehr verändert werden. Dauerhafte Objekte können so im EPROM untergebracht werden und belegen keine RAM–Kapazität des Rechners. Gerade Tabellen, die während der Programmverlaufs unverändert bleiben sollen, gehören schlicht einmalig ins EPROM, statt jeder Task eine eigene Kopie in der `.data`–Section zuzuteilen.

Als konstant vereinbarte Objekte werden von CREST–C in die `.text`–Section des Programmes gelegt. Hier liegt auch der ausführbare Maschinencode. Beachten Sie bei der Verwendung von konstanten Strings, dass derartige Objekte implizit als `const` behandelt werden und nicht mehr wie unter K&R–C noch üblich, als potentiell veränderlich gehandhabt werden. Zugriffe der Art:

```
"Konstanter String"[3] = 'S' ;
```

führen bei geladenen Programmen *quasi* zu selbstmodifizierendem Code, denn der Zugriff langt mitten in die `.text`–Section und kann dort beliebigen Ärger anrichten. Vorzugsweise wird unter CREST–C der Einstieg in die nachfolgende Funktion dezent kaputtgeschrieben und dann gibt es auch im RAM `WRONG OPCODE`'s oder ähnlich anheimelnde Effekte. Im EPROM bekommen Sie für derartige Schandtaten von der CPU einen kurzen und bündigen `BUS ERROR` eingeschenkt. Um klarzumachen, was Sie niemals machen dürfen, sollten Sie sich die folgenden Beispiele gut einprägen und tunlichst derartige Aktionen unterlassen.

```
strcat( "Gib ihm ", "die Kante"        ) ;
strcpy( "und noch", " eine Breitseite" ) ;
sscanf( "%s", str ) ;
```

Der Modifizierer `volatile` stellt in dieser Hinsicht eher das absolute Gegenteil dar. Es verbietet dem Compiler bei seinen Optimierungen einen bestimmten Wert eines Objektes als bekannt vorauszusetzen. Es ist immer dann sinnvoll, wenn ein Programm entwickelt werden soll, bei dem sich der Wert einer bestimmten Speicherstelle ohne direkten Zusammenhang mit dem ablaufenden Programm ändern kann. Ein Beispiel dafür ist die Programmierung von I/O–Bausteinen, bei denen es durchaus Sinn machen kann, mehrmals einen identischen Wert auf die gleiche Adresse zu schreiben. Auch Schleifen, bei denen auf die Antwort eines Peripherie–Gerätes oder einer Interrupt–Routine gewartet wird, sollten tunlichst unter Verwendung dieses Schlüsselwortes entworfen werden.

3.18 Sections unter CREST–C

CREST–C verwaltet fünf verschiedene Sections, in denen Daten abgelegt und angesprochen werden können. Beim Begriff *Section* handelt es sich schlicht um eine begriffliche Festlegung, mittels derer sich unterschiedliche — aber jeweils logisch zusammenhängende — Speicherbereiche mit unterschiedlichen Eigenschaften sprachlich eindeutig trennen lassen.

1. `.text`–Section
2. `.data`–Section
3. `.bss`–Section
4. `.common`–Section
5. `.local`–Section

In den folgenden Abschnitten soll die Bedeutung der einzelnen Sections erläutert werden. Weiterhin soll vorgestellt werden, in welcher Form sich die Sections von CREST–C auf Hochsprachen– und Assemblerebene ansprechen lassen.

3.18.1 Die `.text`–Section

Die `.text`–Section beinhaltet unter CREST–C den unveränderlichen Anteil dessen, was ein Programm ausmacht. Wenn Sie ein Programm entwickeln, so besteht eine grundsätzliche Zweiteilung aus Speicherbereichen, in denen unveränderliche Daten (wie z.B. der Maschinencode, den die CPU ausführen soll) und veränderliche Daten (wie z.B. Ihre Variablen) untergebracht werden müssen. Der CREST–C–Compiler fasst die Anweisungen für die CPU in der `.text`–Section zusammen und betrachtet derartigen Maschinencode grundsätzlich als unveränderlich.

Weiterhin können in der `.text`–Section auch konstante Werte und Tabellen mittels des ANSI–C–Schlüsselwortes `const` abgelegt werden. Beachten Sie bei der Definition von Initialwerten, dass der resultierende Maschinencode nur dann EPROM–fähig ist, wenn alle Adressen im EPROM bereits zur Linkzeit bekannt sind. Es dürfen in konstanten Initialwerten deshalb grundsätzlich keine Adressen von Objekten abgelegt werden, die aus der `.data`–, `.bss`– oder `.local`–Section stammen, da diese erst zur Laufzeit dynamisch ermittelt werden und deshalb **niemals** zur Linkzeit zur Verfügung stehen.

```
       const int  a = 0 ;
static const int  b = 4 ;
void f( void )
```

```
{
   extern const int  c     ;
   static const int  d = 3 ;

   /*
    *  Der Maschinencode von f() liegt in der .text-Section
    */
}
```

Halten Sie bei der Verwendung des Schlüsselwortes `const` die korrekte Schreibweise. Im folgenden Beispiel sind drei Pointer vereinbart. Nur die Objekte `a` und `c` werden in der `.text`–Section abgelegt und sind somit *echte* Konstanten. Das `const` vor der Variable `b` bezieht sich auf die Speicherzelle `0x812`, die so gegen Schreibzugriffe geschützt ist und **nicht** auf den Pointer `b` selbst.

```
      int  * const  a = (void*)0x802 ;
const int  *        b = (void*)0x812 ;
const int  * const  c = (void*)0x822 ;
void f( void )
{   a = (void*)0 ;  /* FALSCH */
    ++a ;           /* FALSCH */
    *a = 1 ;

    b = (void*)0 ;
    ++b ;
    *b = 1 ;        /* FALSCH */

    c = (void*)0 ;  /* FALSCH */
    ++c ;           /* FALSCH */
    *c = 1 ;        /* FALSCH */
}
```

3.18.2 Die `.data`–Section

Die `.data`–Section umfasst die dauerhaften Variablen, die innerhalb Ihrer C–Programme einen Anfangswert zugewiesen bekommen sollen. Die hier gespeicherten Objekte liegen im RAM–Bereich und sind veränderlich. Auf C–Ebene werden alle dauerhaften Variablen in der `.data`–Section abgelegt, denen ein Anfangswert zugewiesen wurde — dazu zählen auch dauerhafte Variablen, die explizit (und unnützerweise) mit einem Null–Muster initialisiert wurden!

Dauerhafte Objekte werden in der `.data`–Section allokiert, wenn ein Initialwert zugewiesen wurde.

```
                int  a = 0 ;
static          int  b = 4 ;
       volatile int  c = 0 ;
static volatile int  d = 4 ;
void f( void )
{
   extern int  x     ; // liegt im .data- oder .bss-Bereich
   static int  y = 3 ;
}
```

3.18.3 Die .bss–Section

Die .bss–Section umfasst die dauerhaften Variablen, die innerhalb Ihrer C–Programme keinen expliziten Anfangswert zugewiesen bekommen haben und folglich implizit mit einem Null–Muster initialisiert werden. Die hier gespeicherten Objekte liegen im RAM–Bereich und sind veränderlich. Auf C–Ebene werden alle dauerhaften Variablen in der .bss–Section abgelegt, denen kein Anfangswert zugewiesen wurde.

Dauerhafte Objekte werden in der .bss–Section allokiert, wenn kein Initialwert zugewiesen wurde.

```
                int  a ;
static          int  b ;
       volatile int  c ;
static volatile int  d ;
void f( void )
{
   extern int  x ; // liegt im .data- oder .bss-Bereich
   static int  y ;
}
```

3.18.4 Die .common–Section

Die .common–Section umfasst die dauerhaften Variablen, die taskübergreifend erreichbar sein sollen. Es lassen sich keine Initialwerte an Variablen der .common–Section zuweisen. Nach dem Laden des Programmes (bei RAM–Code) oder dem Hochlaufen von RTOS–UH (bei EPROM–Code) enthalten die Objekte dieser Section ein Null–Muster. Die hier gespeicherten Objekte liegen im RAM–Bereich und sind veränderlich. Auf C–Ebene werden alle dauerhaften Variablen, die mittels der Speicherklasse absolute angelegt wurden, in der .common–Section abgelegt.

```
       absolute int  a ;
static absolute int  b ;
void f( void )
{
   extern absolute int  c ; // liegt im .common-Bereich
   static absolute int  d ;
}
```

3.18.5 Die .local–Section

Die .local–Section umfasst die dauerhaften Variablen, die taskintern erreichbar sein sollen. Es lassen sich keine Initialwerte an Variablen der .local–Section zuweisen. Nach dem Laden des Programmes (bei RAM–Code) oder dem Hochlaufen von RTOS–UH (bei EPROM–Code) enthalten die Objekte dieser Section ein Null–Muster. Die hier gespeicherten Objekte liegen im RAM–Bereich und sind veränderlich. Auf C–Ebene werden alle dauerhaften Variablen, die mittels der Speicherklasse local angelegt wurden, in der .local–Section abgelegt.

```
       local int  a ;
static local int  b ;
void f( void )
{
   extern local int  c ; // liegt im .local-Bereich
   static local int  d ;
```

```
}
```

Kapitel 4

Der interne Assembler

Der Assembler liegt als integraler Bestandteil des CCC vor. Der ehemals eigenständige Assembler wurde bei den Umstellungen auf die Release 2.000 aus dem CREST–C–Paket entfernt — oder besser ausgedrückt: komplett in den Compiler integriert!

Sie sollten keine zu hohen Erwartungen an den Assembler stellen. Er versteht und assembliert die Syntax, die der Compiler generiert: mehr nicht! Es lag nicht in meiner Absicht, einen Makro–Assembler zu schreiben. Sie sollten den Einbau–Assembler als das betrachten, was er ist: die letzte Phase des Compilers.

Die Notation des 68K–Assemblers entspricht der von Motorola vorgestellten Syntax für MC68020–Prozessoren und deren Nachfolger. Die alte MC68000–Syntax wird nicht unterstützt.

Die wesentlichste Differenz zum RTOS–UH–Assembler besteht im völlig differierenden Ausgabeformat. Während der RTOS–UH–Assembler direkt ladbare S–Records generiert, erzeugt der Assembler binäre Objektfiles, die erst noch zu S–Records gelinkt werden müssen.

4.1 Die `.text`–Section

Der Assembler erwacht mit der Vorgabe, Objektcode für die `.text`–Section zu generieren. Wenn Sie nur CPU– oder FPU–Befehle kodieren wollen, so ist das Umschalten auf eine andere Section demnach redundant. Befinden Sie bei der Abarbeitung eines Assemblerfiles in einer anderen Section, so können Sie mittels der Anweisung `.CODE` in die `.text`–Section wechseln. Innerhalb der `.text`–Section sind folgende Assembleranweisungen zulässig:

- Die Definition von Labels
- Die Kodierung von CPU– und FPU–Befehlen
- Die Initialisierung von Konstanten mittels `.DC.x`
- Die Initialisierung von konstanten Bereichen mittels der Blockanweisung `.DCB.x`.

Die globale Definition eines Symboles aus der `.text`–Section erfolgt mittels der Anweisung `.CODE_DEF name`. Dadurch wird das Symbol `name` für den Linker auch ausserhalb der gerade bearbeiteten Übersetzungseinheit global bekannt.

Eine Referenz auf ein Objekt in der `.text`–Section einer fremden Übersetzungseinheit kann mittels der Anweisung `.CODE_REF name` erzeugt werden. Referenzen auf Funktionen **müssen** mittels `FUNC_REF` erzeugt werden!

Die Objekte der .text–Section lassen sich ausschliesslich PC–relativ oder absolut ansprechen. Die Variable code_section wäre auf Assemblerebene folglich als (code_section.W,PC) oder (code_section.L,PC) adressierbar. Entsprechend sind Funktionen nur absolut als JSR func oder PC–relativ als JSR (func.W,PC) bzw. JSR (func.L,PC) anzusprechen. Ebenso sind implizite relative Adressierungsarten wie BSR.B func, BSR.W func oder BSR.L func zulässig.

4.2 Die .data–Section

Die globale Definition eines Symboles aus der .data–Section erfolgt mittels der Anweisung .DATA_DEF name. Dadurch wird das Symbol name für den Linker auch ausserhalb der gerade bearbeiteten Übersetzungseinheit global bekannt. Innerhalb der .data–Section sind folgende Assembleranweisungen zulässig:

- Die Definition von Labels
- Die Initialisierung von Konstanten mittels .DC.x
- Die Initialisierung von konstanten Bereichen mittels der Blockanweisung .DCB.x.

Eine Referenz auf die .data–Section einer fremden Übersetzungseinheit kann mittels der Anweisung .XREF name erzeugt werden. In diesem Falle geht der CLN davon aus, dass das betreffende Symbol sich in der .data–, oder .bss–Section befindet und löst die Referenzen entsprechend der aufgefundenen Definition des Symbols auf.

Die Objekte der .data–Section lassen sich ausschliesslich relativ zum Adressregister A5 ansprechen. Die Variable data_section wäre auf Assemblerebene folglich als (data_section.W,A5) oder (data_section.L,A5) adressierbar.

4.3 Die .bss–Section

Die globale Definition eines Symboles aus der .bss–Section erfolgt mittels der Anweisung .BSS_DEF name. Dadurch wird das Symbol name für den Linker auch ausserhalb der gerade bearbeiteten Übersetzungseinheit global bekannt. Innerhalb der .bss–Section sind folgende Assembleranweisungen zulässig:

- Die Definition von Labels
- Die Definition von Null-initialisierten Bereichen mittels der Blockanweisung .DS.x.

Eine Referenz auf die .bss–Section einer fremden Übersetzungseinheit kann mittels der Anweisung .DATA_REF name erzeugt werden. In diesem Falle geht der CLN davon aus, dass das betreffende Symbol sich in der .data–, oder .bss–Section befindet und löst die Referenzen entsprechend der aufgefundenen Definition des Symbols auf.

Die Objekte der .bss–Section lassen sich ausschliesslich relativ zum Adressregister A5 ansprechen. Die Variable bss_section wäre auf Assemblerebene folglich als (bss_section.W,A5) oder (bss_section.L,A5) adressierbar.

4.4 Die `.common`–Section

Die globale Definition eines Symboles aus der `.common`–Section erfolgt mittels der Anweisung `.COMMON_DEF name`. Dadurch wird das Symbol `name` für den Linker auch ausserhalb der gerade bearbeiteten Übersetzungseinheit global bekannt. Innerhalb der `.common`–Section sind folgende Assembleranweisungen zulässig:

- Die Definition von Labels
- Die Definition von implizit Null-initialisierten Bereichen mittels der Blockanweisung `DS.x`.

Eine Referenz auf die `.common`–Section einer fremden Übersetzungseinheit kann mittels der Anweisung `.COMMON_REF name` erzeugt werden.

Die Objekte der `.common`–Section lassen sich ausschliesslich mittels absoluter Adressierung ansprechen. Die Variable `common_section` wäre auf Assemblerebene folglich als `common_section.W` oder `common_section.L` adressierbar.

4.5 Die `.local`–Section

Die globale Definition eines Symboles aus der `.local`–Section erfolgt mittels der Anweisung `.LOCAL_DEF name`. Dadurch wird das Symbol `name` für den Linker auch ausserhalb der gerade bearbeiteten Übersetzungseinheit global bekannt. Innerhalb der `.local`–Section sind folgende Assembleranweisungen zulässig:

- Die Definition von Labels
- Die Definition von implizit Null-initialisierten Bereichen mittels der Blockanweisung `.DS.x`.

Eine Referenz auf die `.local`–Section einer fremden Übersetzungseinheit kann mittels der Anweisung `.LOCAL_REF name` erzeugt werden.

Die Objekte der `.local`–Section lassen sich ausschliesslich relativ zum Adressregister `A4` ansprechen. Die Variable `local_section` wäre auf Assemblerebene folglich als `(local_section.W,A4)` oder `(local_section.L,A4)` adressierbar.

4.6 System–Traps

Die System–Traps von RTOS–UH stehen als Einbaukommandos im Assembler zur Verfügung. Die verwendete Nomenklatur und der generierte Code sind den Tabellen 4.1 und 4.2 zu entnehmen. In Hinsicht auf deren Bedeutung und Anwendung sei auf das RTOS–UH–Manual verwiesen.

Trap	Hex–68K	Hex–PPC
.ACT	0xA014	0x3BC0442844000002
.ACTEV	0xA01A	0x3BC0443444000002
.ACTEVQ	0xA056	0x3BC044AC44000002
.ACTQ	0x4E40	0x3BC0408044000002
.CACHCL	0xA05C	0x3BC044B844000002
.CLOCKASC	0xA068	0x3BC044D044000002
.CON	0x4E42	0x3BC0408844000002
.CONEV	0xA01C	0x3BC0443844000002
.CONEVQ	0xA05A	0x3BC044B444000002
.CONQ	0xA050	0x3BC044A044000002
.CSA	0xA03A	0x3BC0447444000002
.DATEASC	0xA066	0x3BC044CC44000002
.DCDERR	0xA06C	0x3BC044D844000002
.DELTST	0xA052	0x3BC044A444000002
.DISAB	0xA034	0x3BC0446844000002
.DVDSC	0xA012	0x3BC0442444000002
.ENAB	0xA032	0x3BC0446444000002
.ENTRB	0xA076	0x3BC044EC44000002
.ERROR	0xA002	0x3BC0440444000002
.FETCE	0x4E48	0x3BC040A044000002
.FREEB	0xA074	0x3BC044E844000002
.GAPST	0xA00E	0x3BC0441C44000002
.IMBS	0xA03C	0x3BC0447844000002
.INTD1	0xA04C	0x3BC0449844000002
.IOWA	0xA00A	0x3BC0441444000002
.IROFF	0x4E4F	0x3BC040BC44000002
.ITBO	0xA006	0x3BC0440C44000002
.ITBS	0xA048	0x3BC0449044000002
.LEAVB	0xA078	0x3BC044F044000002
.LITRA	0xA036	0x3BC0446C44000002
.LITRAV	0xA038	0x3BC0447044000002
.MD2B60	0xA046	0x3BC0448C44000002
.MSGSND	0xA070	0x3BC044E044000002
.PENTR	0x4E4B	0x3BC040AC44000002
.PIT	0xA02E	0x3BC0445C44000002

Tabelle 4.1: Notation der System–Traps Teil 1

Trap	Hex–68K	Hex–PPC
.POT	0xA030	0x3BC0446044000002
.PREV	0xA022	0x3BC0444444000002
.PREVQ	0xA054	0x3BC044A844000002
.QDPC	0x4E43	0x3BC0408C44000002
.QSA	0xA01E	0x3BC0443C44000002
.RBCLOCK	0xA06A	0x3BC044D444000002
.RCLOCK	0xA03E	0x3BC0447C44000002
.RCLOCK50	0xA086	0x3BC0450C44000002
.RELCE	0x4E49	0x3BC040A444000002
.RELEA	0x4E47	0x3BC0409C44000002
.REQU	0x4E46	0x3BC0409844000002
.RESRB	0xA072	0x3BC044E444000002
.RETN	0x4E4C	0x3BC040B044000002
.RSTT	0xA04A	0x3BC0449444000002
.RUBBL	0xA020	0x3BC0444044000002
.RWSP	0xA02A	0x3BC0445444000002
.SBCLOCK	0xA05E	0x3BC044BC44000002
.SCAN	0x4E45	0x3BC0409444000002
.SUSP	0xA028	0x3BC0445044000002
.TERME	0xA000	0x3BC0440044000002
.TERMEQ	0xA058	0x3BC044B044000002
.TERMI	0x4E41	0x3BC0408444000002
.TERV	0xA010	0x3BC0442044000002
.TIAC	0xA016	0x3BC0442C44000002
.TIAC50	0xA07C	0x3BC044F844000002
.TIACQ	0xA024	0x3BC0444844000002
.TIACQ50	0xA07E	0x3BC044FC44000002
.TICON	0xA018	0x3BC0443044000002
.TICON50	0xA080	0x3BC0450044000002
.TICONQ	0xA04E	0x3BC0449C44000002
.TICONQ50	0xA082	0x3BC0450444000002
.TIRE	0xA02C	0x3BC0445844000002
.TIRE50	0xA084	0x3BC0450844000002
.TOQ	0x4E4D	0x3BC040B444000002
.TOV	0x4E4E	0x3BC040B844000002
.TRIGEV	0xA026	0x3BC0444C44000002
.TRY	0xA07A	0x3BC044F444000002
.WFEX	0xA06E	0x3BC044DC44000002
.WSBS	0xA00C	0x3BC0441844000002
.WSFA	0xA008	0x3BC0441044000002
.WSFS	0xA004	0x3BC0440844000002
.XIO	0x4E4A	0x3BC040A844000002

Tabelle 4.2: Notation der System–Traps Teil 2

Kapitel 5

Der Linker cln

Beim CLN handelt es sich um ein Werkzeug zum Linken von Programmen. Er stellt die Endstufe des Verarbeitungsprozesses dar, der die aus den Quelltexten erzeugten Objektdateien eines Projektes mit einem Startupfile versieht und die benötigten Bibliotheksroutinen hinzubindet. Als Resultat liefert der CLN wahlweise frei ladbare oder an feste Addressen gebundene S–Records.

Bei Fehlbedienungen oder Aufruf ohne Parameter gibt der CLN einen kurzen Überblick über die unterstützten Optionen aus. Sie können diese Usage–Meldungen im Anhang (cln68k: Abschnitt C.3; cln-ppc: Abschnitt C.4) nachschlagen.

Als Aufrufparameter erwartet der CLN minimal die Angabe einer Linkdatei. Die Linkdatei muss eine Liste der zu bindenden Startupdatei, Objektdateien und Bibliotheken enthalten. Für ein Projekt `TEST`, das aus der Quelltextdatei `test.c` entstehen soll und das Verhalten einer Task auf einem CPU32–Zielsystem an den Tag legen soll, sieht der beispielhafte Aufbau einer Linkdatei `test.lnk` wie folgt aus:

```
tstart.obj
test.obj
std3fast.lib
```

Erfolgt die Angabe der zu bindenden Datei ohne Verwendung absoluter Pfadangaben, so ist ein optionaler Suchpfad über Umgebungsvariablen oder als Kommandozeilen–Option angebbar (siehe Abschnitt5.1.13).

Als optionaler zweiter Parameter wird der Name der zu generierenden Ausgabedatei erwartet. Fehlt dieser, so bildet der CLN diesen automatisch aus dem Namen der Linkdatei durch Ersetzung der Dateiextension durch die Endungen `.sr` bzw. `.epr`. Die Endung `.epr` wird verwendet, wenn in der Kommandozeile die Option `-T=address` angegeben wird und soll darauf hinweisen, dass es sich bei der generierten Datei nicht um einen *normal ladbaren* S–Record handelt.

5.1 Die Optionen des cln

5.1.1 Ausgaben während der Linkerlaufes

Die Option `-V` veranlasst den CLN, die gerade von ihm durchgeführten Aktionen auf dem Bildschirm auszugeben.

5.1.2 Erzeugung einer .map–Datei

Die Option `-M` generiert eine Mapdatei, deren Name sich aus dem Ausgabefilenamen durch die Ersetzung der Extension mit `.map` ergibt. Die Mapdatei enthält sowohl die Grössen der Sections (siehe Abschnitt 3.18) als auch deren relative bzw. absolute Lage auf dem Zielsystem. Weiterhin sind dort bei gleichzeitiger Verwendung der Option `-z` Informationen über den Stackbedarf der einzelnen Funktionen und Tasks aufgelistet.

5.1.3 Vorgabe von Programm–Namen

Der Name einer Task oder eines Shellmodules lässt sich vom CLN mittels der Option `-N=name` beeinflussen. Dies setzt allerdings zwingend voraus, dass das Projekt eine der mitgelieferten Startupdateien `tstart.obj` oder `sstart.obj` bzw. deren Ableger in der Linkdatei enthält. Als Defaultname ist innerhalb der Startupdateien der Name `X` vorgegeben. Es wird angeraten, die Namen von Shellmodulen in Gross–Schreibweise anzugeben.

5.1.4 Vorgabe der Stackgrösse

Über die Option `-S=size` wird der CLN angewiesen, die Startupdatei so zu verändern, dass der gebundenen Task oder dem Shellmodul `size` KiloBytes an Stack zur Verfügung gestellt werden. Ohne Verwendung dieser Option werden 16kB Stack eingerichtet.

5.1.5 Autostartfähige Task

Unter Verwendung der Startupdatei `tstart.obj` ist es möglich, mittels der Option `-E`, die resultierende Task autostartfähig zu machen.

5.1.6 Taskpriorität

Unter Verwendung der Startupdatei `tstart.obj` ist es möglich, mittels der Option `-Q=prio`, die resultierende Task mit der angegebenen Priorität `prio` anzulegen.

5.1.7 Residenter Taskworkspace

Unter Verwendung der Startupdatei `tstart.obj` ist es möglich, mittels der Option `-U`, die resultierende Task mit residentem Taskworkspace auszustatten.

5.1.8 Erweiterter Taskkopf

Unter Verwendung der Startupdatei `tstart.obj` ist es möglich, mittels der Option `-H=size`, die resultierende Task mit residentem Taskworkspace auszustatten. Der Wert von `size` darf 0x1FFE–Bytes nicht überschreiten.

5.1.9 Tasks für PROM vorbereiten

Sollen C–Programme, die mittels `tstart.obj` gelinkt wurden, später mit dem Bedienbefehl `PROM` bearbeitet werden, so ist es zwingend erforderlich, die Option `-O` zu verwenden.

5.1.10 Verwendung der FPU

Programme, die die Fliesskommaeinheit verwenden, müssen zwingend mittels der Option `--fpu` gelinkt werden, da ansonsten eine notwendige Modifikation der verwendeten Startupdatei unterbleibt. Ohne diesen Eingriff unterbleibt das Retten und Restaurieren der Fliesskommaregister der betreffenden Task oder des Shellmodules bei einem Taskwechsel.

5.1.11 Zahl der FILE–Strukturen

Mittels der Option `-F=files` lässt sich die Zahl der gleichzeitig in Verwendung befindlichen *filehandles* für I/O–Operation einer Task oder eines Shellmoduls beeinflussen. Zulässig Werte sind 0 und Angaben grösser 3. Ein Programm, das mittels `-F=0` gelinkt wurde, kann keinerlei I/O mehr über Bibliotheksroutinen durchführen. Selbst Ein- bzw. Ausgaben über *stdin*, *stdout* und *stderr* sind danach unmöglich. Bei `-F=3` stehen nur diese drei Kanäle zur Verfügung. Das Anwenderprogramm ist jedoch nicht mehr in der Lage, über `open()` bzw. `fopen()` Dateien oder Schnittstellen zu öffnen. Der Defaultwert von 32 ermöglicht es C–Programmen, pro Task maximal 29 eigene Dateien gleichzeitig offen zu halten.

5.1.12 Vorgabe von Ladeadressen

Der CLN ermöglicht unter Verwendung der Option `-T=address`, das resultierende Programm fest auf die Adresse `address` zu binden.

Verwendet das zu linkende Programm Variablen der Speicherklasse `absolute` und liegt das Programm auf dem Zielsystem in *read-only*–Speicher, so ist zudem die Verwendung der Option `-C=address` zwingend notwendig. Mittels dieser Adressvorgabe wird ein Modul benötigter Grösse im RAM–Bereich angelegt. Dieses Modul erhält standardmässig einen auf sechs Buchstaben eingekürzten Namen, der aus dem Prefix # und dem Task– bzw. Shellmodulnamen gebildet wird. Die Option `-R=name` erlaubt es, einen eigenen (maximal 6 Zeichen langen) Namen für das RAM–Modul vorzugeben.

Wenn mehrere unabhängige Projekte einen gemeinsamen RAM–Bereich nutzen wollen, der mittels `absoluter` Variablen verwaltet werden soll, so ist es bei Verwendung der Option `-C=address` zwingend notwendig, die Einrichtung des Variablen–Moduls nur bei einer der beteiligten Tasks zuzulassen. Die Option `-B` unterdrückt die Erzeugung einer Scheibe zur Einrichtung des Moduls.

Mittels der Option `-h` lässt sich der CLN anweisen, den Gebrauch von `absoluten` Variablen in Projekten zu kontrollieren und mit einer Fehlermeldung abzulehnen.

5.1.13 Suchpfade für Startup und Bibliotheken

Der CLN sucht die in der Linkdatei aufgeführten Dateien — sofern keine absoluten Dateinamen angegeben sind — relativ zum aktuellen Workingdirectory.

Kann dort auf die angegebene Datei nicht zugegriffen werden, so wird für 68K–System die Umgebungsvariable `CCC_68K_LIBS` ausgewertet. Für PPC–Systeme lautet der Name der Variable `CCC_PPC_LIBS`. Wo die Umgebungsvariablen vorbesetzt werden können, ist in Abschnitt 2.1 erläutert.

Die Vorgabe der Umgebungsvariable läßt sich mittels der Option `-L=path` übersteuern.

5.1.14 Debuginformationen exportieren

Die Option `-z` weist den CLN an, Debuginformationen, die in den Objektdateien enthalten sind, in eine binäre Debugdatei zu exportieren.

Kapitel 6

Der Linker lnk

Beim LNK handelt es sich um ein Werkzeug zur Erzeugung von binären Bibliotheken.

Bei Fehlbedienungen oder Aufruf ohne Parameter gibt der LNK einen kurzen Überblick über die unterstützten Optionen aus. Sie können diese Usage–Meldungen im Anhang (lnk68k: Abschnitt C.5; lnkppc: Abschnitt C.6) nachschlagen.

Als Aufrufparameter erwartet der LNK minimal die Angabe einer Linkdatei. Die Linkdatei muss eine Liste der zusammenzufassenden Objektdateien und Bibliotheken enthalten.

Als optionaler zweiter Parameter wird der Name der zu generierenden Ausgabedatei erwartet. Fehlt dieser, so bildet der LNK diesen automatisch aus dem Namen der Linkdatei durch Ersetzung der Dateiextension durch die Endung `.lib`.

6.1 Die Optionen des lnk

6.1.1 Ausgaben während der Linkerlaufes

Die Option `-V` veranlasst den LNK, die gerade von ihm durchgeführten Aktionen auf dem Bildschirm auszugeben.

6.1.2 Erzeugung einer `.map`–Datei

Die Option `-M` generiert eine Mapdatei, deren Name sich aus dem Ausgabefilenamen durch die Ersetzung der Extension mit `.map` ergibt. Die Mapdatei enthält sowohl die Grössen der Sections (siehe Abschnitt 3.18) als auch deren relative bzw. absolute Lage in der Bibliothek.

6.1.3 Suchpfade für Bibliotheken

Der LNK sucht die in der Linkdatei aufgeführten Dateien — sofern keine absoluten Dateinamen angegeben sind — relativ zum aktuellen Workingdirectory.

Kann dort auf die angegebene Datei nicht zugegriffen werden, so wird für 68K–System die Umgebungsvariable `CCC_68K_LIBS` ausgewertet. Für PPC–Systeme lautet der Name der Variable `CCC_PPC_LIBS`. Wo die Umgebungsvariablen vorbesetzt werden können, ist in Abschnitt 2.1 erläutert.

Die Vorgabe der Umgebungsvariable läßt sich mittels der Option `-L=path` übersteuern.

6.1.4 Debuginformationen exportieren

Die Option `-z` weist den LNK an, Debuginformationen, die in den Objektdateien enthalten sind, in die `.lib`–Datei zu exportieren.

Kapitel 7

Der Linker ssl

Beim SSL handelt es sich um ein Werkzeug zur Erzeugung von Shared Libraries.

Bei Fehlbedienungen oder Aufruf ohne Parameter gibt der SSL einen kurzen Überblick über die unterstützten Optionen aus. Sie können diese Usage–Meldungen im Anhang (ssl68k: Abschnitt C.7; sslppc: Abschnitt C.8) nachschlagen.

Als Aufrufparameter erwartet der SSL minimal die Angabe einer Linkdatei. Die Linkdatei muss eine Liste der zusammenzufassenden Objektdateien und Bibliotheken enthalten.

Als optionaler zweiter Parameter wird der Name der zu generierenden Ausgabedatei erwartet. Fehlt dieser, so bildet der SSL diesen automatisch aus dem Namen der Linkdatei durch Ersetzung der Dateiextension durch die Endung `.ssl`. Diese Datei enthält einen S–Record, der später auf dem Zielsystem im EPROM abgelegt werden muss.

Weiterhin generiert der SSL eine Datei mit der Endung `.ref`, die in binärer Form die Informationen über den Aufbau der erzeugten *Shared Library* enthält. Diese Datei muss beim Linken von Programmen, die die *Shared Library* nutzen wollen, in der Linkdatei angegeben werden, die der CLN verwendet.

7.1 Die Optionen des ssl

7.1.1 Ausgaben während der Linkerlaufes

Die Option `-V` veranlasst den SSL, die gerade von ihm durchgeführten Aktionen auf dem Bildschirm auszugeben.

7.1.2 Erzeugung einer `.map`–Datei

Die Option `-M` generiert eine Mapdatei, deren Name sich aus dem Ausgabefilenamen durch die Ersetzung der Extension mit `.map` ergibt. Die Mapdatei enthält sowohl die Grössen der Sections (siehe Abschnitt 3.18) als auch deren relative bzw. absolute Lage auf dem Zielsystem.

7.1.3 Vorgabe von Ladeadressen

Der SSL verlangt zwingend die Verwendung der Option `-T=address`, um die resultierende Shared-Library fest auf eine Adresse `address` zu binden.

7.1.4 Suchpfade für Startup und Bibliotheken

Der SSL sucht die in der Linkdatei aufgeführten Dateien — sofern keine absoluten Dateinamen angegeben sind — relativ zum aktuellen Workingdirectory.

Kann dort auf die angegebene Datei nicht zugegriffen werden, so wird für 68K–System die Umgebungsvariable `CCC_68K_LIBS` ausgewertet. Für PPC–Systeme lautet der Name der Variable `CCC_PPC_LIBS`. Wo die Umgebungsvariablen vorbesetzt werden können, ist in Abschnitt 2.1 erläutert.

Die Vorgabe der Umgebungsvariable läßt sich mittels der Option `-L=path` übersteuern.

7.1.5 Debuginformationen exportieren

Die Option `-z` weist den SSL an, Debuginformationen, die in den Objektdateien enthalten sind, in die `.ref`–Datei zu exportieren.

Kapitel 8

Der Library–Manager clm

Beim CLM handelt es sich um die rudimentäre Implemetierung eines Library–Managers.

Bei Fehlbedienungen oder Aufruf ohne Parameter gibt der CLM einen kurzen Überblick über die unterstützten Optionen aus. Sie können diese Usage–Meldungen im Anhang (clm68k: Abschnitt C.9; clmppc: Abschnitt C.10) nachschlagen.

Kapitel 9

Der Objekt–Inspektor cop

Beim COP handelt es sich um ein Werkzeug zur Betrachtung von binären Files des CREST–C–Paketes.

Bei Fehlbedienungen oder Aufruf ohne Parameter gibt der COP einen kurzen Überblick über die unterstützten Optionen aus. Sie können diese Usage–Meldungen im Anhang (cop68k: Abschnitt C.11; copppc: Abschnitt C.12) nachschlagen.

Kapitel 10

Das cmake–Utility

Das unter UNIX weitverbreitete make–Tool erlaubt die leichte Bearbeitung komplexer Projekte. Es ermöglicht die Modularisierung Ihrer Programme, ohne dass Sie dabei die Übersicht über die Einzelkomponenten Ihres Gesamtprojektes verlieren. Ohne grossen Aufwand für den Programmierer hält CMAKE die Bestandteile Ihres Projektes stets auf dem neuesten Stand. Da gerade dieser Verlust an Transparenz des Projektes viele Programmierer vor der Unterteilung Ihrer Programme in wartbare Einzelkomponenten zurückschrecken lässt, entschärft CMAKE ein wesentliches Problem, das der modularen Programmierung oft genug im Wege steht.

Bei Fehlbedienungen oder Verwendung der Option `-?` gibt der CMAKE einen kurzen Überblick über die unterstützten Optionen aus. Sie können diese Usage–Meldungen im Anhang (Abschnitt C.13) nachschlagen.

Die Funktionsweise aller MAKE–Tools ist im Prinzip recht trivial. In einer beliebig benamten Datei (üblicherweise *Makefile*) wird notiert, **welche** Dateien des Projektes **wann** und **wie** bei Änderungen neu zu erzeugen sind.

Für die korrekte Funktionsweise von CMAKE ist eine korrekt gesetzte Uhrzeit auf dem Entwicklungsrechner und aller beteiligten Dateien eines Projektes von essentieller Bedeutung.

10.1 Die Optionen des cmake

CMAKE erwartet als ersten Aufrufparameter die (optionale) Angabe eines Makefiles. Fehlt diese Angabe, so wird versucht, die Datei `Makefile` zu öffnen. Ist diese ebenfalls nicht vorhanden, so generiert CMAKE eine Fehlermeldung.

Vor der Verarbeitung des explizit angegebenen Makefiles oder der impliziten Vorgabe `Makefile` versucht CMAKE, eine Default–Datei zu öffnen und zu verarbeiten. Diese kann dazu dienen, projekt– oder systemspezifische Vorgaben zu hinterlegen. Dabei sucht CMAKE zunächst im Workingdirectory und anschliessend im Wurzelverzeichnis nach der Default–Datei. Es wird je nach Betriebssystem mit den dort üblichen Namenskonventionen vorgegangen, die Sie der Tabelle 10.1 entnehmen können.

Reihenfolge	UNIX	WINDOWS NT	RTOS–UH
1.	`./.cmakerc`	`MAKE.INI`	`make.ini`
2.	`/.cmakerc`	`\MAKE.INI`	*/XX/*`make.ini`

Tabelle 10.1: Namensgebung bei CMAKE–Initialfiles

Die Option `-V` dient dazu, einige Ausgabemeldungen von CMAKE zu aktivieren. So zeigt das CMAKE z.B. die Namen der automatisch gelesenen Dateien und deren Includedateien an.

Mit der Option `-A` lässt sich CMAKE dazu zwingen, alle *Ziel*–Dateien unabhängig von deren Aktualität erneut zu erstellen.

Mittels der Option `-I` wird CMAKE angewiesen, die Verarbeitung des Makefiles unabhängig von Fehlermeldungen bei der Ausführung von *Aktionen* fortzusetzen.

Die Option `-T` erlaubt es, CMAKE in den Tracemodus zu versetzen. Die auszuführenden *Aktionen* werden angezeigt, aber nicht ausgeführt.

Mittels der Option `-#macro[=replace]` lassen sich CMAKE–Makros über die Kommandozeile vorgeben. Eine explizite Beschreibung der *Makros* entnehmen Sie bitte dem Abschnitt 10.3.4

Die weiteren Optionen dienen zur Definition von Kommandozeilen–Makros und sind im Abschnitt 10.3.4.2 näher beschrieben.

10.2 Das erste Makefile

An Hand eines Beispiels soll die grundlegende Notation dargestellt werden. Ein Projekt `test`, bestehend aus den Dateien `test.c` und `test.h`, soll mittels eines Makefiles den ladbaren S–Record `test.sr` erstellen:

```
test.sr: test.c test.h
   ccc68k test.c   test.obj
   cln68k test.lnk test.sr -N=TEST
```

Werden bei Dateiangaben innerhalb von Makefiles keine absoluten Pfade verwendet, so expandiert CMAKE automatisch das aktuelle Workingdirectory vor die betreffenden Dateinamen.

In diesen wenigen Zeilen sind die drei wesentlichen Komponenten zur Erstellung von korrekten Makefiles zu erkennen. In der **ersten Spalte beginnend** wird das *Ziel* des Makelaufes notiert — hier: `test.sr`. Dahinter befindet sich **in der gleichen Zeile**, durch einen **Doppelpunkt** abgetrennt, eine optionale Liste der **Abhängigkeiten** — hier: `test.c test.h` —, bei deren Änderung eine *Aktion* gestartet werden soll. Die *Aktion* — hier: Compiler– und Linkerlauf — besteht aus einer oder mehreren Zeilen von ausführbaren Anweisungen, die CMAKE aufrufen soll. Die Zeilen mit den Aktionen müssen mit einem oder mehreren Blanks oder Tabs eingeleitet werden, dürfen also **nicht in der ersten Spalte beginnen**.

CMAKE sucht bei der Bearbeitung eines Makefiles nach dem zuerst notierten *Ziel* und kontrolliert dessen Existenz und gegebenfalls das letzte Änderungsdatum dieser Datei. Danach verfährt CMAKE auf gleiche Art und Weise mit den *Abhängigkeiten*. Existiert das *Ziel* nicht oder ist eine von den *Abhängigkeitsdateien* jüngeren Ursprungs, so werden die darunter notierten *Aktionen* von CMAKE eingeleitet, um das *Ziel* neu zu erstellen. In diesem Fall werden Compiler und Linker direkt hintereinander aufgerufen, um den gewünschten S–Record zu generieren. Kommt es beim Ausführen von *Aktionen* zu Fehlern, so bricht CMAKE standardmässig mit einer Fehlermeldung ab.

Um die weitere Funktionalität von CMAKE zu demonstrieren, wird das Beispiel nun etwas erweitert, indem dem Projekt nun noch die Dateien `test1.c` und `test1.h` hinzugefügt werden sollen, wobei `test1.h` nur von `test1.c` includiert werden soll, `test1.c` jedoch zusätzlich `test.h` includiert:

```
test.sr: test.c test.h test1.c test1.h
   ccc68k test.c   test.obj
   ccc68k test1.c  test1.obj
   cln68k test.lnk test.sr -N=TEST
```

Sie sehen, dass es ungeschickt ist, das Makefile in dieser Form zu notieren, da bei Änderungen in `test1.h` nun auch `test.c` stets unnötigerweise neucompiliert würde. Es bietet sich deshalb an, die *Abhängigkeiten* folgendermassen aufzuspalten:

```
test.sr: test.obj test1.obj
   cln68k test.lnk test.sr -N=TEST

test.obj: test.c test.h
   ccc68k test.c test.obj

test1.obj: test1.c test.h test1.h
   ccc68k test1.c test1.obj
```

Eine Änderung von `test1.h` führt nunmehr lediglich zur Compilierung vom `test1.c` und zum erneuten Linken des Projektes. Sie sehen, dass die Erstellung des Projektes jetzt von Existenz undÄnderungszeitpunkt der Objektdateien abhängt, statt wie bisher von den Quelltextdateien. Für die *Abhängigkeiten* wird nämlich ebenfalls kontrolliert, ob die entsprechenden Dateien

1. existieren und
2. aktuell sind.

Sind die Abhängigkeitsdateien nicht vorhanden, so muss im Makefile eine *Regel* zu deren Erstellung notiert sein. Eine solche *Regel* besteht aus den bereits bekannten drei Komponenten *Ziel*, *Abhängigkeiten* und *Aktionen*.

Ist eine Datei in den *Abhängigkeiten* bereits vorhanden und es existiert eine *Regel*, so wird diese *Regel* zunächst ausgewertet und kontrolliert, ob deren *Ziel* noch aktuell ist. Da dieser Mechanismus rekursiv abläuft, ist bei korrekter Angabe der *Abhängigkeiten* und erfolgreicher Ausführung aller *Aktionen* stets sichergestellt, dass bei Änderungen im Projekt alle erforderlichen Schritte zur Aktualisierung des ersten *Zieles* in der Makedatei automatisch abgespult werden.

10.3 Syntax von cmake

Nachdem im letzten Abschnitt die grundlegende Funktionalität von CMAKE erläutert wurde, sollen die folgenden Ausführungen Ihnen dabei behilflich sein, mittels CMAKE lesbare und portable Makefiles für ihre Projekte zu erstellen.

10.3.1 Kommentare

CMAKE ermöglicht die Angabe von Zeilenkommentaren innerhalb der Makefiles. Ein Doppelkreuz # in der ersten Spalte leitet einen Kommentar ein, der am Ende einer Zeile endet.

```
#
# Makefile für Projekt "test"
#
# Erstellungsdatum: 05/11/1999
# Programmierer   : Stephan Litsch
#
# Letzte Änderung : ---
#
test.sr: test.c test.h
```

```
   ccc68k test.c   test.obj
   cln68k test.lnk test.sr -N=TEST
```

10.3.2 Zeilenfortsetzung

Da die Syntax von CMAKE bei der Angabe von *Ziel* und *Abhängigkeiten* zwingend vorschreibt, dass diese **in einer Zeile** angegeben werden, kann ein Makefile bei einer langen Liste von Abhängigkeiten schnell unlesbar werden.

CMAKE ermöglicht zu diesem Zweck die Fortsetzung von Zeilen mittels des von C bekannten Backslashes mit direkt darauf folgendem Zeilenumbruch.

```
#
# Demonstration von Fortsetzungszeilen
#
test.sr: test.c  test.h \
         test1.c test1.h
   ccc68k test.c   test.obj
   ccc68k test1.c  test1.obj
   cln68k test.lnk test.sr -N=TEST
```

10.3.3 Wildcards

In einem Projekt mit vielen Dateien wächst die Zahl der *Regeln* schnell an. Unterscheiden sich die *Regeln* dabei nur durch die verwendeten Dateinamen, so lassen sich diese mittels *Wildcards* kompakter notieren. Als Beispiel sei das Projekt aus Abschnitt 10.2 um die Dateien `test2.c` bis `test9.c` sowie die Includedateien `test2.h` bis `test9.h` erweitert. Jede der neuen C–Dateien soll dabei sowohl `test.h` als auch die Headerdatei mit dem identischen Basisnamen includieren. Zudem soll `test.c` in diesem Fall von allen existierenden Headerdateien abhängig sein:

```
#
# Arbeiten mit Wildcards
#
test.sr: test.obj                      \
         test1.obj test2.obj test3.obj \
         test4.obj test5.obj test6.obj \
         test7.obj test8.obj test9.obj
   cln68k test.lnk test.sr -N=TEST

*.obj: *.c test.h *.h
   ccc68k *.c *.obj

test.obj: test.c test.h                \
          test1.obj test2.obj test3.obj \
          test4.obj test5.obj test6.obj \
          test7.obj test8.obj test9.obj
   ccc68k test.c test.obj
```

In CMAKE wird das Multiplikationszeichen `*` als *Wildcardzeichen* eingesetzt. CMAKE versucht bei der Auswertung der *Abhängigkeiten* von `test.sr` eine `Regel` zu jeder *Abhängigkeit* zu finden. Trifft CMAKE auf eine Regel mit *Wildcard*, so versucht es, das *Wildcardzeichen* durch Substitution des Basisnamens der untersuchten *Abhängigkeit* zu ersetzen.

Bei der Suche nach *Regeln* für *Abhängigkeitsdateien* wird die Liste der *Regeln* im Makefile von unten nach oben durchsucht. Die erste *Regel*, die hier überprüft würde, wäre folglich die für `test.obj` und erst danach die für `*.obj`. Sie sollten bei der Verwendung von *Wildcards* also stets darauf achten, Spezialfälle, die nicht durch *Wildcardregeln* behandelt werden sollen, **hinter** der *Wildcardregel* zu notieren.

10.3.4 Makros

Der Umgang mit *Makros* erleichert die Erstellung von allgemeingültigeren Makefiles. Um z.B. ein Projekt sowohl für 68K– als auch für PowerPC–Zielsysteme zu compileren, wären ohne die Verwendung von *Makros* zwei unterschiedliche Makefiles zu erstellen und zu pflegen.

Am folgenden Beispiel soll die Verwendung von *Makros* demonstriert werden:

```
#
# Projekt für 68K und PPC
#
CCC = ccc68k
CLN = cln68k
EXT = 68k

#CCC = cccppc
#CLN = clnppc
#EXT = ppc

test.sr: test.c test.h
   $(CCC) test.c test.obj
   $(CLN) test$(EXT).lnk test.sr -N=TEST
```

Die Definition vom *Makros* erfolgt **ab der ersten Spalte** einer Zeile. Der zu ersetzende *Makrobezeichner* wird mit einem **Gleichheitszeichen** = abgeschlossen. Es folgt ein optionaler *Ersatztext* **in der gleichen Zeile**.

Die Anwendung des *Makros* erfolgt, indem der *Makrobezeichner* mit `$( )` umgeben wird. Die oben aufgeführte Regel wird demnach folgendermassen expandiert:

```
test.sr: test.c test.h
   ccc68k test.c test.obj
   cln68k test68k.lnk test.sr -N=TEST
```

Nach dem Entfernen der Kommentare vor den PPC–spezifischen Makros erhält man folgendes Resultat:

```
test.sr: test.c test.h
   cccppc test.c test.obj
   clnppc testppc.lnk test.sr -N=TEST
```

Sie erkennen daran auch, dass eine erneute Definition eines bereits bekannten *Makros* nicht zu einer Fehlermeldung führt, sondern die alte Definition kommentarlos übersteuert.

Das angeführte Beispiel ist allerdings noch nicht praxistauglich, da es sich nicht zur parallelen Erstellung beider Konfigurationen eignet, und soll nun so erweitert werden, dass die erzeugten Dateien projektspezifisch in den Ordnern `obj68k` und `objppc` abgelegt werden:

```
#
# Projekt für 68K und PPC
```

```
#
EXT = 68k
#EXT = ppc

CCC    = ccc$(EXT)
CLN    = cln$(EXT)
OUTPUT = obj$(EXT)$(S)

$(OUTPUT)test.sr: test.c test.h
   $(CCC) test.c $(OUTPUT)test.obj
   $(CLN) test$(EXT).lnk $(OUTPUT)test.sr -N=TEST
```

Bei der Definition der *Makros* für Compiler, Linker und Ausgabeverzeichnis sehen Sie zunächst, dass die Anwendung von *Makros* auch bei der Definition neuer *Makros* verwendet werden kann.

Weiterhin ist bei der Definition von `OUTPUT` das Einbau–Makros `$(S)` verwendet worden, das CMAKE automatisch zu einem gültigen Pathseparator des jeweiligen Entwicklungssystems expandiert. Verwenden Sie deshalb weder Backslash noch Slash bei der Notation von Pfaden. Resultat dieser kleinen Bemühung ist ein Makefile, das sich sowohl auf UNIX– als auch auf WINDOWS–Entwicklungsumgebungen korrekt verhält.

10.3.4.1 Einbau–Makros

CMAKE stellt neben dem im letzten Abschnitt angesprochenen *Makro* `$(S)` je nach Entwicklungsumgebung weitere *Einbau–Makros* zur Verfügung, um so auf einfache Art innerhalb des Makefiles systemspezifische Auswertungen vornehmen zu können. Für die drei unterstützten Plattformen sind dies:

Makro	Betriebssystem
`__RTOSUH__`	RTOS–UH
`__LINUX__`	LINUX
`__NT__`	WINDOWS

Tabelle 10.2: CMAKE–Einbau–Makros

Die aufgeführten Einbau–Makros waren in früheren Versionen von CMAKE undokumentiert ohne die umgebenden doppelten Underlines implementiert. Die alte Schreibweise ist mit Erscheinen dieser Dokumentation abgekündigt und wird ohne weitere Ankündigung in späteren Versionen von CMAKE verschwinden. Sie sollten alte Makefiles deshalb auf die offizielle Schreibweise umstellen.

10.3.4.2 Spezielle Kommandozeilen–Makros

Über die Kommandozeile lassen sich einige *Makros*, die in der CREST–C–Umgebung sinnvoll einsetzbar sind, mittels spezieller Optionen definieren.

Die aufgeführten Kommandozeilen–Makros waren in früheren Versionen von CMAKE undokumentiert ohne die umgebenden doppelten Underlines implementiert. Die alte Schreibweise ist mit Erscheinen dieser Dokumentation abgekündigt und wird ohne weitere Ankündigung in späteren Versionen von CMAKE verschwinden. Sie sollten alte Makefiles deshalb auf die offizielle Schreibweise umstellen.

Option	Makro
-0	__MC68000__
-2	__MC68020__
-3	_CPU32_
-K	__M68K__
-P	__MPPC__
--fpu	_FPU__
-A	__MAKEALL__

Tabelle 10.3: CMAKE–Kommandozeilen–Makros

10.3.5 Präprozessor

Beim Parsen der Makefiles wird ein kleiner Präprozessor eingesetzt, der folgenden Befehlssatz versteht:

- `!if macro`
- `!if ! macro`
- `!else`
- `!elif macro`
- `!elif ! macro`
- `!endif`
- `!include filename`
- `!error text`
- `!message text`

10.3.5.1 Bedingte Ausführung des Makefiles

Die Kommandos `!if` und `!elif` gestatten es, durch Abfrage der Existenz eines *Makros* Teile des Makefiles bedingt auszuführen. Die Abfrage lässt sich durch Verwendung eines `!` vor dem *Makro* invertieren.

```
#
# Projekt für 68K und PPC unter Verwendung von Makros
#
!if __M68K__
EXT = 68k
!else
EXT = ppc
!endif

CCC    = ccc$(EXT)
CLN    = cln$(EXT)
OUTPUT = obj$(EXT)$(S)

$(OUTPUT)test.sr: test.c test.h
   $(CCC) test.c $(OUTPUT)test.obj
   $(CLN) test$(EXT).lnk $(OUTPUT)test.sr -N=TEST
```

Auf diese Art lässt sich — gesteuert über den Aufruf in der Kommandozeile — automatisch das korrekte Projekt erzeugen. `cmake -0K` generiert Code für den MC68000; `cmake -P` erzeugt das identische Projekt für einen PowerPC als Zielsystem.

Die Kommandos `!else` und `!endif` entsprechen in ihrer Funktionalität den C–Pendants.

10.3.5.2 Includieren von Dateien

Das Kommando `!include` erlaubt es, Dateien innerhalb eines Makefiles einzulesen. Der Dateiname ist dabei in doppelten Hochkommata anzugeben. Es existieren keinerlei Standardsuchpfade. Bei Verwendung relativer Pfadangaben versucht CMAKE zunächst die einzufügende Datei im dem Verzeichnis zu öffnen, in dem auch die aktuell includierende Datei liegt. Gelingt dies nicht, so wird ein Zugriff im aktuellen Workingdirectory versucht.

```
#
# Beispiel einer includierenden Datei
#
!include "test.def"
```

10.3.5.3 Ausgeben vom Meldungen

Die Kommandos `!message text` und `!error text` ermöglichen es, informative Meldungen bzw. Fehlermeldungen in einem Makefile zu generieren. In beiden Fällen wird der nachfolgende Text bis zum Zeilenende ausgegeben.

```
#
# Beispiel von (Fehler-)Meldungen
#
!message Ich bin eine informative Meldung
!error Und ich bin ein echter Fehler
```

Kapitel 11

Der Post–Mortem–Dump pmd

Wer viel programmiert, programmiert viel Mist. Wenn ein C–Programm mal wieder mit einem hämischen `BUS ERROR` die Beine durchstreckt, reicht ein Blick in die Register und die Assemblerlistings meist aus, um den Fehler zu lokalisieren und zu beheben.

Der Post–Mortem–Dump funktioniert — wie der Name erraten lässt — sinnvoll nur mit *toten* Programmen. Laufende Programme sollten Sie damit nicht untersuchen. Es besteht jedoch nicht die Gefahr, dass der PMD bei solchen Aktionen Unfug im System anrichtet. Leider ist die Aussage über die Registerinhalte einer laufenden Task furchtbar redundant. Sie sollten demnach die zu untersuchende Task zunächst suspendieren, wenn RTOS–UH diese Aktion nicht schon durchgeführt hat. Danach liefert der Aufruf `PMD taskname` genauere Informationen über den aktuellen Zustand der Task.

Je nach Art des zu untersuchenden Programmes gibt PMD unterschiedlich reichhaltige Information auf dem Schirm aus. In folgenden Fall wurde ein suspendierter Editor mit dem PMD betrachtet.

```
PC: 00085FC4   4CDF... MOVEM.L (SP)+,D7/A1
Shell-Module   at 00077AD2 PCrel=0000E4F2 CED
Task           at 001FFEE4 TWSP =001FF800 CED/0C                Line: 0000
               SR: 00000000   ttsm0xnzvc                      SSP: 000007F6
    0        1        2        3        4        5        6        7
D0: 0000006A 00000020 00000000 00004055 00000001 00004055 00000001 03E1B6B8
A0: 001FFE12 001FFEE4 001FFE12 001C1576 001FF800 001C1442 04005D9C 001E6054
```

Mittels PMD lassen sich so gesammelt die wichtigsten Informationen übersichtlich auf dem Schirm darstellen, die man sonst mit einer Reihe von RTOS–UH–Befehlen und Rechenoperationen einzeln ermitteln müsste. Die hier untersuchte Task `CED/0C` basiert auf dem C–Shellmodul `CED`, das an der Adresse `$00077AD2` geladen wurde. Der aktuelle `PC` steht auf Position `$00085FC4`. Um Ihnen das Rechnen zu ersparen, gibt der PMD auch noch die relative `PC`–Position innerhalb des geladenen Moduls aus: `$0000E4F2`. Mit diesem Wert können Sie mittels der von Linker erzeugten `.MAP`–Files das File und die Position innerhalb der Datei bestimmen, in der sich der `PC` gerade aufhält. Bei Programmen, die den Zeilencounter der Task setzen, gibt der Eintrag `Line` über dessen Inhalt Auskunft. Wie Sie erkennen, gehört der CED nicht zu den Programmen, bei dem dieser Eintrag Sinn machen würde. Zuletzt wird an der aktuellen `PC`–Position der Code der nächsten zu exekutierenden Zeile disassembliert.

Um die Dekodierung des aktuellen Statusregisterinhaltes zu erleichtern, wird dieses sowohl in Hex– als auch in *Klartext*–Darstellung angezeigt. Hier sind sämtliche Bits inaktiv. Ist in der Zeichenkette `ttsm0xnzvc` ein Zeichen als Grossbuchstabe eingetragen, so bedeutet dies, dass das entsprechende Bit in der Maske gesetzt ist. Die Zahl in der Mitte des Strings gibt den IR–Level des Prozessors an. Der Prozessor läuft also auf IR–Level 0 im Usermode — wie man es bei einem Editor wohl auch erwarten kann.

Bei Programmen, die mit FPU–Unterstützung laufen, werden auch die Registerinhalte des Coprozessors angezeigt. Das folgende Beispiel stellt einen Schnappschuss während eines kleinen Benchmarkprogrammes dar.

```
PC: 0009DDCE   4CDF... MOVEM.L (SP)+,D7/A1
Shell-Module   at 0009C45C PCrel=00001972 TEST
Task           at 001F4098 TWSP =001F3E7A TEST/20               Line: 0000
               SR: 00000010   ttsm0Xnzvc                     SSP: 000007F6
    0        1        2        3        4        5        6        7
D0: 00000001 00000000 00000020 00005B76 00001800 00000004 00000013 03F62FFC
A0: 001BA04A 001F4098 001F40FA 001BA17D 001F3E7A 001BA048 04005D9C 001BFB00
FP:  1.43333333E+0001  4.18604651E-0001  1.61111111E+0000  2.38888889E+0000
FP:  4.00000000E+0000  1.66666667E-0001  7.16666667E+0000  4.18604651E-0001
               FPSR: 00000008  n-z-i-nan
```

Kapitel 12

Bibliotheken von CREST–C

Gerade in Hinsicht auf die Erzeugung kompakter Zielprogramme werden die Bibliotheken des CREST–C–Paketes mit unterschiedlichen Übersetzungsparametern für die verschiedenen Prozessorreihen ausgeliefert.

fast-Libs	long-Libs	stack-Libs	debug-Libs	stackdebug-Libs
std0fast.lib	std0long.lib	std0stack.lib	std0debug.lib	std0stackdebug.lib
flt0fast.lib	flt0long.lib	flt0stack.lib	flt0debug.lib	flt0stackdebug.lib
ffp0fast.lib	ffp0long.lib	ffp0stack.lib	ffp0debug.lib	ffp0stackdebug.lib

Tabelle 12.1: Verfügbare 68000er–Bibliotheken

fast-Libs	long-Libs	stack-Libs	debug-Libs	stackdebug-Libs
std3fast.lib	std3long.lib	std3stack.lib	std3debug.lib	std3stackdebug.lib
flt3fast.lib	flt3long.lib	flt3stack.lib	flt3debug.lib	flt3stackdebug.lib
ffp3fast.lib	ffp3long.lib	ffp3stack.lib	ffp3debug.lib	ffp3stackdebug.lib

Tabelle 12.2: Verfügbare CPU32–Bibliotheken

fast-Libs	long-Libs	stack-Libs	debug-Libs	stackdebug-Libs
std2fast.lib	std2long.lib	std2stack.lib	std3debug.lib	std3stackdebug.lib
flt2fast.lib	flt2long.lib	flt2stack.lib	flt2debug.lib	flt2stackdebug.lib
ffp2fast.lib	ffp2long.lib	ffp2stack.lib	ffp2debug.lib	ffp2stackdebug.lib
fpu2fast.lib	fpu2long.lib	fpu2stack.lib	fpu2debug.lib	fpu2stackdebug.lib

Tabelle 12.3: Verfügbare 68020er–Bibliotheken

fast-Libs	long-Libs	stack-Libs	debug-Libs	stackdebug-Libs
stdpfast.lib	stdplong.lib	stdpstack.lib	stdpdebug.lib	stdpstackdebug.lib
fltpfast.lib	fltplong.lib	fltpstack.lib	fltpdebug.lib	fltpstackdebug.lib
fpupfast.lib	fpuplong.lib	fpupstack.lib	fpupdebug.lib	fpupstackdebug.lib

Tabelle 12.4: Verfügbare PowerPC–Bibliotheken

Die ersten drei Buchstaben des Namens geben darüber Aufschluss, ob es sich um reine Integer–Bibliothen handelt oder welche Art von Fliesskommaarithmetik von der Bibliothek unterstützt wird.

- std: Integer–Bibliothek ohne Fliesskomma-Unterstützung
- flt: Fliesskommaemulations–Bibliothek ohne FPU–Unterstützung
- ffp: Fliesskommaemulations–Bibliothek ohne FPU–Unterstützung und FastFloatingPoint–Implementierung für den Datentyp `float`

- fpu: Fliesskomma–Bibliothek mit FPU–Unterstützung

Das vierte Zeichen des Namens beinhaltet die Prozessorreihe:

- 0: MC68000
- 2: MC68020
- 3: CPU32
- p: PowerPC

Der Anhang des Namens informiert über den unterschiedlichen Aufbau der Bibliothen, wie der Tabelle 12.5 zu entnehmen ist.

Extension	CCC**–Optionen**
`fast`	`-R=3`
`long`	`-R=[0|2] -D=1`
`stack`	`-R=[0|2] -D=1 -U`
`debug`	`-R=[0|2] -D=1 -z`
`stackdebug`	`-R=[0|2] -D=1 -U -z`

Tabelle 12.5: Übersetzungsparameter der Bibliotheken

Die `fast`–Varianten sind für Programme geeignet, deren Codebereich 32kB nicht überschreitet. Sie stellt die kompakteste Bibliotheks–Variante dar. In den `fast`–Bibliotheken sind nur 16Bit–Sprünge enthalten. Für grösserer Projekte stehen die `long`–Versionen zur Verfügung, die externe Funktionen durch 32Bit–Sprungbefehle adressieren. Selbst bei grossen Projekten, die nur wenige Bibliotheksroutinen verwenden, ist es oftmals möglich, erfolgreich mit den `fast`–Varianten zu linken — einen Versuch ist es immer wert!

Die `stack`– und `stackdebug`–Varianten sind `long`–Bibliotheken mit aktiver Stacküberwachung. Die Endung `debug` zeigt zudem an, dass in diesen Bibliotheken alle Informationen zur Verwendung des Debuggers enthalten sind.

Kapitel 13

CREST–C–Programme

CREST–C ermöglicht die Generierung unterschiedlicher Arten von Code für das Betriebssystem RTOS–UH.

- C–Shellmodule

  Die Kodierung eines Programmes als C–Shellmodul führt zu dem unter anderen Betriebssystemen geläufigen Verhalten. Es lassen sich Übergabeparameter mittels `argc` und `argv` an die Hauptfunktion `main()` übergeben und bei der Beendigung liefert das Programm einen Rückgabewert an seinen Aufrufer. Innerhalb eines C–Shellmoduls stehen alle wesentlichen Features gemäss ANSI–C zur Verfügung.

  Der Aufruf eines C–Shellmoduls führt zur Erzeugung einer eigenständigen Task. Ein mehrmaliger Aufruf des identischen C–Shellmoduls liefert eine entsprechende Anzahl unabhängiger Tasks, die völlig asynchron auf dem identischen Maschinencode ablaufen, jedoch mit unterschiedlichen Datenbereichen arbeiten.

- C–Tasks

  Bei der Kodierung eines Programmes als C–Task gibt es lediglich einen Unterschied bezüglich des Quelltextes. C–Tasks sind nicht in der Lage, Parameter über die Kommandozeile zu übernehmen oder einen Rückgabestatus an den Aufrufer zu liefern. Innerhalb einer C–Task stehen alle wesentlichen Features gemäss ANSI–C zur Verfügung.

  Jeder Aufruf einer C–Task führt zur einer Aktivierung der betreffenden Task. Wie unter RTOS–UH üblich, sind führen Mehrfach–Aktivierungen von C–Tasks zu einer Pufferung der Aktivierung, solange die betreffende Task bereits läuft. Erst nach der Beendigung der laufenden C–Task wird die gepufferte Aktivierung ausgeführt. Ein mehrfacher Aufruf einer C–Task führt demnach zu einer sequentiellen Abarbeitung der hintereinanderfolgenden Aktivierungen.

- C–Subtasks

  Bei den als Subtask bezeichneten Funktionen handelt es sich um Tasks, die dynamisch zur Laufzeit von C–Shellmodulen oder C–Tasks aus gebildet werden können. Sie erben weitesgehend die Eigenschaften des erzeugenden Prozesses und sind nur solange lebensfähig, solange die `main()`–Task, von der sie abstammen, im System existiert. Innerhalb einer C–Subtask stehen alle wesentlichen Features gemäss ANSI–C zur Verfügung.

- Systemtasks

  Als Systemtasks werden alle RTOS–UH–Tasks bezeichnet, die nicht mit einer CREST–C–Laufzeitumgebung ausgestattet sind. CREST–C ist in der Lage, diverse Sonderformen von Systemtasks zu generieren. Die Kodierung von Systemtasks wird jedoch nur für spezielle Aufga-

benstellungen angeraten und ist nicht als die Standardform eines CREST–C–Programmes zu verstehen.

- Interface–Funktionen zum Aufruf von PEARL–Funktionen

  Innerhalb von C–Shellmodulen, C–Tasks und C–Subtasks besteht die Möglichkeit, PEARL–Funktionen aufzurufen. CREST–C bietet für einen Grossteil der unter PEARL üblichen Datentypen die entsprechenden Parameter–Schnittstellen.

- C–Funktionen, die von PEARL–Tasks aufgerufen werden können

  Hier bietet sich die Möglichkeit, Unterprogramme in C zu schreiben, die von PEARL aus verwendet werden können. Es besteht innnerhalb dieser Funktionen jedoch keinerlei Unterstützung durch die CREST–C–Laufzeitumgebung!

- Interrupt– und Exception–Handler

  CREST–C erlaubt die Generierung von hardwarenahen Funktionen, die direkt auf asynchrone Ereignisse reagieren können. Die betreffenden Funktionen unterstützen dabei die unter RTOS–UH gültigen Konventionen. Eine CREST–C–Laufzeitumgebung existiert innerhalb dieser Codebereiche nicht!

- Kalt– und Warmstartcode

  Hierbei handelt es sich um Funktionen, die in der Startphase des Betriebssystems automatisch ausgeführt werden. Es steht weder die volle Funktionalität von RTOS–UH noch eine CREST–C–Laufzeitumgebung zur Verfügung.

13.1 C–Shellmodule

Ein in C kodiertes Shellmodul verhält sich für den Bediener entsprechend zu den meisten Einbaukommandos wie z.B. `COPY`. C–Shellmodule erlauben die Übergabe von Parametern an die `main()`–Funktion des Programmes und ebenso die Übermittlung eines Rückgabewertes an den Aufrufer. Die Funktion `main()` könnte demnach in der gewohnten Form

```
int main( void ) {...}
int main( int argc, char  *argv[] ) {...}
```

erfolgen. Der Rückgabewert auf Shellebene beschränkt sich dabei lediglich auf die Meldungen ”Shellmodul *ordnungsgemäss* oder *mit Fehlern* terminiert”. Bei der Funktion `exit()` ist demnach als Argument `EXIT_FAILURE` anzugeben, wenn das Shellmodul mit fehlerhaftem Status abgebrochen werden soll. Dagegen bewirkt `EXIT_SUCCESS`, dass das Shellmodul als ordnungsgemäss terminiert betrachtet wird.

13.1.1 C–Shellmodule fürs RAM

Ein mit `sstart.s` gelinktes Programm liefert einen S–Record, der an beliebiger Stelle mit `LOAD` in den RAM–Speicher geladen werden kann. Der Lader überträgt die im S–Record enthaltenen Informationen in den RAM–Speicher und löst dabei auch etwaige Referenzen, die in der Datei an den `G`– bis `Z`–Symbolen zu erkennen sind, auf. Nach dem Laden ist das C–Shellmodul in diversen Speicherketten ordnungsgemäss eingetragen und der (oder die Namen) der Shellextension sind mit dem Kommando `?` abfragbar.

Die Abbildung 13.1 zeigt in der oberen linken Ecke den Aufbau eines derartig geladenen C–Programmes.

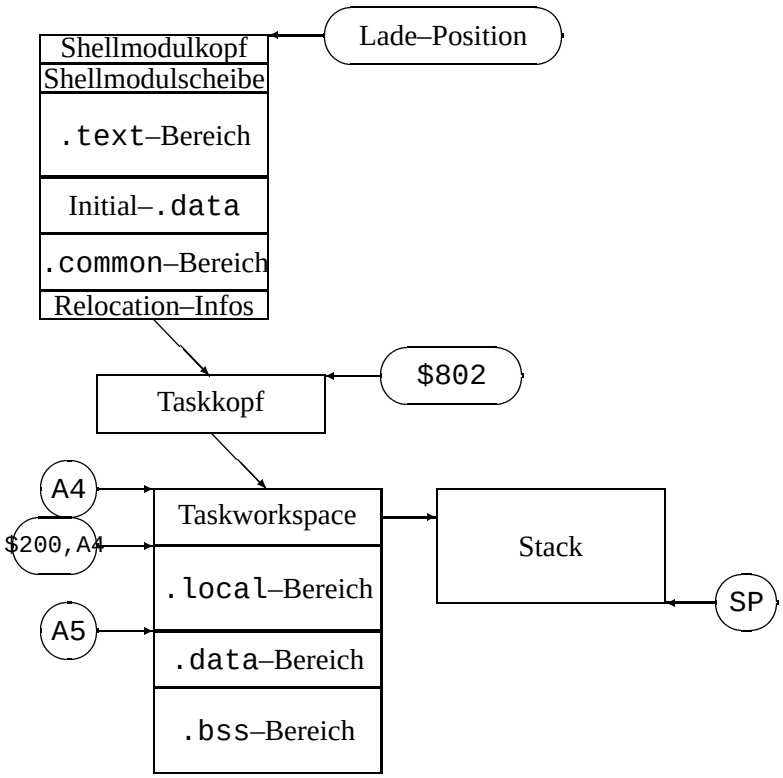


Abbildung 13.1: RAM–Shellmodul

Hinter dem Modulkopf und dem oder den Namen der Extensions beginnt der `.text`–Bereich, in dem der eigentliche Maschinencode abgelegt ist. Hier liegen auch konstante Strings und Tabellen. Dahinter folgen die Initialdaten des Programmes. Für Sie als C–Programmierer bedeutet es, dass für jede dauerhafte Variable, der Sie einen Startwert zugewiesen haben, hier der Initialwert als Konstante abgelegt wird. Es folgt — sofern Ihr Programm absolute Variablen definiert hat — ein vom Lader gelöschter Speicherbereich: der `.common`–Bereich. Es folgt eine Liste mit Informationen, die vom Startupcode dazu verwendet werden, etwaige Initialdaten im `.data`–Bereich ihres Programmes beim Start der Task korrekt aufzusetzen.

Beim Aufruf eines C–Shellmoduls generiert RTOS–UH zunächst einen Taskkopf, der mit den Informationen im Kopf des C–Shellmoduls korrespondiert. Dieser dynamische Taskkopf ist jetzt mit dem Code der Shellextension verknüpft und enthält zudem eine Reihe von Standardvorbesetzungen. Weiterhin legt die Shell einen zugehörigen Taskworkspace für die neu entstandene Task an.

Nun analysiert die Shell die Kommandozeile, die Sie hinter dem Namen der Shellextension angegeben haben. Dabei werden Sachen herausgefiltert, die auf Shellebene eine Bedeutung besitzen wie z.B. Semikolon und Doppelstriche `'--'`. Der Rest wird in einen Puffer mit maximal 255 Zeichen kopiert.

Jetzt wird erstmals der Startupcode des C–Shellmoduls in die Aktionen einbezogen. RTOS–UH unterstellt, dass es sich stets um ein PEARL–Shellmodul handelt und transferiert deshalb die Daten im für PEARL–Prozeduren üblichen Verfahren. CREST–C–Programme tarnen sich entsprechend und verhalten sich beim Anlaufen wie PEARL–Prozeduren. Der Aufbau der Übergabeparameter sieht dabei wie folgt aus:

1. Adresse DATION-Block stdin
2. Adresse DATION-Block stdout
3. Adresse DATION-Block stderr
4. Länge der Commandline

5. Pointer auf die Commandline
6. Pointer auf die Return–Zelle

Die drei DATION–Blöcke (deren Aufbau in Abbildung 13.2 dargestellt ist) enthalten das Wissen der Shell über die aktuell eingestellten Standard–Kanäle bei Aufruf der Shellextension. Interessant sind für CREST–C nur die Einträge ldn, drive und file_name. Wenn Sie selbst sich an diesen Daten vergehen möchten, dann bedenken Sie bitte, dass file_name im RTOS–UH–üblichen Format vorliegt und mittels eines Abschlussbytes mit geseztem achten Bit terminiert wird.

```
typedef struct DATION_Block
{  UWORD       io        ;
   UBYTE       ldn       ;
   UBYTE       drive     ;
   UWORD       stat      ;
   UWORD       tfu       ;
   UWORD       info      ;
   FileName    file_name ;
}  DATION_Block ;
```

Abbildung 13.2: Aufbau eines DATION–Blockes

Weiterhin sind nun ein Pointer auf die Kommandozeile und deren Länge bekannt. Wenn Sie selbst ein Startup–File kodieren wollen, so sollten Sie nicht der Versuchung erliegen, darauf gedankenlos mit C–Stringbefehlen einzuschlagen, da kein abschliessendes Nullbyte enthalten ist.

Jetzt braucht die C–Task erstmal etwas mehr Speicher. Die übergebenen Parameter wurden bislang in dem Speicherbereich untergebracht, der beim Parametertransfer angefordert wurde. Hier stand als Grösse der spätere Stackbereich, den RTOS–UH intern als PWSP verwaltet.

Problematisch wird zu diesem Zeitpunkt die nette Geste von RTOS–UH, dem Shellmodul beim Anlegen des Taskkopfes einen Taskworkspace spendiert zu haben. Schade, denn CREST–C braucht mehr Speicher, als eine übliche PEARL–Task. Da CREST–C noch viel mit diesem Taskworkspace vorhat, besteht die erste Aktion des Startup's darin, den alten TWSP durch ein etwas üppigers Exemplar zu ersetzen. Es sei dringend davon abgeraten, an diesen Stellen des Startupcodes Veränderungen vorzunehmen.

Der Speicherblock, der nun als TWSP eingeklinkt ist, enthält neben dem Platz für die RTOS–UH–internen Daten nun auch die .local, .data– und .bss–Section von CREST–C. Dabei gilt grundsätzlich, dass die .local–Section ab Adressoffset $200 beginnt.

Jetzt müssen die transferierten Parameter und die Initialdaten des Programmes in den .data–Bereich kopiert werden. Anschliessend steht der alte PWSP–Block dem C–Programm als Stack zur Verfügung.

13.1.2 C–Shellmodule fürs EPROM

Der Unterschied zwischen dem Startup fürs RAM, der im letzten Kapitel beschrieben wurde, und der EPROM–Version ist nur relativ winzig. Der Startup sstart.s enthält zusätzlich zum ladbaren Modulkopf eine als Scheibe codierte Variante.

Ein Shellmodul, das fürs EPROM erstellt werden soll, muss mit der CLN–Option -O=address gelinkt werden. Das Resultat besteht in einem S–Record, der keine Referenzen mehr enthalten darf und bei dem die Verschiebeinformationen bereits vom CLN für eben die angegebene EPROM–Adresse aufgelöst wurden.

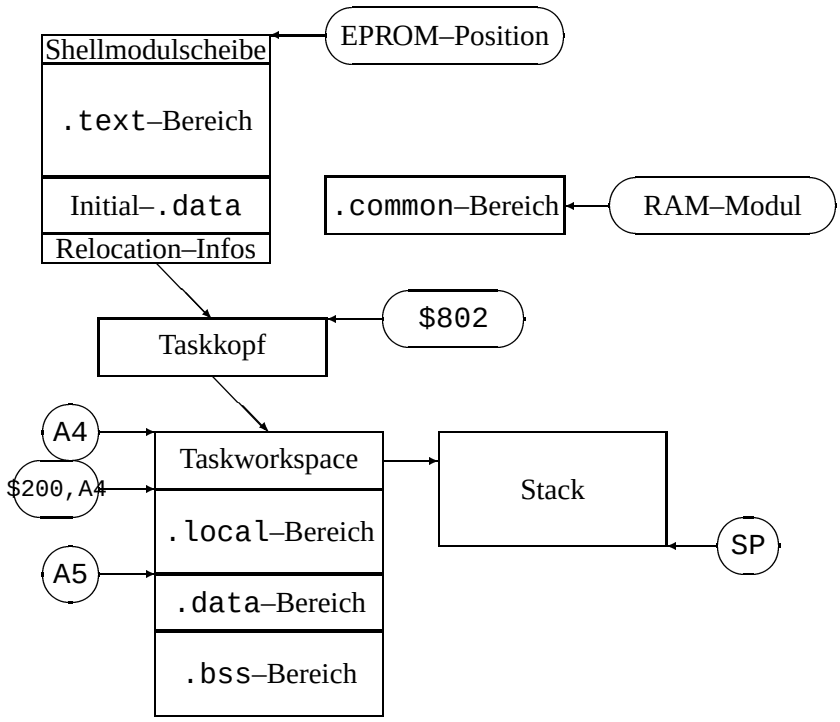


Abbildung 13.3: EPROM–Shellmodul

13.2 C–Tasks

Die Kodierung eines Programmes als C–Task hat nach aussen hin ein anderes Verhalten als das der C–Shellmodule. So lassen sich bei der Aktivierung einer Task keine Übergabeparamter an die `main()`–Funktion transferieren oder ein Rückgabestatus definieren. Ein C–Programm, das als Task ablaufen soll, sollte die Funktion `main()` als

```
void main( void ) {...}
void main( int  argc, char  *argv[] ) {...}
```

anlegen. Die Angabe des Rückgabedatentyps `int` ist nicht schädlich, hat aber auch keinerlei Auswirkung. In der zweiten Variante ist `argc` grundsätzlich gleich Eins und `argv[0]` enthält den Tasknamen. Weitere Parameter lassen sich nicht entgegennehmen.

13.2.1 C–Tasks für RAM oder EPROM

Der interne Aufbau von `tstart.s` unterscheidet sich erheblich von seinem Pendant für C–Shellmodule. `tstart.s` ist ein Zwitter. Wird ein Programm mit diesem Startup gelinkt, so existiert nach dem Laden mit `LOAD` automatisch ein Taskkopf im Speicher und braucht nicht erst dynamisch erzeugt zu werden. In der EPROM–Variante existiert entsprechender Code im Startup, um RTOS–UH zum automatischen Einrichten eines Taskkopfes bereits in der Hochlaufphase zu bewegen. Wenn das System aufgewacht ist, existiert folglich bereits ein Taskkopf für das Programm.

Der Vorteil eines statischen Taskkopfes besteht darin, dass Sie durch Vorgabe des Tasktyps residente oder autostartfähige Tasks anlegen können — wie in der Beschreibung des CLN bei den Optionen `-E` und `-U` erläutert. Weiterhin lässt sich auch eine Defaultpriorität der Task vorgeben — was bei C–Shellmodulen leider auch nicht möglich ist. Die Angabe einer Priorität sollte grundsätzlich **nie** im Startupfile vorgenommen werden, sondern mittels des CLN und der Option `-T=prio` erfolgen.

```
.DC.L   0,0                 ; RTOS-MODUL  Header
.DC.W   $01                 ; TYP: TASK
.DC.L   _MoName-*           ; Pointer auf Tasknamen
.DC.W   0                   ; Relativ-Kennung fuer Tasknamen
.DC.W   _Prio               ; Prio der Task
.DC.L   _SumSegSize         ; _LocaleSize+_DataSize+_BssSize
.DC.L   0,0
.DC.W   _Prio               ; Prio der Task
.DC.L   _main               ; START PC
.DC.L   0,0,0,0,0,0,0,0
.DC.L   0,0,0,0,0,0,0,0
```

Abbildung 13.4: Taskkopf fürs RAM und/oder EPROM

Die Beschaffung der Speicherblöcke für CREST–C fällt hier deutlich leichter als bei C–Shellmodulen, da die Grösse des `TWSP` direkt im Taskkopf anzugeben ist — und der CLN die erforderliche Grösse bereits beim Linken eintragen kann und nicht erst Laufzeitcode abgespult werden muss, um RTOS–UH davon zu überzeugen, dass PEARL und das Wissen um dessen Konventionen, reichlich überflüssig sind. Die entsprechenden Labelnamen im Startup sollten beibehalten bleiben, weil der CLN beim Linken diese Symbole zwingend benötigt, um die errechneten Daten für das zu bindende Programm korrekt an den diversen Positionen im Startupcode aufzusetzen.

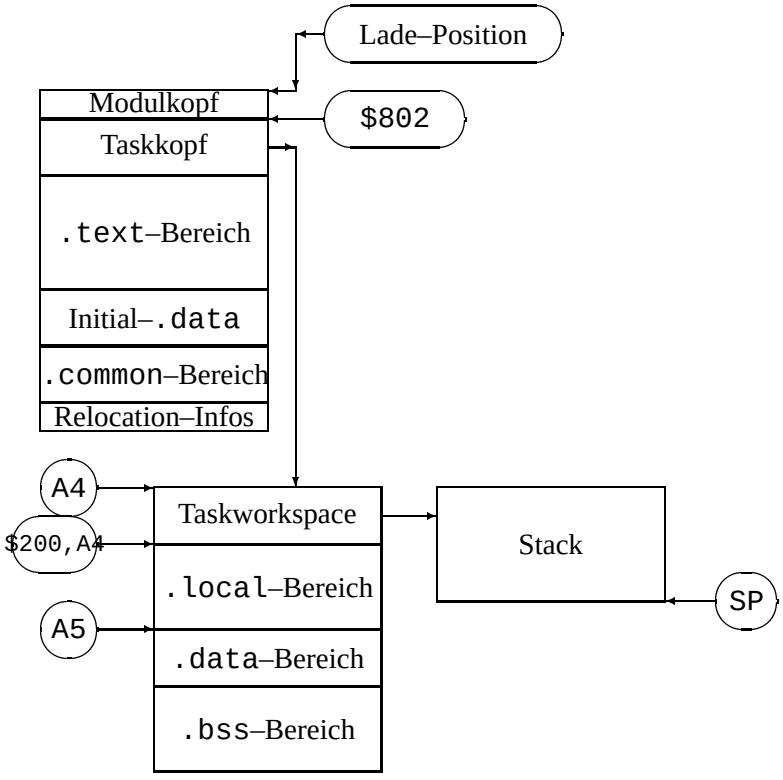


Abbildung 13.5: C–Task im RAM

Wird der Code ins EPROM gebrannt, so durchsucht das Betriebssystem seinen Scan–Bereich beim Hochlaufen nach Scheiben vom Typ `1`. Da Taskköpfe im RAM liegen müssen, wird für jede gefundenen Scheibe dieser Art aus den im EPROM aufgefundenen Informationen ein Taskkopf generiert. Den Aufbau einer solchen `SLICE-1` können Sie der Abbildung 13.6 entnehmen.

Danach besteht aus Nutzersicht keinerlei Unterschied mehr zwischen einer Task, die aus dem EPROM oder dem RAM aufgewacht ist. Eine Darstellung der Lage der Speichersektionen können Sie der Abbildung 13.7 entnehmen.

```
.SLICE  1

.DC.W   $0100         ; TYP: TASK
.DC.L   _MoName-*     ; Pointer auf Tasknamen
.DC.W   0             ; Relativ-Kennung fuer Modulnamen
.DC.W   _Prio         ; prio of task
.DC.L   _SumSegSize   ; _LocaleSize+_DataSize+_BssSize
.DC     _main-*       ; Start-PC
.DC.L   0
```

Abbildung 13.6: Taskkopf fürs EPROM

Wesentlich ist dabei die Lage der `.common`–Section, die von der Vorgabe der CLN–Option `-A=address` abhängig ist. Während beim Startup `tstart.s` die `.common`–Section bereits beim Laden hinter der `.text`–Section des Programmes vorgehalten wird, patcht der CLN bei der Verwendung der `-A`–Option den Startupcode so um, dass eine `SLICE 13` im EPROM–Code entsteht. Somit richtet RTOS–UH beim Hochlaufen automatisch ein Modul mit gelöschtem Datenbereich für die `.common`–Section ein.

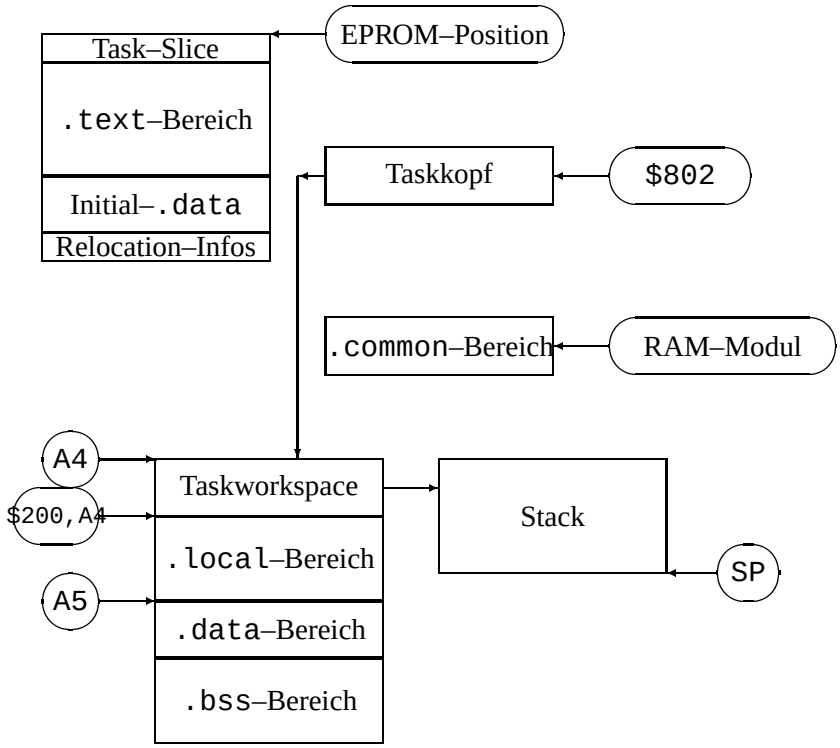


Abbildung 13.7: C–Task im EPROM

13.3 C–Subtasks

RTOS–UH zeichnet sich neben der Echtzeitfähigkeit dadurch aus, dass eine nahezu ungegrenzte Zahl verschiedenster Prozesse quasi gleichzeitig ihr Unwesen im Rechner treiben können. Mittels der bislang vorgestellten C–Shellmodule und C–Tasks bietet sich Ihnen bislang lediglich die Chance, mehrere unabhängige Prozesse anzulegen, die keinerlei direkte Kommunikationsmöglichkeiten besitzen. Durch die Shellmoduleigenschaft der CREST–C–Programme kennen sich die dynamisch generierten C–Tasks selbst dann untereinander nicht, wenn Sie auf identischen Codebereichen ablaufen. In den meisten Fällen ist das recht praktisch, denn schliesslich wollen Sie — an einem Editor sitzend — nicht

unbedingt die Auswirkungen der Kommandos Ihres Nebenmannes, der ebenfalls ein Briefchen hackt, auf Ihrem Schirm erleben. Zumindest sollte das nicht passieren können.

In anderen Fällen ist es jedoch wünschenswert, wenn zwei oder mehr Prozesse direkt miteinander kommunizieren können. ANSI–C bietet dafür leider keinerlei Ansätze. Die UNIX–Lösungen zu diesem Thema setzen den Einsatz einer MMU zwingend voraus. Alles nun Folgende ist eine implementierungsbedingte Eigenschaft von CREST–C und stellt einen Versuch dar, die Möglichkeiten von RTOS–UH sinnvoll auszunutzen ohne dabei das Sprachkonzept von C allzusehr mit Füssen zu treten.

Beim Starten von CREST–C–Programmen werden Sie schon festgestellt haben, dass Sie z.B. mehrere C–Compiler gleichzeitig laufen lassen können. Der Witz besteht darin, dass der Code des Programmes nur einmal im Speicher vorliegen muss, das System aber eine Reihe unabhängiger Prozesse generiert, die auf diesem Codebereich unabhängig voneinander arbeiten, und dass jeder dieser Prozesse seinen eigenen Datenbereich besitzt.

Wenn Sie die Notwendigkeit erblicken, innerhalb eines C–Programmes mehrere nebenläufige Tasks zu verwenden, so kann dies mittels eines `#pragma`–Kommandos bewerkstelligt werden. Sie kodieren die zukünftige C–Subtask als normale Funktion **innerhalb** Ihres Programmpaktes. Dabei können selbstverständlich auch Argumente übergeben werden. Die Rückgabedatentypen sind entweder `void`, `signed char*`, `unsigned char*` oder `Task*`.

Was spielt sich dabei genau ab? Sie rufen die zukünftige C–Subtask wie eine normale Funktion auf. Die aufrufende Funktion legt die Argumente auf den Stack, generiert einen Taskkopf für die neue C–Subtask, kopiert eine Reihe von Informationen, die der Vater an den Sohn weiterleiten möchte und erklärt das Söhnchen dann für startklar. Ob das Kind gleich losläuft, hängt direkt von der zugewiesenen Priorität ab. Sie haben jetzt einen nebenläufigen Prozess, der mit eigenem Stack und eigenen lokalen Daten arbeitet — aber auf die globalen Daten der Vater–Task zugreifen kann!

Beispiele für die Syntax bei der Verwendung des `#pragma SUBTASK`–Kommandos könnten wie folgt lauten:

```
#pragma SUBTASK PRIORITY 20 STACKSIZE 4096
#pragma SUBTASK PRIORITY 20
#pragma SUBTASK STACKSIZE 20000 PRIO 5
#pragma SUBTASK
```

Die `default`–Priorität eine C–Subtask ist 20 und die `default`–Stackgrösse der aufgesetzten Subtask beträgt 4 kB. **Die Angabe der Stackgrösse erfolgt in Bytes.**

Es sind keine Argumentdeklarationen mit offener Parameterliste, soll heissen mit den drei Pünktchen, erlaubt. Wenn Sie wirklich darauf Wert legen sollten, besteht natürlich immer noch die Möglichkeit, einen derartigen Parametertransfer mittels des von `main()` verwendeten Verfahrens zu kodieren. Die Subtask liefert dem Aufrufer auf Wunsch ihren Namen oder den Pointer auf ihren Taskkopf zurück. Wenn der Name von Interesse ist, muss lediglich der Rückgabedatentyp als Character–Pointer vereinbart werden. Der Speicherbereich, in dem der Name abgelegt wird, ist dann mittels `malloc()` auf der Aufruferseite beschafft worden und kann vom Vater der erzeugten Subtask mittels `free()` freigegeben werden.

Folgende Punkte sind beim Umgang mit C–Subtasks zu beachten:

- C–Subtasks lassen sich nur von C–Shellmodulen, C–Tasks und anderen C–Subtasks aus aufsetzen.
- Terminieren Sie die Task `main()` **niemals**, bevor alle C–Subtasks beendet sind. Sie ziehen damit den Sohnprozessen den `.text`–, `.data`– und `.bss`–Bereich unter dem Hintern weg. Die schnellste Möglichkeit zur Blockierung von `main()` besteht darin, ein schlichtes `for (;;) rt_suspend() ;` am Ende von `main()` einzubauen. Durch die Endlosschleife wären dann

auch die Spielkinder unter den Bedienern entschärft, die nur so zum Spass (oder aus purer Verzweiflung) alle möglichen und unmöglichen Tasks fortzusetzen pflegen, wenn eine Anwendung unerwartet hängen bleibt.

- Es gilt zu beachten, dass alle dauerhaften Variablen von allen nebenläufigen Prozessen erreicht werden können. Nur automatische und registerinterne Variablen sind im ausschliesslichen Besitz der jeweiligen Tasks. Objekte vom Typ `local` sind völlig getrennt von denen der Sohntasks.
- Die Stacküberwachung ist auch für C–Subtasks möglich und oft sogar sehr angebracht.
- Bei jedem Aufruf einer als Subtask generierten Funktion entsteht eine neue Subtask. Sie können nahezu beliebig viele davon im Speicher haben. Es kommt nicht zu Mehrfachaktivierungen, da grundsätzlich ein neuer Taskkopf vom System generiert und nicht nur der interne Aktivierungszähler eines vorhandenen Taskkopfes erhöht wird.

Eine derartig aufgesetzte Subtask verschwindet nach einem `return` aus dem Speicher. Sie sollten **niemals** mit `exit()` oder ähnlichen Notausstiegen eine Subtask terminieren. Die Funktionen `exit()` und `abort()` wehren sich gegen Aufrufe von Subtasks, um zu verhindern, dass `main()` durch unzulässige Aktionen seiner Kinder quasi enthauptet wird.

```
abort()_impossible_in_subtask_(TERMI)
exit()_impossible_in_subtask_(TERMI)
```

Das Beispielprogramm in Abbildung 13.8 zeigt, wie eine nahezu beliebige Anzahl von C–Subtasks erschaffen wird und in einem globalen Array Zähler erniedrigt. Wenn Sie die Quelle unter dem Namen SUBTASK compilieren, erzeugt der Aufruf `SUBTASK 10` 10 Subtasks. Eine Ausgabe–Subtask gibt jeweils drei Sekunden nach der letzten Ausgabe den aktuellen Stand der Zähler kund. `main()` überwacht die Kinder und terminiert, wenn alle erzeugten Subtasks mit der Arbeit fertig sind.

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

#define NUMBER_OF_SUBTASKS   (100)

long   max_tasks = 0               ;
long   count[ NUMBER_OF_SUBTASKS ] ;
int    continue_dump = 1           ;

#pragma SUBTASK PRIORITY 1 STACKSIZE 16000
char  *dumper( void )
{  long   t ;
   for (; continue_dump ;)
   {
      rt_resume_after( 3000 ) ; // Nach 3 Sekunden weiter
      for ( t=0 ; t<max_tasks ; ++t )
         printf( "%ld -> %ld\n", t, count[ t ] ) ;
   }
}

#pragma SUBTASK PRIORITY 20 STACKSIZE 8000
char  *counter( long  number )
{  register long  number_count = 10000 ;
   while ( number_count )
   {
      /* Globale Zelle runterzaehlen */
      count[ number ] = --number_count ;

      /* Die Subtasks machen unterschiedlich lange Mittagspausen */
```

```
      rt_resume_after( number ) ;
   }
}

void main( int  argc, char  *argv[] )
{
   long   t                              ;
   char  *sub_name[ NUMBER_OF_SUBTASKS ] ;
   char  *dump                           ;
   int    fertig, weiter                 ;

   printf( "main ist als %s angelaufen\n", argv[ 0 ] ) ;

   if ( argc == 2 ) max_tasks = atol( argv[ 1 ] ) ;

   if ( max_tasks <= 0 || max_tasks > NUMBER_OF_SUBTASKS )
      max_tasks = NUMBER_OF_SUBTASKS ;

   printf( "Es werden %ld Subtasks gestartet !\n", max_tasks ) ;

   for ( t=0 ; t<max_tasks ; ++t )
   {
      /*  Subtask starten und Namen merken */
      sub_name[ t ] = counter( t ) ;
      printf( "Subtask %s aktiviert\n", sub_name[ t ] ) ;
   }

   /*  Subtask zur zyklischen Ausgabe des count-Feldes bauen */
   dump = dumper() ;
   printf( "Subtask %s aktiviert\n", dump ) ;

   /*
    *  Jetzt warten wir, bis alle counter-Kinder tot sind.
    *  Wenn wir main() einfach terminieren, so ziehen wir den
    *  Kindern den Code und die Daten unter dem Hintern weg.
    */
   for ( weiter=1 ; weiter ; )
   {
      /* Signale waeren schoener, aber mit zyklischen Pollen
       * geht es auch
       */
      rt_resume_after( 1000L ) ;
      for ( t=0, fertig=1 ; t<max_tasks ; ++t )
      {
         if ( rt_task_status( sub_name[ t ] ) == 0xFFFFFFFFL )
         {
            /* Die Task ist schon tot */
            continue ;
         }
         else
         {
            /* Mindestens eine Counter-Task laeuft noch... */
            fertig = 0 ;
            break ;
         }
      }
      weiter = ! fertig ;
   }
```

```
   /* dann den Dumper rauswerfen... */
   for (;;)
   {  continue_dump = 0 ;
      rt_resume_after( 100L ) ;
      if ( rt_task_status( dump ) != 0xFFFFFFFFL ) continue ;
      break ;
   }
   printf( "\nFERTIG\n" ) ;
}
```

Abbildung 13.8: Beispiel zur Generierung von C–Subtasks

Sie sollten sich gut überlegen, mit welcher Priorität die Subtasks gestartet werden. Wenn `main()` mit niedriger Priorität läuft, so kann es zu dem unerwünschten Zustand kommen, dass die gerade erzeugten Kinder dem erzeugenden Programm die Prozessor–Zeit vollständig entziehen. Der Aufruf `SUBTASK PRIO 70 100` erzeugt z.B. 100 Subtasks, und `main()` wird schon in der Generierungsphase fast vollständig lahmgelegt.

13.3.1 Langlebige Subtasks

Wenn ein eigenständiger Prozess mit dem langlebigen Verhalten einer PEARL–Task dynamisch benötigt wird, so kann unter CREST–C dieser Prozess zur Laufzeit erschaffen werden.

Das kann Nachteile haben, da es z.B. nicht möglich ist, autostartfähige Shellmodule zu kodieren, die selbstständig aus dem EPROM hochkommen. Der Grund dafür liegt im bereits erwähnten und leider nicht vorhandenen Taskkopf der Shellmodule.

Wenn Sie unter CREST–C eine nebenläufige Task benötigen, dann handelt es sich in der Regel um eine recht flüchtige Angelegenheit. Die Task wird erzeugt, wenn man sie braucht und verschwindet automatisch, wenn sie ihre Aufgabe erfüllt hat. Für jeden Aufruf der als Funktion getarnten Subtask muss ein neuer Prozess aufgesetzt werden, und das kostet bekanntlich Zeit. Wie bereits erläutert, ist es aber durchaus möglich, eine Subtask etwas langlebiger zu installieren. Dazu ist nur eine einfache Einplanung der Subtask notwendig. Auch Aktivierungen, die während der normalen Lebensdauer einer solchen Subtask auflaufen, verhindern das automatische Entladen des Taskkopfes nach Abschluss des Subtask–Codes — diese Methode ist allerdings so unsicher, dass Sie sie nicht weiter in Betracht ziehen sollten. Wenn solche Mehrfachaktivierung auftreten oder ein Start auf Grund eines Ereignisses forciert wird, auf das die Subtask eingeplant wurde, führt das zu einem erneuten Start der *alten Subtask* mit den *alten Parametern*.

```
#pragma TASK
char *Test( int  a )
{
   /* Irgendwas */
   a = 17 ;
}

Task *Test1( int  a )
{
   /* Irgendwas */
   a = 18 ;
}

void main( void )
{
```

```
    char  *name1 = Test( 23 ) ; /* erzeugt z.B. Test/01 */
    char  *name2 = Test( 12 ) ; /* erzeugt z.B. Test/02 */
    Task  *tid1  = Test1( 45 ) ; /* erzeugt z.B. Test/03 */
    Task  *tid2  = Test1( 87 ) ; /* erzeugt z.B. Test/04 */

    rt_activate      ( name1, prio1 ) ; /* wie Test( 23 ) */
    rt_activate      ( name2, prio2 ) ; /* wie Test( 12 ) */
    rt_activate_quick( tid1 , prio2 ) ; /* wie Test( 45 ) */
    rt_activate_quick( tid2 , prio2 ) ; /* wie Test( 87 ) */
}
```

In dem kleinen Beispiel werden dauerhaftere Taskköpfe erzeugt. Der Aufruf in der `#pragma`–Zeile lautet dazu lediglich `TASK` anstelle von `SUBTASK`. Wie bei CREST–C üblich, werden diese C–Subtasks erst durch explizite *Funktions*–Aufrufe — hier in `main()` — erzeugt und gestartet. Für das System sind diese Tasks unter den Namen erreichbar, die im Beispiel mittels der beiden Pointer `name1` und `name2` gespeichert werden. Für Sie als Entwickler hat das den Nachteil, dass Sie nicht in der Lage sind, sich ein Büschel von Tasks anzulegen, mit denen Sie schon vor dem Start des Haupttask herumspielen können. Es gibt sie einfach noch nicht und sie entstehen erst unter Programmkontrolle. Für echte RTOS–UH–Freaks mag das zunächst als eine lästige Einschränkung erscheinen. Der Sicherheit von Programmen tut es jedoch zweifellos recht gut, wenn z.B. Tasks, die gerade keinen Sinn machen würden oder bei unsachgemässer — weil frühzeitiger — Aktivierung gar Schaden anrichten könnten, dem Spieltrieb des Anwenders komplett entzogen werden. Die bisherige Resonanz zu diesem Thema deutet darauf hin, dass diese Meinung auch von den meisten Anwendern geteilt wird...

Durch die Verwendung der `#pragma`–Anweisung `TASK` bleiben die Taskköpfe der C–Subtasks auch nach der Ausführung des `return`–Statements im System. Explizite Einplanungen oder rechtzeitige Neuaktivierungen sind jetzt nicht mehr notwendig, um den Taskkopf am Leben zu erhalten. Jede erneute Aktivierung sollte nun allerdings mittels `rt_activate()` erfolgen, da Sie ansonsten das System recht bald in einem Wust von Tasks ersticken, die sich alle hartnäckig weigern, automatisch zu verschwinden. Sie haben so auch die Möglichkeit, die `default`–Priorität der Subtask zu übersteuern und jede weitere Aktivierung mit variabler Priorität vorzunehmen.

Um solche dauerhafteren Tasks programmgesteuert zu ermorden, dient die Funktion `rt_unload_task()`. Sie wirkt wie ein `UNLOAD name` von der Shell aus, nimmt die Ausplanung der angegebenen Task vor, terminiert sie und verschrottet hinterher den Taskkopf.

13.3.2 Umgang mit Subtasks

Bei der Generierung einer Subtask in CREST–C über die `#pragma`–Anweisungen `TASK` oder `SUBTASK` werden üblicherweise die Namen der Sohnprozesse aus dem jeweiligen C–Funktionsnamen und dem auch von der Shell verwendeten Zähler zusammengesetzt. Eine Subtask `Test` wäre dann über den Namen Test/xy erreichbar. Die Zahl der eindeutig durch den Namen zu identifizierenden Tasks, die aus einer Funktion generiert werden können, ist demnach auf 256 begrenzt. Diese Angabe ist allerdings optimistisch, da sie keinen direkten Einfluss auf den internen Zähler haben, der für die Extension hinter dem Funktionsnamen verantwortlich zeichnet. Im ungünstigsten Fall ist es sogar denkbar, dass Sie fünf Subtasks erzeugen, die alle mit identischen Namen aufgesetzt werden, weil der zyklische Zähler durch Aktionen anderer Tasks inzwischen seine Runde beendet hat und wieder auf dem alten Wert angekommen ist.

Der namentliche Umgang mit einer dynamisch erzeugten Task ist demnach — wenigstens potentiell — etwas mit Vorsicht zu geniessen. Wenn Sie mehr als eine Subtask gleichzeitig aus einer C–Funktion abspalten wollen, besteht der sichere Weg, sich als Vaterprozess die `TID` der Sohntask mitteilen zu lassen. Dazu muss lediglich der Rückgabetyp der Subtask als `Task*` (ist in <`rtos.h`> vereinbart)

angegeben werden. Wenn Sie zusätzlich den Namen der Sohntask benötigen, so kann dies unter Verwendung der Funktion `rt_get_taskname()` geschehen. Sie liefert für eine gültige `TID` den zugehörigen Funktionsnamen. Der Speicherbereich, auf den der Pointer zeigt, wurde mit `malloc()` beschafft und kann folglich mittels `free()` wieder freigegeben werden.

```
char  *rt_get_taskname( Task  *tid ) ;
```

Ein Zugriff über die `TID` ist immer eindeutig — kann aber auch furchtbar in die Hose gehen, wenn die Task, die ehemals diese `TID` besass, sich schon aus dem System verabschiedet hat und der Vaterprozess nun über eine ungültige `TID` im Speicher rumstochert! Es gibt keinen eindeutigen Weg im Umgang mit C–Subtasks, den man unter RTOS–UH als *absolut narrensicher* bezeichnen könnte.

- Über Namen sollten Sie nur zugreifen, wenn Sie sichergestellt haben, dass im System keine Namenskonflikte auftreten — was vom Betriebssystem selbst nicht kontrolliert wird! Sie erwischen dann stets die erste Task, die unter diesem Namen in der Systemkette angetroffen wird.
- Über eine `TID` sollten Sie nur zugreifen, wenn Sie sicher sind, dass sich hinter diesem Zeiger auch noch Ihre gewünschte Task verbirgt! Wenn Sie böswillig eine Task über die Shell entladen, sollten Sie sich auch nicht darüber wundern, was beim nächsten Programmzugriff eines anderen Prozesses mittels der nunmehr ungültigen `TID` alles passieren mag.

In Hinsicht auf die Namensgebung von Subtasks ist die automatische Vergabe von Namensextensions oft lästig. Wenn Sie exakt eine Task aus einer C–Funktion generieren wollen, spricht in Hinblick auf die Eindeutigkeit der Namensgebung nichts dagegen, dieser Sohntask schlicht den Funktionsnamen zu geben. Dies kann durch die Angabe des Schlüsselwortes `USE_FUNCTION_NAME` hinter den `#pragma`–Kommandos `TASK` oder `SUBTASK` erreicht werden.

```
#pragma TASK USE_FUNCTION_NAME
void Test_A( int  a ){ ... }

#pragma TASK USE_FUNCTION_NAME
void Test_B( int  a ){ ... }
```

In diesem Beispiel würden zwei Tasks `Test_A` und `Test_B` entstehen.

Um zu verhindern, dass eine dauerhafte Task unmittelbar nach der Generierung durch den *Funktionsaufruf* startet, wurde das Schlüsselwort `NO_TASKSTART` impementiert.

```
#pragma TASK NO_TASKSTART USE_FUNCTION_NAME
void SchlafWeiter( void ){ ... }

#pragma TASK USE_FUNCTION_NAME
void WachAuf( void ){ ... }

void main( void )
{
   // Erzeugt die Task
   SchlafWeiter() ;
   // Aktiviert die Task
   rt_activate( "SchlafWeiter", 1 ) ;

   // Erzeugt und aktiviert die Task
   WachAuf() ;
   // Aktiviert die Task zum 2. Mal
   rt_activate( "WachAuf", 1 ) ;
```

```
    rt_suspend() ;
}
```

Der Funktionsaufruf `SchlafWeiter()` in `main()` erzeugt dann zwar den betreffenden Sohnprozess sofort, dieser geht jedoch direkt nach dem Anlaufen in den Zustand `DORM` über und verharrt so bis zu einer expliziten Aktivierung.

13.3.3 Benutzung der FPU bei Sohn–Tasks

Jede Task hat unter RTOS–UH das Recht und die Pflicht, sich selbst um die Register zu kümmern, die bei einem Kontextwechsel durch den Dispatcher gerettet werden müssen. Die Daten– und Adressregister und der Kontext der CPU werden dabei ohne weiteres Zutun des Benutzers grundsätzlich gerettet. Bei Benutzung einer FPU ist es anders, denn hier können Sie selbst entscheiden, ob und wenn ja welche FPU–Register gerettet werden sollen. Unter CREST–C stellt sich die Frage, welche FPU–Register in Deckung gebracht werden sollen nicht, denn im Gegensatz zum PEARL–Compiler können Sie als Benutzer nicht bestimmen, welche FPU–Register benutzt werden sollen. CREST–C verwendet bei Fliesskommaprogrammen üblicherweise alle FPU–Register, die der Stein zu bieten hat und deshalb werden auch sämtliche Register `FP0` bis `FP7` bei einem Kontextwechsel gerettet.

Beim Linken eines Projektes weisen Sie mittels der CLN–Flags `--fpu` den CLN an, den Startupcode so zu modifizieren, dass die `main()`–Task später die FPU–Register rettet. Bei Sohn–Tasks, die zur Laufzeit generiert werden können, wird standardmässig bei der Erzeugung der Sohn–Task nachgeschaut, wie die Vater–Task es mit den FPU–Registern hält und dementsprechend der Eintrag beim Sohn vorgenommen.

```
#pragma SUBTASK
#pragma SUBTASK USE_FPU
#pragma SUBTASK NO_FPU
#pragma TASK
#pragma TASK USE_FPU
#pragma TASK NO_FPU
```

Mit der `#pragma`–Anweisung `TASK` oder `SUBTASK` kann jedoch auch explizit angegeben werden, wie die Kinder sich später verhalten sollen. So macht es z.B. durchaus Sinn, Sohn–Tasks, die nur mit Integer–Arithmetik umgehen, beim Kontextwechsel nicht durch unnütze FPU–Zugriffe auszubremsen. Das Schlüsselwort `NO_FPU` dient dazu, eine Sohn–Task so aufzusetzen, dass grundsätzlich keine FPU–Register gerettet werden. Umgekehrt wird mittels `USE_FPU` der Compiler dazu angehalten, bei Sohn–Tasks unabhängig vom Verhalten der Vater–Task alle FPU–Register zu retten.

Bei derartigen Aktionen ist darauf zu achten, dass das Schlüsselwort `USE_FPU` nur dann zu Aktionen führt, wenn das CCC–Flag `--fpu` aktiv war. Ansonsten würde sowieso kein FPU–Code vom Compiler erzeugt und ein Retten von FPU–Registern wäre entsprechend sinnlos.

13.4 Systemtasks

Unter RTOS–UH besteht auch die einfache Möglichkeit, Tasks mit Sonderfunktionalitäten zu installieren, die bereits beim Hochlaufen des Systems eingebunden und gegebenenfalls automatisch gestartet werden können. Ähnlich wie beim Kalt– oder Warmstartcode werden solche Tasks nur dann vom System erkannt, wenn sie im Scanbereich des Systems abgelegt wurden (siehe Kapitel 17.1). Verwechseln Sie bitte diese Form der Tasks nicht mit denen, die mittels des Kommandos `#pragma`

`TASK` von CREST–C generiert werden können. Auch Programme, die unter Zuhilfenahme des Startups `tstart.obj` gelinkt werden, erscheinen zwar als `TASK` oder `ATSK` in der Speicherliste, sind aber gänzlich unterschiedlich zu den Systemtasks. Bei Systemtasks handelt es sich quasi um die Urform von RTOS–UH–Tasks, bei denen keinerlei Laufzeitumgebung für die CREST–C–Bibliotheken existiert.

Derartige Tasks haben immer dort ihre Berechtigung, wenn es darum geht, Systemdienste einzurichten, die ohne die recht komplexen Dienste der CREST–C–Bibliotheken auskommen. Nehmen wir auch hier ein Beispiel. Sie wollen externe Hardware an ihr System anschliessen, die mit Interrupts auf sich aufmerksam macht. Zu diesem Zwecke richten Sie einen Interrupt–Handler ein, wie es im Abschnitt (13.5) beschrieben wird. Um nicht *endlos* in der Interruptroutine zu bleiben, ist es angebracht, auf Interrupt–Ebene nur die notwendigsten Aufgaben abzuhandeln, um das Tasking nicht unnötig zu blockieren.

Zu den notwendigen Aufgaben eines Interrupt–Handlers gehört normalerweise nur die Beruhigung der Interrupt–Quelle und das Abnehmen oder Abliefern von Daten. Weitere Aktionen sollten nach Möglichkeit von geeigneten Tasks übernommen werden. Und hier kommen die Systemtasks ins Spiel. Der Sinn von Betreuungstasks liegt z.B. darin, Daten, die auf Interrupt–Ebene entgegengenommen wurden, in geeigneter Form aufzubereiten oder Daten aufbereitet einem Interrupt–Prozess zur Verfügung zu stellen. Gerade bei solchen — meist zeitkritschen — Angelegenheiten fällt es meist nicht unangenehm auf, dass man `printf()` und die meisten anderen Bibliotheksfunktionen nicht benutzen darf, weil sowieso schon der kleinste Ausgabebefehl über eine Schnittstelle auf ein Terminal das Tasking so durcheinanderbringen würde, dass meist nur die allererste Debugausgabe einen echten Fehler beschreiben würde — der Rest der Ausgaben wäre dann meist eine direkte Folge der ersten Ausgabe und des dadurch zerstörten Timings...

Als Arbeitstiere sind die — aus C–Sicht — reichlich dummen Systemtasks jedoch absolut ideal. Zur direkten Kommunikation mit Peripheriebausteinen, Pufferverwaltung und ähnlich triviale Dinge, reichen die normalen Sprachmittel von C in 99% aller Fälle aus, ohne das dynamische Speicherallokationen oder komplexe I/O–Operationen über ANSI–C–Funktionen jemals benötigt werden. Tasks, die Betreuungsaufgaben für Interrupt–Routinen übernehmen, sollen klein und schnell sein, weil sie lediglich als verlängerter Arm der Interruptroutinen anzusehen sind. Der Vorteil dieser Aufteilung ist darin zu sehen, dass die CPU bei der Abarbeitung von Systemtasks offen für Interrupts oder höherpriorisierte Tasks ist. Würden auch die Verwaltungsaufgaben eines hochprioren Interrupts auf Interruptlevel der CPU abgehandelt, so wäre die Wahrscheinlichkeit recht hoch, dass Interruptquellen mit niedrigerer Priorität sich mit ihren Anforderungen nicht durchsetzen könnten. Durch die Ausgliederung von Code der Interrupt– auf die Taskebene, ist auch eine deutlich flexiblere Ausnutzung des Prioritätenkonzepts von RTOS–UH zu erreichen.

Hier ohne jeglichen Sinn den kompletten Startupcode für C–Tasks zu exekutieren, um eine Task zu generieren, die die dadurch verfügbaren Features nicht nutzt, wäre grober Unfug. Soviel zur Theorie und nun zur Implementierung. Der CREST–C–Compiler wird mittels eines `#pragma`–Kommandos dazu veranlasst, speziellen Code für **die jeweils nachfolgende Funktion** zu generieren. Um allen Möglichkeiten des RTOS–UH gerecht zu werden, gibt es hier eine ganze Reihe von Möglichkeiten, einen geeigneten Taskkopf für die gewünschte Art von Systemtask erzeugen zu lassen. Die Einleitung ist stets identisch.

```
#pragma SYSTEMTASK
```

Werden keine weiteren Angaben gemacht, so erzeugt der CCC standardmässig einen Taskkopf für eine normale Task mit Priorität 20 und einem Stack von 4kB Grösse. Priorität und Stackgrösse lassen sich nach bekanntem Muster mittels der Schlüsselworte `PRIO` bzw. `PRIORITY` und `STACKSIZE` übersteuern.

Die Angabe der Stackgrösse hat in Bytes zu erfolgen — anders als in der CLN–Option `-S=size`, bei der die Angaben in Kilobytes zu erfolgen haben!

```
#pragma SYSTEMTASK PRIORITY 20 STACKSIZE 4096
#pragma SYSTEMTASK PRIORITY 20
#pragma SYSTEMTASK STACKSIZE 20000 PRIO -3
```

Im Gegensatz zu normalen C–Subtasks sind hier auch negative Prioritäten zulässig. Sie sollten allerdings **niemals** Systemtasks generieren, die höher priorisiert sind als die #ERROR–Task Ihres Systems, damit diese unter allen Umständen lauffähig bleibt.

Durch die zusätzliche Angabe der Schlüsselworte RESIDENT und/oder AUTOSTART lassen sich residente, autostartfähige oder resident–autostartfähige Systemtasks erzeugen. Der Name der Systemtask wird bei Verwendung des Schlüsselwortes RESIDENT vom Compiler automatisch um ein Doppelkreuz # erweitert, um unbeabsichtigtes Entladen zu vermeiden.

Weiterhin lässt sich unter Verwendung der Schlüsselworte QUEUE, INTERFACE oder ERROR spezifizieren, ob die Tasks für eine besondere Verwendung ausgelegt sein soll.

```
#pragma SYSTEMTASK PRIO -20 ERROR
```

Da es maximal eine ERROR–Task im System geben darf, ist die Möglichkeit, eine zusätzliche Task dieser Art in CREST–C zu schreiben, eher von akademischer Bedeutung und soll hier auch nicht näher erläutert werden.

```
#pragma SYSTEMTASK PRIO -4 RESIDENT INTERFACE 21 STACKSIZE 0x300
```

Eine INTERFACE–Task könnte etwa so vereinbart werden, liefe auf Priorität -4, wäre resident und für LDN=21 verantwortlich. Eine Stackgrösse von $300 Bytes wäre für eine solche Task durchaus hinreichend. Vertreter dieser INTERFACE–Tasks in ihrem System wären die #USERxx, die die Shell aufwecken, wenn Sie auf Ihrem Terminal ein Ctrl–A gedrückt haben.

Ein typisches Beispiel für eine QUEUE–Task wären die Betreungstasks für die Schnittstellen ihres Systems. Eine QUEUE–Task für LDN=21 könnte dann folgendermassen vereinbart werden.

```
#pragma SYSTEMTASK PRIO -5 RESIDENT QUEUE 21 STACKSIZE 1024 \
        TASKHEADSIZE 0x0200
```

Ebenso wie bei Kalt– und Warmstartscheiben, Interrupt– und Exception–Handlern, spielen Sie bei der Kodierung von Systemtasks ohne Netz und doppelten Boden. Sie haben zwar eigenen Stack von freiwählbarer Grösse — was gegenüber Interrupt–Handlern schon eine deutliche Verbesserung darstellt — aber immer noch keinen Zugriff auf irgendwelche globalen Variablen. Auch der Aufruf von Funktionen, die auf globale Variablen zugreifen, dürfte rasch zum Crash führen. Wenn Sie mit der Aussenwelt Daten austauschen wollen, dann geschieht das üblicherweise über Communication–Elemente — denn um die Dinger zu verwalten, werden die Betreuungstasks schliesslich geschrieben — oder alternativ über absolute Variablen in der .common–Section.

Eine weitere Möglichkeit besteht darin, sich einen grösseren Taskkopf anzufordern und hinter den internen Informationen des RTOS–UH die eigenen Daten abzulegen. Mittels des Schlüsselwortes TASKHEADSIZE size lassen sich size Bytes zusätzlich im Taskkopf anfordern. Einen Zeiger auf den Speicher, der Ihnen nun zur freien Verfügung steht, erhalten Sie mit z.B. mittels der folgenden Konstruktion:

```
void  *my_memory = rt_my_TID() + 1 ;
```

Um von einer fremden Task an diesen Speicher heranzukommen, muss diese lediglich die TID der Task mit dem erweiterten Taskkopf kennen oder ermitteln. Dies kann z.B. mittels:

```
#pragma SYSTEMTASK PRIO -1 TASKHEADSIZE 0x0200 STACKSIZE 1024
void MyTask( void ) { ... }
```

```
void main( void )
{
   void  *common_memory = rt_search_task( "MyTask" ) + 1 ;
}
```

erfolgen, wobei der Pointer auf den erweiterten Taskkopf der Task `SysTask` zurückgeliefert wird. Bedenken Sie bitte, dass ein relevanter Unterschied zwischen der Adresse einer Task aus Sicht des C–Compilers und der Adresse einer Task aus Sicht von RTOS–UH besteht. In der Sprachdefinition von C wäre die Adresse der Funktion `MyTask` die Stelle, an der der Maschinencode für den Einsprung in die betreffende Funktion zu finden ist. RTOS–UH versteht unter der Adresse einer Task die Stelle im RAM, an der der Taskkopf zu finden ist. Diese ist aber erst zur Laufzeit bekannt. Die Zuweisung `common_memory = ( (Task*)&MyTask ) + 1 ;` wäre demnach zwar syntaktisch korrekt, würde allerdings Äpfel mit Birnen mischen und zu einem Pointer auf eine absolut sinnlose Stelle im Adressraum führen.

Abschliessend sollte noch Erwähnung finden, dass Systemtasks nicht mit Parametern aufgerufen werden können und selbstredend auch keine direkte *Funktions*–Antwort liefern können.

13.5 Interrupts und Exceptions

Die CPU ist in der Lage, zwischen zwei unterschiedlichen Betriebsarten zu unterscheiden. Normalerweise laufen Ihre Programme im User–Mode. Das Betriebssystem selbst arbeitet meist auf einer höheren Ebene: dem Supervisor–Mode. Im Supervisor–Mode stehen den abzuarbeitenden Programmen spezielle Befehle des Prozessors und erweiterte Zugriffsrechte zu Verfügung, die die Arbeit derartig übergeordneter Programme erleichtern oder erst ermöglichen (siehe auch Abschnitt 15.7). Unter RTOS–UH stehen Systemaufrufe zur Verfügung, um normalen Tasks einen Wechsel in den privilegierten Modus zu ermöglichen. Allerdings besitzt die CPU auch die Fähigkeit, selbstständig den Modus zu wechseln. Es gibt drei wesentliche Fälle, die dabei zu unterscheiden sind.

- Gewollte Unterbrechungen des laufenden Programms durch entsprechende Anweisungen wie `TRAP`, `ILLEGAL` und ähnliche Befehle, die zu Ausnahmebehandlungen führen können.
- Abbrüche durch Fehler bei der Programmausführung. Dazu zählen Division durch Null, Zugriff auf verbotene Adressbereiche, Schreibzugriffe auf EPROM's, unsinniger Code, fehlerhafte Dekodierung von Daten und viele schöne Sachen mehr.
- Unterbrechungen durch externe Ereignisse.

Tritt ein solcher Fall ein, so wird der normale Programmfluss unterbrochen und eine Ausnahmebehandlung eingeleitet. Das neudeutsche Wort dafür lautet *Exception*. Die CPU wechselt in den priviligierten Modus und greift über eine Vektortabelle auf eine Adresse zu, an der der Programmablauf fortgesetzt werden soll. Der so ermittelte Vektor wird angesprungen und der zugehörige Code im Supervisor–Mode exekutiert. Es gibt 256 Vektoren, die zu diesem Zwecke verfügbar sind. Ein grosser Teil davon ist nur für administrative Aufgaben des Betriebssystems von Bedeutung (wie z.B. in Tabelle 13.4 an Hand der RTOS–UH–Traps zu sehen ist oder ist von Motorola für die Ausnahmebehandlung reserviert. So macht es z.B. wenig Sinn, mit dem `ILLEGAL INSTRUCTION`–Vektor zu spielen, da das Auftreten einer solchen Exception in Nutzerprogrammen darauf hindeutet, dass in Ihrem Code (oder Stack) bereits der Super–GAU eingetreten ist.

Zu den Intterrupt–Vektoren, die für Anwenderprogramme von Bedeutung sind, zählen z.B. die in Tabelle 13.1 aufgeführten Auto–Vektoren, sowie die Vektoren im Bereich von `$100` bis `$3FC`, die als Non–Auto–Vektoren bezeichnet werden.

Name	Adresse	Interrupt Auto Vector
IR1	$64	Level 1
IR2	$68	Level 2
IR3	$6C	Level 3
IR4	$70	Level 4
IR5	$74	Level 5
IR6	$78	Level 6
IR7	$7C	Level 7

Tabelle 13.1: Auszug aus der Exception–Vektor–Tabelle

Wenn Sie externe Hardware an Ihren Rechner anschliessen und diese Hardware Interrupts auslösen kann, werden diese Vektoren für Sie interessant. Erreicht ein externes Signal die CPU, so prüft diese zunächst intern, ob der Interrupt zur Zeit bearbeitet werden soll. Es gibt zu diesem Zwecke sieben unterschiedliche Level, die die CPU unterscheiden kann. Im Statusregister der CPU ist vermerkt, bis zu welchem Level Interrupts ignoriert werden sollen. Läuft die CPU auf Level 4, dann werden Interrupts der Stufen 1 bis 4, die während der Abarbeitung eines Level–4–Interrupts auftreten, erstmal ignoriert und gelangen erst dann zur Ausführung, wenn alle höherprioren Interrupts abgearbeitet wurden. Schlägt jedoch ein Interrupt von Level 5 bis 7 auf, dann lässt die CPU unverzüglich den Hammer fallen und wendet sich der höherpriorisierten Aufgabe zu.

Sie versetzt sich in den priviligierten Modus, rettet wichtige Daten über den Kontext des unterbrochenen Programmes, holt den Vektor des Interrupts, der gerade anliegt und exekutiert den Code, der zu dieser Unterbrechung vorgesehen ist. Im einfachsten Falle kann dieser Vektor auf ein `RTE` (Return From Exception) zeigen und die Interruptquelle schlicht ignorieren — sofern diese sich ignorieren lässt. In den nächsten Kapitel geht es darum, wie man Code an Stelle eines `RTE` unterbringen kann, um etwas sinnvoller auf eine Unterbrechung zu reagieren.

Ein Interrupt–Handler unterscheidet sich von einer normalen Funktion in zwei wesentlichen Punkten. Erstens wird er direkt von der CPU aufgerufen und Sie haben keinerlei Einfluss auf die Parameter, die die CPU Ihnen übergibt. Eigene Funktionsparameter an Interrupt–Handler zu übergeben, ist demnach nicht möglich. Zweitens will niemand wissen, was die Routine zu antworten gedenkt, denn einen direkten Aufrufer auf Funktionsebene gibt es nicht. Schlimmer noch, denn der unterbrochene Prozess soll typischerweise nichts davon bemerken, dass ihm zwischenzeitlich die CPU entzogen war. Der Interrupt–Handler ist also stets als `void`–Funktion anzulegen.

Der CCC wird mittels der Anweisung `#pragma INTERRUPT` angewiesen, aus der nachfolgenden Funktion einen Interrupt–Handler zu generieren.

```
#pragma INTERRUPT VECTOR 0x78 EPROM EVENT 0x00000001 IROFF
void InterruptHandler1( void ){ ... }
```

Das Beispiel würde einen Interrupt–Handler für den Level 6 erzeugen und zudem durch das Schlüsselwort `EPROM` eine Scheibe (`.SLICE 14`) generieren, die den Vektor bereits beim Hochlaufen des Systems automatisch initialisiert. Mittels `EVENT 0x00000001` wird Code erzeugt, der beim Verlassen der Interruptroutine RTOS–UH anweist, einen Software–Event für die angegebene Maske zu feuern. Das Schlüsselwort `IROFF` teilt dem Compiler mit, dass der Handler vom Eintritt in den Interrupt an auf Level 7 laufen soll — will heissen: alle anderen Interrupts werden gnadenlos ausgesperrt!

```
#pragma INTERRUPT LEVEL 6 EPROM EVENT 0x80000000
void InterruptHandler2( void ){ ... }
```

Wie Sie diesem Beispiel entnehmen können, wurde diesmal statt `VECTOR 0x78` in diesem Falle die Kurzschreibweise `LEVEL 6` verwendet. Die Umsetzung können Sie der Tabelle 13.1 entnehmen. Es wird wieder Code für den Einsatz in einem EPROM–System erzeugt, ein anderes Event gefeuert und

die Interruptroutine läuft diesmal ab, ohne die Interrupts zu sperren.

```
#pragma INTERRUPT VECTOR 0x124
void InterruptHandler3( void ){ ... }
```

Diesmal handelt es sich um eine Interruptroutine, die für nachladbaren Code geeignet ist. Sie ist zudem nicht dazu gedacht, RTOS–UH–Tasks per Event über den aufgetretenen Interrupt zu informieren. In ihrem Programm müssen Sie ohne die Angabe von `EPROM` den Vektor `0x124` selbstständig versorgen und per Hand auf ihre Routine setzen. Die Angabe von `VECTOR` sollte dennoch korrekt vorgenommen werden, denn der Compiler benötigt diese Information, um den Rückfallmechanismus für diesen Interrupt im Code vorzusehen. Apropos Vektoren und Vektornummern: der CCC überprüft ihre Angabe daraufhin, ob es sich bei der Vektornummer um eine durch Vier teilbare Zahl handelt — da die Vektortabelle der CPU auf Langwortgrenzen ausgerichtet ist, sind andere Angaben zumeist von wenig Erfolg gekrönt. Wenn Ihnen dennoch die Meldung:

```
ERROR:  Not a vectornumber -- must be a multiple of 4
```

begegnet, dann hat der Compiler aufgepasst und gerade die Generierung von heimtückischem Code verhindert, durch den RTOS–UH beim Systemstart übel auf die Nase gefallen wäre.

```
#pragma INTERRUPT NO_VECTOR
void InterruptHandler3( void ){
   *(UWORD*)0x7FE = (UWORD) (..hier die Vektornummer..) ;}
```

In wenigen Fällen macht es auch Sinn, ganz auf die automatische Generierung des Sicherheits– und Rückfallmechanismus des Compilers zu verzichten; wenn Sie z.B. erst zur Laufzeit des Programmes die Information aus der Hardware auslesen müssen, welcher Vektor eigentlich versorgt werden muss. In diesem Falle macht es selbstverständlich keinen Sinn, den Compiler einen Dummy–Vektor eintragen zu lassen, den Sie dann überschreiben müssten. Mittels des Schlüsselwortes `NO_VECTOR` wird der Code, um die Systemspeicherstelle `IID` (`$7FE.W`) mit der Vektornummer der laufenden Interruptroutine zu versorgen, unterdrückt. Der erste Code, den ihre Interruptroutine abzuspulen hat, **muss** darin bestehen, diese Zelle zu versorgen.

```
#pragma INTERRUPT NO_VECTOR
void InterruptHandler4( void ){ ... }
```

Um das Retten und Restaurieren der alten Vektornummer brauchen Sie sich jedoch nicht zu kümmern. Diese Aufgabe nimmt Ihnen der Compiler ab.

Wenn Sie komplett auf die Versorgung der `IID`–Zelle verzichten wollen — Sie sollten dabei allerdings ernsthaft wissen, was Sie tun —, dann ist die Verwendung der Schlüsselworte `NO_IID` oder `NO_MALFUNCTION` die Tat der Stunde.

```
#pragma INTERRUPT NO_IID
void InterruptHandler5( void ){ ... }

#pragma INTERRUPT NO_MALFUNCTION
void InterruptHandler6( void ){ ... }
```

Der erzeugte Code entspricht dann allerdings nicht mehr so ganz dem, was das Betriebssystem erwartet und bei Fehlern innerhalb des Interrupt–Handlers knallt es ganz furchtbar, weil der Rücksturzmechanismus von RTOS–UH nicht aufgesetzt wurde. Wenn ein Fehler auftritt, dann ist das Resultat für den Rechner final!

13.5.1 Laufzeitcode eines Interrupt–Handlers

CREST–C ist es zunächst völlig gleichgültig, ob der Code im Supervisor– oder im User–Mode abläuft. Das Problem beginnt allerdings dann, wenn man irgendwo Daten unterbringen möchte. Interrupts haben die unangenehme Angewohnheit, asynchron zum normalen Programmverlauf aufzutreten. Für Sie bedeutet dies, dass beim Eintritt in den Interrupt–Handler unbrauchbarer Müll in den Registersätzen der CPU steht, der zwar für die gerade unterbrochene Task — welche auch immer das sein mag — von vitaler Bedeutung, für den Interrupt–Handler jedoch völlig wertlos ist. Das einzige Register, auf das Sie sich verlassen können, ist der Stackpointer `A7`. Da Sie sich im Supervisor–Mode befinden, handelt es sich dabei dummerweise um den unter RTOS–UH recht winzig ausgefallenen Supervisor–Stack, der im Bereich von `0x600` bis `0x7FE` liegt. Die einzige Methode, Daten abzulegen, ist also dieser Stack, eine absolute Position innerhalb des RAM–Bereichs Ihres Rechners oder der Zugriff auf `absolute`–Variablen.

Das oberste Prinzip muss in jedem Falle sein, den Stack nach Möglichkeit wenig zu belasten. Der Aufruf einer Funktion, die erstmal kräftig rekursiv absteigt und damit alles, was unterhalb des Stacks liegt, ordentlich aufmischt, verbietet sich aus diesem Grunde von selbst. Weiterhin ist der Gebrauch dauerhafter C–Variablen nicht möglich, denn woher soll der Interrupt–Handler wissen, wo diese Dinger zu suchen wären — so ganz ohne gültige Registerinhalte ist das mächtig schwierig. Das Adressregister `A5`, das in normalen CREST–C–Nutzerprogrammen für derartige Zugriffe verwendet wird (siehe Abschnitt 3.7), enthält hier garantiert keine sinnvollen Werte und wird bei der Codegenerierung eines Interrupt–Handlers nicht anders behaltet, als die anderen Adressregister. Ebenso ist der Aufruf von Funktionen, die direkt oder indirekt von der Existenz eines korrekten `A5` ausgehen, tödlich. Sie werden erraten haben, dass auch `local`–Variablen in der `.local`–Section nicht verfügbar sind, weil Interrupt–Handler nunmal keine Tasks sind. Das dazu benötigte Adressregister `A4` ist auf Interruptebene nicht besetzt und wird bei der Codegenerierung verwendet.

Nur `auto`– und `register`–Variablen, die CREST–C auf dem Stack allokiert bzw. CPU–Registern zuordnet, sind sinnvoll einsetzbar. Während der Laufzeit des Interrupt–Handlers können Sie mit diesen Zellen arbeiten. Sie sollten jedoch auch hier beachten, dass der Platz, der zu diesem Zwecke verfügbar ist, nur eine erschreckend endliche Grösse besitzt. Der folgende Code in Abbildung 13.9 lässt sich zwar übersetzen, führt aber mit Sicherheit bei der ersten Aktivierung zur Auslöschung sämtlicher lebenswichtiger Systemvektoren von RTOS–UH. Ein entsprechend schneller Abgang ins Nirwana ist da eine Selbstverständlichkeit.

```
#pragma INTERRUPT LEVEL 3
void KillMe( void )
{
   register int   t            ;
   auto     char  test[ 5000 ] ;

   for ( t=0 ; t<5000 ; )
      test[ t++ ] = 0 ;
}
```

Abbildung 13.9: `Stackoverflow` auf Interruptlevel

13.5.2 Die Interrupt–Data–Pointer

RTOS–UH bietet eine standardisierte Möglichkeit, die Daten eines Interrupt–Handlers in die Aussenwelt zu transferieren. In den RTOS–UH–Systemvariablen existieren 7 Pointer, die auf Datenbereiche zeigen, in denen Interrupts der Level 1 bis 7 ihre Datenbestände ablegen können. Diese Pointer bzw. die entsprechend einzurichtenden Datenbereiche können bereits bei Konfigurierung des Betriebssystems initialisiert werden. Dazu dienen auf Assemblerebene die Systemscheiben 2 bis 8, mit deren Hilfe sich

Datenpuffer der gewünschten Grösse einrichten lassen. CREST–C erlaubt mittels der Anweisung:

```
#pragma ALLOCATE_INTERRUPT_BUFFER LEVEL 3 SIZE 0x100
```

die Generierung derartiger Scheiben (`.SLICE 2` bis `.SLICE 8`). Nach einem Kaltstart steht dann ein entsprechender Puffer für den entsprechenden Interrupt–Level zur Verfügung. Hinter `LEVEL` dürfen die Zahlenwerte 1 bis 7 angegeben werden. Die Grösse des Buffers wird mit `SIZE` vorgegeben. Beachten Sie, dass die Summe aller Buffer (deutlich) kleiner als 32kB bleiben muss!

Zunächst soll nur die Existenz dieser Pointer interessieren, die RTOS–UH nach dem Kaltstart und dem Auffinden der eben beschriebenen Scheiben für Sie einrichtet. In Tabelle 13.2 sind die Adressen der `IDP`'s (Interrupt–Data–Pointer) aufgeführt.

Name	Adresse
IDP1	$832
IDP2	$836
IDP3	$83A
IDP4	$83E
IDP5	$842
IDP6	$846
IDP7	$84A

Tabelle 13.2: Adressen der Interrupt Data Pointer

Der weiterer Vorteil der `IDP`'s besteht darin, dass hierdurch unter RTOS–UH im Bereich der konstanten Kernelvariablen ein paar Pointer zur Verfügung stehen, die sich trefflich für eigene Zwecke missbrauchen lassen.

13.5.3 Kommunikation mit Interrupt–Handlern

In der Folge soll an Hand von drei Beispielen verdeutlicht werden, welche Möglichkeiten prinzipiell bestehen, um Daten mit einem Interrupt–Handler auszutauschen. Es ist beim Design des Datenaustausches mit Interrupt–Handlern stets zu beachten, dass die Adressen gemeinsam genutzter Resourcen bereits zur Compile– oder Linkzeit festgelegt oder über gewisse Konventionen vom Interrupt–Handler bestimmt werden müssen.

13.5.3.1 Interrupt–Handler zum Nachladen

Als Beispiel ist ein kleiner Interrupt–Handler entstanden, der sich *zur Laufzeit* auf Level 4 einklinkt, Daten von einer imaginären I/O–Karte liest und über einen Puffer an eine Betreuungstask weiterleitet. Hier soll ein `IDP` quasi als globaler Pointer eingesetzt werden. Wenn die `IDP`'s von einem Anwenderprogramm verbogen werden, dann ist dieser Trick nur mit den IDP's 2 bis 7 machbar, da `IDP1` eine Sonderbedeutung besitzt — eine Veränderung dieses Pointers zur Laufzeit führt beim nächsten `ABORT` zu interessanten Umgestaltungen Ihrer RAM–Inhalte, die mit grosser Wahrscheinlichkeit dem Betriebssystem den Todesstoss versetzen.

```
typedef struct
{  Task   *tid          ;
   char   *reader_ptr   ;
   char   *writer_ptr   ;
   char    buffer[ 256 ] ;
}DataSpace ;
```

```
#define DPC    ( *(WORD* )0x800L      )
#define IR4    ( *(void**)0x70L       )
#define IDP4   ( *(void**)0x83EL      )
#define I_O    ( *(char*)0xFFFF6803UL )

#pragma INTERRUPT LEVEL 4
void Level4( void )
{
   DataSpace  *bptr = IDP4 ;
   *bptr->writer_ptr++ = I_O ;
   /* Jetzt Betreuungstask wieder anwerfen */
   *bptr->tid->block &= ~BLKBSU ;
   --DPC ;
}

void main( void )
{
   DataSpace *bptr ;
   char       c    ;
   if ( ( bptr = malloc( sizeof( DataSpace ) ) ) == NULL )
      exit( 0 ) ;
   bptr->tid        = rt_my_TID()        ;
   bptr->reader_ptr = &bptr->buffer[ 0 ] ;
   bptr->writer_ptr = &bptr->buffer[ 0 ] ;
   IDP4 = bptr ;
   IR4  = Level4 ;
   /* Interrupt jetzt scharfmachen !! */
   for (;;)
   {
      /* Warten auf Freigabe durch den Interrupthandler */
      rt_suspend() ;
      c = *bptr->reader_ptr++ ;
   }
}
```

Abbildung 13.10: Kommunikation über dynamischen Speicher

Das Beispiel ist nur dazu gedacht, die Vorgehensweise zum Einklinken von Interrupts und Betreuungstasks zur Laufzeit zu demonstrieren. Es ist nicht lauffähig und die Synchronisation zwischen Interrupt– und Grundebene mittels `rt_suspend()` zu realisieren, ist wohl ebenfalls nur als abschreckendes Beispiel gedacht! Etwas mehr Aufwand ist schon noch erforderlich, um einen Ringpuffer sauber zu verwalten und keine Interrupts zu verschlafen.

Zuerst kommt die Beschaffung eines Speicherblockes, der die Daten des Interrupt–Handlers aufnehmen soll. Geschickterweise verwaltet man den Speicher in Form einer Struktur, um auf die Elemente namentlich zugreifen zu können. Dann kommt eine Initialsequenz, um den Block auf seine spätere Aufgabe vorzubereiten.

In der Verwaltungsstruktur sei die `TID` der Grundebene, ein Lese–, ein Schreibzeiger und ein Datenblock vorgesehen. Damit lässt sich bereits ein Ringpuffer verwalten, der selbstverständlich gross genug sein sollte, um eventuelle Überläufe zu verhindern, wenn sich der Datenproduzent nicht abschalten lässt.

Anschliessend wird der Puffer auf `IDP4` eingeklinkt. Das Umhängen des Interrupt–Vektors `IR4` kann auf verschiedenen Rechnern den Wechsel in den Supervisor–Mode notwendig machen. Danach spielt `main()` den Konsumenten und wartet darauf, bis der Interrupt–Handler wieder ein Zeichen liefern kann. In Hinsicht auf die Verwaltung des Datenblockes geht es im Beispiel etwas grausam zu. Der Interrupt–Handler schert sich nicht um die Abmessungen des Puffers und die Betreuungstask ebenso-

wenig. Aber schliesslich geht es nur ums Prinzip!

Auch über die Mechanismen, die zur Synchronisation zwischen Interrupt– und Grundebene verwendet werden, sei hier zunächst nichts ausgesagt.

13.5.3.2 Interrupt–Handler in EPROM's

Im letzten Kapitel wurde demonstriert, wie man die Interrupt–Data–Pointer des RTOS–UH zur Laufzeit für seine Zwecke missbrauchen kann. Sauberer geht es natürlich, wenn man die `IDP`'s nicht gewalttätig verbiegt, sondern diese Aufgabe RTOS–UH beim Hochlaufen überlässt. Das vorherige Beispiel vereinfacht sich für ein EPROM–System deutlich. Die Speicherbeschaffung und das Eintragen des Vektors wird bereits bei der Konfiguration des Betriebssystems erledigt.

```
typedef struct
{  Task   *tid          ;
   char   *reader_ptr   ;
   char   *writer_ptr   ;
   char    buffer[ 256 ] ;
}  DataSpace ;

#pragma ALLOCATE_INTERRUPT_BUFFER LEVEL 4 SIZE ( sizeof( DataSpace ) )

#define DPC    ( *(WORD* )0x800L      )
#define IDP4   ( *(void**)0x83EL      )
#define I_O    ( *(char*)0xFFFF6803UL )

#pragma INTERRUPT LEVEL 4 EPROM
void Level4( void )
{
   DataSpace  *bptr = IDP4 ;
   *bptr->writer_ptr++ = I_O ;
   /* Jetzt Betreuungstask wieder anwerfen */
   *bptr->tid->block &= ~BLKBSU ;
   --DPC ;
}

void main( void )
{
   DataSpace  *bptr = IDP4 ;
   char       c            ;

   bptr->tid        = rt_my_TID()        ;
   bptr->reader_ptr = &bptr->buffer[ 0 ] ;
   bptr->writer_ptr = &bptr->buffer[ 0 ] ;
   /* Interrupt jetzt scharfmachen !! */
   for (;;)
   {
      /* Warten auf Freigabe durch Interrupt-Handler */
      rt_suspend() ;
      c = *bptr->reader_ptr++ ;
   }
}
```

Abbildung 13.11: Kommunikation über systemeigene–`IDP`–Puffer

Beachten Sie bitte, dass zur Übersetzung des Beispiels die Compileroption `-q` aktiv sein muss (siehe Abschnitt 3.1.8), um die Expressionauswertung für die `SIZE`–Angabe des anzufordernden Interrupt-

puffers zu forcieren.

Die Initialisierung der Datenstruktur, auf die der `IDP4` verweist, wird selbstverständlich nicht von RTOS–UH vorgenommen. Achten Sie bitte stets darauf, dass es durchaus Sinn macht, diese Aktionen durchzuführen, **bevor** der betreffende Interrupt freigegeben bzw. der Vektor auf den Interrupt–Handler eingeklinkt wird.

Verwenden Sie zum Aufsetzen der `IDP`–Puffer **nie** Kaltstart–Code! Kaltstart–Routinen werden **vor** der Initialisierung der `IDP`'s ausgeführt. Die korrekte Reihenfolge zum Aufsetzen eines Interrupt–Handlers im EPROM könnte korrekt folgendermassen ablaufen:

1. Erzeugung einer Funktion mittels `#pragma COLDSTART`. Diese Funktion sollte die interrupterzeugende Hardware aufsetzen.
2. Erzeugung einer Funktion mittels `#pragma WARMSTART`. Diese Funktion sollte die nunmehr verfügbaren Kommunikationspuffer initialisieren und bei Bedarf auch schon die Interrupts hardwaremässig freigeben.
3. Erzeugung einer Grundebenentask. Es bietet sich meist an, diese mittels `#pragma SYSTEMTASK` zu generieren und autostartfähig zu machen.

13.5.3.3 Interrupt–Handler mit `absolute`n Variablen

Das die Kommunikation von Grundebene und Interrupt–Handler auch ohne die Verwendung von `IDP`'s funktionieren kann, zeigt das nachfolgende Beispiel. Hier werden C–Variablen der Speicherklasse `absolute` verwendet, um eine gemeinsame Datenbasis zu garantieren.

```
typedef struct
{  Task   *tid           ;
   char   *reader_ptr    ;
   char   *writer_ptr    ;
   char    buffer[ 256 ] ;
}  DataSpace ;

absolute DataSpace  Buffer ;

#define DPC    ( *(WORD* )0x800L      )
#define IR4    ( *(void**)0x70L       )
#define I_O    ( *(char*)0xFFFF6803UL )

#pragma INTERRUPT LEVEL 4
void Level4( void )
{
   *Buffer.writer_ptr++ = I_O ;
   /* Jetzt Betreuungstask wieder anwerfen */
   bptr->tid->block &= ~BLKBSU ;
   --DPC ;
}

void main( void )
{
   char  c ;

   Buffer.reader_ptr = &Buffer.buffer[ 0 ] ;
   Buffer.writer_ptr = &Buffer.buffer[ 0 ] ;
   IR4  = Level4 ;
   /* Interrupt jetzt scharfmachen !! */
```

```
    for (;;)
    {
       /* Warten auf Freigabe durch Interrupthandler */
       rt_suspend() ;
       c = *Buffer.reader_ptr++ ;
    }
}
```

Abbildung 13.12: Kommunikation über `absolute` Variablen

Diese Methode funktioniert deshalb, weil der Interrupt–Handler die Adresse der Variablen `Buffer` bereits vom CLN zugeteilt bekommt.

13.5.4 Das Verlassen einer Interrupt–Routine

Normalerweise sollen Exceptions und Interrupts unter RTOS–UH mittels des Dispatchers verlassen werden. Die Methode, einen Interrupt–Handler durch ein `RTE`–Kommando zu terminieren, ist eigentlich nur nach vorheriger Kontrolle der Speicherzelle `DPC` (`$800.W`) zulässig. Diese 16–Bit–Variable dient dem Dispatcher des RTOS–UH als Kontrollzelle, ob ein Dispatcherlauf erforderlich ist. Enthält sie einen Wert kleiner Null, so startet der Dispatcher. Ansonsten wird der Handler korrekt über `RTE` verlassen. Um also einen Dispatcherprozess zu forcieren, sollten Sie bei Bedarf diese Speicherstelle selbst dekrementieren, wie es in den vorausgegangenen Abschnitten mehrfach demonstriert wurde. Unterbleibt diese Aktion, so greift z.B. das Aufheben der Blockierbedingung der Grundebenentask erst dann, wenn ein anderer Prozess einen Dispatcherlauf auslöst — was einem deterministischen Verhalten doch arg im Wege steht.

Nach dieser langen Vorrede dürfte klar sein, dass sich auch eine Abart von Interruptcode erzeugen lässt, die die Interruptroutine ohne Test der `DPC`–Zelle verlässt. Mittels des Schlüsselwortes `NO_DISPATCHER_CALL` unterbleibt der Test und der Code des Interrupthandlers terminiert mittels eines `RTE`.

```
#pragma INTERRUPT NO_DISPATCHER_CALL
void BrutalerAusstieg( void ){ ... }
```

13.5.5 Synchronisation von Grund– und Interruptebene

Eine weitere Schwierigkeit bei der Erstellung von Interrupt–Handlern besteht darin, Ereignisse, die bereits zu Aktionen auf Interruptebene geführt haben, der oder den interessierten Tasks der Grundebene mitzuteilen.

Dabei sollten Sie stets folgendes im Auge behalten:

- Interrupt–Routinen gehören zu den privilegierten Prozessen, im System, was ihre *Priorität* betrifft. Dabei laufen Sie auf Supervisor–Mode der CPU und haben somit *quasi* eine Priorität, die gemeine Tasks mit den handelsübliche Methoden des Taskings **nie** erreichen können.
- Interrupt–Routinen gehören zu den benachteiligten Prozessen, im System, was die Nutzung der allermeisten Systemdienste betrifft. Es ist schlicht all das untersagt, was dazu führt (oder führen könnte), dass Veränderungen an den diversen Speicherverkettungen von RTOS–UH vorgenommen werden — was nahezu alle Traps betrifft!

Im Prinzip sind Sie nur auf der sicheren Seite, wenn Sie keinerlei Traps von Interruptebene aus aufrufen. Das wiederum verlangt eine ganz andere Herangehensweise, wenn man sich dennoch genötigt sieht, das Tasking in der normalen RTOS–UH–Welt interruptgesteuert zu beeinflussen.

Wenn zeitkritische Operationen abgeschlossen wurden, sollten Sie umgehend den Interrupt–Level wieder verlassen und etwaige zeitaufwendige Nachbehandlungen bzw. die eigentliche Behandlung der Daten einer Grundebenentask überlassen. Beachten Sie bei der Kodierung von Interrupt–Handlern die wesentliche Tatsache, dass ein Multitaskingsystem nur Sinn macht, wenn das Tasking nicht blockiert ist.

Die Abarbeitung von Interrupts blockiert das Tasking!

In den folgenden Abschnitten sollen zwei unterschiedliche Methoden vorgestellt werden, um von Interruptebene Einfluss auf das Tasking nehmen zu können.

13.5.5.1 Interrupts entblockieren Tasks

Wenn die Aufgabenstellung sich darauf beschränkt, dass eine einzige oder zumindest überschaubare Anzahl von Tasks auf einen Interrupt–Prozess reagieren soll, so besteht in Assembler oder C die Möglichkeit, eine sehr schnelle Möglichkeit zu realisieren.

Im Abschnitt 15.2 ist das *offizielle* Tasking unter RTOS–UH beschrieben — der zugehörigen Tabelle 15.1 sind die geläufigen Taskzustände und Zustandsübergänge aufgeführt. Die nun beschriebene Methode werden Sie dort jedoch weder beschrieben noch dargestellt finden. Sie basiert auf der Tatsache, dass jede Task unter RTOS–UH mittels diverser Flags im Taskkopf vom Zustand *laufwillig* oder `RUN` in nahezu jeden anderen Zustand überführt werden kann — ohne dazu den zugehörigen offiziellen Weg zu gehen. Üblicherweise sollte man als Programmierer diesen Weg meiden.

13.5.5.2 Interrupts feuern Events

Wenn Sie zum Beispiel eine FFT durchführen wollen, könnte der Interrupt–Handler so geschrieben sein, dass er erstmal 256 Werte aufsammelt und erst dann die Task, die die Fourier–Transformation berechnen soll, aufweckt und die Daten durchreicht.

Dieses Aufwecken einer Task ist unter RTOS–UH über Prozessinterrupts bzw. Events geregelt. Sie stellen das softwareseitige Gegenstück zu Hardware–Interrupts dar. RTOS–UH verwaltet dazu einen internen Vektor mit 32 Bits. Jedes Bit ist einem Prozessinterrupt zugeordnet. Wenn ein Interrupt–Handler eine Task ansprechen möchte, so muss er nur ein Event feuern. Jede Task, die sich auf dieses Ereignis eingeplant hat, wird beim nächsten Dispatcherlauf lauffähig gemacht und könnte entsprechend ihrer Prioriät auch die CPU zugeteilt bekommen, um auf den Prozessinterrupt zu reagieren.

```
#pragma INTERRUPT LEVEL 3 EPROM EVENT 0x00000001
void InterruptHandler( void ){}

#pragma SYSTEMTASK AUTOSTART
void MyTask( void )
{
   for (;;)
      rt_event_resume( 0x00000001 ) ;
}
```

Im Beispiel würde bei jedem Auftreten eines Level–3–Interrupts die Task `MyTask` fortgesetzt, die das wiederum zum Anlass nimmt, sich erneut auf diesen Prozessinterrupt einzuplanen und sich aufs Ohr zu legen.

13.5.6 Fehlerbehandlung auf Interrupt–Level

Dem RTOS–UH–Handbuch können Sie entnehmen, dass es zum Abfangen von Fehlern auf Interrupt–Level eine sogenannte `malfunction` gibt. Auch dahinter versteckt sich nichts sonderlich Geheimnisvolles. Tritt ein `BUS ERROR` oder eine ähnlich peinliche Begebenheit auf Interrupt–Level auf, so versucht das System, noch einige Rettungsmassnahmen durchzuführen, um nicht absolut hilflos im Walde zu stehen. Dazu dient die Angabe eines Pointers auf den Code, der den Schiffbruch vermeiden soll. Bei CREST–C besteht die `malfunction` lediglich aus dem üblichen Epilog–Code des Interrupt–Handlers. Es werden im Fehlerfall die alten Register restauriert, der Stack wieder auf den Ursprungswert gesetzt und dann der Handler geordnet verlassen. Doch vorsicht, denn diese Methode funktioniert nur, wenn der Interrupt–Handler in einem Stück kodiert wurde. In unserem Falle heisst das:

Funktionsaufrufe auf Interruptebene legen den Rückfall–Mechanismus im Fehlerfall lahm und führen zum Systemabsturz!

RTOS–UH findet im Fehlerfalle die `malfunction` nicht mehr, wenn der Stack durch Funktionsaufrufe modifiziert wurde. Wenn Sie sicher sind, dass keine Fehler in der Interrupt–Routine auftreten können, steht es Ihnen selbstverständlich frei, beliebig Unterprogramme zu verwenden. Sollte darin dennoch ein Fehler auftreten, kommt es unweigerlich zum GAU für das Betriebssystem.

13.6 Exception–Handler in C

Das Thema Exceptions wurde im vorangegangen Kapitel bereits kurz angesprochen. Da RTOS–UH in näherer Zukunft in der Lage sein soll, Signale auf Hochsprachenebene abzufangen und zu verarbeiten, betrachten Sie die hier vorgestellte Implementierung bitte als einen Notnagel, der nur bis zur endgültigen Lösung des Problems auf Systemebene Bestand haben wird.

Die nun folgende Beschreibung reisst das Thema nur oberflächlich an, kann aber als Grundlage eines kompletten Exception–Handlers dienen. Eine wesentliche Einschränkung ist dabei jedoch zu beachten. Das RTOS–UH hat ein gestörtes Verhältnis zu den Exception–Stackframes der Motorola–Prozessoren. Man merkt hier nur zu deutlich, dass der MC68000 der Stammvater des Betriebssystems ist. Fast alle Nachfolgemodelle bieten erweiterte Möglichkeiten an, die bei der Erkennung und Behebung derartiger Ausnahmebedingungen nützlich oder absolut unumgänglich sind. RTOS–UH macht davon keinerlei Gebrauch und kürzt die ankommenden Stackframes, die die CPU brav abgelegt hat, auf ein für den MC68000 verständliches Format ein. Diese kurzen Stackframes sind dann allerdings nicht mehr zum Debuggen geeignet.

Was nun folgt, stellt keinen Exception–Handler dar, der den des RTOS–UH ersetzen soll. Dazu sind die betreffenden Routinen zu stark ins System integriert. Eine `BUS ERROR`–Routine muss nicht nur in einem speziellen Einsatzfall für eine spezielle Task korrekt funktionieren, sondern für alle denkbaren Situationen gerüstet sein — nach Murphy treten die dann auch alle auf. Deshalb sollten Sie schon über gewisse Grundlagen in Assemblerprogrammierung verfügen und die entsprechenden Handbücher für ihre CPU neben sich liegen haben, um auch nur den Hauch einer Chance zu besitzen. Blutigen Anfängern sei vom Weiterlesen abgeraten, denn hier geht es hautnah an die CPU und tief in den Kern des RTOS–UH.

```
#pragma EXCEPTION IROFF
void Trace( void  *registers , void  *stackframe ){ ... }
```

Mittels dieses Programmrumpfes entsteht Code für einen Exception–Handler. Das Schlüsselwort `IROFF` gibt zudem an, dass der Handler komplett auf Level 7 laufen soll und nicht von Interrupts unterbrochen werden kann. Der Startupcode für solche Exception–Handler begeistert zwar nicht durch

seine Laufzeit–Effizienz, gibt Ihnen jedoch die Möglichkeit, auf Hochsprachenebene an alle wichtigen Register und Speicherinhalte zu gelangen. Der Compiler generiert Code, der beim Auftreten einer Exception eine Kopie der aktuellen CPU–Register auf dem Stack anfertigt. Anschliessend wird eine Funktion aufgerufen, in der Sie persönlich entscheiden können, was im Falle der Exception zu geschehen hat.

```
      .CODE
      .EVEN
   Trace:
      ORI.W    #$0700,SR
      MOVE.L   SP,-(SP)
      MOVEM.L  D0-A6,-(SP)    ; Register retten
      PEA      (64.W,SP)      ; Pointer auf ESF
      PEA      (4.W,SP)       ; Pointer auf Registersatz
      BSR      ~~Trace
      ADDQ.W   #$8,SP         ; Super-Stack korrigieren
      MOVEM.L  (SP)+,D0-A6    ; Register restaurieren
      MOVEA.L  (SP)+,SP
      TST.W    $800.W
      BMI      ~~DISP_Trace
      RTE                     ; Normal end of exception Trace
   ~~DISP_Trace:
      JMP      $906.W         ; Exit via dispatcher
   ~~Trace:
      RTS                     ; End of function Trace
```

Abbildung 13.13: Einstieg in Exception–Handler

Bei der Behandlung derartiger Exceptions stehen Ihnen die beiden Pointer `registers` und `stackframe` zur Verfügung. `registers` zeigt auf eine Struktur, deren Aufbau Sie Abbildung 13.14 entnehmen können.

```
   typedef struct StoredRegisterSet
   {  unsigned long   _D0,  _D1,  _D2,  _D3  ;
      unsigned long   _D4,  _D5,  _D6,  _D7  ;
      char           *_A0, *_A1, *_A2, *_A3  ;
      char           *_A4, *_A5, *_A6, *_SSP ;
   }  StoredRegisterSet ;
```

Abbildung 13.14: Gerettete Register bei Exception–Handlern

Der Pointer `stackframe` könnte z.B. mittels einer Struktur gemäss Abbildung 13.15 verwaltet werden.

Zu Beginn liegt immer ein Wort mit den Statusregisterinhalt (SR) des unterbrochenen Prozesses — Ausnahmen bestätigen auch hier die Regel. Es folgt ein Langwort mit dem *program counter* der Task und ein weiteres Wort mit der *vector number* der aufgetretenen Exception. Diesem letzten Eintrag können Sie entnehmen, ob und wie es auf dem Stack weitergeht. Als Beispiel sind hier in der folgenden Union ein paar denkbare Stackframes aufgeführt. Aber da kann Ihnen nur ein gutes CPU–Handbuch weiterhelfen. Eine ausführliche Beschreibung ist im Rahmen eines Compiler–Manuals wohl kaum möglich.

Bei allen Prozessoren, die vom MC68000 abweichende Stackframes benutzen, ist es zwingend notwendig, die abgelegten Stackframes zu modifizieren, bevor der Exception–Handler in den Dispatcher hüpft. Ansonsten sind Sie zum letzten Mal gehüpft — um es mal so auszudrücken!

Bei derartigen Veränderungen muss auch der Supervisor–Stackpointer angepasst werden. Dies ge-

```
typedef struct ExceptionStackFrame
{  unsigned short     sr            ;
   void              *pc            ;
   unsigned short     format_vector ;
   /* CPU-abhaengig */
   union
   {  SixWordFrame         six_word        ;
      MidInstructionFrame  mid_instruction ;
      ShortBusCycleFrame   short_bus_cycle ;
      LongBusCycleFrame    long_bus_cycle  ;
   }  ext ;
}  ExceptionStackFrame ;
```

Abbildung 13.15: Aufbau eines Exception–Stackframes

schieht von der Ebene der Hochsprache über die Modifikation der Struktur, auf die `registers` weist. Diese Struktur dient nicht nicht der Sicherung der Registerinhalte der unterbrochenen Task, sondern kann zur Veränderung des Supervisor–Stackpointers herangezogen werden. Nach Beendigung des von Ihnen kodierten Handlers übernimmt der Rechner automatisch alle Registerinhalte der Struktur (siehe Abbildung 13.13).

13.6.1 Exceptions für EPROM–Systeme kodieren

Um Exception–Handler zu kodieren, die sich in der Startphase von RTOS–UH selbsttätig auf den zu betreuenden Vektor einklinken, steht das Schlüsselwort `VECTOR` zur Verfügung. Mittels `VECTOR addr` wird der CCC angewiesen, eine entsprechende Scheibe im Code zu generieren, um den Pointer auf die nachfolgende Funktion — eben auf den Exception–Handler — im Vektor `addr` einzutragen.

```
#pragma EXCEPTION  VECTOR 0x08  IROFF
void EXCEPTION_BusError( StoredRegisterSet    *registers ,
                         ExceptionStackFrame  *stackframe  )
{
   /* Vector 0x08 == BUS-ERROR */
   stackframe = BusAddress( registers, stackframe ) ;

   /* Meldung BUS-ERROR (TRAP) ausgeben */
   CommonErrorHandler( registers, stackframe, 0x8015 ) ;
}
```

Tabelle 13.3: Beispiel eines Exception–Handlers in C

Das Beispiel zeigt, wie das `#pragma EXCEPTION`–Kommando verwendet werden kann, um den `BUS ERROR`–Vektor (`0x08`) durch eine eigene C–Funktion zu ersetzen.

13.6.1.1 Kodierung von Traps

CREST–C stellt mittels des Schlüsselwortes `TRAP` eine Möglichkeit bereit, eigene Traps zu kodieren — was angesichts der Tatsache, dass RTOS–UH diese Vektoren selbst benutzt (siehe Tabelle 13.4), nur der Vollständigkeit halber implementiert wurde.

Die Schreibweise `TRAP nr` trägt an der Position `0x80+4*nr` den Pointer auf den Exception–Handler ein. `nr` muss ein Wert zwischen `0` und `15` sein.

```
#pragma EXCEPTION  TRAP 0  IROFF
void TRAP_ActivateTask_Quick( StoredRegisterSet  *registers )
```

TRAP	Vektor	Bezeichnung
0	$80	ACTQ
1	$84	TERMI
2	$88	CON
3	$8C	QDPC
4	$90	----
5	$94	SCAN
6	$98	REQU
7	$9C	RELEA
8	$A0	FETCE
9	$A4	RELCE
10	$A8	XIO
11	$AC	PENTR
12	$B0	RETN
13	$B4	TOQ
14	$B8	(TOV)
15	$BC	IROFF

Tabelle 13.4: Belegung der RTOS–UH–Traps

```
{
   /*
    *    Vector 0x80 == TRAP 0
    *    D1 -- Prioritaet der Task
    *    A1 -- Adresse der Task
    *    Die Task, deren TID in A1 uebergeben wurde, wird mit der
    *    Prioritaet D1 gestartet.
    */
   register Task   *task = (Task*)registers->_A1 ;
   register Prio    prio = (Prio) registers->_D1 ;

   ActivateTask( task, prio ) ;
}
```

Tabelle 13.5: Beispiel eines Traps in C

13.6.1.2 Kodierung von Line–A–Funktionen

Die Option `LINE-A A0xx` bietet die Möglichkeit, eigene Line–A–Routinen zu schreiben. `LINE-A A0xx` trägt an der Position `0x400+xx*2` den Pointer auf die betreffende Funktion ein. Der RTOS–UH–eigene Line–A–Handler springt dann Ihre selbstgeschriebene Line–A–Funktion entsprechend dem vorgefundenen Operationcode, der die Line–A–Exception ausgelöst hat, an.

`xx` muss ein gerader Wert zwischen 0x00 und 0xFE sein. Die Indizes von `0x00` bis `0x7E` sind für das Betriebssystem reserviert. Für eigene Line–A–Funktionen stehen die 64 Einträge von `0x80` bis `0xFE` zur freien Verfügung.

```
#pragma EXCEPTION  LINE-A 0xA028  IROFF
void TRAP_Suspend( void )
{
   /*
    *    LINE-A: $A028
    *
    *    wird auf Vektor 0x400 + 0x28*2 == 0x450 eingeklinkt
```

```
     *
     *    Die gerade laufende Task wird suspendiert.
     */
    SYSVAR( TID )->block |= BLKBSU ; /* Suspend-Bit setzen */
    --SYSVAR( DPC ) ;
}
```

Tabelle 13.6: Beispiel einer Line–A–Funktion in C

13.7 Kalt– und Warmstart–Code

Bekanntlich konfiguriert sich das RTOS–UH beim Hochlaufen mittels eines Scheiben–Konzepts selbstständig. Beim Systemstart durchläuft das Betriebssystem eine Reihe von Initial–Routinen und durchsucht dabei bestimmte Bereiche nach Modulen, aus denen sich das komplette System zusammensetzen soll. Bei einem Kaltstart — z.B. nach dem Einschalten oder dem Kommando `SYSTEM_RESET` — wird dabei nach Scheiben mit den Kennungen 15 und 18 gesucht. Dabei steht die Scheibe 18 für Routinen, die nach jedem Kaltstart automatisch ausgeführt werden müssen. Es werden zunächst alle 18er–Scheiben in der Reihenfolge exekutiert, in der sie von RTOS–UH gefunden wurden. Ist diese Schritt abgeschlossen, so wird dieser Vorgang mit 15er–Scheiben wiederholt. Die Warmstartscheiben werden, wie der Name schon erahnen lässt, auch nach jedem `ABORT` exekutiert.

Hinter diesen beiden Scheiben steckt eigentlich nichts geheimnisvolles. Es bedeutet nur, dass das System nach Codebereichen forscht, die ohne Bedienerkommandos automatisch ausgeführt werden sollen. Da Sie sich zudem noch im Supervisor–Mode befinden, wenn die Routinen abgearbeitet werden, ist Vorsicht geboten. Es gilt noch eine weitere Einschränkung, die sich auf den Gebrauch der RTOS–UH–Systemdienste bezieht. Im Grunde genommen ist das Betriebssystem zu diesem Zeitpunkt noch in der Wachstumsphase und noch nicht besonders willig, Ihnen zu Diensten zu sein. Sie sollten deshalb davon Abstand nehmen, in der Hochlaufphase Systemkommandos zu verwenden. Das gilt selbstverständlich auch für CREST–C–Bibliotheksroutinen, die sich auf dem System abstützen. Zugriffe auf Variablen, die `A4`– oder `A5`–relativ angesprochen werden, sind unzulässig. Es gibt weder eine zugeordnete C–`main()`–Task, deren globale Variablen über `A5` adressiert werden könnten, noch existiert überhaupt eine Task, deren Taskworkspace über `A4` erreichbar wäre.

Auch alle Bemerkungen aus dem Abschnitt 13.5 bezüglich des Aufrufers, der Verfügbarkeit von globalen Variablen, Rückgabewerten und Übergabeparameter treffen auch hier zu. Da Sie auch hier auf dem Supervisor–Stack arbeiten, sollten Sie es mit den lokalen Parametern nicht übertreiben, da das System sich sonst mit grosser Wahrscheinlichkeit schon beim Hochlaufen kommentarlos von Ihnen verabschiedet.

Also ein kleines Beispiel: wenn Sie Peripherie angeschlossen haben, die nach dem Systemstart Streicheleinheiten benötigt, so funktioniert das mittels der folgenden `#pragma`–Syntax:

```
#pragma COLDSTART
void Kaltstart( void ){ ... }
```

Entsprechend funktioniert die Angelegenheit, wenn Sie nach jedem `ABORT` bestimmte Anweisungen ausführen müssen:

```
#pragma WARMSTART
void Warmstart( void ){ ... }
```

Kapitel 14

Über Programme, Tasks und Shellmodule

Da CREST–C–Programme sich nicht im luftleeren Raum bewegen, sondern fest eingebetet unter RTOS–UH dessen Features nutzen, folgt nunmehr eine kleine Einführung in einige ausgewählte Bereiche des Betriebssystems. Um das Verhalten von C–Programmen verstehen zu können, sollten Sie unbedingt diese Kapitel durchlesen und auch begreifen. Für Anwender, die von Fremdbetriebssystemen kommen, sind manche Dinge zunächst ungewöhnlich und oder verwirrend. Als Lösung für ein schnelles Echtzeitbetriebssystem sind die gewählten Ansätze von RTOS–UH jedoch meist sehr elegant — aber leider teilweise auch hochgradig gewöhnungsbedürfig.

14.1 Taskkopf und Taskworkspace

Wenn man sich anschaut, was eine Task unter RTOS–UH ausmacht, dann stolpert man zunächst über den Begriff des Taskkopfes. Wäre RTOS–UH kein Multi–Tasking–System, so würde der nackte Motorola–Maschinencode ausreichen, um ein Programm auf der CPU ablaufen zu lassen. Der nackte Chip braucht prinzipiell nur seinen Start–PC und etwas korrekten Maschinencode und schon kann man von einem Programm sprechen. Soll ein Rechner mehrere Programme verwalten können, so muss vom Betriebssystem — was ja selbst schon das *erste* Programm repräsentiert — einige Verwaltungsarbeit geleistet werden.

Die Verwaltungsinformationen werden unter RTOS–UH im Taskkopf abgelegt. Dort findet die Verknüpfung von Maschinecode mit einem symbolischen Namen — dem Tasknamen — statt. Ebenso werden hier die Informationen über Prioritäten, Einplanungen, Aktivierungen und wesentliche Systembedingungen gehalten. Die Abbildung 14.1 zeigt den Aufbau eines Taskkopfes.

Eine Task benötigt also erstmal eine solche Verwaltungsstruktur. Das entspricht in etwa der Geburtsurkunde im realen Leben, wenn man sich an ein Amt wenden möchte. Wer einen Personalausweis beantragen möchte, sollte eben auch besser ein Papier über sein Existenz als Staatsbürger vorweisen können — ansonsten könnten *Echte Beamte* darauf insistieren, dass man eigentlich nicht existiert.

Das Betriebssystem arbeitet beim Umgang mit Tasks grundsätzlich mittels der Taskköpfe. Oder anders ausgedrückt: **Ein Anwenderprogramm unter** RTOS–UH **hat immer die Form einer Task!**

Zum wirklichen Leben genügt ein nackter Taskkopf aber noch nicht. Man kann eine Task zwar identifizieren und ansprechen, aber zum Anlaufen benötigt sie weitere Schützenhilfe vom Betriebssystem. Wird eine Task aktiviert, so sind im Vergleich zur bereits existierenden (aber schlafenden) Task weitere Verwaltungsinformationen notwendig.

Auch hier greift das Beispiel mit dem Amtsschimmel: wenn Sie als Bürger Ihre vier Wände verlassen wollen, dann benötigt jeder Beamte, auf den Sie nunmehr stossen, ein Papierchen für Einträge und

```
struct Task
{  MemSectionHeader    head                      ;
   MemSectionName      name                      ;
   Prio                default_prio              ;
   WorkSpaceLen        workspace                 ;
   Task               *fors                      ;
   Task               *backs                     ;
   Prio                run_prio                  ;
   void               *start_pc                  ;
   Block               block                     ;
   Schedule            sched                     ;
   Taskworkspace      *wto                       ;
   ActCnt              activation_count          ;
   CeCnt               CE_count                  ;
   TimeSchedule        time_continue             ;
   TimeSchedule        time_activate             ;
   TimeSchedule        time_interval             ;
   TimeSchedule        time_last                 ;
   Breakpoint          brkadr                    ;
   Prio                schd_prio                 ;
   ConfFlag            newstart_flag             ;
   ConfFlag            fpu_flag                  ;
   StatusReg           initial_SR                ;
   void               *message_link              ;
   void               *signal_link               ;
   Ce                 *t_ce_fors                 ;
   Ce                 *t_ce_backs                ;
   Prio                t_ce_prio                 ;
   UWORD               dummy                     ;
   Prio                actbuf[ MAX_ACTIVATIONS ] ;
} ;
```

Abbildung 14.1: Aufbau eines Taskkopfes

Kontrollen — den Zettel für die Stempeluhr! Eine Task, die anlaufen möchte, lässt sich zu diesem Zwecke einen weiteren Verwaltungsblock vom Betriebssystem zuteilen, in dem RTOS–UH dann zu jeder beliebigen Zeit eintragen kann, wo und wie die Task gerade im System unterwegs ist.

Diese Verwaltungsstruktur wird als Taskworkspace (`TWSP`) bezeichnet. Im Taskkopf wird über die Struktureinträge `workspace` (für die Länge des `TWSP`) und `wto` (als Pointer auf den Speicherblock) eine Verkettung von Taskkopf und Taskworkspace vorgenommen. Der Aufbau des `TWSP` kann der Abbildung 14.2 entnommen werden.

Auf Rechnern, die keine FPU besitzen oder für Tasks, die nur pure Integerarithmetik betreiben, spart RTOS–UH den Platz für die Einträge `fpu_regs` und `fpu_format` ein — entsprechend ist der Eintrag `workspace` im Taskkopf auch deutlich kleiner und das Byte `fpu_flag` enthält eine Null.

Nun kann eine Task endgültig anlaufen. Wenn RTOS–UH zwischen den Tasks hin und her schaltet, so wird der jeweils aktuelle CPU/FPU–Zustand, mit dem die gestoppte Task unterwegs war, in den `TWSP` geschrieben und von dort auch wieder nachgeladen, wenn RTOS–UH beschliesst, der Task wieder CPU–Zeit zur Verfügung zu stellen. Solange eine Task nicht auf der Systemuhr nachschaut, merkt sie bei einer Fortsetzung absolut nichts davon, dass ihr der Prozessor für eine gewisse Zeit entzogen wurde.

Terminiert sich eine Task, so wird — von den Ausnahmen abgesehen — der `TWSP` wieder als frei-

```
struct Taskworkspace
{  MemSectionHeader      head              ;
   MemSectionHeader     *fort              ;
   MemSectionHeader     *backt             ;
   Task                 *TID_of_owner      ;
   PRegister             A7UW              ;
   PRegister             A7SW              ;
   PRegister             data     [ 8 ]    ;
   PRegister             address  [ 7 ]    ;
   UWORD                 sys_stack[ 6 ]    ;
   MemSectionName        opname            ;
   TimeSchedule          opfati            ;
   TimeSchedule          opintv            ;
   TimeSchedule          oplti             ;
   UWORD                 pearl_buffer[ 21 ] ;
   LineNo                lineno            ;
   WORD                  __dummy           ;
   long double           fpu_regs  [   9 ] ;
   char                  fpu_format[ 184 ] ;
} ;
```

Abbildung 14.2: Aufbau des System–Taskworkspaces

er Speicher ans Betriebssystem zurückgegeben. Auch die Speicherketten, die während der Laufzeit der Task angelegt wurden, werden wieder abgebaut. Die Terminierung einer Task bewirkt eine Art Aufräumaktion im Speicher. Es werden sämtliche zur Laufzeit der Task angeforderten Speicherbereiche (das betrifft den Procedureworkspace und auch die CE–Ketten) wieder freigegeben — oder so behandelt, dass sie, wenn sie ihre Aufgabe erfüllt haben, automatisch verschwinden.

Und jetzt kurz zu den erwähnten Ausnahmen. Eine Task, die mit dem Typ `TASK resident` bzw. `ATSK resident` (siehe Tabelle 14.1) eingetragen ist, behält ihren TWSP auch nach der Terminierung. Der Vorteil besteht darin, dass bei einer erneuten Aktivierung bereits ein Taskworkspace existiert und nicht neu geschaffen werden muss.

Besteht zum Zeitpunkt der Terminierung einer Task bereits eine Einplanung auf ein weiteres Ereignis, so wird ebenfalls der TWSP nicht freigegeben — weil das Betriebssystem ökonomisch *denkt* und sich sagt, dass ein solcher Taskworkspace demnächst — wann auch immer das sein mag — wieder benötigt wird und es keinen Sinn macht, das Teil erst zu verschrotten und dann neu zu beschaffen…

14.2 Taskköpfe bei Shellmodulen

Während für jedes Maschinenprogramm, das als Task kodiert ist, bereits ein Taskkopf vorhanden sein muss, ist es für Shellmodule etwas beschwerlicher. Ein Shellmodul hat keinen eigenen Taskkopf! Wenn RTOS–UH über die Shell feststellt, dass ein Shellmodul angesprochen wird, so generiert es erstmal einen eigenständigen Taskkopf für eben diese Anforderung und verknüpft diesen Taskkopf mit dem Maschinencode, dessen Position im Shellmodulkopf angegeben ist. **Anschliessend gibt es eine neue Task im System!** Nunmehr besteht *kein wesentlicher* Unterschied mehr zwischen einer Task, die bereits mit eigenem Taskkopf kodiert wurde und einer Task, deren Taskkopf dynamisch für ein Shellmodul angefordert wurde.

Unterschiede werden erst wieder akut, wenn die reinrassige (statische) Task bzw. die Task des Shellmoduls zu dem Entschluss kommen, dass die Arbeit nun getan sei und sich terminiert. Eine Task, die

aus einem Shellmodul entstand — auch als **Subtask** oder **dynamische Task** bezeichnet — ist als kleines Helferlein konzipiert, die eine Aufgabe erfüllen und dann wieder aus dem System verschwinden soll. Der übliche Abschluss einer dynamischen Task besteht demnach darin, RTOS–UH darüber zu informieren, dass sie ihre Aufgabe beendet hat und gerne verschwinden möchte. Wenn keine weiteren Aktivierungen oder Einplanungen für die dynamische Task vorliegen, so räumt RTOS–UH die internen Ketten der Task ab, vernichtet den `TWSP` — worin bis jetzt noch kein Unterschied zu einer normalen Task besteht — **und** schmeisst den zuvor generierten Taskkopf der Subtask aus dem Speicher: die dynamische Task ist tot und aus dem System verschwunden!

Bei normalen Tasks bleibt der Taskkopf auch nach der Terminierung der Task im Speicher liegen und steht demnach bei Neuaktivierungen sofort wieder zur Verfügung. Die Generierung einer dynamischen Task aus einem Shellmodul kostet dagegen etwas mehr Zeit, da der Taskkopf erst noch geschaffen werden muss. Sie sollten also bei der Auswahl, ob ein C–Programm als Task oder als Shellmodul laufen soll, immer abwägen, ob Sie wirklich Übergabeparameter oder den Vorteil, dynamisch mehrere Tasks auf identischem Code ablaufen lassen zu können, benötigen. Der riesige Nachteil der Shellmodule liegt zudem darin, dass Sie keine autostartfähigen Shellmodule kodieren können! Eine Task vermerkt ihre Autostartfähigkeit im Taskkopf und Shellmodule besitzen eben keinen.

14.3 Speichersektionen

RTOS–UH unterteilt den Speicher eines Rechners in verschiedenartige Speichersektionen. Dieser Vorgang findet dynamisch statt und passt sich den jeweiligen Anforderungen des Nutzers und der Tasks im System an — der Vorteil des dynamischen Verhaltens des Betriebssystems, was die Zahl der Speicherbereiche angeht, wird dadurch erkauft, dass das Laufzeitverhalten bei manchen Operationen nur noch sehr begrenzt als deterministisch zu bezeichnen ist.

RTOS–UH verwaltet den RAM–Speicher des Rechners als eine doppeltverkettete Liste von Sektionen. Es gibt keine MMU, die irgendwelche virtuellen Adressen zur Verfügung stellt. Eine jede Speichersektion beginnt grundsätzlich mit einem Verwaltungskopf, dem `MemSectionHeader`, dessen Aufbau Sie Abbildung 14.3 entnehmen können. Beispiele für deren Verwendung sind z.B. die bereits vorgestellten Strukturen von Taskkopf (siehe Abbildung 14.1) und Taskworkspace (siehe Abbildung 14.2).

```
typedef struct MemSectionHeader
{  struct MemSectionHeader  *forward  ;
   struct MemSectionHeader  *backward ;
   UserNumber                owner    ;
   MemSection                typ      ;
}  MemSectionHeader ;
```

Abbildung 14.3: Aufbau eines MemSectionHeader's

Mittels `forward` und `backward` wird die bereits erwähnte Verkettung der Sektionen verwaltet. Sie als Programmierer sollten sehr vorsichtig sein, wenn Sie mit derartigen Ketten spielen. In C ist das Spielen selbst kein Problem — das Problem beginnt erst dann, wenn Sie nicht begriffen haben, was Sie da tun und RTOS–UH mal so eben mit Fehlermeldungen nach Ihnen wirft oder gar gen Himmel schwebt. Unsinnige Veränderungen in der Kette führen **garantiert** zum schnellen Ableben des Betriebssystems.

Wenn Ihr Rechner z.B. beim `S`– oder `L`–Kommando ein paar Zeilen ausgibt und dann mit `BUS ERROR` nach Ihnen wirft, dann hat es mit hoher Wahrscheinlichkeit diese Kette des Systems erwischt. Mit etwas Glück kann man eine solche angemackelte Kette noch mit ein paar gezielten `SM`–Operationen reparieren, aber meist führt selbst ein derartiger handwerklicher Eingriff nicht zu Erfolg — zumindest, solange Sie nicht ganz genau wissen, was da wirklich im System abgeht...

Im Eintrag `owner` im `MemSectionHeader` enthält die Usernummer des jeweiligen Eigentümers

einer Speichersektion. Der Bediener an der Hauptkonsole (meist #USER1) hat hier die Nummer 0. Wenn Sie über andere Schnittstellen in den Rechner hineinschauen und Speicherblöcke anfordern, so enthalten diese im owner–Eintrag die Kennung des jeweiligen USER's.

Der Eintrag typ gibt an, um welche Art von Speichersektion es sich bei dem jeweiligen Block handelt. Beim S–Kommando werden die zulässigen Bitmuster in lesbare Namen umgesetzt. In Tabelle 14.1 ist eine Zusammenstellung der dokumentierten Sektionstypen aufgeführt.

Name	**Bitmuster**	**Bedeutung**
FREE	0x00	Freier Speicherbereich
MARK	0x00	Start/Ende owner $80
TASK	0x01	Normale Task
ATSK	0x41	Task: autostartfähig
TASK Resident	0x81	Task: resident
ATSK Resident	0xC1	Task: autostartfähig+resident
TWSP	0x02	Task–Workspace
CWSP	0x04	Communication–Workspace
PWSP	0x08	Procedure–Workspace
MDLE	0x10	Module
SMDL	0x50	Shellmodul
PMDL	0x90	prombares Modul
EDTF	0x20	Editor-File

Tabelle 14.1: MemSection–Typen

Kapitel 15

RTOS–UH — Der Einstieg

Es fällt den meisten Einsteigern schwer, einen sinnvollen Zugang zum Betriebssystem RTOS–UH zu finden. Das System ist schnell und effizient, aber nicht sonderlich nett zu seinen Programmierern. Wer als blutiger Anfänger das RTOS–UH–Manual in die Hand nimmt, hat harte Zeiten vor sich — und so lange ist es noch nicht her, dass ich selbst voller Verzweifelung und Abscheu versucht habe, die entscheidenden Kapitel zu finden, die die Erleuchtung bringen sollten. Es gibt sie nicht! Das RTOS–UH–Manual ist als Nachschlagewerk und nicht als Lehrbuch konzipiert und erfüllt diese Aufgabe auch recht gut.

Als Einsteiger in dieses Betriebssystem ist aber erstmal die Hürde eines gewissen Grundverständnisses zu überwinden. Das beginnt schon mit der verwendeten Nomenklatur. Das Stichwortverzeichnis des RTOS–UH–Manuals ist bestimmt nicht so schlecht, wie es oft dargestellt wird, aber das Manko besteht darin, dass man zu Beginn nicht den geringsten Schimmer hat, was sich hinter den Begriffen verbirgt, die dort aufgeführt sind und zudem praktisch keine Zusammenhänge erfährt.

Zunächst ein paar Worte zur Echtzeitfähigkeit des RTOS–UH. Umsteiger von *Single–User/Single–Tasking*–Betriebssystemen kennen das Problem wohl kaum. Plötzlich toben auf einem Rechner mehrere Programme gleichzeitig durch den Prozessor. Das ist auf Grossrechnern schon ewig kein weltbewegendes Thema mehr und auch PC–Anwender kommen langsam in den Genuss dieser Möglichkeiten. Auch unter Betriebssystemen wie LINUX, OS/2 und WINDOWS–NT besteht die Möglichkeit, mehrere Prozesse, Jobs oder Tasks quasi–parallel auf demselben Prozessor (oder gar mehreren Prozessoren) ablaufen zu lassen. Das Betriebssysten schaltet mit mehr oder weniger ausgeklügelten Algorithmen zwischen diesen Jobs hin und her, sodass jeder Prozess für eine bestimmte Dauer Rechenzeit zugewiesen bekommt und damit *irgendwann* seine Arbeit fertigstellen kann. Gerade in diesem Punkt unterscheidet sich RTOS–UH von derartigen Betriebssystemen. In Fällen, in denen *irgendwann* vielleicht den Zeitpunkt darstellt, an dem Sie — oder Ihre Kunden — bereits die dritte Harfenstunde auf Wolke 17 genommen haben, scheint es sinnvoll, sich etwas mit den Vorgaben bei der Entwicklung der Betriebssysteme zu beschäftigen.

Echtzeitbetriebssysteme unterscheiden sich von normalen Grossrechnerbetriebssystemen in erster Linie durch die unterschiedliche Bewertung der Wichtigkeit verschiedener Aufgaben. In Rechenzentren kommt es darauf an, die zur Verfügung stehende CPU–Zeit möglichst demokratisch an die Benutzer zu verteilen. Auf PC's geht es meist nur darum, einen einzelnen Benutzer glücklich zu machen — was sich zumeist in einer hohen Priorität der Benutzerführung niederschlägt, um genervte Mäusebesitzer nicht zusehr zu verärgern.

Die reale Welt der Prozesssteuerung und –regelung sieht da deutlich anders aus. Hier kommt es nicht auf eine gleichmässige Verteilung der Rechenzeiten, eine gute Ausnutzung des Gesamtsystems oder auf die Befriedigung des Bedieners an. Hier gilt es, möglichst sofort auf Ereignisse zu reagieren. Ein Zeitscheibenverfahren, das Ihnen alle drei oder vier Sekunden etwas Rechenzeit zugesteht, um den

Status einer Turbine zu überprüfen, dürfte hier allenfalls ausreichen, um einem durchgegangenen Läufer auf dem Weg durchs Hallendach ein freundliches Lebwohl nachzurufen. In einem solchen Falle würde es Sie bestimmt auch nicht versöhnlicher stimmen, zu erfahren, dass das Betriebssystem sich keiner Schuld bewusst war und Ihre restlichen Kollegen pflichtschuldigst beim Schach– oder Tetrisspielen an ihren Terminals bedient hat.

Bei der Bedienung von Prozessereignissen gilt es für ein Betriebssystem, möglichst sofort zu reagieren. Dieses *möglichst sofort* stellt selbstverständlich nur ein Ideal dar, da es natürlich *echt Zeit* beansprucht, einem Haufen dusseliger Chips zu verraten, dass es etwas zu tun gibt. Der Begriff der Echtzeitfähigkeit lässt sich auf viele Arten definieren. Mit brutaler Gewalt in Gestalt schneller und kostspieliger Computer lassen sich heute bereits viele Aufgaben mit konventionellen Betriebssystemen lösen, für die früher hochspezialisierte Echtzeitkerne notwendig waren. Dennoch ist es reichlich übertrieben, Produkte wie OS/2 und WINDOWS–NT, die nur von Taktfrequenz und Chiptechnologie leben, als echtzeitfähig zu bezeichnen. Ein Flugzeugträger ist nicht dafür gebaut, an Schnellbootrennen teilzunehmen, obwohl die PS–Ausstattung einer solchen schwimmenden Festung die kleinen Renner ziemlich blass aussehen lässt.

15.1 Das Betriebssystem

RTOS–UH soll als Betriebssystem die Resourcen eines Rechners (Speicher, Prozessor, I/O–Schnittstellen) verwalten und Anwenderprogrammen Dienstleistungen zur einfachen Nutzung dieser Resourcen zur Verfügung stellen. Um diese Aufgabe effizient zu erfüllen, bedient sich RTOS–UH dreierlei Hilfsmittel: es besteht aus einem Betriebssystemkern, dem sogenannten Nukleus, der über Systemaufrufe, Traps, die Manipulation von Tasks kontrolliert und den Systemspeicher verwaltet, aus Interrupt–Routinen, die grundlegende I/O–Operationen abwickeln, und aus Systemtasks, die komplexere Verwaltungsaufgaben erledigen. In den folgenden Kapiteln erfolgt zunächst eine grundsätzliche Betrachtung des Taskings unter RTOS–UH. Der Begriff der Task ist fundamental für das Verständnis des Systemverhaltens.

15.1.1 Nomenklatur der C–Funktionen

CREST–C unterstützt die Möglichkeiten des RTOS–UH durch die Bereitstellung aller *wesentlichen* RTOS–UH–Traps auf Funktionsebene. Die Namensgebung der Systemaufrufe folgt nicht den Mnemonics der entsprechenden Traps, die im RTOS–UH–Handbuch festgelegt sind. Stattdessen wurden sprechende Namen gewählt, die sich leichter verstehen lassen und — meiner Ansicht nach — fast selbstdokumentierend sind.

Der Basisname der Trap– oder RTOS–UH–spezifischen Anschlüsse lautet **`rt_`**. Darauf folgt eine Art Kurzbeschreibung der jeweiligen Funktion. Die erforderlichen Prototypen sind in der Includedatei `<rtos.h>` enthalten. Die Verwendung dieser Funktionen führt mit absoluter Gewissheit zu ernsthaften Problemen bei der Portierung auf ein anderes Betriebssystem. Die folgenden Erklärungen betreffen alle Funktionen, die die entsprechenden Parameter enthalten und sind dort nicht mehr explizit beschrieben.

15.1.1.1 Relative oder absolute Zeitangaben

Zeiten können relativ oder absolut angegeben werden. Die Angabe erfolgt grundsätzlich in Millisekunden. Absolute Zeiten beschreiben feste Zeitpunkte, z.B. `19:45:05`. Relative Zeiten geben eine Zeitdauer ab dem aktuellen Zeitpunkt an, z.B. nach 5 Sekunden. Unterschieden werden absolute und relative Zeiten durch das oberste Bit des betreffenden 32–Bit–Langwortes. Ist es gesetzt, so wird die

Zeit als relativ zum Aufrufzeitpunkt der Funktion interpretiert. Aus einer absoluten Zeit kann folgendermassen eine relative gemacht werden:

```
TimeSchedule startzeit = 5000 | 0x80000000UL ; // 5000 ms
```

15.1.1.2 Trapinterne Tasksuche

Es gibt für die Traps, die mit einer Task arbeiten sollen, zwei unterschiedliche Mechanismen, um diese zu identifizieren.

- Der Trap bekommt einen Pointer auf den Tasknamen übergeben. Dann sucht er in der Speicherverwaltung, ob eine solche Task bekannt ist. Diese Suche kostet natürlich Zeit, um so mehr, je voller der Speicher ist. Allerdings erhält man für die eingezahlte Zeit die Sicherheit, keinen Unfug im Betriebssystem anrichten zu können. Wird die Task nicht gefunden, erhält man eine Fehlermeldung und der Aufrufer wird suspendiert. Namen, die mit einem Doppelkreuz # beginnen, werden vom Betriebssystem nicht gefunden!
- Es wird ein Pointer übergeben, der direkt auf die Task (Taskkopf) zeigt. Hier findet keine weitere Überprüfung statt. Ist an der angegebenen Stelle keine Task vorhanden, wird eine beliebige Speicherstelle manipuliert, was im Regelfall zum Abschuss des Betriebssystems führt! Haben Sie z.B. die Task von Hand entladen und vergessen, dass andere Tasks noch etwas von der entladenen Task wollen, so sind Sie für die Folgen selbst verantwortlich!

Alle Traps die im Namen ein `_quick` enthalten, suchen Tasks nach der zweiten Methode und bedürfen besonderer Vorsicht.

15.2 Tasks und Tasking

Grundlegend für RTOS–UH ist der Begriff *Task*, der ein eigenständig ablauffähiges Programm kennzeichnet. Unter RTOS–UH können diese Tasks verschiedene Eigenschaften haben und sich in verschiedenen Zuständen befinden. Das Betriebssystem verteilt die verfügbare Rechenkapazität auf die einzelnen Tasks (Multi–Tasking), und zwar derart, dass der Wechsel von einer Task zur anderen jederzeit möglich ist (Echtzeit–Fähigkeit). Bei Mehrprozessor–Rechnern können durchaus mehrere Tasks gleichzeitig aktiv sein, bei einem Einprozessor–System kann zu einem Zeitpunkt natürlich nur eine Task exekutiert werden. Im folgenden Text wird stets von einem Einprozessor–System ausgegangen. Die analoge Übertragung auf ein Mehrprozessor–System ist jedoch relativ einfach.

15.2.1 Task–Eigenschaften

Unter einer Task versteht RTOS–UH ein eigenständig arbeitendes Programm. Um unter RTOS–UH als Task anerkannt zu werden, muss dieses Programm einige Eigenschaften haben. Diese Eigenschaften müssen gemäss den Spielregeln von RTOS–UH in einem Verwaltungsblock, dem Taskkopf, notiert werden. Hochsprachen–Programmierer brauchen sich um die Generierung dieser Taskköpfe normalerweise keine Gedanken machen, dies übernimmt der Compiler, Assembler–Programmierer müssen diesen Taskkopf jedoch selbst generieren. Nach dem Laden eines Programmes muss dieser Taskkopf im Speicher vorhanden sein — oder wird von CREST–C dynamisch erzeugt. Spricht man unter RTOS–UH von der Adresse einer Task, so meint man die Adresse des Taskkopfes, die sogenannte `TID` (Task–Identifier). Neben den hier explizit aufgeführten Informationen sind im Taskkopf noch wesentlich mehr (zum Teil nur betriebssysteminterne) Daten enthalten, auf den genauen Aufbau und die weiteren enthaltenen Daten wird detailliert zu späterer Zeit eingegangen.

15.2.1.1 Priorität

Die wichtigste Eigenschaft einer Task ist für RTOS–UH ihre Dringlichkeit, die Priorität. RTOS–UH verteilt die Prozessorkapazität unter den lauffähigen Tasks gemäss deren individueller Priorität. Prioritäten werden als Ganzzahl im Bereich –32768 .. 32767 angegeben [1], in der Reihenfolge sinkender Priorität. Eine Task mit der Priorität 1 ist also dringlicher als eine Task mit der Priorität 5. Im Gegensatz zu den Zahlenwerten sagt man auch, die Task mit der Priorität 1 habe eine höhere Priorität als die Task mit der Priorität 5. Negative Werte sind, zumindest von der Intention her, dem Betriebssystem selbst vorbehalten.

Jedesmal, wenn es für RTOS–UH eine Veranlassung gibt, den Prozessor einer Task zuzuteilen, d.h. eine Task auszuführen, inspiziert RTOS–UH eine Liste der vorhandenen Tasks (die Dispatcher–Kette) und teilt den Prozessor der laufwilligen Task mit der höchsten Priorität zu. Für den Fall, dass einmal keine Task laufwillig sein sollte, enthält RTOS–UH eine „Leerlauf"–Task, die Task mit dem Namen `#IDLE`, die die niedrigstmögliche Priorität besitzt und stets laufwillig ist. Diese Task erfüllt keinerlei Aufgaben, ist aber für die Funktion des Betriebssystems sehr wichtig.

Die folgenden zwei Bibliotheksfunktionen dienen dazu, die Priorität von Tasks *zur Laufzeit* zu ändern bzw. abzufragen. Die Prototypen befinden sich in <`rtos.h`>.

```
Prio rt_set_prio( Prio  new_prio ) ;
```

Mittels `rt_set_prio()` kann die aufrufende Task ihre Run– bzw. Laufzeitpriorität neu setzen. Wird als Aufrufparameter eine Null angegeben, so wird die Defaultpriorität der Task eingesetzt. Der Rückgabewert entspricht der zuvor gültigen Laufzeitpriorität. Sie sollten dabei stets beachten, dass negative Prioritäten unter RTOS–UH den hochprioren Systemtasks vorbehalten bleiben sollten.

```
Prio rt_get_prio( int  flag ) ;
```

Mittels `rt_get_prio()` kann die aufrufende Task wahlweise ihre Laufzeit– bzw. ihre Defaultpriorität abfragen. Wird eine Null übergeben, so liefert die Funktion die Defaultpriorität. Jeder andere Wert führt zu Ausgabe der Laufzeitpriorität.

15.2.1.2 Taskname

Der Taskname ist, neben der `TID`, ein weiteres wichtiges Kennzeichen einer Task. Alle Tasks in einem Rechner sollten nicht nur über ihre `TID`, sondern auch über den Namen eindeutig voneinander unterscheidbar sein. Probleme bei Namensgleichheit treten spätestens dann auf, wenn eine Task über ihren Namen gestartet werden soll.

15.2.1.3 Speicherbedarf

Um möglichst viele Tasks im Rechner halten zu können, teilt RTOS–UH diesen den für lokale Variable erforderlichen Speicherplatz erst beim Starten zu. Daher muss die Grösse des erforderlichen Task–Arbeitspeichers, des sog. Taskworkspace (`TWSP`), im Taskkopf eingetragen sein. Beim Starten einer Task, unter RTOS–UH spricht man vom Aktivieren einer Task, teilt RTOS–UH dieser Task dann Speicherplatz in der erforderlichen Grösse zu. Beendet eine Task ihre Arbeit, unter RTOS–UH spricht man vom Terminieren einer Task, so wird dieser `TWSP` wieder zu freiem Speicher und kann vom Betriebssystem anderweitig zugeteilt werden.

[1] Nur PEARL beschneidet den Wertebereich auf 1 bis 255! Unter CREST–C sind Sie als Systemprogrammierer frei in der Wahl der Priorität von Tasks. Alleine die Trap–Anschlüsse kontrollieren, dass Sie keine Tasks mit negativen Prioritäten anwerfen können.

15.2.1.4 Residente Tasks

Erklärt sich eine Task als resident, so wird Ihr der TWSP nur bei der allerersten Aktivierung vom Betriebssystem neu zugeteilt. Sie behält diesen Speicherbereich, auch wenn sie terminiert ist. Dieses Verfahren hat den Vorteil, dass bei häufig aktivierten Tasks nicht jedesmal neu Speicher zugeteilt werden muss (Zeitgewinn) und ermöglicht andererseits einer Task, sich statische Variablen zu halten, d.h. Variablen, deren Wert über eine Terminierung bis zur nächsten Aktivierung erhalten bleibt.

15.2.1.5 Autostart–Fähigkeit

Gerade für *stand–alone*–Systeme wichtig: erklärt sich eine Task als Autostart–Task, so wird sie direkt nach dem Start des Betriebssystems von diesem aktiviert. Alle Tasks, die diese Eigenschaft nicht besitzen, werden nicht automatisch aktiviert. Sie können nur von anderen Tasks oder über Bedienbefehle gestartet werden.

Unter RTOS–UH gibt es keine andere Möglichkeit für Tasks, aus eigenem Antrieb lauffähig zu werden.

15.3 Multi–Tasking

Das Konzept von RTOS–UH beruht auf der Idee des Multi–Tasking, der Fähigkeit des Betriebssystems, mehrere eigenständige (und allein lauffähige) Programme (quasi–) gleichzeitig betreuen zu können. Diese Tasks können gleichzeitig im Rechner vorhanden sein, und das Betriebssystem verteilt die verfügbare Prozessorkapazität auf die laufwilligen Tasks.

15.3.1 Task–Zustände

Das kleine Wörtchen „laufwillig“ kennzeichnet schon, dass RTOS–UH für eine Task unterschiedlichste Betriebszustände kennt. RTOS–UH verwaltet die Tasks in ihren unterschiedlichen Betriebszuständen und stellt Mechanismen zur Veranlassung und Kontrolle von Zustandsübergängen bereit. In der Abbildung 15.1 sind die wichtigsten Betriebszustände sowie die möglichen Übergänge nebst den diese veranlassenden Operationen exemplarisch aufgeführt.

Neben den grundsätzlichen Taskzuständen DORM (schlafend), RUN (laufwillig), SUSP (ausgesetzt), SCHD (eingeplant) und SEMA (warten auf Synchronisation), die direkt durch Task– oder Bedieneroperationen erzielt werden können, gibt es zudem die eher systeminternen Zustände I/O? (wartend auf Abschluss einer I/O–Operation), PWS? (wartend auf Speicher), ???? (Mehrfachblockierung) und CWS? (warten auf I/O–Speicher), die vom Zustand RUN aus erreicht werden können, wenn eine Task eine Aktion einleitet, die vom Betriebssystem nicht unmittelbar befriedigt werden kann.

Der Zustand SCHD wird in aktuellen Systemen etwas spezifischer dargestellt, um den aktuellen Einplanungszustand genau wiederzuspiegeln. Die Bezeichnungen sind der Tabelle 15.1 zu entnehmen.

Name	Task wartet auf	Beispiel
TIAC	Timed Activate	AT,AFTER xx ACTIVATE
CYAC	Cyclic Activate	ALL xx ACTIVATE
EVAC	Event Activate	WHEN EV xxx ACITVATE
TICO	Timed Continue	AT, AFTER xxx CONTINUE
EVCO	Event Continue	WHEN EV xxx CONTINUE

Tabelle 15.1: Einplanungszustände

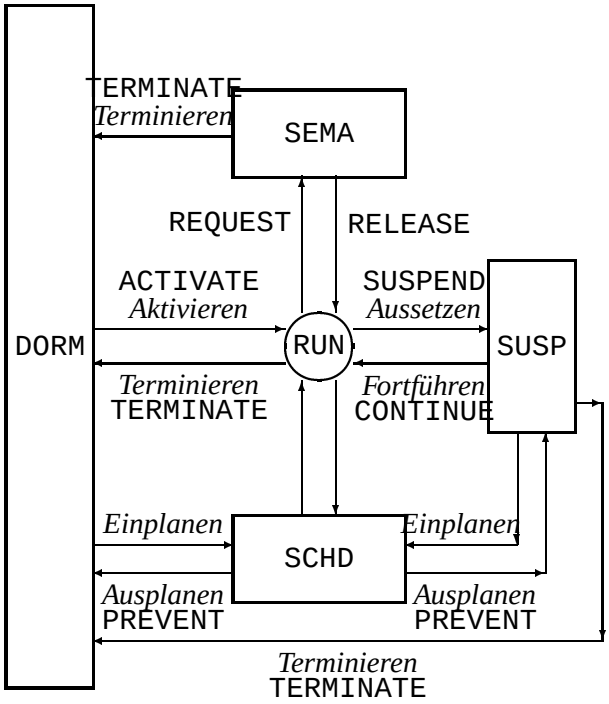


Abbildung 15.1: Zustandsübergänge

Um von C–Programmen aus an die Informationen über den aktuellen Taskzustand einer Task heranzukommen, gibt es mehrere Möglichkeiten. Ist nur der Name der zu begutachtenden Task bekannt, so dient die Funktion `rt_task_status()` dazu, die Task zu finden und den gerade aktuellen Zustand der gefundenen Task als Langwort abzuliefern.

```
ULONG  rt_task_status( char  *name ) ;
```

Konnte die Task nicht im System gefunden werden, so liefert die Funktion das Bitmuster `0xFFFFFFFF-FUL`. Ansonsten ist die Antwort gemäss Abbildung 15.2 zu interpretieren.

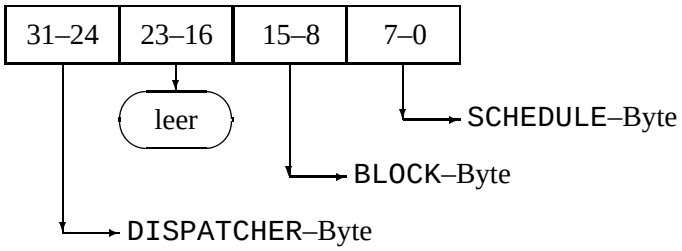


Abbildung 15.2: Rückgabewert von `rt_task_status()`

Wenn das höchste Bit im `DISPATCHER`–Byte gelöscht ist, so befindet sich die Task aktuell in der Dispatcher-Kette des Betriebssystems und ist in irgendwelche Aktivitäten verwickelt. Der Aufbau des `BLOCK`–Bytes kann Tabelle 15.2 entnommen werden

Entsprechendes gilt für das `SCHEDULE`–Byte und die Tabelle 15.3.

Die Bitmuster, die sich hinter den symbolischen Namen verbergen, sind bei Bedarf der Includedatei `<rtos.h>` zu entnehmen. Natürlich können die Informationen über die Task zum Zeitpunkt ihrer Auswertung schon überholt und völlig veraltet sein — das ist eben das grosse Elend beim Multitasking! Es handelt sich immer nur um einen Schnappschuss des jeweils aktuellen Taskzustandes — der sich beim nächsten Timerinterrupt schon wieder geändert haben kann.

Name	Task wartet auf	Zustand
BLKBWA	Beendigung einer I/O	I/O?
BLKBAC	Aktivierung	DORM
BLKBSU	Fortsetzung	SUSP
BLKBCE	ein CE	CWS?
BLKBWS	auf Workspace	PWS?
BLKBSE	auf Semaphore	SEMA

Tabelle 15.2: Belegung des BLOCK–Bytes

Name	Eingeplant	Zustand
SCHBAC	zur Aktivierung	SCHD
SCHBTA	zur zeitlichen Aktivierung	TIAC
SCHBCA	zur zyklischen Aktivierung	CYAC
SCHBWA	auf Aktivierung bei Event	EVAC
SCHBTC	zur zeitlichen Fortsetzung	TICO
SCHBWC	auf Fortsetzung bei Event	EVCO

Tabelle 15.3: Belegung des SCHEDULE–Bytes

Um die Vorgehensweise von `rt_task_status()` zu verdeutlichen, können Sie dem nachfolgenden Quelltext entnehmen, was sich dabei intern abspielt und wo RTOS–UH die betreffenden Informationen ablegt.

```
ULONG rt_task_status( char  *taskname )
{
   ULONG  status   ;
   Task  *task_ptr ;

   if ( ( task_ptr = rt_search_task( taskname ) ) != NULL )
   {
      if ( task_ptr->fors )
         status = 0x00000000UL ; /* Irgendwas ist gerade los        */
      else
         status = 0x80000000UL ; /* Nicht in Aktivitaeten verwickelt */

      status |= ( ( (ULONG)task_ptr->block ) << 8 )
             |      (ULONG)task_ptr->sched        ;
   }
   else
      return( 0xFFFFFFFFUL ) ;  /* Unbekannte Task */

   return( status ) ;
}
```

Tabelle 15.4: Implementierung von `rt_task_status`

Wenn Sie sich sicher sind, dass eine zu überwachende Task, deren Zustand Sie interessiert, dauerhaft im Speicher bleibt, so kann nach einmaligem Aufruf der Funktion `rt_search_task()` die gelieferte `TID` gespeichert und direkt im Taskkopf der zu überwachenden Task herumgestochert werden. Dabei ist jedoch zu beachten, dass RTOS–UH sich darauf versteift, Tasks, deren Namen mit einem Doppelkreuz # beginnen, nicht finden zu wollen. Es handelt sich dabei um eine Art Abwehrhaltung des Betriebssystems, das verständlicherweise verhindern möchte, dass Nutzer an systemeigenen Tasks herumspielen können und deshalb so tut, als wären sie nicht in der Speicherverwaltung eingetragen.

```
Task  *rt_search_task( char  *name ) ;
```

Ein Task kann sich selbstverständlich auch in den eigenen Taskkopf schauen. Durch den Aufruf der Funktion `rt_my_TID()` kann eine Task ihren eigenen Taskkopf lokalisieren.

```
Task  *rt_my_TID( void ) ;
```

Alternativ zu diesem Systemaufruf kann eine Task sich selbst lokalisieren, indem sie direkt in der Systemzelle `$802.L` für die aktuell laufende Task nachschaut. Der Zugriff könnte z.B. `*(Task**)(0x802UL)` lauten und spart den Overhead des Funktionsaufrufes `rt_my_TID()`. Sie sollten allerdings **niemals** an der Speicherzelle `$802.L` herumspielen und immer nur lesend darauf zugreifen. Wenn Sie dort Unfug eintragen, knallt der Dispatcher des RTOS–UH beim nächsten Kontextwechsel mit der Nase an die Wand und RTOS–UH verlässt meist schweigend diese Welt.

Nachdem nun bekannt ist, wie man den Taskzustand ermitteln kann, sollen nunmehr die einzelnen Taskzustände etwas genauer betrachtet werden.

15.3.1.1 DORM — Schlafend

Der einfachste Betriebszustand wird `DORM` genannt, engl. dormand, dt. schlafend. Eine Task im Zustand `DORM` ist im Rechner geladen, aber nicht laufwillig. Sie wird nie Prozessorkapazität erhalten, es sei denn, ihr Betriebszustand ändert sich durch äussere Einwirkung. Eine schlafende Task beansprucht vom Betriebssystem lediglich Speicherplatz für ihren Code und ggf. (residente Tasks!) für statische Variablen. In allen anderen Betriebszuständen besitzt eine Task mit Sicherheit Speicherplatz für ihren `TWSP` (einzige Ausnahme siehe Übergang Aktivierung).

15.3.1.2 RUN — Laufwillig

Im Zustand `RUN` ist eine Task laufwillig. Sobald keine höher priorisierte Task laufwillig ist, wird diese Task den Prozessor erhalten und damit ihre Aufgabe bearbeiten können. Der Übergang vom Zustand `DORM` in den Zustand `RUN` wird Aktivierung dieser Task genannt. Sobald die Task den Prozessor zugeteilt erhält, beginnt sie ihre Abarbeitung beim Programmanfang. Die Aktivierung kann nur von aussen, d.h. von einer anderen Task oder vom Bediener über den Befehl „ACTIVATE Taskname“ erfolgen. Es ist klar, dass eine Task sich nicht von selbst aus dem Zustand `DORM` in den Zustand `RUN` bewegen kann, da sie ja im schlafenden Zustand keine Prozessorkapazität erhält und somit ihre eigene Aktivierung nicht veranlassen kann.

Allerdings kann eine einmal laufende Task sich selbst aktivieren. In diesem Fall wird die Aktivierung gepuffert, d.h. für später aufgehoben. Will oder soll diese Task später wieder in den Zustand `DORM` übergehen, so stellt RTOS–UH diese gepufferte Aktivierung fest und versetzt die Task wieder in den Zustand `RUN`. Die Task wird dann erneut von Beginn an ausgeführt. RTOS–UH kann max. 3 Aktivierungen für jede Task puffern.

15.3.1.3 SUSP — Ausgesetzt

Eine Task im Zustand `SUSP` (engl. suspended, dt. unterbrochen) ist zwar im Prinzip laufwillig und hat auch schon ein– oder mehrmals Prozessorkapazität erhalten, will aber zur Zeit keine Prozessorkapazität erhalten. Sie hat die Bearbeitung ihrer Aufgabe freiwillig oder von aussen gesteuert für eine undefinierte Zeitdauer ausgesetzt. Bei der Verteilung der Prozessorkapazität übergeht RTOS–UH eine suspendierte Task.

Der Zustand `SUSP` kann vom Bediener über den Befehl `SUSPEND Taskname` hergestellt werden. Die betroffene Task setzt ihre Ausführung an der Stelle, an der sie gerade ist, aus und kann später über den Befehl `CONTINUE Taskname` wieder fortgesetzt werden.

15.3.1.4 SCHD – Eingeplant

Eine Task ist engl. scheduled, dt. eingeplant. Der Zustand ist dem SUSP insofern ähnlich, als diese Task im Prinzip laufwillig ist, jedoch zur Zeit keine Prozessorkapazität erhalten will. Allerdings ist für diese Task schon eine Bedingung definiert, unter der sie in den Zustand RUN überführt wird. Diese Bedingung kann sowohl eine zeitliche Einplanung (z.B. vom Bediener `AT 17:00:00 ACTIVATE Taskname`) als auch eine Einplanung auf einen Event (z.B. vom Bediener `WHEN EV 00000001 ACTIVATE Taskname`) sein.

Der Zustand SCHD stellt den übergeordneten Begriff der Taskzustände TIAC, CYAC, EVAC, TICO und EVCO dar, die in Tabelle 15.3 aufgeführt sind.

Der Übergang in den Zustand der Einplanung ist aus allen anderen Zuständen mit Ausnahme des Zustandes SEMA möglich. Insbesondere ist allein durch den Zustand SCHD nicht gesagt, ob diese Task schon einmal Prozessorkapazität erhalten hat, oder ob sie erst durch die Einplanung aktiviert wird.

Wird die gegebene Einplanungsbedingung erfüllt, überführt RTOS–UH die entsprechende Task automatisch in den Zustand RUN und teilt ihr den Prozessor zu, falls keine höher priorisierte Task laufwillig ist.

15.3.1.5 I/O? — Blockiert durch I/O–Vorgang

Eine Task wartet auf den Abschluss einer Ein– oder Ausgabe. Unter RTOS–UH werden die tatsächlichen Ein– und Augabeoperationen von langsamen Geräten nicht direkt von der veranlassenden Task ausgeführt, sondern von sogenannten Betreuungstasks. Ist eine Task im Zustand I/O?, so hat sie eine Ein– oder Ausgabeoperation angestossen und wartet auf deren Abschluss. Für diese Zeit erhält sie keine Prozessorkapazität zugeteilt. Sobald die I/O–Operation abgeschlossen ist, überführt RTOS–UH diese Task wieder in den Zustand RUN.

15.3.1.6 PWS? — Blockiert durch Speicheranforderung

Eine Task wartet auf die Zuteilung von Speicher, i.a. Procedureworkspace, PWSP. Sie hat vom Betriebssystem Speicher in einer Grösse verlangt, die zur Zeit nicht zur Verfügung steht. Da RTOS–UH den Speicher dynamisch verwaltet, kann jedoch jederzeit wieder ausreichend freier Speicher entstehen. RTOS–UH befriedigt dann diese Speicheranforderung (bei mehreren wartenden Tasks in der Reihenfolge der Priorität) und überführt die Task automatisch wieder in den Zustand RUN.

15.3.1.7 CWS? — Blockiert durch CE–Anforderung

Eine Task wartet auf die Zuteilung von Speicherbereich für I/O–Operationen. Dies ist ein Sonderfall des Status PWS?, der nicht daher rührt, dass im System nicht mehr genügend freier Speicher vorhanden ist, sondern aus dem Überschreiten des Speicherkontingents für I/O–Operationen resultiert. RTOS–UH begrenzt den für jede einzelne Task bei jeder einzelnen Aktivierung für I/O–Operationen bereitgestellten Speicherplatz auf zur Zeit 3kB, um zu verhindern, dass eine einzelne Task durch zahlreiche Ausgaben den Systemspeicher übermässig verkleinert.

15.3.1.8 SEMA? — Blockiert durch SEMA–Anforderung

Eine Task wartet auf die Zuteilung einer Semaphore oder einer Bolt–Variable. Dieser Taskzustand wird immer dann erreicht, wenn zwei Tasks konkurrierend auf Resourcen zugreifen wollen und die kritischen

Pfade mit Semaphoren oder Bolts abgesichert haben. Stellt eine Task fest, dass eine (oder mehrere Tasks) den kritischen Pfad bereits betreten haben und einen weiteren Zugang verbieten, so bleibt die anfragende Task im Zustand SEMA liegen bis der kritische Bereich von der (oder den Tasks) wieder freigegeben wurde.

15.3.1.9 ???? – Mehrfachblockierung

Eine Task ist mehrfach blockiert, z.B. wenn sie im Zustand PWS? zusätzlich vom Bediener suspendiert wurde. Die Ausgabe von ???? als Taskzustand ist hierbei nur als Verzweiflungstat des Betriebssystems zu werten, das den Taskzustand nicht mehr in einem Wort berichten kann. Intern wird die Kombination unterschiedlicher Zustände korrekt behandelt.

Allerdings kann die Ausgabe von ???? als Taskzustand auch auf eine Notwehrreaktion des Betriebssystems hindeuten: hat eine Task eine unerlaubte Aktion durchgeführt, die das Betriebssystem nicht mehr abfangen, sondern nur beobachten konnte (z.B. Speicherzugriffsfehler bei privilegierten Speicherzugriffen), so versetzt es diese Task in den Zustand ????, um ein Weiterlaufen der Task zuverlässig zu verhindern.

15.3.2 Taskzustands–Übergänge

Nachdem die einzelnen Zustände, in denen sich Tasks befinden können, nun geklärt sind, werden hier die Möglichkeiten, Zustandsübergänge zu veranlassen und zu beeinflussen, erläutert. Dies betrifft nur die grundsätzlichen Taskzustände DORM, RUN, SUSP, SCHD und SEMA. Die Operationen, die Zustandsübergänge veranlassen können, sind *Aktivieren*, *Terminieren*, *Aussetzen*, *Fortfahren*, *Einplanen*, *Ausplanen* und Synchronisationsoperationen sowie das Eintreten von Zeitpunkten und Events.

Hier werden die Wirkung der Operation sowie die veranlassenden Bedienbefehle (und die analogen C–Anweisungen) grob beschrieben, detailliertere Erläuterungen finden sich im RTOS–UH–Handbuch im Kapitel Bedienbefehle, sowie für Assemblerprogrammierer im Kapitel über die Traps.

15.3.2.1 Aktivieren

Aktivieren hat nur auf Tasks im Zustand DORM direkte Wirkung: sie werden in den Zustand RUN versetzt, und die Programmausführung beginnt bei der Zuteilung des Prozessors am Programmanfang. Eine Aktivierung einer Task in einem anderen Zustand ist möglich und hat eine sogenannte gepufferte Aktivierung zur Folge. RTOS–UH merkt sich (im Taskkopf), dass diese Task aktiviert wurde, und startet sie von Programmanfang an neu, sobald sie wieder in den Zustand DORM geraten will.

Bei der *Aktivierung* teilt RTOS–UH zunächst der Task den Speicherplatz für ihre lokalen Variablen zu (TWSP), anschliessend wird die Task vom Anfang an gestartet. Steht bereits für diese Aktion nicht genügend Speicherplatz zur Verfügung, so wird die Task in den Zustand PWS? versetzt. RTOS–UH überprüft dann bei jeder Änderung der Speicherbelegung, ob die Anforderung bedient werden kann, und teilt den Prozessor erst zu, wenn der Task auch TWSP zur Verfügung steht. Die Aktivierung kann sowohl vom Bediener als auch von einer beliebigen Task veranlasst werden, Tasks können sowohl sich selbst als auch andere Tasks aktivieren.

Eine *Aktivierung* wird über den Bedienbefehl ACTIVATE Taskname, auch in der Kurzform A Taskname oder allein durch die Angabe des Tasknamens veranlasst. In CREST–C stehen die Funktionen rt_activate() und rt_activate_quick() zur Verfügung.

```
void rt_activate      ( char  *name, Prio prio ) ;
void rt_activate_quick( Task  *task, Prio prio ) ;
```

Die Task wird mit der Priorität `prio` aktiviert. Bei `prio=0` wird die Defaultpriorität der Task verwendet. Diese Defaultpriorität lässt sich beim Compilieren einer Task explizit angeben. Bei C–Shellmodulen wird diese Priorität standardmässig auf den Wert 20 vorbesetzt. Negative Prioritäten sind nicht zulässig und führen zu einer Fehlermeldung. Befindet sich die Task im Zustand `DORM` wird sie gemäss ihrer Priorität aktiviert. Steht die angesprochene Task im Zustand „waiting for activation“ in der Dispatcher–Kette, so wird sie entsprechend ihrer Priorität gestartet. Läuft die Task, wird die Aktivierung gepuffert. Bei einem Überlauf des Aktivierungspuffers geht die Aktivierung verloren, und es erfolgt eine Fehlermeldung.

15.3.2.2 Warten

RTOS–UH ermöglicht ab Nukleus 7.x das Warten auf die Terminierung einer Task. Die C–Funktion `rt_wait_for_exit()` realisiert hierzu den Hochsprachenanschluss. Dabei untersucht die aufrufende Task zunächst, ob die Task, auf die gewartet werden soll, tatsächlich in der Dispatcherkette des Betriebssystems eingetragen ist.

```
ReportCode  rt_wait_for_exit( Task  *task ) ;
```

Ist dies nicht der Fall, so kehrt die Funktion sofort zurück und liefert dem Aufrufer den Status der Operation als Langwort zurück. Die Antwort wird als Reportcode bezeichnet und ist zudem im Taskkopf der aufrufenden Task gespeichert. Ein Reportcode mit dem Wert `$FFFFFFFF` entspricht einem Fehlschlag der Funktion und bedeutet, dass RTOS–UH die spezifierte Task nicht in der Dispatcherkette lokalisieren und folglich keine weiteren Operationen einleiten konnte.

Wurde die angegebene Task hingegen gefunden, so wird die aufrufende Task mit Status `SEMA` blockiert und erst wieder lauffähig, wenn die andere Task sich selbst terminiert oder extern terminiert wird. Steht die angesprochene Task im Zustand „waiting for activation“ im Dispatcherring, so wird diese Blockierung aufgehoben. In jedem Falle liefert `rt_wait_for_exit()` entsprechende Rückgabewerte an die aufrufende Task. Eine Antwort von `$00000000` deutet auf ein ereignisloses Leben und einen erwartungsgemässen Tod der zu überwachenden Task hin.

Um die überwachende Task auch über Sonderkonditionen der überwachten Task zu informieren, sind vom System her weitere Bitmuster reserviert. Ein Reportcode von `$00000001` bedeutet, dass die überwachte Task *von der Seite abgeschossen wurde*, sich also nicht selbstständig terminiert hat.

```
void        rt_set_report_code( ReportCode  rep_cod ) ;
ReportCode  rt_get_report_code( void ) ;
```

Die Reportcodes `$FFFFFFFF`, `$0` bis einschliesslich `$10` sind für Systemmeldungen reserviert. Möchte eine überwachte Task dem Wächter weitere Informationen zukommen lassen, so lassen sich mittels der Funktionen `rt_set_report_code()` und `rt_get_report_code()` entsprechende Einträge vom Sohnprozess vornehmen und später von Vaterprozess auslesen. Dabei trägt der Sohnprozess die spätere Antwort in seinem eigenen Taskkopf ein. Terminiert sich der Sohn ordnungsgemäss selbst, so vererbt er diese Information als eine seine letzten Lebensbekundungen an seinen Vater. Dieser erhält die Information entweder über den Rückgabewert von `rt_wait_for_exit()` oder kann ihn mittels `rt_get_report_code()` später auslesen.

15.3.2.3 Terminieren

Terminierung ist die zur Aktivierung entgegengesetzte Operation. Die Ausführung einer Task wird an der Stelle, an der sie sich gerade befindet, abgebrochen. Die Task wird sofort in den Zustand `DORM` überführt, es sei denn, eine gepufferte Aktivierung liegt vor.

Bei der Terminierung wird der Task jeglicher zugeteilter Speicher entzogen (Ausnahme: TWSP residenter Tasks und I/O–Speicherbereiche) und wieder zu freiem Speicher erklärt. Hat eine Task schon I/O–Operationen angestossen, die noch nicht beendet sind, so wird wie folgt verfahren: alle Ausgaben bleiben weiterhin gültig; alle Eingaben, die ohne MODMWA abgeschickt wurden, sowie alle Eingaben bis auf eventuell gerade in Bearbeitung befindliche, werden verworfen. Das heisst: hat eine Task Eingaben veranlasst, so werden nur diejenigen Eingaben, die gerade von einer I/O–Betreuungstask bearbeitet werden, von der Betreuungstask zu Ende geführt; alle anderen Eingaben werden verworfen.

Eine Terminierung ist in jedem Taskzustand möglich. Auch eine Task, die schon DORM ist, kann terminiert werden; das Betriebssystem führt in diesem Fall allerdings keine Aktionen durch. Die Terminierung kann sowohl vom Bediener als auch von einer Task veranlasst werden, für Tasks ist sowohl Selbst–Terminierung als auch Fremdterminierung zulässig.

Die Terminierung wird durch den Bedienbefehl `TERMINATE Taskname`, auch in der Kurzform `T Taskname` veranlasst. Unter CREST–C stehen dazu die zwei Funktionen `rt_terminate_external()` und `rt_terminate_external_quick()` zur Terminierung fremder Tasks zur Verfügung. Bestehende Einplanungen für Aktivierungen werden dabei nicht berührt. Einplanungen auf Fortsetzung werden hingegen gelöscht.

```
void rt_terminate_external      ( char  *name ) ;
void rt_terminate_external_quick( Task  *task ) ;
```

Zur Selbst–Terminierung ist die Funktion `rt_terminate_internal()` zu verwenden.

```
void rt_terminate_internal( void ) ;
```

Eine weitere Form der Selbst–Terminierung besteht in der Verwendung der Funktion `rt_terminate_and_vanish()`, die nach erfolgreicher Terminierung der aufrufenden Task zusätzlich deren Taskkopf aus dem System entfernt. Dies stellt den üblichen Abschied von Tasks dar, die aus C–Shellmodulen generiert wurden und deren Taskköpfe nach Terminierung (ohne aufgelaufene Neuaktivierungen oder Einplanungen) überflüssig sind.

```
void rt_terminate_and_vanish( void ) ;
```

Die kompletteste Form der Task–Terminierung besteht im Bedienbefehl `UNLOAD Taskname`. Die angegebene Task wird ausgeplant, terminiert und der Taskkopf aus dem System entfernt. Es handelt sich um eine Kombination mehrerer RTOS–UH–Traps, um Tasks in beliebigem Zustand aus dem System zu entfernen. Die C–Funktionen `rt_unload_task()` und `rt_unload_task_quick()` bewirken dieser Verhalten für eine externe Task. Ein Selbst–Entladen der aufrufenden Task ist jedoch nicht möglich!

```
void rt_unload_task      ( char  *name ) ;
void rt_unload_task_quick( Task  *task ) ;
```

15.3.2.4 Aussetzen

Durch *Aussetzen* geht eine Task vom Zustand RUN in den Zustand SUSP über. Die Ausführung der Task wird für eine beliebige Zeit ausgesetzt, die Task sozusagen in ihrem gegenwärtigen Zustand eingefroren. Eine ausgesetzte Task wird bei der Zuteilung von Prozessorkapazität übergangen. Alle lokalen Variablen bleiben erhalten, jeder zugeteilte Speicher ebenfalls. Von der Task angestossene I/O–Operationen werden ausgeführt.

Aussetzen kann sowohl vom Bediener als auch von Tasks veranlasst werden; Tasks können sowohl sich selbst als auch fremde Tasks in der Ausführung aussetzen. Für eine einzelne Task ist auch das Aussetzen, verbunden mit der Definition einer Fortsetzbedingung, möglich, siehe Einplanen mit Aussetzen.

Aussetzen wird veranlasst durch den Bedienbefehl `SUSPEND Taskname`, Kurzform `SU Taskname`. Die CREST–C–Funktion `rt_suspend()` bewirkt eine Selbstsuspendierung der aufrufenden Task.

```
void rt_suspend( void ) ;
```

Die folgenden Funktion dienen zur Suspendierung fremder Tasks über deren Namen bzw. über deren `TID`.

```
void  rt_suspend_external      ( char  *name ) ;
void  rt_suspend_external_quick( Task  *task ) ;
```

15.3.2.5 Fortsetzen

Fortsetzen ist die zum Aussetzen entgegengesetzte Operation. Eine Task wird aus dem Zustand `SUSP` in den Zustand `RUN` gebracht, d.h. sie wird ab sofort wieder bei der Zuteilung der Prozessorkapazität berücksichtigt. Die Task nimmt ihre Tätigkeit an genau der Stelle, an der sie ausgesetzt wurde, wieder auf. Für die Task selbst ist bis auf eine Änderung der Uhrzeit kein Unterschied zu ununterbrochener Ausführung merkbar.

Die *Fortsetzung* kann sowohl vom Bediener als auch von einer anderen Task veranlasst werden. Klar dürfte sein, dass eine ausgesetzte Task keine Möglichkeit hat, sich selbst wieder fortzusetzen, es sei denn, sie beginnt das Aussetzen mit einer Einplanung zur Fortsetzung (siehe Einplanen).

Die Fortsetzung kann mittels des Bedienbefehls `CONTINUE Taskname`, oder kurz `C Taskname` veranlasst werden. Auf C–Ebene stehen die Funktionen `rt_continue()` und `rt_continue_quick()` zur Verfügung. Wenn die Task suspendiert ist, wird sie fortgesetzt. Anderfalls erhält man eine Fehlermeldung.

```
void rt_continue      ( char *name ) ;
void rt_continue_quick( Task *task ) ;
```

15.3.2.6 Einplanen

Einplanungen sind das Herz von RTOS–UH. Die Fähigkeit, Aktivierung und Fortsetzung von Tasks von dem Eintreten unterschiedlichster Ereignisse abhängig zu machen, ist ein wesentliches Merkmal des Betriebssystems und ermöglichen die Erstellung von Programmpaketen, die zum einen sehr schnell und zum anderen sehr flexibel auf äussere Situationen reagieren können.

Einplanungen lassen sich zum einen nach der Art des Ereignisses, Zeit oder Events, und zum anderen nach der Art der eingeplanten Operation, Aktivierung oder Fortsetzung, unterscheiden.

Allen Einplanungen gemeinsam ist die Änderung des Taskzustands von `DORM`, `RUN` oder `SUSP` in den Zustand `SCHD`. Die Einplanungen können vom Bediener oder von einer Task vorgenommen werden, auch ein selbstständiges Einplanen einer Task ist möglich (natürlich nicht, wenn sich diese Task im Zustand `DORM` befindet).

- zeitlich eingeplante Aktivierung

  Eine zeitlich eingeplante Aktivierung führt zum Start einer Task zu einem angegebenen Zeitpunkt oder nach einer angegebenen Zeitdauer. Weiterhin kann festgelegt werden, dass diese Aktivierung in bestimmten Zeitabständen zu wiederholen ist, und diese zyklische Einplanung kann auf eine bestimmte Zeitdauer eingegrenzt werden. RTOS–UH verwaltet Uhrzeit und Zeitdauer mit der Auflösung von 1 ms, die längste verarbeitbare Zeitdauer sind ca. 24 Tage.

Die Einplanung auf einen Zeitpunkt erfolgt mit dem Bedienbefehl `AT Uhrzeit ACTIVATE Taskname`, die Einplanung über eine Zeitdauer mit dem Befehl `AFTER Zeitdauer ACTIVATE Taskname`. Unter CREST–C stehen die zwei Funktionen `rt_timed_activate()` und `rt_timed_activate_quick()` zur Verfügung.

```
void rt_timed_activate      ( char        *name     ,
                              Prio         prio     ,
                              TimeSchedule startzeit,
                              TimeSchedule intervall,
                              TimeSchedule endzeit  ) ;

void rt_timed_activate_quick( Task       *task     ,
                              Prio         prio     ,
                              TimeSchedule startzeit,
                              TimeSchedule intervall,
                              TimeSchedule endzeit  ) ;
```

Zusätzlich zu den Parametern bei `rt_activate()` werden noch eine Startzeit, Intervallzeit und Endzeit angegeben. Die Task kann zu oder nach einer bestimmten Zeit (oberstes Bit der Zeitangabe) gestartet werden. Eine sofortige Aktivierung — soll heissen beim nächsten Clock–Tick — erreicht man durch Angabe von `0x80000000UL`. Soll die Einplanung zyklisch wiederholt werden, ist eine Intervallzeit einzutragen, die sonst 0 sein sollte (negative sind auch zulässig, aber nicht zu empfehlen). Um die Einplanung zu einem bestimmten Zeitpunkt oder nach einer bestimmten Zeitdauer (oberstes Bit der Zeitangabe) zu beenden, ist die Endzeit anzugeben. Wird die Endzeit mit `0x7FxxxxxxUL` angegeben — nur das erste Byte ist hier relevant —, so laufen die Einplanungen endlos. Bestehende Einplanungen werden gelöscht.

Um die Angabe von üblichen Befehlen zu vereinfachen, wurden einfachere Formen zyklischer Aktivierungen in der Bibliothek aufgenommen, die automatisch absolute Zeitpunkte voraussetzen (`at`–Formen).

```
void rt_at_activate      ( char        *taskname  ,
                           Prio         prio     ,
                           TimeSchedule startzeit ) ;

void rt_at_activate_quick( Task        *task     ,
                           Prio         prio     ,
                           TimeSchedule startzeit ) ;
```

Ebenso stehen Funktion bereit, die mit relativen Zeitangaben umgehen (`after`–Formen).

```
void rt_after_activate      ( char        *taskname   ,
                              Prio         prio      ,
                              TimeSchedule startdauer ) ;

void rt_after_activate_quick( Task        *task      ,
                              Prio         prio      ,
                              TimeSchedule startdauer ) ;
```

- zyklische Einplanung

  Eine zyklische Einplanung erfolgt mit dem Bedienbefehl `ALL Zeitdauer ACTIVATE Taskname` und kann in der Form `ALL Zeitdauer DURING Zeitdauer ACTIVATE Taskname` auf eine bestimmte Zeitspanne eingegrenzt werden. Statt der Eingrenzung auf eine bestimmte Zeitdauer kann auch eine Uhrzeit für die letzte Aktivierung in der

Form `ALL Zeitdauer UNTIL Uhrzeit ACTIVATE Taskname` gegeben werden. Die entsprechenden CREST–C–Anschlüsse sind Sonderfälle der bereits erläuterten Funktionen `rt_timed_activate()` und `rt_timed_activate_quick()`.

Zyklische Einplanungen in der einfachen Form haben eine sofortige Aktivierung zur Folge, d.h. die erste der zyklischen Aktivierungen findet sofort statt. Ist dies nicht erwünscht, so kann die zyklische Einplanung mit der Einplanung auf einen Zeitpunkt oder über eine Zeitdauer kombiniert werden, z.B. `AFTER Zeitdauer ALL Zeitdauer DURING Zeitdauer ACTIVATE Taskname` oder `AT Uhrzeit ALL Zeitdauer UNTIL Uhrzeit ACTIVATE Taskname`.

- auf Event eingeplante Aktivierung

  Sehr wichtig für ein Echtzeit–Betriebssystem ist die Fähigkeit, schnell auf externe Ereignisse reagieren zu können. Externe Ereignisse sind Ereignisse, die asynchron zum Programmablauf auftreten und dem Prozessor von der Peripherie signalisiert werden, sogenannte Unterbrechungen (im regulären Programmablauf) oder Interrupts. Die entsprechenden Interrupt–Handler haben die Aufgabe, die Hardware–Interrupts in entsprechende Software–Events (oder auch als Prozessinterrupt bezeichnet) umzusetzen.

  RTOS–UH kann die Aktivierung von Tasks vom Auftreten dieser Events abhängig machen, d.h. tritt ein (Hardware–)Interrupt auf, so wird der entsprechende Interrupt–Handler sich um die Hardware kümmern und anschliessend das zugehörige (Software–)Event auslösen. Daraufhin wird die hierauf eingeplante Task aktiviert. Hat diese Task eine hohe Priorität, so wird ihr sofort der Prozessor zugeteilt, und Tasks niedriger Priorität können die Behandlung dieser Unterbrechungssituation nicht behindern.

  Die Einplanung auf einen Event erfolgt mit dem Befehl `WHEN Eventkennzeichnung ACTIVATE Taskname`. In CREST–C werden dazu die zwei Funktionen `rt_event_activate()` und `rt_event_activate_quick()` verwendet.

```
void rt_event_activate      ( char  *name, Prio prio, Event mask ) ;
void rt_event_activate_quick( Task  *task, Prio prio, Event mask ) ;
```

  Zusätzlich zu den Parametern bei `rt_activate()` wird noch eine Event–Maske übergeben, in der die Events auf „1“ gesetzt sind, auf die die Task eingeplant werden soll. Bestehende Einplanungen werden gelöscht.

- zeitlich eingeplante Fortsetzung

  Zeitlich eingeplante Fortsetzungen entsprechen den zeitlich eingeplanten Aktivierungen, wobei weder eine zyklische Einplanung noch die Begrenzung auf einen bestimmten Zeitraum oder bis zu einem bestimmten Zeitpunkt möglich sind. Eine zeitlich eingeplante Fortsetzung wird mit den Bedienbefehlen `AT Uhrzeit CONTINUE Taskname` oder `AFTER Zeitdauer CONTINUE Taskname` definiert. Die C–Funktionen `rt_timed_continue` und `rt_timed_continue_quick` erlauben analog zu den `rt_timed_activate`–Funktionen eine Fortsetzung einer Task zu oder nach einer bestimmten Zeit (oberstes Bit der Zeitangabe).

```
void rt_timed_continue      ( char  *name, TimeSchedule  zeit ) ;
void rt_timed_continue_quick( Task  *task, TimeSchedule  zeit ) ;
```

  Die folgenden Funktionsaufrufe setzen die Bits in der Zeitangabe automatisch und unterscheiden sich ansonsten nicht von den vorgenannten Basisfunktionen.

```
void rt_at_continue         ( char  *name, TimeSchedule  zeit ) ;
void rt_at_continue_quick   ( Task  *task, TimeSchedule  zeit ) ;
void rt_after_continue      ( char  *name, TimeSchedule  iv   ) ;
```

```
void rt_after_continue_quick( Task  *task, TimeSchedule  iv   ) ;
```

Will eine Task selbst ihre Ausführung für eine bestimmte Zeitdauer aussetzen, so kann dies mittels der C–Funktion `rt_timed_resume()` erreicht werden. Auch hier gilt die Nomenklatur für die Angabe von absoluten und relativen Zeiten.

```
void rt_timed_resume( TimeSchedule  zeit ) ;
```

Im Sinne der besseren Bedienbarkeit steht auch hier eine `at`– und eine `after`–Form als Funktionsaufruf in der Bibliothek zur Verfügung.

```
void rt_resume_at   ( TimeSchedule  zeit ) ;
void rt_resume_after( TimeSchedule  zeit ) ;
```

Die Task überführt sich hierbei selbst in den Zustand `SCHD` und verzichtet bis zum Erreichen des gewünschten Zeitpunktes oder nach dem Verstreichen der gewünschten Zeitdauer auf die Zuteilung von Prozessorkapazität.

- auf Event eingeplante Fortsetzung

  Die auf einen Interrupt eingeplante Fortsetzung ist analog der auf einen Interrupt eingeplanten Aktivierung zu verstehen. Für eine Task wird definiert, dass sie beim Eintreten eines Events fortgesetzt werden soll. Voraussetzung ist natürlich, dass diese Task bis zum Eintreten des Interrupts ausgesetzt ist, andernfalls erfolgt beim Eintreten des Events die Fehlermeldung `Taskname NOT SUSPENDED`, da die Fortsetzung einer nicht ausgesetzten Task unmöglich ist. Die Einplanung wird über den Bedienbefehl `WHEN Eventkennzeichnung CONTINUE Taskname` definiert. Unter CREST–C stehen die Funktionen `rt_event_continue()` und `rt_event_continue_quick()` zur Verfügung.

  ```
  void rt_event_continue      ( char *name, Event mask ) ;
  void rt_event_continue_quick( Task *task, Event mask ) ;
  ```

  Die Task wird zur Fortsetzung auf die in der Eventmaske gesetzten Bits eingeplant. Bestehende Einplanungen werden gelöscht.

  Will eine Task sich selbst gleichzeitig aussetzen und zur Fortsetzung bei einem Event einplanen, so kann dazu die C–Funktion `rt_event_resume()` verwendet werden, die eine Kombination der Funktionen `rt_event_continue_quick()` und `rt_suspend()` darstellt.

  ```
  void  rt_event_resume( Event  mask ) ;
  ```

15.3.2.7 Ausplanen

Ausplanen ist die zum Einplanen entgegengesetzte Operation. Ausplanungen können für eine Task nur pauschal vorgenommen werden, d.h. eine Differenzierung nach der Art der Einplanung ist nicht möglich. Durch eine Ausplanung werden alle für eine Task vorgenommenen Einplanungen gelöscht, der Zustand der Task nach der Ausplanung ergibt sich aus dem Taskzustand bei der Einplanung.

Die häufigsten Übergänge sind `SCHD` ⇒ `DORM`, falls eine Task nur zur Aktivierung eingeplant war. Eine Task, die auf eine Fortsetzung eingeplant war und sich folglich nicht im Status `DORM` befand, bleibt nach einer Ausplanung weiterhin im Zustand `SUSP`. Eine Task kann auch ihre eigenen Einplanungen löschen.

Die Ausplanung wird durch den Bedienbefehl `PREVENT Taskname`. Unter CREST–C stehen die Funktionen `rt_prevent_task()` und `rt_prevent_task_quick()` zur Verfügung. Alle bestehenden Einplanungen der Task werden gelöscht. Der aktuelle Laufzustand ändert sich dadurch nicht.

```
void rt_prevent_task      ( char  *name ) ;
void rt_prevent_task_quick( Task  *task ) ;
```

15.3.3 Synchronisationsoperationen

In einem Multi–Tasking–Betriebssystem tritt häufig der Fall ein, dass mehrere Tasks auf den gleichen Datenbestand zugreifen müssen. Ändert eine Task diesen Datenbestand, so muss gewährleistet sein, dass die anderen Tasks, die den Datenbestand nur auslesen, stets konsistente Daten erhalten. Diese Forderung kann nur allein über die Prioritätenwahl nicht erfüllt werden:

- Hat die ändernde Task die höchste Priorität, so kann sie zwar stets alle Änderungen ungestört vornehmen, unterbricht aber möglicherweise eine auslesende Task mitten im Lesevorgang, was bei der auslesenden Task zu inkonsistenten Daten führt.
- Hat die ändernde Task die niedrigste Priorität, so kann sie mitten während einer Datenänderung unterbrochen werden, wodurch eine auslesende Task wiederum inkonsistente Daten erhält.

Zur Lösung dieses Problems stellt RTOS–UH Synchronisationsvariable, sog. Semaphore, ital. für Ampeln, und Bolts, engl. für Riegel, zur Verfügung.

15.3.3.1 Semaphore

Semaphore lassen sich in ihrer Funktion gut durch die Analogie zu Ampeln erklären. Als Beispiel sei eine Engstelle in der Fahrbahn gegeben, die durch beidseitige Ampeln gegen das Einfahren von Fahrzeugen sperrbar ist. An beiden Einfahrstellen sind jeweils in Fahrtrichtung Induktionsschleifen installiert, über die ein Einfahrwunsch gemeldet und das Verlassen der Engstelle erkannt werden kann.

Der einfachste, zu betrachtende Fall besteht aus zwei Fahrzeugen, die sich aus entgegengesetzten Richtungen der Engstelle nähern. Das Fahrzeug, das als erstes seine Einfahrt–Induktionsschleife erreicht, erhält Grün für die Weiterfahrt und setzt automatisch Rot für die Gegenseite, so dass der Einfahrtwunsch der Gegenseite abgelehnt wird. Erst mit dem Überfahren seiner Austritts–Induktionsschleife gibt das Fahrzeug die Engstelle frei und erwirkt Grün für das wartende Fahrzeug.

Die Übertragung auf Semaphore unter RTOS–UH erfordert nur eine neue Terminologie: das Erfragen der Einfahrterlaubnis durch Überfahren der Einfahrt–Induktionsschleife wird `REQUEST`–Operation genannt, das Verlassen der Engstelle `RELEASE`–Operation. Die einzelnen Tasks sind wie die Fahrzeuge zu betrachten: Task A führt ein `REQUEST` auf die Semaphore *Engstelle* durch und belegt die *Engstelle* hiermit. Diese Operation hat keine Auswirkungen auf den Zustand der Task A. Fährt nun Task B, die z.B. durch ein Unterbrechungssignal oder einen Bedienereingriff lauffähig geworden ist und wegen ihrer höheren Priorität den Prozessor zugeteilt erhalten hat, ebenfalls eine `REQUEST`–Operation durch, so wird sie suspendiert, d.h. ihre Ausführung wird trotz der höheren Priorität ausgesetzt und ihr Zustand auf `SEMA`, d.h. wartend auf das Freiwerden einer Semaphore, gesetzt. Es wird eine Neuzuteilung des Prozessors notwendig, und Task A hat gute Chancen, nun die höchstpriorisierte, laufwillige Task zu sein und den Prozessor zugeteilt zu erhalten.

Verlässt Task A die Engstelle, so führt sie eine `RELEASE`–Operation auf die Semaphore *Engstelle* durch. Hiermit wird die Engstelle wieder freigegeben. Auf den Computer übertragen, bedeutet dies die Notwendigkeit einer erneuten Prozessorzuteilung, und hierbei wird Task B auf Grund ihrer höheren Priorität den Prozessor zugeteilt erhalten.

Umgesetzt von Fahrbahn–Engstellen auf Datenbereiche ist also durch Einsatz der Synchronisations–Operationen `REQUEST` und `RELEASE` die Integrität des Datenbestandes gewahrt geblieben.

Von Bedienerebene her sind Semaphore nur über ihre Adresse mit den Anweisungen `REQUEST Se-`

maphoradresse und RELEASE Semaphoradress ansprechbar; zur Vereinfachung ist auch eine Anweisung RELEASE Taskname möglich, die die Semaphore, auf die eine Task im Zustand SEMA wartet, freigibt. Man beachte, dass die letztgenannte Form der Anweisung zum einen nicht spezifisch eine bestimmte Semaphore anspricht, da die Adresse der betroffenen Semaphor–Variablen nicht angegeben wird, zum andern aber stets genau eine und nur eine Semaphore betrifft, da eine Task im Zustand SEMA stets auf nur eine Semaphore warten kann.

In Erweiterung zu dem bisher Geschilderten sind Semaphore unter RTOS–UH mehrwertig implementiert, d.h. je nach Vorbesetzung der Semaphore können mehrere REQUEST–Operationen ohne Blockierung durchgeführt werden. Damit ist dann auch die Synchronisation zweier als Erzeuger und Verbraucher tätiger Tasks möglich: Führt der Verbraucher vor Verwendung der erzeugten Daten (die z.B. in einem Ringpuffer abgelegt werden können) eine REQUEST–Operation durch, und führt der Erzeuger stets nach Bereitstellung eines Datensatzes eine RELEASE–Operation durch, so ist die Synchronisation beider gewährleistet.

Von CREST–C aus stehen drei Funktionen zur Verfügung, um Semaphoren zu bearbeiten. In C existiert keine Basisdatentyp SEMA. Stattdessen liegt eine Typenvereinbarung Sema in der Includedatei <rtos.h> vor. Bei Semaphoren handelt es sich aus Sicht des C–Compilers um normale Objekte vom Typ signed short. Damit lassen sich Initialisierungen mit Startwerten wie normale Variablenzuweisungen handhaben. Die Anforderung und Freigabe einer Semaphore **muss** jedoch durch Aufruf von Systemtraps erfolgen. Die REQUEST–Operation wird mittels der Funktion rt_request_sema() durchgeführt. Die angegebene Semaphore wird um Eins erniedrigt. Ist der neue Wert grösser oder gleich Null, läuft die aufrufende Task normal weiter. Ein negativer Wert wird auf -1 korrigiert und die Task wird blockiert (SEMA).

```
void rt_request_sema( Sema  *sema ) ;
```

Die RELEASE–Operation wird durch die Funktion rt_release_sema() ausgelöst. Die Semaphore wird um Eins erhöht. Die aufrufende bleibt dabei grundsätzlich lauffähig. Stellt RTOS–UH bei der Freigabe der Semaphore fest, dass eine (oder mehrere Tasks) auf diese Sema wartet, so wird die höchstpriorisierte Task, die auf diese Semaphore gewartet hat, lauffähig gemacht.

```
void rt_release_sema( Sema  *sema ) ;
```

Eine weitere Funktionalität der C–Bibliothek besteht darin, eine Semaphore *nichtblockierend* anzufordern. Die Funktion rt_try_sema() funktioniert ähnlich wie die Funktion rt_request_sema(), blockiert die aufrufende Task jedoch nicht, wenn die Semaphore zum Zeitpunkt der Anforderung bereits vergeben war.

```
Sema rt_try_sema( Sema  *sema ) ;
```

Im Erfolgsfall wird die Semaphore belegt und die Funktion liefert den Wert der Semaphore **vor** der Anforderung zurück — also immer einen Wert grösser Null. Wenn die Funktion den Wert Null liefert, so haben war die Semaphore blockiert und konnte **nicht** requestet werden.

In Hinsicht auf Multiprozessorsysteme ist zudem der MC680xxx–Maschinenbefehl TAS von Bedeutung, der es erlaubt, mittels eines unteilbaren Lese–Schreibzyklus ein Byte im Speicher zu modifizieren. Der TAS–Befehl wurde als Funktionsanschluss bereitgestellt.

```
int  rt_Test_And_Set( void  *ptr ) ;
```

Die Funktion rt_Test_And_Set() liefert 0, wenn der TAS–Befehl gescheitert ist und einen Rückgabewert ungleich Null, wenn das getestete Byte Null enthielt.

15.3.3.2 Bolts

Bolts arbeiten gegenüber Semaphoren wesentlich differenzierter. Sie können zwischen nur lesenden Tasks und Tasks, die ggf. auch Daten modifizieren, unterscheiden. Lesende Prozesse können durch die Operation `ENTER Boltvariable` den Beginn und mit der Operation `LEAVE Boltvariable` das Ende ihres Zugriffs auf den kritischen Datenbereich signalisieren. Schreibende Prozesse, d.h. Tasks, die Daten modifizieren, benutzen stattdessen die Operation RESERVE beim Eintritt und FREE beim Verlassen des kritischen Pfades. Analog stehen dazu die folgenden C–Funktionen zur Verfügung.

```
void  rt_reserve_bolt( Bolt  *bolt ) ;
void  rt_free_bolt   ( Bolt  *bolt ) ;
void  rt_enter_bolt  ( Bolt  *bolt ) ;
void  rt_leave_bolt  ( Bolt  *bolt ) ;
```

Ziel dieser unterschiedlichen Operationen ist es, lesende Prozesse nicht zu behindern, wenn kein Schreiber die Daten benutzen will. Solange kein Schreiber eine RESERVE–Operation durchgeführt hat, dienen die ENTER– und LEAVE–Operationen nur dazu, die Benutzung der Daten zu markieren. Erst bei Abarbeitung einer RESERVE–Operation greift der mit den Bolts verknüpfte Synchronisationsmechanismus. Ist zu dieser Zeit ein lesender Prozess im kritischen Pfad, so wird die Ausführung des das RESERVE ausführende Schreibers ausgesetzt, bis der oder die Leser den kritischen Pfad verlassen haben. Gleichzeitig wird der Eintritt in den kritischen Pfad für Leser und weitere Schreiber gesperrt. Sind keine Leser oder Schreiber mehr im kritischen Pfad, d.h. haben alle Leser ihre LEAVE– bzw. ein Schreiber seine FREE–Operation durchgeführt, so wird der höchstpriore Prozess mit einer RESERVE–Operation in den Zustand lauffähig versetzt und erhält ggf. den Prozessor zugeteilt. Nach Abarbeitung der dem RESERVE zugeordneten FREE–Operation wird der kritische Pfad wieder freigegeben.

15.3.3.3 Interne Blockierbedingung

Unter CREST–C wurden zusätzlich zwei Funktionen implementiert, um die Tasksynchronisation etwas flexibler gestalten zu können.

```
void  rt_wait_for_activation ( void ) ;
void  rt_unblock_waiting_task( Task  *tid ) ;
```

Eine Task, die `rt_wait_for_activation()` aufruft, verhält sich *beinahe* so, als sei sie auf eine belegte Semaphore aufgelaufen oder habe sich selbst suspendiert. Im Unterschied zu den offiziellen Taskzuständen SUSP und SEMA lauert die Task jedoch nicht auf Fortsetzung oder die Zuteilung einer Semaphore, sondern lediglich die die Aufhebung einer internen Blockierbedingung, die z.B. durch eine Aktivierung erfolgen kann. Nach aussen hin tritt die Task als eingeplant in Erscheinung (Zustand ist SCHD). Nur: tritt eine Aktivierung auf, so setzt die Task die Arbeit hinter der Funktion `rt_wait_for_activation()` fort und **nicht** mit einem Neustart der Task, legt also eher das Verhalten einer Sema–blockierten oder selbstsuspendierten Task an den Tag.

Die Funktion `rt_wait_for_activation()` kontrolliert dabei intern, ob weitere Aktivierungen für die aufrufende Task vorliegen. Sind keine Aktivierungen gepuffert, so legt sich die Task in den Zustand SCHED, bleibt also eingeplant. Eine andere Task kann die schlummernde Task nun mittels `rt_unblock_waiting_task()` aufwecken.

Das folgende Beispiel demonstriert die Möglichkeiten, die diese spezielle Form der Selbstblockierung bietet:

```
Verarbeitungstask(..)
{
   for (;;)
```

```
    {
       while ( HabDatenBekommen() ) VerarbeiteDaten() ;

       rt_timed_activate_quick( rt_my_TID(), (Prio)0,
                                0x80000000L | 5, // Start nach 5 ms
                                0L,              // kein Intervall
                                0L  ) ;          // keine Endezeit
       rt_wait_for_activation() ;
    }
}

EineDerAnlieferungsTasks(..)
{
   GibIhmDaten() ;
   rt_unblock_waiting_task( tid_der_Verarbeitungstask ) ;
}
```

Die Verarbeitungstask plant sich nach Ablauf von fünf Millisekunden für eine Eigenaktivierung ein und legt sich mit der neuen Funktion schlafen bis:

1. eine Aktivierung von draussen erfolgt wie z.B. von der Task `EineDerAnlieferungs-Tasks()`...
2. fünf Millisekunden abgelaufen sind und die Eigenaktivierung greift.

Diese Methode ist deutlich flexibler in der Anwendung, als Timeout–Überwachungen innerhalb der Task mittels `rt_resume_after()` zu kodieren oder externe Watchdog–Tasks mit dieser Aufgabe zu betreuen.

Grundsätzlich sollte die Funktion `rt_unblock_waiting_task()` zur Aktivierung einer mittels `rt_wait_for_activation()` eingeschläferten Task Verwendung finden. Zwar funktionieren die normalen Funktionen zur Aktivierung von Tasks ebenfalls ordnungsgemäss, aber `rt_unblock_waiting_task()` besitzt den grossen Vorteil, dass lediglich die Blockierbedingung der betreffenden Task aufgehoben wird und diese damit wieder laufwillig wird. Der interne Zähler für aufgelaufene Aktivierungen wird dabei stets auf den Wert für **genau eine** Aktivierung gesetzt; auf diese Weise werden die unerwünschten `OVERFLOW (ACT)`–Meldungen vermieden.

Nachdem die Vorteile nun behandelt wurden, sollen die Besonderheiten, die Sie sich bei der Verwendung dieser Funktionen berücksichtigen müssen, nicht verschwiegen werden.

- Eine Task, die mittels `rt_unblock_waiting_task()` fortgesetzt werden soll, **muss** bereits in der Dispatcher–Kette stehen. Eine Task, die nicht bereits auf eine derartige Fortsetzung durch Aufhebung der Blockierbedingung wartet, wird auch nicht implizit aktiviert.
- Normalerweise erfolgt die Aktivierung einer Task mittels `rt_activate_...()` unter Angabe einer Priorität. Wenn eine Task mittels `rt_wait_for_activation()` schlafengelegt wurde, dann reagiert sie zwar auch auf Aktivierungen, verwendet allerdings immer die gerade aktuelle Laufzeitpriorität der Task. `rt_unblock_waiting_task()` verzichtet deshalb sogar ganz auf diesen Parameter.

15.3.4 Ereigniseintritt

Ein Übergang aus dem Zustand `SCHD` in den Zustand `RUN` ist nur durch den Eintritt des bei der Einplanung festgelegten Ereignisses möglich, sei es durch das Erreichen einer Uhrzeit, durch das Verstreichen einer Zeitdauer oder durch das Auftreten eines Events.

Der Bediener kann — und darf — hierauf nur begrenzt Einfluss nehmen. Eine Beeinflussung der Uhrzeit ist mit dem Befehl `CLOCKSET` durchaus möglich. Sie sollten dabei stets im Hinterkopf behalten, dass ein Umstellen der Uhrzeit bei laufenden Anwenderprogrammen unter RTOS–UH leider oft im absoluten Chaos endet. Das Betriebssystem verwaltet Einplanungszeitpunkte intern — auch wenn diese relativ angegeben wurden — stets mittels absoluter Uhrzeiten! Wenn Sie dem Betriebssystem mitteilen, dass eine Operation *nach 5 Sekunden* auszuführen ist, so rechnet sich RTOS–UH intern den Zeitpunkt aus, der der Angabe nach 5 Sekunden entspricht und speichert sich nur diesen Zeitpunkt! Ändern Sie nun die Systemuhrzeit, so gerät so ziemlich alles in Trudeln...

Die Funktion `rt_read_clock()` liefert die aktuelle Systemuhrzeit in Millisekunden zurück.

```
Time  rt_read_clock( void ) ;
```

Wenn das System eine batteriegepufferte Uhr besitzt, so kann aus einem C–Programm heraus mittels der folgenden beiden Funktionen darauf zugegriffen werden.

```
void  rt_write_battery_clock( ULONG  date, Time  time ) ;
void  rt_read_battery_clock( ULONG  *date, Time *time ) ;
```

Die Funktion `rt_read_battery_clock()` aktualisiert die Systemuhrzeit entsprechend den Vorgaben der Hardwareuhr. Umgekehrt lässt sich die Hardwareuhr mittels `rt_write_battery_clock()` neu stellen — dies hat jedoch keinerlei Auswirkungen auf die Systemuhrzeit. Bei `date == 0` wird in `time` die Zahl der Millisekunden seit Mitternacht erwartet und die Uhr neu gesetzt. Das Datum verändert sich durch diesen Aufruf nicht. Bei `date != 0` wird ist der Parameter `time` redundant. Es wird nur das Datum gesetzt. Das Format von `date` hat wie folgt auszusehen:

```
rt_write_battery_clock( 0xddmmyyyy, 0 ) ;

dd   - Tag
mm   - Monat
yyyy - Jahr
```

Ein Interrupt kann über den Bedienbefehl `TRIGGER Eventkennzeichnung` simuliert werden. Dazu ist es zuvor notwendig, den betreffenden Prozessinterrupt — auch als Event bezeichnet — für das System freizugeben. Freigabe und Sperren von Prozessinterrupts erfolgen über die Bedienbefehle `ENABLE Interruptkennzeichnung` bzw. `DISABLE Interruptkennzeichnung`. Die CREST–C–Funktionen `rt_enable_event()` und `rt_disable_event()` erlauben das Setzen und Zurücksetzen der Event–Maske des Betriebssystems. Beachten Sie bei der Verwendung dieser Befehle stets daran, dass es **nur exakt eine** Event–Maske für den gesamten Rechner gibt und unsinnige Zugriffe sich deshalb stets furchtbar global auf alle Tasks im System auswirken. Es werden alle Prozessinterrupts, die in der Maske auf „1“ gesetzt sind, freigegeben.

```
void rt_enable_event( Event  mask ) ;
```

Entsprechend können mittels `rt_disable_event()` alle Prozessinterrupts, die in der Maske auf „1“ gesetzt sind, gesperrt werden. Auch die Sperrung gilt global im gesamten System.

```
void rt_disable_event( Event  mask ) ;
```

Das eigentliche „Feuern“ des Prozessinterrupts erfolgt mittels der C–Funktion `rt_trigger_event()` . Mittels dieser Funktion lässt sich bei Tests fehlende Hardware simulieren.

```
void rt_trigger_event( Event  mask ) ;
```

15.4 Interrupt–Routinen

Nach dem Konzept der Tasks ist das zweitwichtige Element für die Effizienz von RTOS–UH der planvolle Einsatz von Interrupt–Routinen und Software–Events.

Interrupts, Unterbrechungen, sind Ereignisse, die von der Peripherie bzw. den Treiberbausteinen für Peripheriegeräte unter bestimmten Umständen ausgelöst werden. Sie unterbrechen den regulären Ablauf eines Programms und werden von speziellen Programmteilen, den Interruptroutinen, bearbeitet. Die Ursache für den Einsatz von Interrupt–Routinen liegt in der im Vergleich zum Prozessor langsamen Arbeitsweise von Peripheriegeräten. Bei einer seriellen Schnittstelle dauert z.B. die Übertragung eines Zeichens bei 9600 Baud ca. 1 ms; selbst der alte MC68000–Prozessor könnte die Daten jedoch um einen Faktor von ca. 1000 schneller senden oder empfangen. Da die Übertragung eines Zeichens von der Hardware selbständig erledigt wird und den Prozessor nicht benötigt, ist es sinnvoll, den Prozessor in dieser Zeit andere Aufgaben erledigen zu lassen. Erst bei Ende der Übertragung eines Zeichens wird der Prozessor wieder benötigt, sei es, um ein neues Zeichen bereitzustellen, oder um das Ende der Übertragung zu erkennen. RTOS–UH nutzt diese Pausen durch Verteilung der Prozessorkapazität an laufwillige Tasks. Lediglich dann, wenn der Prozessor für die Betreuung eines I/O–Bausteins erforderlich ist, wird er auch hierfür eingesetzt. Die I/O–Bausteine lösen in diesem Fall einen Interrupt aus, der die regulär arbeitenden Programme unterbricht. RTOS–UH hält für diese Fälle spezielle Interruptroutinen zur Betreuung der Bausteine bereit. Hiermit wird erzielt, das an keiner Stelle des Betriebssystems Prozessorkapazität durch wiederholtes Abfragen von Peripheriebausteinen verschwendet wird.

Interrupt–Routinen liegen ausserhalb des Task–Konzeptes von RTOS–UH, da die Interruptbeantwortung schon im Prozessor als Sonderfall angelegt ist. Für die Dauer der Bearbeitung einer Interrupt–Routine ist der Taskwechsel–Mechanismus von RTOS–UH paralysiert. Um diese Paralyse–Zeiten möglichst gering zu halten, arbeiten sämtliche Interrupt–Routinen von RTOS–UH mit Systemdiensten zusammen, die im Regelfall als Tasks angelegt sind. Auch hier werden die RTOS–UH–eigenen Mechanismen der Taskzustandsänderung ausgenutzt: die Systemtasks setzen ihre Abarbeitung aus oder planen sich ein, um dann von den Interrupt–Routinen wieder fortgeführt zu werden. Kontrollierten Zugriff auf diese Interrupt–Routinen hat lediglich der Systemprogrammierer: Anwenderprogramme bedienen sich hierzu stets der Systemtasks.

15.4.1 System–Interrupt–Handler

RTOS–UH benutzt Interrupts überall da, wo es gilt, nicht unnötig auf langsame Peripherie zu warten. Sie machen überall dort Sinn, wo angestossene I/O–Vorgänge so langsam sind, dass Prozessor–Restkapazität zu verteilen ist.

15.4.1.1 Timer–Interrupt

Eine besondere Rolle spielt der Timer–Interrupt. Für die Systemuhr wird ein periodischer Interrupt benutzt, der in den meistens RTOS–UH–Systemen jede Millisekunde ausgelöst wird, um die Systemzeit selbstständig zu verwalten. Die hierzugehörende Interrupt–Routine prüft bei jedem Uhr–Interrupt, auch Clock–Tick genannt, ob ein Zeitpunkt vorliegt, zu dem eine Einplanung existiert. Ist dies der Fall, so werden die erforderlichen Massnahmen zur Taskzustandsänderung durchgeführt und der Taskumschalter gestartet, um den Prozessor der nun höchstpriorisierten Task zuzuteilen.

Selbstverständlich ist es auch möglich, eigene Timer zu programmieren, um z.B. periodische Interrupts mit höherer Frequenz als dem üblichen Zeitatom von RTOS–UH (1kHz) zu erzeugen. Bei der Erzeugung von hochfrequenten periodischen Interrupts sollten Sie jedoch stets im Auge behalten, dass dadurch die für Anwendertasks freie Prozessorleistung herabgesetzt wird. Das kann auf langsamen

Maschinen dazu führen, dass das System sich nur noch mit der Verwaltung von Unterbrechungen beschäftigt und der „normale“ Betrieb stark eingeschränkt oder völlig paralysiert wird.

15.4.1.2 Schnittstellen–Interrupt

Wie schon oben erwähnt, ist die Ausgabe von Zeichen über die meisten seriellen und parallelen Schnittstellen langsam im Vergleich zur Prozessorgeschwindigkeit. Daher gibt es in RTOS–UH für jede Schnittstelle eine Interrupt–Routine, die die Ein– und Ausgabe von Zeichen unabhängig von jeder Task durchführt. Lediglich für Anfang und Ende des I/O–Vorgangs arbeiten diese Interruptroutinen mit ihren Systemtasks zusammen. Für die Dauer der tatsächlichen Ein– oder Ausgabe setzen diese Systemtasks ihre Abarbeitung aus (sind im Zustand `SUSP`) und werden von den Interrupt–Routinen nach Beendigung der I/O wieder fortgesetzt.

15.4.1.3 Floppy–Interrupt

Auch beim Betrieb von Massenspeichern treten längere Wartephasen auf. Daher existieren auch hier im Regelfall Interrupt–Routinen, die es ermöglichen, dass der Prozessor in Wartephasen den laufwilligen Tasks zugeteilt wird. Die zugehörige Systemtask hält für diese Zeit ihre Abarbeitung ebenfalls an und wird bei Bedarf von der Interrupt–Routine fortgesetzt.

15.5 I/O unter RTOS–UH

Wesentlicher Bestandteil eines jeden Betriebssystems ist die Kommunikation mit der Peripherie. In einem Single–Tasking–Betriebssystem ist die Behandlung von I/O–Vorgängen relativ simpel. Hier regt sich niemand auf, wenn der Rechner auf Bedienerebene *steht*, nur weil man eben mal `DIR` gesagt hat. In einem Echtzeit–Multitasking–System sind die Anforderungen an die I/O schon erheblich rabiater. Einerseits bewerben sich nun mehrere Parteien quasi parallel um die Betriebsmittel und andererseits ist der Gedanke, dass eine Task mit hoher Priorität die Rechenzeit der CPU in einer Poll–Schleife verheizt, nicht gerade verlockend.

Viele Betriebssysteme (DOS, viele UNIX–Systeme, OS/9, etc. . .) sind in dieser Hinsicht eher schlecht drauf und warten *leider nur zu oft* tapfer auf die Fertigmeldung langsamer Peripherie, indem sie ohne Erbarmen die I/O–Ports immer und immer wieder abfragen. RTOS–UH hat geht hier andere Wege, die für neue Anwender des Betriebssystems erstmal ungewohnt erscheinen.

15.5.1 Direkte Speicherzugriffe

Oftmals ist ein schneller Zugriff auf I/O–Karten erwünscht. In diesen Fällen bietet C mit den normalen Sprachmitteln ausreichende Möglichkeiten, um 99% der auftretenden Fälle zu erschlagen. Es wird ein Pointer mit geeignetem Basisdatentyp auf die entsprechende Speicherstelle gelegt und über `normale` Zuweisungen darauf zugegriffen. Diese Methode funktioniert allerdings nur, wenn der Zugriff über `MOVE.x`–Anweisungen gestattet ist.

15.5.1.1 Überwachte Speicherzugriffe

In manchen Fällen ist es ganz hilfreich, Speicherzugriffe auf geschützte oder illegale Adressbereiche machen zu können, ohne dabei gleich einen `BUS-ERROR` auszulösen. Dieser Zustand ist hochgradig peinlich, weil er das Aus für die verursachende Task darstellt. Einsatzfälle sind z.B. Adressbereiche,

die nur im Supervisor–Mode zugreifbar sind oder Zugriffe über den Bus auf Bereiche, von denen man sich nicht sicher sein kann, ob sie gültig sind oder die entsprechende Karte gerade nicht im Rechner steckt. Derartige Zugriffe mit der Gefahr eines BUS-ERROR's sind nur im Supervisor–Mode abzufangen. Speicherbereiche, die **nicht** im Supervisor–Mode erreicht werden können — wie z.B. entsprechend gejumperte VME–Bus–Karten — lassen sich folglich nicht mittels dieser Funktionen testen, da in solchen Fällen grundsätzlich beim Zugriff ein BUS--ERROR erzeugt wird.

```
int rt_read_memory_byte( void  *address, UBYTE  *value ) ;
int rt_read_memory_word( void  *address, UWORD  *value ) ;
int rt_read_memory_long( void  *address, ULONG  *value ) ;

int rt_write_memory_byte( void  *address, UBYTE  value ) ;
int rt_write_memory_word( void  *address, UWORD  value ) ;
int rt_write_memory_long( void  *address, ULONG  value ) ;
```

In den CREST–C–Bibliotheken stehen sechs Funktionen zur Verfügung, die vor dem Zugriff in den privilegierten Modus wechseln, einen möglichen BUS-ERROR wegfangen und hinterher über den Rückgabewert berichten, ob der Zugriff geklappt hat (Rückgabewert 1) oder mit BUS-ERROR bestraft wurde (Rückgabewert 0). Der Zugriff auf den Speicher erfolgt über MOVE.x–Befehle. Das gelesene Datum ist bei den rt_read_memory_...–Funktionen über den Parameter value verfügbar. Die rt_write_memory_...–Funktionen schreiben das Datum value auf die Speicheradresse address.

Und weil die Welt halt hart und grausam ist, sind die gerade beschriebenen Routinen natürlich restlos unbrauchbar, wenn das Anwenderprogramm sich bereits im Supervisor–Mode befindet. Der Aufruf rt_write_...– und rt_read_...–Funktionen führt in solchem Falle nämlich zum ungewollten Rücksturz in den Usermode — was dann schnell ins Chaos mündet.

Wenn Sie sich bereits im Supervisor–Mode befinden, so sind die folgenden Funktionen zwingend erforderlich, um BUS-ERROR–überwachte Zugriffe durchzuführen.

```
int rt_super_read_memory_byte( void  *address, UBYTE  *value ) ;
int rt_super_read_memory_word( void  *address, UWORD  *value ) ;
int rt_super_read_memory_long( void  *address, ULONG  *value ) ;

int rt_super_write_memory_byte( void  *address, UBYTE  value ) ;
int rt_super_write_memory_word( void  *address, UWORD  value ) ;
int rt_super_write_memory_long( void  *address, ULONG  value ) ;
```

Sie sind von der Funktionsweise identisch mit den korrespondierenden Funktionen ohne das_super–Kürzel im Namen.

15.5.1.2 Peripherie Ein/Ausgabe

Diese Traps erlauben spezielle I/O–Operationen. Der Zugriff ist implementationsabhängig. Auf vielen RTOS–UH–Systemen sind die zugehörigen Traps nicht angeschlossen und der Aufruf führt zu einer WRONG OPCODE–Meldung des Betriebssystems.

Diese Funktionen rt_peripheral_input() und rt_peripheral_output() sind beim Umgang mit C normalerweise unnötig, solange die Zugriffe sich auf MOVE.B, MOVE.W oder MOVE.L beschränken, die der CCC bei Pointerzugriffen generiert. Sind jedoch auf Ihrer RTOS–UH–Implementierung die Traps PIT und POT explizit beschrieben, so lassen sich diese mittels der rt_peripheral_...–Funktionen ansprechen.

Mittels rt_peripheral_input() wird ein lesender Zugriff auf eine Peripherie–Adresse durch-

geführt. Der Rückgabewert ist implementierungsabhängig.

```
ULONG rt_peripheral_input( void  *p_addr ) ;
```

Entsprechend wird bei `rt_peripheral_output()` das Datum `data` nach der jeweiligen Implementierungsvorschrift auf `p_addr` ausgegeben.

```
void rt_peripheral_output( void  *p_addr, ULONG  data ) ;
```

15.5.2 Von CE's, Queues und Betreuungstasks

Zunächst eine kleine Vorbemerkung zum Thema Aktualtität. Was in dieser Dokumentation über das Verhalten von RTOS–UH beim Umgang mit `CE`'s festgehalten ist, entspricht dem Stand des Nukleus 6.x. Ab der Version 7.0 hat sich intern im RTOS–UH eine Menge geändert.

Mein Entschluss, jetzt auf Internas einzugehen, die im Prinzip schon überholt sind, beruht auf der schlichten Tatsache, dass tausende dieser alten 6.x–Kerne heute stabil im Einsatz sind. DieÄnderungen, die sich durch den neuen Nukleus ergeben, wirken sich auf Hochsprachenprogrammierer nur unwesentlich aus. CREST–C läuft seit Anfang 1992 auf dem damals brandneuen Kernel der Uni Hannover. In der aktuellen Version wurden die neuen Trapanschlüsse übernommen und auch dokumentiert. Sie lassen sich jedoch erst mit dem neuen Kern ab NUK 7.x nutzen.

15.5.2.1 Anforderung eines `CE`'s

Unter RTOS–UH findet eine Abkopplung einer Task, die von langsamer Peripherie lesen möchte und dem eigentlichen physikalischen Lesevorgang statt. Eine Task, die I/O–Vorgänge ausführen möchte, muss sich zu diesem Zwecke zunächst vom Betriebssystem eine Datenstruktur anfordern, die zur Versendung derartiger Kommunikations–Aufträge dient. Den Aufbau eines als `Communication Element` oder kurz `CE` bezeichneten Paketes können Sie Abbildung 15.3 oder der Includedatei <`rtos.h`> entnehmen.

```
struct Ce
{  MemSectionHeader   head         ;
   MemSectionHeader  *fort         ;
   MemSectionHeader  *backt        ;
   Task              *tid_of_owner ;
   Ce                *fors         ;
   Ce                *backs        ;
   Prio               prio         ;
   char              *buffer       ;
   IOlen              reclen       ;
   IOstatus           status_of_io ;
   IOqueue            ldn          ;
   IOmode             mode         ;
   IOdrive            drive        ;
   FileName           file_name    ;
} ;
```

Abbildung 15.3: C–Struktur zur Verwaltung eines `CE`'s

Bevor Sie sich ein `CE` vom Betriebssystem beschaffen, müssen Sie sich darüber klar werden, dass es sich bei einer solchen Aktion um die Beschaffung dynamischen Speichers handelt. RTOS–UH holt für die anfordernde Task einen Block aus dem Freispeicher. Um zu verhindern, dass eine Task sich unkontrolliert mit dynamischem Speicher vollsaugt und das ganze System plötzlich aus Speicherplatzmangel

liegenbleibt, führt RTOS–UH darüber Buch, wieviel CE–Speicher jede Task aktuell im Besitz hat. Wird das Kontingent einer Task überschritten, so stoppt das Betriebssystem den Gierschlund. Die Task bleibt mit dem Status CWS? liegen und wird erst wieder lauffähig, wenn CE's der Task abgearbeitet wurden und die Kontingentgrenze wieder unterschritten wird. An der Adresse $896.W liegt der Wert, der die CE–Kontingentangabe in Bytes enthält, die für ihr RTOS–UH gültig ist.

Erstmal klingt das ziemlich gemeingefährlich, aber es macht in einem Betriebssystem, das ohne MMU arbeitet und deshalb mit dem physikalisch vorhandenen RAM haushalten muss, durchaus Sinn, den Benutzer von sinnlosem Speicherhamstern abzuhalten. Derartige Blockierungen können gemeinerweise auch ohne böse Absicht auftreten. Wenn Sie mit einer niedrigprioren Task ohne Wartebedingung ständig einen Text auf ein angeschlossenes Terminal schicken und das Gerät sich permanent mit XOFF dagegen wehrt, würde ohne Kontigentbegrenzung nach einiger Zeit der Rechner mit CE's gefüllt sein und auch hochpriore Tasks, die dann noch Speicher anfordern würden, hätten in diesem Fall ein echtes Problem.

Anderseits ist es natürlich auch unerwünscht, dass eine Task gnadenlos gestoppt wird, nur weil sie etwas mehr CE–Speicher benötigt, als RTOS–UH für angemessen hält. Deshalb gibt es eine Chance, sich am Kontingent vorbeizumogeln.

Die Beschaffung von CE–Speicher hat stets mittels der Funktion rt_fetch_ce() zu erfolgen. Sie sollten niemals dem zunächst verlockenden Gedanken erliegen, sich mal eben eine Struktur struct Ce vom Compiler hinlegen zu lassen. Das böse Erwachen kommt sehr schnell bei der Verwendung eines solchen Blockes, denn ordentlich beschaffte CE's hängen in zwei internen Speicherketten von RTOS–UH. Sind diese Werte nicht korrekt vorbesetzt, kann so ziemlich alles im System passieren — bis zum totalen Stillstand.

Also verwenden Sie **grundsätzlich** die Funktion rt_fetch_ce() zur Beschaffung von CE's.

```
Ce  *rt_fetch_ce( size_t  len ) ;
```

Als Parameter wird eine Länge gefordert und als Resultat erhalten Sie grundsätzlich einen Pointer auf ein gültiges CE. Im CE sind anschliessend einige Einträge vorbesetzt. Die Priorität prio entspricht der der anfordernden Task. Das Strukturmitglied buffer zeigt auf den internen Puffer des CE's. status_of_io ist direkt nach der Anforderung auf Null vorbesetzt und file_name enthält ein einfaches (terminiertes) Blank.

Es gibt zwei Möglichkeiten eines Fehlschlags, bei denen die anfordernde Task blockiert wird. Die Angelegenheit mit der Kontigentüberschreitung mit CWS?–Blockierung (fehlender **C**ommunication–**W**ork–**S**pace) wurde ja bereits erklärt. Wenn im Rechner schon vor der Kontingentausschöpfung keine ausreichend grossen FREE–Blöcke mehr verfügbar sind, bleibt die Task ebenfalls liegen. Diesmal lautet der Status PWS? (fehlender **P**rocedur–**W**ork–**S**pace) und die Task muss sich gedulden, bis eine andere Task Speicher in ausreichender Grösse ans System zurückgibt. Anschliessend wird sie automatisch wieder lauffähig und taucht aus dem Funktionsaufruf auf.

Bei der Längenangabe len handelt es sich um einen 32–Bit–Wert, in dem die Grösse eines Puffers angegeben werden kann. *Wozu ein Puffer?* werden Sie sich vielleicht beim Vergleich mit anderen Betriebssystemen fragen. Im Abschnitt 15.5.2.2 werden die Möglichkeiten beim Umgang mit CE's weitergehend erläutert. Für die CE–Anforderung ist es erstmal nur wichtig, sich darüber klarzuwerden, ob man die Daten eines I/O–Vorgangs in eigenen Puffern (C–Variablen oder dynamisch angefordertem Procedureworkspace) halten möchte oder den Puffer in einem Rutsch zusammen mit dem CE–Verwaltungsblock neu anfordern möchte oder muss.

Wenn Sie auf einen CE–internen Puffer verzichten wollen, ist len=0 zu übergeben. Sie erhalten dann einen Pointer auf den nackten CE–Kopf. len–Werte grösser Eins führen dazu, dass direkt im Anschluss an den CE–Kopf ein nicht initialisierter Speicherblock allokiert wird. Die Abbildung 15.4 verdeutlicht, dass man um Puffer bei CE's nicht herumkommt, wenn Daten gelesen oder geschrieben werden sollen. Der Eintrag buffer im CE muss auf einen Speicherbereich zeigen, der für den I/O–Vorgang verwendet

werden soll. Wenn Sie ein CE mit internem Puffer anfordern, so ist dieser Pointer bereits initialisiert. Ansonsten müssen Sie buffer selbst auf einen Speicherbereich vorbesetzen.

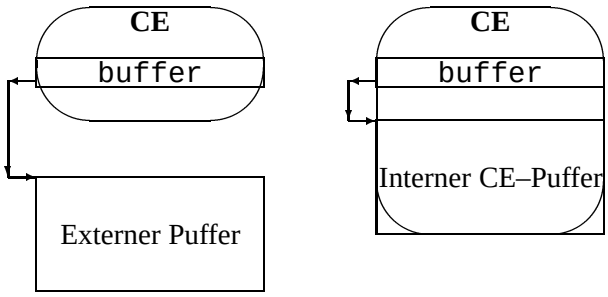


Abbildung 15.4: CE mit eigenem oder externem Puffer

Über die Längenangabe wird auch gesteuert, ob das CE–Kontingent überschritten werden darf. Ungerade Werte für das Argument len übersteuern die Kontingentüberprüfung bei der Anforderung eines CE's. Vorsicht: das unterste Bit im Parameter len wird von rt_fetch_ce() nur als Flag interpretiert! Wenn Sie z.B. einen Puffer für 1 Byte anfordern wollen und dabei sicherstellen müssen, dass RTOS–UH nicht die CWS?–Vollbremsung einleitet, so muss der Aufruf mit len=3 erfolgen. Um ein Byte mit Kontigentüberprüfung anzufordern, muss len=2 eingetragen werden. Die Regeln für Puffergrössen lauten:

1. Sie können niemals ungerade Pufferlängen anfordern. Wenn Sie unbedingt eine ungerade Puffergrösse benötigen, müssen Sie grundsätzlich auf den nächsten geraden Wert aufrunden — und für unbedingte Anforderungen sogar noch ein weiteres Byte draufschlagen.
2. Wenn Sie CE's ohne Kontingentüberprüfung anfordern wollen, müssen Sie das unterste Bit in der Längenangabe anknipsen.
3. Auch CE's ohne Kontigentüberprüfung werden dem Kontingent der anfordernden Task zugerechnet. Nach erfolgter Kontingentüberschreitung führt bereits das kleinste CE, das Sie mit Kontingentüberwachung anfordern, zur CWS?–Blockierung der Task.
4. Bei CE's, die später zur I/O verwendet werden sollen, machen Längenangaben nur im Bereich von 0x0000 bis 0x7FFF Sinn, da nur die unteren 15 Bit des Eintrags reclen als Recordlänge des CE–Puffers interpretiert werden.

15.5.2.2 Verschicken eines CE's

Nach der Beschaffung eines Communication Elements geht es nun darum, das CE so aufzufüllen, dass der I/O–Auftrag auch in geeigneter Form von RTOS–UH bearbeitet werden kann. Der Abbildung 15.5 ist nochmals der Aufbau eines CE's dargestellt. Die fettgedruckten Namen der Strukturmitglieder sind als Spielwiese für den normalsterblichen Programmierer freigegeben. Der obere Teil des CE's besteht aus internen Verwaltungsinformationen, die RTOS–UH dringend benötigt. Die Einträge von head bis backs sind für Sie absolut tabu! Willkürliche Änderungen in diesem Bereich führen sehr schnell zum Crash.

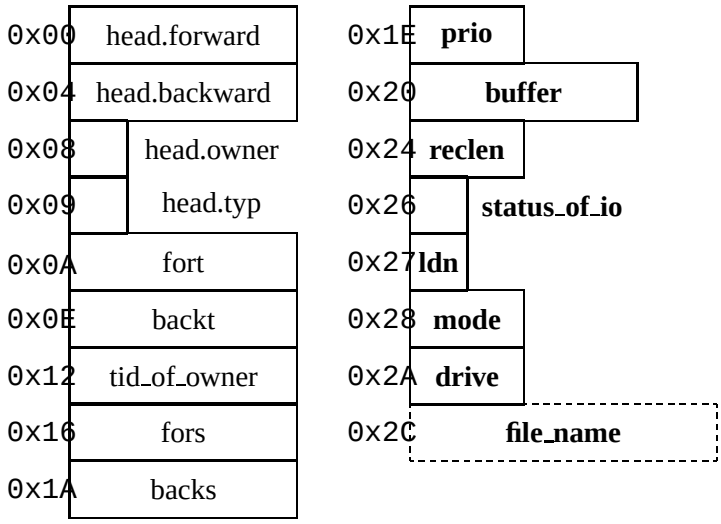


Abbildung 15.5: Darstellung eines `CE`'s

Interessant wird es im unteren Bereich der Struktur. Hier wird darüber entschieden, wer das `CE` bearbeiten soll (`ldn`, `drive`), unter welchem Namen der Auftrag verschickt wird (`file_name`), wo die Daten liegen (`buffer`) und wieviel Bytes von diesem `CE` verarbeitet werden können (`reclen`). Nun fehlt nur noch die Angabe über die Art des Auftrags (`mode`, `status_of_io`) und schon kann das `CE` verschickt werden. Wenn Sie zudem den I/O–Vorgang mit einer anderen Priorität als der ihrer Task starten wollen, können Sie den Default–Wert von `prio` ändern.

Und nun noch mal langsam zum Mitschreiben. Ein `CE` ist eigentlich nur ein einfaches Daten–Päckchen, das man allerdings penibel beschriften muss, um nicht unwissentlich eine Packetbombe in die Welt hinauszuschicken. Wenn ein `CE` mit `rt_fetch_ce()` beschafft wurde, so ist die anfordernde Task fortan der Eigentümer dieses speziellen Speicherblocks. In `tid_of_owner` ist die Task–ID dieser Task eingetragen. Dieser Pointer ist quasi die Absenderangabe des Paketes. Nebenbei erfolgt über die beiden Pointer `fort` und `backt` eine doppeltverzeigerte Verkettung mit dem Taskworkspace des Eigentümers. Diese Kleinigkeiten hat RTOS–UH bei der Beschaffung des `CE`'s bereits für uns initialisiert.

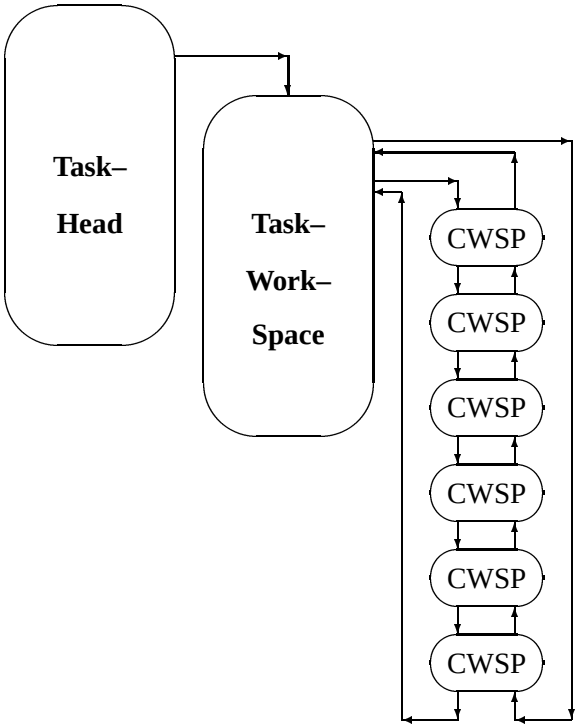


Abbildung 15.6: Verkettung der CE's mit der Task

Jetzt geht es darum, dem Betriebssystem mitzuteilen, wen wir mit der Bearbeitung des I/O–Auftrages beschäftigen wollen. Der Eintrag ldn (**L**ogical–**D**rive–**N**umber) dient dabei als eine Art Postfachnummer. RTOS–UH kann maximal 128 derartige Postfächer verwalten — die Nummern laufen von $00 bis $7F. In der Regel sind es aber ein paar weniger, denn RTOS–UH schaut bei der Konfiguration des Systems nach der höchsten LDN aller Betreuungstasks und richtet nur bis zu dieser Nummer auch Briefkästen ein. An der Position $89A.W finden Sie im Speicher den Wert, der die höchste im System unterstützte LDN plus Eins enthält — also die Zahl der eingerichteten Tabelleneinträge für I/O–Queues. Nähere Informationen entnehmen Sie bitte dem Abschnitt 15.5.2.3.

Der Versuch, ein CE an eine LDN zu verschicken, die grösser als die maximale LDN des Systems ist, wird von RTOS–UH mit WRONG LDN (XIO) beantwortet — was soviel heisst wie: *Adressat unbekannt!* Auftreten kann sowas eigentlich nur, wenn Sie Schrott ins CE eingefüllt haben (der Eintrag ldn ist dann der Bösewicht) oder ein Treiber angesprochen wurde, der nicht installiert wurde. Im zweiten Fall ist erstmal zu kontrollieren, ob die Hardware beim Hochlaufen des Systems ordnungsgemäss erkannt wurde — in diesem Falle melden sich die Treiber aus dem EPROM im Systemprolog der RTOS–UH–Startmeldung. Fehlt dort ein Treiber, den Sie da eigentlich erwartet hätten, so deutet alles darauf hin, dass eine Karte in ihrem Rechner fehlt (der positive Fall) oder die betreffende Hardware seit dem letzten Einschalten das Handtuch geschmissen hat — dann haben Sie ein echtes Problem...

Auch bei nachladbaren Treibern kann es zu derartigem Verhalten kommen. In 99% aller Fälle wurde schlicht vergessen, den Treiber zu laden oder der Treiber wurde bereits angesprochen, bevor er sich installiert hat. Lachen Sie nicht: das ist wirklich ernst gemeint und fasst stundenlange Online–Fehlersuche bei verzweifelten Anrufern knapp und bündig zusammen.

Mit ? -D können Sie sich anschauen, welche LDN's in ihrem System belegt sind. Das betrifft allerdings nur die Treiber, die beim Systemstart erkannt wurden. Nachgeladene Treiber werden in dieser Tabelle nicht aufgeführt — bedauerlich, aber leider ein Faktum. Einzelheiten entnehmen Sie bitte dem Abschnitt 15.5.2.6 und der Beispieltabelle 15.10. Die Tabelle der möglichen LDN's muss nicht unbedingt vollständig belegt sein. Das Verschicken von Nachrichten an existente Postfächer ohne registrierte Kunden führt ebenfalls zur Meldung WRONG LDN (XIO).

Zur Abfrage, ob und welche Betreuungstask für eine LDN existiert, steht unter CREST–C die Funktion rt_LDN_to_Tid() zur Verfügung.

```
Task  *rt_LDN_to_Tid( IOqueue  ldn ) ;
```

Die Funktion liefert die TID der Betreuungstask einer I/O–Queue — sofern sowohl die LDN gültig ist und eine Betreuungstask existiert. Sonst wird ein NULL–Pointer zurückgegeben.

Manchmal ist es auch von Interesse, aus einer bekannten LDN das zugehörige Userinterface zu ermitteln. Mittels der Funktion rt_LDN_to_USER_Tid() lässt sich die TID des korrespondierenden Kommandointerfaces bestimmen.

```
Task  *rt_LDN_to_USER_Tid( IOqueue  ldn ) ;
```

Die Funktion liefert die TID des Users einer I/O–Queue — sofern sowohl die LDN gültig ist, als auch ein Userinterface für diese LDN existiert — oder NULL.

Damit wäre also der Eintrag ldn im CE erschöpfend beschrieben. Der Wert von drive kommt thematisch aus der gleichen Ecke. Die LDN spezifiziert üblicherweise eine Betreuungstask für ein bestimmtes Betriebsmittel. Über drive werden spezielle Informationen für die Betreuungstask auf den Weg gebracht. Bei seriellen Schnittstellen kann das z.B. die Angabe der Betriebsart sein (A1, B1, C1) und bei Festplatten die anzusprechende Partition (H0, H1, etc...).

Mit diesen beiden Werten versehen, würde unser CE bereits das Ziel erreichen. Für viele Betreuungstasks sind diese Adressangaben bereits hinreichend. Eine Betreuungstask, die z.B. eine Floppy zu verwalten hat, benötigt für einen Zugriff auf eine einzelne Datei zusätzlich noch eine Möglichkeit, die jeweilige Datei zu identifizieren. Unter RTOS–UH erfolgt diese Identifizierung grundsätzlich über die Angabe des Dateinamens und des kompletten Zugriffspfades, unter dem die Datei bei diesem Betriebsmittel anzusprechen ist.

Für Sie stellt sich nun das Problem, in jedem CE den kompletten Dateinamen (ohne die Angabe des Gerätes) unter file_name einzutragen und den String ordnungsgemäss mit $FF zu terminieren, um RTOS–UH und die auswertenden Betreuungstasks bei Laune zu halten. Bitte denken Sie daran, diese Aktion nicht gedankenlos mit strcpy() durchzuführen, da RTOS–UH mit der abschliessenden Null herzlich wenig anfangen kann. Der komplette Eintrag in file_name darf 23 Zeichen plus Terminator nicht überschreiten — wenigstens gilt das für die meisten gebräuchlichen Systeme mit einer maximalen Pfadlänge von 24 Zeichen. Neuere RTOS–UH–Systeme verwalten die maximal zulässige Namenslänge sauber über eine Variable, die an der Position $8F2.W im Speicher zu finden ist.

Ein gültiger Eintrag für einen Filenamen könnte z.B. wie folgt vorgenommen werden:

```
strncpy( ce->file_name,
                 ÖRDNER/RTOS/NAME\xFF",
         sizeof( ÖRDNER/RTOS/NAME\xFF" ) - 1 ) ;
```

Solange man sicher davon ausgehen kann, dass es nicht zur Überschreitung der maximalen Namenslänge kommt, tut es allerdings auch die folgende Variante:

```
strcpy( ce->file_name, "ORDNER/RTOS/NAME\xFF" ) ;
```

Wohin das CE gelangt, haben wir jetzt geklärt. Jetzt beschäftigen wir uns etwas eingehender damit, was man mit den Communication Elementen eigentlich anstellen kann. Bevor man losgeht und willkürlich Kommandos generiert, ist es ganz nützlich, zunächst abzuklären, welche Befehle eine Betreuungstask überhaupt versteht und unterstützt.

Unter RTOS–UH besteht mittels des Bedienbefehls SD (Set Device Parameter) und DD (Display Device Parameter) die Möglichkeit, die Eigenschaften von Gerätetreibern abzufragen.

Auf C–Ebene korresponieren dazu die zwei Funktionen `rt_set_device_descriptor()` und `rt_get_device_descriptor()`.

`rt_get_device_descriptor()` liefert den Device–Parameter der angesprochenen `ldn`. Die Funktion überprüft nicht, ob die angegebene `ldn` überhaupt existiert! Wenn nicht, wird ein zufälliger Speicherinhalt zurückgeliefert.

```
Device rt_get_device_descriptor( IOqueue ldn ) ;
```

Der Device–Parameter einer `ldn` kann auch gesetzt werden. Es findet keine Überprüfung statt, ob die angegebene `ldn` existiert. Veränderungen sollten also nur vorgenommen werden, wenn dies sichergestellt ist; ansonsten ist die Wahrscheinlichkeit eines Systemabsturzes eher gross!

```
void rt_set_device_descriptor( IOqueue ldn, Device mask ) ;
```

Makro	Wert	Bedeutung
IOFMRE	0x8000	Rewindable
IOFMOC	0x4000	Open/closable
IOFMLF	0x2000	add Linefeed
IOFMDI	0x1000	dialog possible
IOFMNE	0x0800	no echo
IOFMER	0x0400	errasable
IOFMOU	0x0200	output possible
IOFMIN	0x0100	input possible
IOFMFL	0x0080	dir possible
IOFMFO	0x0040	formatierbar
IOFMCF	0x0020	changeable
IOFMSD	0x0010	subdirectory
IOFMRD	0x0008	sync, seek, save
IOFMNW	0x0002	no wrap
IOFMCE	0x0001	cursor by escape

Tabelle 15.5: Bitmuster für Geräte–Eigenschaften

15.5.2.3 I/O–Queues und Gerätetreiber

Wenn eine Task unter RTOS–UH einen I/O–Auftrag mit `rt_transfer_ce()` losschickt, dann gelangt das versandte `CE` nicht unmittelbar zum zuständigen Gerätetreiber sondern landet zunächst in der für die angegebene `LDN` zuständigen I/O–Queue (auch Warteschlange genannt). Für jede gültige `LDN` existiert eine kleine Struktur, mittels derer sich eine prioritätengesteuerte doppeltverkettete Liste verwalten lässt. Der Abbildung 15.7 können Sie die zugehörige C–Struktur entnehmen. Fassen Sie diese Struktur bitte als kleinen Überblick und nicht als Aufforderung, daran herumzubasteln auf. RTOS–UH erlaubt den Zugriff auf seine Innereien nur über die amtlichen Traps bzw. C–Funktionsaufrufe.

```
struct
{  Ce       *fors  ;
   Ce       *backs ;
   Prio      prio  ;
}  IOqueue[ 128 ];
```

Abbildung 15.7: Interner Aufbau einer Warteschlange

Ein einlaufendes `CE` wird zunächst (äusserst rudimentär) auf Gültigkeit geprüft. Anschliessend wird es — gemäss der eingetragenen Priorität der verschickenden Task — so in die Kette der zugehöri-

gen LDN eingereiht, dass es hinter allen CE's mit höherer oder gleicher Priorität auftaucht. Gerade Anfängern erscheint es immer etwas uneinsichtig, dass Bildschirmausgaben verschiedenpriorisierter Tasks sich nicht in das gewohnte Denkschema des zeitlichen Nacheinanders einreihen lassen. In den Warteschlangen geht es eben absolut undemokratisch zu. Ein CE mit hoher Priorität drängelt sich in der Warteschlange soweit vor, bis es auf ein CE mit identischer oder höherer Priorität stösst. Wenn zwei Tasks unterschiedlicher Priorität *pausenlos* CE's zu einer Warteschlange schicken, so werden Sie nie die Bearbeitung eines CE's der niederprioren Task beobachten — mal ein langsames Ausgabegerät vorausgesetzt, das bereits mit den CE's einer Task ausgelastet ist. Dieser Effekt führt bei der Interpretation von Taskausgaben oft zu verzweifeltem Staunen über die kausalen Widersprüche der Ausgabemeldungen.

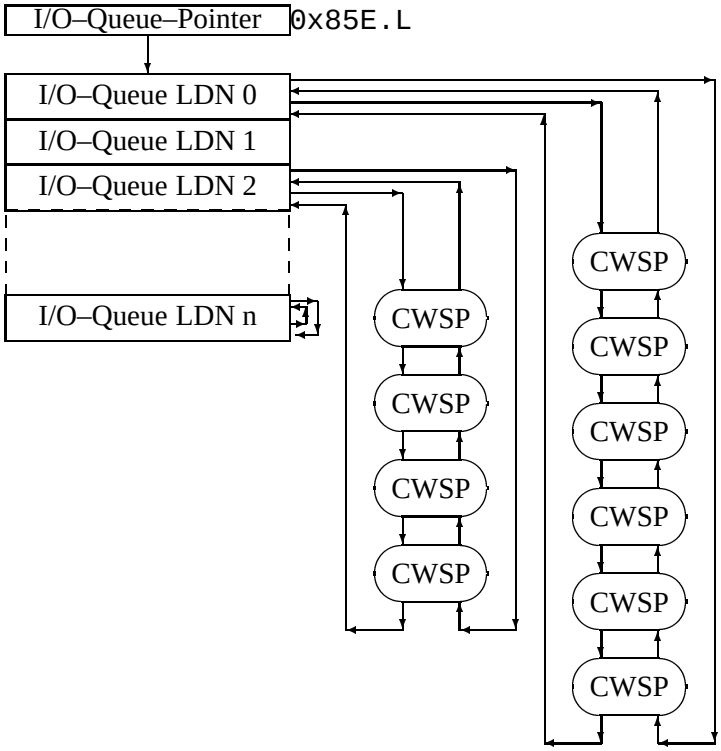


Abbildung 15.8: Beispiel einer Warteschlange

Jetzt steht unser CE also in der I/O–Queue und die Funktion `rt_transfer_ce()` geht daran, dem Empfänger Bescheid zu sagen, dass ein Paket für ihn da ist. Dazu gibt es unter RTOS–UH ein Feld von Taskpointern, in denen sich jede beliebige Task einklinken kann, die die Betreuung einer bestimmten LDN zu übernehmen beabsichtigt. Der Abbildung 15.9 können Sie den prinzipiellen Aufbau entnehmen. Den Pointer auf den Start dieses `LDN-to-TID`–Feldes finden Sie an der Position `$852.L`. Der Zugriff auf die Betreuungstask mit `LDN=7` wäre also mit `((Task**)0x852L)[7]` möglich. LDN's, die nicht von Treibern unterstützt werden, enthalten `NULL`–Pointer. Zur Abfrage, ob und welche Betreuungstask für eine LDN existiert, steht unter CREST–C die Funktion `rt_LDN_to_Tid()` zur Verfügung.

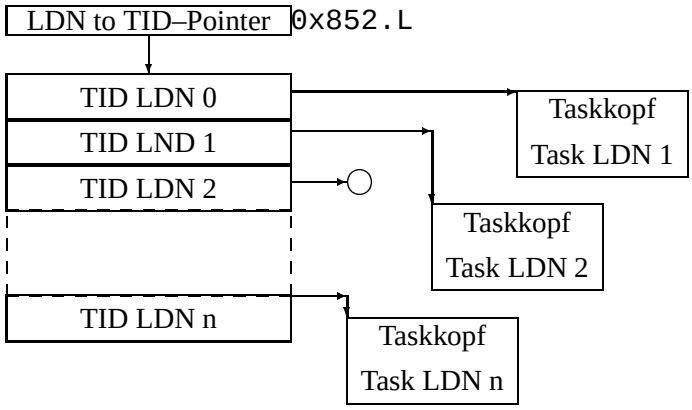


Abbildung 15.9: Verwaltung der Betreuungstask–Adressen

Wenn Sie selbst eine Betreuungstask für eine Warteschlange kodieren wollen, dann gibt es unter CREST–C zwei Möglichkeiten, RTOS–UH diese Neuigkeit höflich beizubringen. Im Abschnitt 13.4 ist beschrieben, wie man für Eprom–Systeme die Konventionen erfüllt, damit das Betriebssystem beim Kaltstart die entsprechende C–Funktion als Betreuungstask einträgt. Zur Laufzeit ist es allerdings ebenfalls noch möglich, nachträgliche Einträge in der `LDN-to-TID`–Tabelle vorzunehmen. CREST–C stellt die zwei Funktionen `rt_make_drive()` und `rt_delete_drive()` zur Verfügung, mit der sich die aktuell laufende Task als Betreuungstask für die übergebene `LDN` eintragen und später auch wieder austragen kann.

```
int rt_make_drive  ( IOqueue ldn ) ;
int rt_delete_drive( IOqueue ldn ) ;
```

Über Sinn oder Unsinn der übergebenen `LDN` müssen Sie sich allerdings weitestgehend selbst den Kopf zerbrechen. Existiert unter der angegebenen Nummer bereits ein Eintrag, wird er gnadenlos überschrieben bzw. gelöscht. War die übergebene `LDN` völliger Unfug, so liefert die Funktion eine Null als Rückgabewert.

Zurück zu unserem `CE`. Irgendwann kommt die passende Betreuungstask ans Laufen und kann nachschauen, wer etwas von ihr will. Dazu befragt sie die zugehörige I/O–Queue mit der Funktion `rt_take_of_queue()`.

```
Ce  *rt_take_of_queue( IOqueue  ldn ) ;
```

Wenn ein `NULL`–Pointer geliefert wird, so ist die I/O–Queue aktuell leer. Ansonsten erhalten Sie einen Pointer auf das prioritätsmässig höchste `CE` in der betreffenden Warteschlange und können es nun bearbeiten. Der nachfolgende Quelltext zeigt die wesentlichen Bestandteile, die zum Aufbau einer Betreuungstask notwendig sind. Im Beispiel wird für die übergebene `LDN` eine Betreuungstask eingerichtet, die lediglich Lese– und Schreiboperationen unterstützt und jedes eingelaufene `CE` ohne Fehlermeldung gleich wieder zurückgibt. Dank des Subtask–Konzepts von CREST–C können Sie derartige Treiber mehrmals starten und damit ohne Explosion der Codegrösse z.B. mehrere unterschiedliche `LDN`'s mit dem identischen Treibercode unabhängig voneinander verwalten.

```
#pragma SUBTASK PRIO -1
void Betreuungstask_zur_Laufzeit( register IOqueue  ldn )
{
   register Ce      *work_ce   ;
   register void    *sys_stack ;

   // Eigenschaften der Betreuungstask definieren
   rt_set_device_descriptor( ldn, IOFMIN | IOFMOU ) ;
```

```
   // Bei RTOS als Betreuungstask eintragen
   rt_make_drive( ldn ) ;

   for (;;)
   {
      // Warteschleife bis zum Eintreffen eines CE's
      for (;;)
      {
         if ( ( work_ce = rt_take_of_queue( ldn ) ) == NULL )
         {
            // Betreuungstask geht jetzt schlafen
            rt_wait_for_activation() ;
         }
         else
            break ;
      }
      /* Eintragen, der aktuellen User-Nummer aus dem CE,  */
      /* um Fehlermeldungen richtig zuordnen zu koennen... */
      if ( work_ce->tid_of_owner )
         rt_my_TID()->head.owner = work_ce->head.owner ;

      /*****************************************/
      /* Hier koennen dann beliebige Dinge mit den */
      /* Daten des CE's angestellt werden...       */
      /*****************************************/

      // Und hier wird das CE wieder freigegeben
      rt_release_ce( work_ce ) ;
   }
}
```

15.5.2.4 Warten auf Beendigung eines I/O–Vorgangs

Die Funktion `rt_wait_for_ce()` überprüft, ob sich das angegebene `CE` noch in einer Warteschlange oder Bearbeitung einer Betreuungstask befindet. Ist dies der Fall, wird die aufrufende Task blockiert (`I/O?`). Sie läuft weiter, wenn das `CE` freigegeben wurde, d.h. wenn die Betreuungstask die Bearbeitung des I/O–Vorgangs abgeschlossen hat.

```
void rt_wait_for_ce( Ce  *akt_ce ) ;
```

Die Verwendung dieser Funktion empfiehlt sich, wenn ein `CE` ohne Wait–Bit (`MODMWA`) abgeschickt wurde und nun weiter verwendet werden soll.

15.5.2.5 Freigeben eines `CE`'s

Um ein `CE`, das nicht länger benötigt wird, an das Betriebssystem zurückzugeben, existiert die Funktion `rt_release_ce()` . Das bezeichnete `CE` soll wieder in freien Speicher verwandelt werden. Zeigt `akt_ce` nicht auf ein `CE`, so sind Systemabstürze vorprogrammiert. Besonders ekelhaft ist es, wenn man durch Unvorsichtigkeit ein bereits freigegebenes `CE` erneut freigibt. `rt_release_ce()` überprüft, ob es sich bei dem adressierten Speicherbereich überhaupt um ein `CE` handeln kann und ignoriert eine Doppelfreigabe, wenn dies nicht der Fall ist. Die Funktion überprüft jedoch nicht, ob der Speicherbereich inzwischen von RTOS–UH an ein völlig anderes `CE` vergeben wurde. Die hässliche Folge besteht bei einer solchen Fehlfreigabe darin, dass nun eine nichtsahnende Fremd–Task, der man ihr `CE`

heimtückisch entrissen hat, kurz danach voll daneben greift — viel Spass beim Suchen nach solchen Fehlern!

```
void rt_release_ce( Ce  *akt_ce ) ;
```

Bei der Freigabe sind mehrere Fälle zu unterscheiden:

- Das `CE` ist nicht in einen Ein– oder Ausgabevorgang verwickelt. Es wird sofort in freien Speicher verwandelt.
- Das `CE` steht in der Warteschlange einer Betreuungstask, ist aber noch nicht in Bearbeitung. Es wird das Release–Bit gesetzt.
- Das `CE` ist in Bearbeitung einer Betreuungstask das `CE` hat keinen Besitzer mehr. Es wird das Release–Bit gesetzt.
- Das `CE` ist in Bearbeitung einer Betreuungstask, die aber nicht Eigentümer ist. Ist kein Eigentümer vorhanden, wird das `CE` zu freiem Speicher. Ist noch ein Besitzer eingetragen und das Release–Bit gesetzt wird das `CE` ebenfalls zu freiem Speicher, ansonsten wird kenntlich gemacht, dass das `CE` nicht mehr in eine I/O–Operation verwickelt ist.

15.5.2.6 Über Dateinamen und Pfade

Bei RTOS–UH gibt es verschiedene Methoden, von der Shell aus Geräte und Dateien anzusprechen. Diese Möglichkeiten sind historisch gewachsen und folglich nicht sonderlich konsistent. Bei CREST–C wird ausschliesslich die aktuellste Methode unterstützt, die in ihrer Syntax an UNIX angelehnt ist. Als Pathseparator wird der Slash '/' verwendet. Eine vollständige Angabe auf C–Ebene besteht aus dem Gerät (Device), dem Zugriffspfad und dem Namen des Files. Eine Angabe ohne führende Geräteangabe wird als relativer Pfad zum aktuellen Working–Directory betrachtet.

Wenn Sie direkt mit `CE`'s arbeiten wollen, besteht ein Zugriffspfad aus der Angabe von `ldn`, `drive` und dem kompletten Namen mit abgetrennter Gerätekennung.

Kompletter Name	`/h0/c/cc/h/p/test.c`
Kompletter Name ohne Gerät	`c/cc/h/p/test.c`
Gerätename	`/h0/`
Zugriffspfad	`c/cc/h/p`
Dateiname	`test.c`
Basisname der Datei	`test`
Endung der Datei	`c`

Tabelle 15.6: Nomenklatur bei DOS–ähnlichen Pfadangaben

Die Geräteangabe wird grundsätzlich mit einem Slash eingeleitet und auch mit einem Slash abgeschlossen. Innerhalb des Betriebssystem existiert eine Umsetzungstabelle, die den Gerätenamen die entsprechenden `ldn`/`drive`–Kennung zuordnet. Mit `? -D` erhalten Sie die Liste der auf Ihrem System unterstützten Devices. In Abbildung 15.10 liegt ein Auszug aus solch einer Liste vor.

Geräte, die nicht in dieser symbolischen Tabelle existieren, deren `ldn`/`drive`–Kennung jedoch bekannt ist, können ebenfalls direkt von CREST–C angesprochen werden. So kann das Laufwerk `/M0/` auf dem System, dessen Device–Tabelle Sie hier vorliegen haben, alternativ als `/LD/3,16/` verwaltet werden. Bei der Entwicklung von Software für verschiedene Zielsysteme ist die symbolische Schreibweise der `LDN`/`DRIVE`–Angabe stets vorzuziehen, da `ldn` und `drive` von System zu System variieren können und es somit die sauberere Methode darstellt, RTOS–UH zur Laufzeit die korrekte Zuordnung zu überlassen.

```
RTOS Devices  (LDN/Drive)

ED:.....01/00  EDB:.....01/01  EDX:.....01/02
XC:.....09/00  NIL:.....0F/00  A1:.....00/00
A2:.....02/00  UL:.....02/03  B1:.....00/02
B2:.....02/02  C1:.....00/06  C2:.....02/06
PP:.....0A/00  A3:.....04/00  B3:.....04/02
C3:.....04/06  D1:.....0B/00  D2:.....0C/00
D3:.....0E/00  F0:.....03/00  F1:.....03/01
M0:.....03/10  M1:.....03/11
```

Abbildung 15.10: Auszug aus einer Device–Tabelle

Um den Weg über die direkte Angabe von `ldn` und `drive` nicht zu verschütten, sollten Sie bei der Erstellung eigener Gerätenamen von der Verwendung des Device–Namens `/LD/` tunlichst absehen.

Bei der Angabe von Pfadnamen sind auch die von UNIX oder DOS bekannten Angaben `/./` und `/../` zulässig. Der einzelne Punkt stellt einen Selbstverweis auf das aktuelle Directory dar, ist also im Grunde genommen redundant. Mit `/../` lässt sich das jeweils übergeordnete Directory erreichen.

Die Angabe eines Datei– oder Verzeichnisnamens orientiert sich an der DOS–Schreibweise, besteht also aus maximal 8 Zeichen für den Basisnamen und einem optionalen Punkt als Einleitung für eine maximal drei Zeichen lange Dateiendung.

Entsprechend wäre die Schreibweise des oben angeführten Beispiels auch wie folgt noch syntaktisch korrekt:

```
/h0/c/./cc/h/dummy/../p/test.c
```

liefert:

```
/h0/c/cc/h/p/test.c
```

Es ist dabei zu berücksichtigen, dass während der Auswertung derartiger Pfadangaben kein physikalischer Zugriff auf die entsprechenden Verzeichnisse stattfindet. Da nach der Expansion des Beispiels kein Zugriff auf das Directory `dummy` stattfindet, kann der Zugriff auf `test.c` auch dann erfolgreich sein, wenn das Verzeichnis `dummy` nicht existiert. Ein Wechsel auf das übergeordnete Verzeichnis wird nicht ausgeführt, wenn bereits die Wurzel des Devices erreicht wurde.

```
/h0/../probe/../../c/./cc/h/dummy/../p/test.c
```

liefert ebenfalls:

```
/h0/c/cc/h/p/test.c
```

Nach der Expansion der Pfadangabe (ohne Gerätenamen) darf die resultierende Zeichenkette die Länge von 23 Zeichen nicht überschreiten — respektive der unter `$8F2.W` aufzufindenden maximalen Pfadlänge des Systems.

15.5.3 Praktische Anwendung von `CE`'s

Wenn Sie nunmehr die Grundlagen der `CE`'s verstanden haben, soll nun an Hand von ein paar gebräuchlichen Beispielen aufgezeigt werden, wie man in der Praxis damit umzugehen hat.

15.5.3.1 Serielle Schnittstellen

Unter RTOS–UH gehört der Umgang mit seriellen Schnittstellen aller Art zum Programmiereralltag, wie das tägliche Brot. Serielle Schnittstellen sind (wie alle Hardware) lebende Wesen — und sie sind böse! Im System sind Sie üblicherweise als /A1, /B2, /C3 etc. anzusprechen, wobei die Zahl quasi die Kanalnummer angibt und der führende Buchstabe die Betriebsart, in der der Treiber die Schnittstelle zu verwalten hat.

15.5.3.1.1 `ldn` und `drive` bestimmen: Die erste Aktion besteht immer darin, aus dem symbolischen Namen eines Gerätes das `ldn`/`drive`–Pärchen zu ermitteln, das zum Zugriff auf den Treiber benötigt wird. CREST–C stellt zu diesem Zwecke eine Funktion zur Verfügung, die die Klartextinformationen in die interne Darstellung umwandelt. Diese liegt in der Includedatei <`rtos.h`> als Prototyp zusammen mit der benötigten Datenstruktur vor.

```
typedef struct RFILE
{
   IOqueue   ldn       ;
   IOmode    mode      ;
   IOdrive   drive     ;
   FileName  file_name ;
}
   RFILE ;

int  rt_get_filedata( RFILE *rfile, char *name ) ;
```

Mit folgendem C–Code lassen sich z.B. die Informationen über die Schnittstelle /A3 erhalten. Die Funktion `rt_get_filedata()` liefert als Rückgabe einen von Null unterschiedlichen Wert, wenn die Dekodierung des übergebenen Namens `name` geklappt hat. In diesem Falle ist die übergebene Struktur `rfile` in den Einträgen `ldn` und `drive` mit den Werten versorgt worden, die auf diesem System für das betreffende Mnemonic des Gerätes gültig sind. Im Struktureintrag `file_name` ist der überarbeitete und von der Gerätekennung befreite Name in C–Nomenklatur eingetragen.

```
void A3_Info( void )
{
   auto  RFILE   port_info_A ;

   if ( rt_get_filedata( &port_info_A, "/A3/name" ) )
   {
      printf( Ä3: LDN=%3d DRV=%3d\n",
              (int)port_info_A.ldn,
              (int)port_info_A.drive ) ;
   }
}
```

Liefert die Funktion `rt_get_filedata()` dagegen eine Null, so ist die Dekodierung des übergebenen Namens gescheitert und der undefinierte Inhalt die Struktur `rfile` sollte besser nicht zur Kodierung eines `CE`'s herangezogen werden. Das kann z.B. folgende Gründe haben:

1. Die Gerätekennung ist dem System unbekannt. So werden z.B. willkürliche Angaben wie `/Hugo/`, `/Egon/` oder `/Waltraud/` nur auf den wenigsten RTOS–UH–Systemen zum *Aha!*–Erlebnis führen und sinnvolle `ldn`/`drive`–Kombinationen liefern[2].

[2] Warum bin ich mir nur so sicher, dass Sie jetzt ihren nächsten Treiber `/Emil/` oder `/Willy/` taufen werden?

2. Die Geräteangabe fehlt ganz. Nicht weiter schlimm, denn dann versucht die Funktion, das aktuelle Workingdirectory zu ermitteln und liefert `ldn/drive` für das aktuelle Laufwerk. Wenn jedoch kein Workingdirectory eingestellt ist, kommt es zum Fehlerabbruch...
3. Der Name ist unsinnig — soll heissen: zu lang, um vom RTOS–UH–System, auf dem der Aufruf stattfand, akzeptiert zu werden. Sie wissen schon: 23 Zeichen plus Terminator `0xFF`; respektive an der Position `$8F2.W` im Speicher die maximale Pfadlänge sauber auslesen.

15.5.3.1.2 Ein Ausgabe–`CE` aufbereiten und wegschicken Als Beispiel soll zunächst mit einer einfachen Übung begonnen werden. Über die Schnittstelle `/A2` soll ein kleiner ASCII–Text ausgegeben werden. Die Beispielfunktion sei so angelegt, dass sie im Fehlerfall Null und im Erfolgsfall einen Wert ungleich Null liefert.

Es wird damit begonnen, die Parameter der Schnittstelle zu ermitteln, was man selbstverständlich in einer echten Anwendung nicht in der Ausgaberoutine, sondern vielmehr einmal vorneweg machen würde.

```
int A2_Ausgabe( char  *str )
{
   auto     RFILE   port_info_A ;
   register Ce     *wce_A       ;

   if ( ! rt_get_filedata( &port_info_A, "/A2/A2_Ausgabe" )
      return( 0 ) ;
   else
   {
```

Hat diese Aktion zum Erfolg geführt, dann kann man sich mit der Beschaffung eines `CE`'s beschäftigen. Zunächst soll mal ein einfacher Fall durchgespielt werden und das `CE` soll schlicht den Puffer ausgeben, dessen Pointer von draussen reingereicht wurde. Dazu benötigt das `CE` keinen eigenen Pufferspeicher und kann folglich mit der Puffergrösse Null vom System angefordert werden.

```
      wce_A = rt_fetch_ce( 0 ) ;
```

Soweit kein Problem, denn der Pointer `wce_A` enthält immer einen gültigen Zeiger auf ein `CE`, wenn die Funktion aus dem Betriebssystem zurückkehrt. Üblicherweise geht das ganz fix, aber Sie entsinnen sich der grundsätzlichen Ausführungen über `CE`'s und Kontigentüberwachung des `CE`–Speicherplatzes einer Task? Wenn Sie sich ganz dumm stellen und in einer Schleife zehntausendmal ein `CE` vom Betriebssystem anfordern, das die Puffergrösse Null besitzt, ohne zwischendurch auch wieder `CE`'s freizugeben, dann greift ein Sicherheitsmechanismus, der verhindern soll, dass eine einzelne Task sich über Gebühr mit Speicherplatz vollsaugt. Die Task wird beim Überschreiten einer magischen Grenze eisenhart blockiert, bis wieder `CE`'s freigegeben werden und das Kontingent der Task für weitere Anforderungen unterschritten wird. Also achten Sie darauf, dass hier für Task die geniale Möglichkeit besteht, sich selbst zu blockieren, wenn sie zu gierig (und gedankenlos) `CE`'s zu hamstern versucht...

Wenn das `CE` vom Betriebssystem angeliefert wurde — respektive der Pointer auf ein solches *Communication Element* — gilt es zunächst, den späteren Empfänger einzutragen. Zu diesem Zwecke werden die gerade ermittelten Informationen über die anzusprechende Schnittstelle ins `CE` geschaufelt.

```
      wce_A->ldn   = port_info_A.ldn   ;
      wce_A->drive = port_info_A.drive ;
      strcpy( wce_A->file_name, "WRITE_A\xFF" ) ;
```

Bei der Gelegenheit sollte man dem `CE` auch gleich einen Namen verpassen. Wie gesagt: für serielle Schnittstellen macht das keinen direkten Sinn, weil der Treiber den Namen sowieso ignoriert, aber

die Zeit sollte schon übrig sein, um das CE mit einem *sprechenden* Namen zu versehen. Die vorgestellte Variante hat sich bei mir eingebürgert, um kurze CE–Namen einzutragen. Vergessen Sie **nie** das abschliessende 0xFF bzw. ein anderes Byte mit gesetztem höchsten Bit an den eigentlichen Namen anzuhängen. RTOS–UH hat nunmal eine andere Vorstellung davon, wie ein Terminator–Zeichen auszusehen hat. Das 0x00, das der strcpy() zusätzlich anhängt, wäre alleine nicht ausreichend, um den Namen RTOS–UH–konform zu terminieren. Der Dummy–Name sollte deshalb 23 Zeichen (einschliesslich des RTOS–UH–Terminators) nicht überschreiten, um bei der Verwendung von strcpy() nicht nachfolgende Speicherketten zu vernichten — ein netter Fehler, den man wochenlang mit wachsender Begeisterung zu suchen pflegt...

Wenn Sie keinen CE–Namen einsetzen, dann ist das auch egal. Bei der Beschaffung des CE's wird bereits der Dummy–Name '-' eingetragen.

Jetzt geht es an die Beschreibung, was das CE denn letztlich tun soll. Bei seriellen Schnittstellen ist das relativ simpel. Man kann lesen oder schreiben! Dazu ist der Parameter mode im CE entsprechend zu besetzen.

Die zugehörigen Makros sind den Tabellen 15.7, 15.8 und 15.9 bzw. der Includedatei <rtos.h> zu entnehmen. Das mode–Wort besitzt eine Dreiteilung in Bezug auf die Funktionalität der Bits.

Makro	Wert	Bedeutung
MODMWA	0x8000	Wait until return
MODMOU	0x4000	Output direction
MODMCR	0x2000	Auto-stop after CR
MODMLF	0x1000	Auto-stop after LF
MODMEO	0x0800	Auto-stop after EOT
MODMSC	0x0400	Suppress command
MODMNE	0x0200	No echo on input
MODBIN	0x0100	binary transfer

Tabelle 15.7: Steuerbedingungen im mode–Byte eines CE's

Die acht oberen Bits des mode–Wortes bestimmen das I/O–Verhalten des Gerätetreibers, der ein solches CE empfängt. Die wichtigsten Bits sind hierbei MODMOU — legt die I/O–Richtung fest — und MODMWA, das darüber entscheidet, ob die anfordernde Task auf die Beendigung des des I/O–Vorganges warten soll. Bei MODMCR, MODMLF und MODMEO handelt es sich um die Abbruchbedingungen, die ein Gerätetreiber bei der Interpretation eines Ein/Ausgabe–Datensatzes zu berücksichtigen hat.

IOCEF	0x0080	EOF-Bit bei Eingabe
IOCNE	0x0040	No error messages
EXCLU	0x0020	Exclusive access

Tabelle 15.8: Sonderkonditionen im mode–Byte eines CE's

Die drei Bits $xxEx legen gewisse Sonderkonditionen fest.

Die fünf niederwertigsten Bits $xx1F sind zur Kodierung des eigentlichen Kommandos bestimmt.

```
wce_A->mode  =  MODMWA   // Auf Beendigung der Ausgabe warten
             |  IOCNE    // Fehlermeldungen ins CE schreiben
             |  MODMOU   // Output einstellen
             |  IOCRWI ; // Befehl: READ-WRITE
```

Wie Ihnen nicht entgangen sein wird, geschah hier noch so manch andere Sache. Das Kommando für *Lesen/Schreiben* wird durch das Bit IOCRWI gesteuert. Die Richtung, in der die Operation erfolgen soll, wird durch das Bit MODMOU gesteuert. Ist MODMOU gesetzt, so wird ausgegeben — ansonsten handelt es sich eben um eine Leseoperation.

IOCRW	0x0000	READ/WRITE OLD
IOCER	0x0001	ERASE
IOCRTN	0x0004	RETURN
IOCCLO	0x0006	CLOSE
IOCRWI	0x0007	READ/WRITE ANY
IOCREW	0x0008	REWIND OLD
IOCAP	0x0009	APPEND FILE
IOCFL	0x000C	FILES
IOCFRE	0x000D	FREE
IOCDI	0x000E	DIR
IOCSYN	0x0010	SYNC
IOCTOU	0x0011	TOUCH
IOCLNK	0x0012	LINK
IOCSEK	0x0013	SEEK
IOCSAV	0x0014	SAVE POINTER
IOCRWS	0x0015	REWIND ANY
IOCRWN	0x0016	REWIND NEW
IOCFOS	0x0017	FORMAT SINGLE
IOCFOD	0x0018	FORMAT DOUBLE
IOCCF	0x0019	CF
IOCMDI	0x001A	MKDIR
IOCRDI	0x001B	RMDIR
IOCREN	0x001C	RENAME
IOBLCK	0x001F	R/W BLOCK

Tabelle 15.9: Kommandos im `mode`–Byte eines CE's

Aber dem `CE` werden noch zwei weitere Bits mit auf den Weg gegeben. Wird der angesprochene Treiber mit einem `CE` belästigt, das das Bit `MODMWA` gesetzt hat, so bedeutet dies, dass RTOS–UH die aufrufende Task solange blockiert, bis die Operation vom Treiber durchgeführt wurde — was in diesem Falle bedeutet, dass der Auftrag, Zeichen über `/A2` auszugeben, abgeschlossen wurde. Weiterhin wurde dem Treiber mittels `IOCNE` gesagt, dass eventuell auftretende Fehler bei der Operation nicht in Form von dummen Sprüchen auf dem Terminal auszugeben sind, sondern nur still und leise im `CE` zu erfolgen haben.

Achtung: Die Verwendung von `IONCE` (soll auf deutsch heissen: `No Error Messages!`) ist unter RTOS–UH arg gewöhnungsbedürftig. Sie können wählen, ob ein Spruch auf dem Bildschirm erscheint oder ob ein entsprechender Eintrag im `CE` erfolgen soll! Das Motto lautet: **Entweder — Oder!** und nicht *Sowohl — als auch!*, wie man es eigentlich als Programmierer erwartet hätte. Aber da es nunmal so von RTOS–UH festgelegt ist, betrachten Sie es bitte wohlwollend als Feature und nicht als Bug...

In diesem Falle soll es uns egal sein, ob die Ausgabe klappt oder voll an die Backe geht und deshalb ist dieses Bit schlicht gesetzt, um auch den Nutzer am Terminal von schlechten Nachrichten zu verschonen.

Jetzt kommt die nächste Übung, um das `CE` nicht zum Selbstzweck verkommen zu lassen. Nutzdaten sind eine feine Angelegenheit und das `CE` davon zu informieren, wo es diese Werte herbekommt und wieviele es denn sein mögen, ist Ihre nächste Aufgabe:

```
      wce_A->buffer =         str   ; // Puf-
fer auf den String setzen
      wce_A->reclen = strlen( str ) ; // und dessen Laenge eintragen
```

Bei ASCII–Daten ist das kein Problem. Die Länge kann mit einem schlichten `strlen()` ermittelt werden. Der Pufferzeiger des `CE`'s wird zudem schlicht auf den Eingabestring gesetzt. Das war's dann

auch schon. Jetzt kann das CE auf die Reise geschickt werden.

```
      rt_transfer_ce( wce_A ) ;
```

Jetzt bekommt das Betriebssystem das CE durchgereicht, stellt fest, dass das MODMWA–Bit gesetzt ist und blockiert die aufrufende Task. Anschliessend wird versucht, einen Treiber ausfindig zu machen, der sich um das CE zu kümmern hat. Ist die LDN, an die das CE geschickt wurde, nicht betreut, gibt es eine Fehlermeldung; die aufrufende Task wird fortgesetzt und bekommt die Niederlage mitgeteilt.

Ansonsten bekommt der Treiber für die angebene LDN das CE übergeben, treibert so ein wenig vor sich hin und teilt (wegen des gesetzten IOCNE–Bits) seine Meinung über die Qualität des CE's in dem Eintrag reclen mit. Werte kleiner oder gleich Null sind Fehlermeldungen. Bei Erfolg, bleibt der alte Eintrag in reclen schlicht erhalten.

Gut, irgendwann (entsprechend der Priorität des CE's und der Belastung der Schnittstelle und des Rechners allgemein) ist der Auftrag (positiv oder negativ) ausgeführt. Der Treiber ist in diesem Falle Kumpel und setzt den Aufrufer wieder fort.

Wenn es Sie interessiert, ob die Ausgabe von Erfolg gekrönt war, sollten Sie in reclen nachschauen, ob dort ein Wert kleiner oder gleich Null steht, was eine Niederlage bedeutet.

Hier soll das mal egal sein, weil Beispiel immer nur das zeigen, was man im realen Betrieb tunlichst unterlassen sollte. Die Ausgabe ist also abgeschlossen und nun haben wir noch so ein lästiges CE am Hals, das uns nicht mehr interessiert. Also schnell weg damit und dem Aufrufer den (zweifelhaften) Erfolg melden:

```
      rt_release_ce( wce_A ) ;
      return( 1 ) ;
   }
}
```

15.5.3.1.3 Ein Output–CE an die Duplex–Schnittstelle schicken: Das erste Beispiel war bewusst trivial behalten und sollte nur die Möglichkeit demonstrieren, ASCII–Daten über eine serielle Schnittstelle auszugeben. Oft besteht aber die Notwendigkeit, über eine physikalische Schnittstelle sowohl Daten zu senden, als auch zu empfangen — und das möglichst auch noch gleichzeitig! Zu diesem Zwecke wurde unter RTOS–UH die Möglichkeit geschaffen, Duplex–Verbindungen aufzubauen. Die Ausgabe einer Duplexverbindung über die Schnittstelle *x* hat über das Gerät /D*x* zu erfolgen. Gleichzeitig können konfliktfrei Eingabeaufforderungen über einen der Kanäle /A*x*, /B*x* oder /C*x* an die gleiche Schnittstelle gesendet werden.

Die Ausgabe über einen D–Kanal unterscheidet sich nicht von der Ausgabe über den A–Kanal. Um das folgende Beispiel nicht zu trivial werden zu lassen, sollen gleichzeitig andere Möglichkeiten von RTOS–UH–Treibern demonstriert werden.

```
int D2_Ausgabe( char  *str )
{
   auto     RFILE   port_info_D ;
   register Ce     *wce_D       ;

   if ( ! rt_get_filedata( &port_info_D, "/D2/D2_Ausgabe" )
      return( 0 ) ;
   else
   {
```

Bis hierher noch kein Unterschied. Aber jetzt soll ein CE mit internem Puffer angefordert werden, der

den String aufnehmen kann. Sinn der Übung soll sein, den String auszugeben, ohne auf die Beendigung der Ausgabe warten zu müssen.

```
wce_D = rt_fetch_ce( ( ( strlen( str ) + 1 ) & ~1L ) + 1 ) ;
```

So, und das war gleich noch etwas mehr. Hier wurde die Länge des Strings ermittelt und auf die nächsthöhere ungerade Zahl aufgerundet, um die Kontingentüberwachung auszuschalten. Aber dafür geht es jetzt weiter, wie schon bekannt:

```
wce_D->ldn   = port_info_D.ldn   ;
wce_D->drive = port_info_D.drive ;
strcpy( wce_D->file_name, "WRITE_D\xFF" ) ;
```

Jetzt stimmt der Adressat! Beim Übertragungsmodus kommen jetzt aber wieder ein paar kleine Änderungen, um die Sache spannender zu gestalten:

```
wce_D->mode  =  MODMOU   // Output einstellen
             |  IOCRWI ; // Befehl: READ-WRITE
```

Diesmal wird das `CE` als pures Ausgabe–`CE` auf die Reise geschickt. Das Fehlen der Bits `MODMWA` und `IOCNE` bewirkt zweierlei:

1. Ohne `MODMWA`–Bit hält das Betriebssystem die Task, die das `CE` absendet, beim Empfang des `CE`'s nicht an. Die Task läuft schlicht weiter, wenn es ihre Priorität erlaubt. Währendessen kann RTOS–UH das `CE` an den Treiber weiterreichen und der sich mit dem Teil vergnügen...
2. Das Fehlen des `IOCNE`–Bits bewirkt, dass Fehler bei der Behandlung des `CE`'s sich in Bildschirmausgaben äussern — Sie kennen das: *Bing! Something is wrong with your program!*

Auch das Aufsetzen des Puffers erfolgt nun anders. Hier muss nicht der Pointer des `CE`'s auf einen externen Puffer gesetzt, sondern vielmehr der externe Puffer in den `CE`–Puffer kopiert werden.

```
memcpy( wce_D->buffer, str, strlen( str ) ) ;
wce_D->reclen = strlen( str ) ;
```

Das war's aber nun wirklich — oder wenigstens beinahe:

```
wce_D->status_of_io |= STABRE ;
```

Da wir uns schon entschlossen haben, auf die Beendigung der Ausgabe nicht zu warten und das `CE` mangels `IOCNE`–Bit auch keine Fehlermeldungen zu liefern bereit ist, können wir den Treiber auch damit beauftragen, das `CE` nach Beendigung der Ausgabe selbst zu verschrotten und uns nicht mit dieser Aufgabe zu belästigen. Das `STABRE`–Bit bewirkt exakt dieses Verhalten.

Makro	Wert	Bedeutung
STABRE	0x02	Send and forget
STABFL	0x80	User Bit

Tabelle 15.10: Bitmuster für das `status`–Byte eines CE's

Wenn Sie jetzt scharf nachdenken, werden Sie zweifelsohne selbst darauf kommen, dass dieses Bit bei Eingabe–`CE`'s nur von zweifelhaftem Nutzen ist. Im Eintrag `status_of_io` besteht zudem noch die Möglichkeit, ein einzelnes Bit `STABFL` frei zu verwenden, das innerhalb von RTOS–UH keinerlei Berücksichtigung findet.

Und jetzt noch das `CE` abschicken und vergessen:

```
rt_transfer_ce( wce_D ) ;
```

```
      return( 1 ) ;
   }
}
```

15.5.3.1.4 Ein Eingabe–CE aufbereiten und wegschicken: Bei Eingaben gibt es drei wesentliche Möglichkeiten, dem Treiber Daten zu entlocken.

1. Sie wollen die Daten ab dem Augenblick haben, an dem Sie ihre Eingabe ans System schicken ⇒ lesen vom A–Kanal.
2. Sie wollen auch Daten lesen, die seit dem letzten Leseauftrag eingelaufen sind ⇒ lesen vom B–Kanal.
3. Sie wollen die Daten haben, die aktuell schon empfangen wurden und Sie wollen nicht blockiert werden, wenn nicht alle Daten da sind, die Sie sich gewünscht hätten ⇒ lesen vom C–Kanal.

Sie sollten alle Beispiele durchlesen und möglichst begreifen, wenn Sie sich ernsthaft vorgenommen haben, unter RTOS–UH I/O–Operationen erfolgreich durchzuführen.

15.5.3.1.4.1 Eine Eingabe vom A–Port: Fangen wir mit einer einfachen Lese–Operation vom A–Port an. Die Schnittstelle sei diesmal /A1.

```
int A1_Eingabe( char  *str )
{
   auto  RFILE     port_info_A ;
   register Ce    *rce_A       ;
   register int    retval      ;

   rt_get_filedata( &port_info_A, "/A1/A_Port" ) ;

   rce_A = rt_fetch_ce( 1 ) ;

   rce_A->ldn   = port_info_A.ldn   ;
   rce_A->drive = port_info_A.drive ;
   strcpy( rce_A->file_name, "READ_A\xFF" ) ;
```

Bis hierher dürfte es sich inzwischen um eine vertraute Übung handeln. Die Überprüfung, ob rt_get_filedata die Schnittstelle A1 als fehlerhaft einstufen könnte, sei diesmal als rundant betrachtet. Diesmal wurde ein CE ohne eigenen Puffer und ohne Kontigentüberwachung angefordert, weil beabsichtigt ist, den in der Funktion übergebenen Puffer str direkt zu füllen. Achtung: ein beliebter Anfängerfehler besteht darin, zwar einen Pointer zu übergeben, aber keinen Pointer auf ausreichend grossen Speicher oder gar gar mit undefinierten Pointern zu arbeiten! Es kommt reichlich unkomisch, ein Eingabe–CE dazu zu missbrauchen, eine nicht näher definierte Speicherstelle zu patchen oder Feldüberläufe vom Nukleus realisieren zu lassen...

Jetzt wird der Modus des CE's gesetzt:

```
   rce_A->mode   =  MODMWA   // Auf Beendigung des I/O-
Vorgangs warten
                 |  IOCNE    // Fehlermeldungen in das CE schreiben.
                 |  IOCRWI ; // Befehl: READ-WRITE
   rce_A->reclen =     5   ; // Hier die Laenge eintragen
   rce_A->buffer =    str  ; // Und mit unserem Argument arbeiten
```

Das Bit IOCRWI für *Lesen/Schreiben* ist immer noch identisch. Das Fehlen des Bits MODMOU selektiert als Richtung der Operation den Input. Der Wunsch, Fehlermeldungen im CE zu erhalten — durch IOCNE angegeben —, ist bei Leseoperationen durchaus **sehr** sinnvoll. Im Beispiel soll nach dem Abschicken des Leseauftrages auf dessen Beendigung gewartet werden — gesteuert über MODMWA. Im Beispiel sollen 5 Zeichen gelesen werden. Jetzt kann das CE auf die Reise geschickt werden:

```
rt_transfer_ce( rce_A ) ;
```

Das CE landet beim Treiber für die Schnittstelle /A1 und die aufrufende Task wird einstweilen blockiert. Wenn der Treiber das CE aus seiner CE–Kette holt und feststellt, dass es sich um eine Anforderung an die Betriebsart A–Port handelt, so wird zunächst der gesamte interne Empfangspuffer der Schnittstelle /A1 gelöscht — alle bislang empfangenen Zeichen sind dann futsch. Anschliessend werden die nächsten 5 einlaufenden Zeichen brav an die Stelle geschrieben, auf die unser CE mit dem Parameter buffer verweist. Solange nicht auch das fünfte Zeichen empfangen wurde, verharrt die aufrufende Task brav im Zustand I/O?. Gesetzt den Fall, dass tatsächlich 5 Zeichen empfangen wurden — oder vom Treiber eine Fehlerbedingung erkannt wurde —, so kehrt das CE zum Aufrufer zurück. Die erste Aktion sollte stets darin bestehen, die Fehlerzelle in reclen abzufragen. Wenn Sie das unterlassen, lesen Sie mit Gewissheit im realen Betrieb eines schönen Tages Schrott — gerade, weil den hundert simulierten Tests nie die Notwendigkeit bestanden haben mag, auf Fehler zu reagiern. Betrachten Sie es als erwiesene Tatsache, dass der Verzicht auf die Abprüfung von Fehlerkonditionen bei I/O–Operationen mit absolut tödlicher Sicherheit zum GAU ihrer Algorithmen im laufenden Betrieb führen wird...

```
if ( rce_A->reclen <= 0 )
{  /* FEHLER */
   retval = 0 ;
}
```

Was Sie im Fehlerfall anzustellen gedenken, bleibt Ihre Sache. Aber tun sollten Sie definitiv etwas, was der Situation angemessen ist. In diesem Falle besteht die Reaktion schlicht darin, der aufrufenden Funktion mittels des Rückgabewerts 0 mitzuteilen, dass da gerade etwas schiefgegangen ist und der runtergereichte Puffer nicht mit 5 Zeichen gefüllt werden konnte.

```
   else
   {
      // Puffer wurde korrekt gefuellt
      retval = 1 ;
   }
   rt_release_ce( rce_A ) ; // Und das CE wieder verschrotten.

   return( retval ) ;
}
```

In jedem Fall sollten Sie vor dem Verlassen der Funktion das angeforderte CE wieder freigeben — möglichst erst, nachdem Fehlerstatus und Nutzdaten ausgelesen wurden, weil die Programme sonst dazu neigen, stochastische Resultate zu erbringen oder gar abzustürzen. Aber warum erzähle ich das: Sie werden diesen Fehler mit absoluter Sicherheit begehen (irgendwann und irgendwo) und dann beginnt zwangsläufig der Kampf mit dem prioritätengesteuerten Multitasking, um die Stelle ausfindig zu machen, an der die Bombe gelegt wurde. Eine Abhilfe schafft die Methode, **jeden** CE–Zeiger, definitiv nach einem rt_release_ce() ungültig zu machen — soll heissen: auf eine Stelle verweisen zu lassen, die garantiert einen BUS-ERROR auslöst, wenn ein Zugriff erfolgt. In diesem Falle wäre rce_A nach dem Release z.B. auf rce_A = 0xAFFEDEAD ; zu setzen, um garantiert herauszubekommen, wenn irgendein Schlingel über den *toten Affen* noch Zugriffe versuchen sollte.

15.5.3.1.4.2 Eine Eingabe vom B–Port: Der B–Port unterscheidet sich in einer wichtigen Eigenschaft vom A–Port: wenn ein CE zu B–Port geschickt wird, dann wird nicht erst der interne Empfangspuffer gelöscht, sondern zunächst die bereits gelesenen Zeichen ins CE umkopiert. Der B–Port stellt somit am ehesten das dar, was unter anderen Betriebssystemen als Eingabeverhalten bekannt ist: die gepufferte Eingabe!

Der A–Port kommt zwar im Alphabet zuerst, aber für den Zyklus *Einlesen–Verarbeiten–Einlesen* ist er meistens ungeeignet, weil der Prozess der Verarbeitung der gelesenen Zeichen niemals in Nullzeit erfolgt und somit bis zum nächsten Aufsetzen eines Lese–CE's 0 bis n Zeichen verloren gehen können.

```
#pragma HEADER  "SerialExample=1.0"

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <ctype.h>

#define MY_BUFFER_SIZE   (1024)

#define PORT_NR  "1"  // Mit den Ports /A1, /B1, /C1, /D1 spielen

/**********/
/*  main  */
/**********/

   void

main( void )

{
   auto  RFILE     port_info_A, port_info_B    ;
   auto  RFILE     port_info_C, port_info_D    ;
   register Ce    *wce_A, *wce_D               ;
   register Ce    *rce_A, *rce_B, *rce_C       ;
   auto     char   my_buffer[ MY_BUFFER_SIZE ] ;
   register char  *ptr                         ;
   register int    count                       ;

   /*
    *    LDN und DRIVE fuer die Ports holen
    */
   rt_get_filedata( &port_info_A, "/A" PORT_NR "/A_Port" ) ;
   rt_get_filedata( &port_info_B, "/B" PORT_NR "/B_Port" ) ;
   rt_get_filedata( &port_info_C, "/C" PORT_NR "/C_Port" ) ;
   rt_get_filedata( &port_info_D, "/D" PORT_NR "/D_Port" ) ;

   printf( "A" PORT_NR ": LDN=%3d DRV=%3d\n",
           (int)port_info_A.ldn, (int)port_info_A.drive ) ;
   printf( "B" PORT_NR ": LDN=%3d DRV=%3d\n",
           (int)port_info_B.ldn, (int)port_info_B.drive ) ;
   printf( "C" PORT_NR ": LDN=%3d DRV=%3d\n",
           (int)port_info_C.ldn, (int)port_info_C.drive ) ;
   printf( "D" PORT_NR ": LDN=%3d DRV=%3d\n",
           (int)port_info_D.ldn, (int)port_info_D.drive ) ;

   /*
    * Schreib-CE fuer A-Port aufsetzen
    * z.B. mit einer Puffergroesse von 512 Bytes
```

```
    * und einem kleinen Ausgabetext.
    */
   wce_A = rt_fetch_ce( 512 ) ; // Bleibt liegen, bis das CE da ist
                                // Bei Ueberlauf des CE-Kontingents
                                // wird die aufrufende Task blockiert
   wce_A->ldn   = port_info_A.ldn   ;
   wce_A->drive = port_info_A.drive ;
   wce_A->mode  =  MODMWA   // Auf Beendigung der Ausgabe warten
                |  MODMOU   // Output einstellen
                |  IOCNE    // Keine Fehlermeldungen aufs Terminal
                |  IOCRWI ; // Befehl: READ-WRITE
   strcpy( wce_A->file_name, "WRITE_A\xFF" ) ; // Einen CE-Namen eintragen
   strcpy( wce_A->buffer,  "A-Text" ) ; // Einen Ausgabetext in den
   wce_A->reclen = strlen( "A-Text" ) ; // Puffer schreiben und
                                        // dessen Laenge im CE
                                        // vermerken...

   rt_transfer_ce( wce_A ) ; // CE an den Treiber schicken.
                             // Die Funktion kehrt zurueck,
                             // wenn das CE verarbeitet wurde,
                             // soll heissen: wenn der Treiber
                             // alle Zeichen abgeschickt hat.

   rt_release_ce( wce_A ) ;  // Und das CE wieder verschrotten.
                             // Danach ist der Pointer 'wce_A'
                             // nicht mehr zu verwenden.


   /*
    * Schreib-CE fuer (D)uplex-Port aufsetzen. Damit ist es z.B.
    *  moeglich, ueber /A1, /B1 oder /C1 Daten zu lesen und
    * gleichzeitig ueber /D1 Daten zu senden --- ueber eine
    * physikalische Verbindung!
    * Hier ein Sendepuffer von 128 Bytes und ein kleiner
    * Ausgabetext.
    */
   wce_D = rt_fetch_ce( 128+1 ) ;  // Bleibt liegen, bis das CE da ist.
                                   // Durch die Angabe der ungera-
den Groesse
                                   // erfolgt *KEINE* CE-Kontingentueber-
                                   // wachung. Achtung: Damit kann man sich
                                   // bei Fehlern das Sy-
stem bis aufs letzte
                                   // Byte zumuellen...
   wce_D->ldn   = port_info_D.ldn   ;
   wce_D->drive = port_info_D.drive ;
   wce_D->mode  = MODMOU           // Wieder auf Output einstellen
                | IOCRWI ;         // Befehl: READ-WRITE
                                   // Aber diesmal ohne MODMWA (also nicht
                                   // auf die Beendi-
gung des Schreibvorgangs
                                   // warten und ohne IOCNE (also Fehler-
                                   // meldungen aufs Terminal schreiben.

   strcpy( wce_D->file_name, "WRITE_D\xFF" ) ;
   strcpy( wce_D->buffer,  "D-Text" ) ; // Namen und Laenge wie gehabt...
   wce_D->reclen = strlen( "D-Text" ) ;
   wce_D->status_of_io |= STABRE ; // Der Treiber soll das CE nach Gebrauch
                                   // verschrot-
```

```
ten und nicht an den Aufrufer
                                  // zurueckliefern...

   rt_transfer_ce( wce_D ) ; // CE an den Treiber schicken. Die Funktion
                             // kehrt sofort zurueck, wenn das CE vom
                             // Betriebssystem angenommen wurde...

   /*-------------------------------------------------------------*/
   /* Waehrend diese Stelle hier erreicht ist, arbeitet RTOS im */
   /* Hintergrund weiter und gibt das CE *irgendwann* auch      */
   /* ueber die Schnittstelle aus. Das Anwenderprogramm kann    */
   /* waehrend dieser Zeit anderen Aufgaben nachgehen.          */
   /*-------------------------------------------------------------*/

   /*
    * Lese-CE fuer A-Port aufsetzen.
    * Hier mit einem Puffer von 5 Byte.
    */
   rce_A = rt_fetch_ce( 6 ) ; // Achtung: 5 waere falsch, weil das
                              //          unterste Bit nur zu Ein-
                              //          stellung der Kontingentueber-
                              //          wachung dient...
   rce_A->ldn   = port_info_A.ldn   ;
   rce_A->drive = port_info_A.drive ;
   rce_A->mode  =  MODMWA     // Auf Beendigung des I/O-Vorgangs warten
                |  IOCNE      // Fehlermeldungen im CE liefern.
                |  IOCRWI ;   // Befehl: READ-WRITE
   strcpy( rce_A->file_name, "READ_A\xFF" ) ;
   rce_A->reclen = 5 ; // Hier die echte Laenge eintragen

   rt_transfer_ce( rce_A ) ; // CE an den Treiber schicken. Der Treiber
                             // loescht alle bis dahin eingelaufenen
                             // Zeichen in seinem Puffer (A-Betrieb) und
                             // fuellt den Puffer mit den naechsten 5
                             // Zeichen...

   /*
    * Wenn man hier angekommen ist, ist der Puffer entweder
    * gefuellt oder ein Fehler aufgetreten...
    */
   if ( rce_A->reclen <= 0 )
   {
      /* FEHLER */
   }
   else
   {
      /*
       * Hier stehen die Daten im Puffer!
       */
      ptr = rce_A->buffer ; // Pointer auf Puffer

      for ( count=0 ; count<rce_A->reclen ; ++count )
         printf( "%d.Zeichen: %08lX\n", count+1, ptr[count] & 0xFFUL) ;
   }

   rt_release_ce( rce_A ) ; // Und das CE wieder verschrotten.


   /*
```

```
 *    Lese-CE fuer B-Port aufsetzen.
 *    Hier mit einem Puffer von 256 Byte.
 */
rce_B        = rt_fetch_ce( 256 ) ;
rce_B->ldn   = port_info_B.ldn    ;
rce_B->drive = port_info_B.drive  ;
rce_B->mode  =  MODMWA    // Auf Beendigung des I/O-Vorgangs warten
             |  MODBIN    // Schnittstelle im binaeren Betrieb fahren
             |  IOCNE     // Fehlermeldungen im CE liefern.
             |  IOCRWI ;  // Befehl: READ-WRITE
strcpy( rce_B->file_name, "READ_B\xFF" ) ;
rce_B->reclen = 256 ;

rt_transfer_ce( rce_B ) ; // CE an den Treiber schicken. Der Treiber
                          // liefert zunaechst aus seinem internen
                          // Puffer Zeichen, bis die Laenge des CE's
                          // erfuellt ist. Wenn Zeichen fehlen, wird
                          // auf weitere Zeichen gewartet.

/*
 *    Wenn man hier angekommen ist, ist der Puffer entweder
 *    mit der gewünschten Anzahl von Zeichen gefuellt oder
 *    es ist ein Fehler aufgetreten...
 */
if ( rce_B->reclen <= 0 )
{
   /* FEHLER */
}
else
{
   /*
    *    Hier stehen die Daten im Puffer!
    */
   ptr = rce_B->buffer ; // Pointer auf Puffer

   for ( count=0 ; count<rce_B->reclen ; ++count )
      printf( "%d.Zeichen: %08lX\n", count+1, ptr[count] & 0xFFUL) ;
}

rt_release_ce( rce_B ) ; // Und das CE wieder verschrotten.


/*
 *    Lese-CE fuer C-Port aufsetzen.
 *    Hier mit einem externen Puffer von 1024 Byte.
 *
 */
rce_C        = rt_fetch_ce( 1 )  ; // Das CE *IMMER* beschaffen
rce_C->ldn   = port_info_C.ldn   ;
rce_C->drive = port_info_C.drive ;
rce_C->mode  =  MODMWA    // Auf Beendigung des I/O-Vorgangs warten
             |  MODBIN    // Schnittstelle im binaeren Betrieb fahren
             |  IOCNE     // Fehlermeldungen im CE liefern.
             |  IOCRWI ;  // Befehl: READ-WRITE
strcpy( rce_C->file_name, "READ_C\xFF" ) ;
rce_C->reclen = MY_BUFFER_SIZE ;
rce_C->buffer =   my_buffer    ; // Hier den eigenen Puffer einklinken

rt_transfer_ce( rce_C ) ; // CE an den Treiber schicken. Der Treiber
```

```
                          // liefert zunaechst aus seinem internen
                          // Puffer Zeichen, bis der interne Puffer
                          // leergelesen ist. Zeichen, die waehrend
                          // dieser Operation noch hereinkommen,
                          // werden ebenfalls an das CE weitergegeben.

/*
 *    Wenn man hier angekommen ist, ist der Puffer entweder
 *    gefuellt oder ein Fehler aufgetreten...
 */
if ( rce_C->reclen <= 0 )
{
   /* FEHLER */
}
else
{
   /*
    *    Hier stehen die Daten im Puffer!
    *    Aus 'reclen' kann man entnehmen, wieviele Bytes es sind.
    *    Bei 'reclen==1' und 'buffer[0]==0' ist kein Zeichen
    *    eingelaufen...
    */
   ptr = rce_C->buffer ; // Pointer auf Puffer oder auch:
                         //   ptr = my_buffer ;

   for ( count=0 ; count<rce_C->reclen ; ++count )
      printf( "%d.Zeichen: %08lX\n", count+1, ptr[count] & 0xFFUL) ;
}

rt_release_ce( rce_C ) ; // Und das CE wieder verschrotten.


/*
 *    Und jetzt mit dem Timeout rumspielen:
 *
 *       Es sollen 50 Zeichen vom A-Port gelesen werden.
 *       Wenn nach 5 Sekunden die Zeichen noch nicht da
 *       sind, soll ein Fehlerabbruch stattfinden...
 */

/*
 *    Lese-CE fuer A-Port aufsetzen.
 *    Hier mit einem Puffer von 50 Byte.
 *
 */
rce_A        = rt_fetch_ce( 50 ) ;
rce_A->ldn   = port_info_A.ldn   ;

/*
 *    Das Timeout wird im oberen Byte des 'drive'-Wortes abgelegt. Die
 *    Zeit wird in Vielfachen von 512ms angegeben. Um RTOS ueber diese
 *    Betriebsart zu informieren, muss das oberste Bit des Wortes
 *    angeschaltet werden.
 */
rce_A->drive = port_info_A.drive
             | ( (5000/512) | 0x80 ) << 8 ; // ca. 5000 ms

rce_A->mode  =  MODMWA    // Auf Beendigung des I/O-Vorganges warten
             |  IOCNE     // Fehlermeldungen im CE liefern.
```

```
             |  IOCRWI ;  // Befehl: READ-WRITE
strcpy( rce_A->file_name, "READ_A_TIMEOUT\xFF" ) ;
rce_A->reclen = 50 ;      // Hier Laenge eintragen

rt_transfer_ce( rce_A ) ; // CE an den Treiber schicken.

/*
 *    Wenn man hier angekommen ist, ist der Puffer entweder
 *    gefuellt oder ein Fehler aufgetreten...
 */
if ( rce_A->reclen <= 0 )
{
   /*    FEHLER
    *
    *    Wenn beim Ablaufen des Timeout's 49 der 50 Zeichen eingelaufen
    *    sind, dann werden Sie das nie erfahren --- die sind jetzt
    *    schlicht alle futsch...
    */
}
else
{
   /*
    * Hier stehen die 50 Datenbytes im Puffer!
    */
   ptr = rce_A->buffer ; /* Pointer auf Puffer */

   for ( count=0 ; count<rce_A->reclen ; ++count )
      printf( "%d.Zeichen: %08lX\n", count+1, ptr[count] & 0xFFUL) ;
}

/*
 * Und jetzt mit dem noch existierenden CE weiterspielen
 * und ein paar Abbruchbedingungen ausprobieren...
 *
 * Das Lese-CE fuer den A-Port umkonfigurieren auf 25 Zeichen
 * (der angeforderte Puffer reicht schliesslich fuer 50 Zeichen
 * und weniger Zeichen sind nicht schaedlich)...
 */
rce_A->drive = port_info_A.drive ; // Timeout-Kennzeichnung
                                   // wegschalten...
rce_A->mode  =  MODMWA    // Auf Beendigung des I/O-Vorgangs warten
             |  MODMNE    // Echo bei der Eingabe abschalten
             |  MODMCR    // Bei Carriage Return abbrechen...
             |  MODMLF    // Bei Linefeed abbrechen
             |  MODMEO    // Bei End-Of-File (^D) abbrechen
             |  IOCNE     // Fehlermeldungen im CE liefern.
             |  IOCRWI ;  // Befehl: READ-WRITE
strcpy( rce_A->file_name, "READ_A_CONDITIONAL\xFF" ) ;
rce_A->reclen = 25 ; // Hier Laenge eintragen

rt_transfer_ce( rce_A ) ; // CE an den Treiber schicken.

/*
 *    Wenn man hier angekommen ist, ist der Puffer entweder
 *    gefuellt oder ein Fehler aufgetreten...
 */
if ( rce_A->reclen <= 0 )
{
   /* FEHLER */
```

```
}
else
{
   /*
    *    Hier stehen jetzt maximal 25 Datenbytes im Puffer!
    *    In 'reclen' steht die wahre Anzahl der gelesenen Daten.
    */
   ptr = rce_A->buffer ; // Pointer auf Puffer

   for ( count=0 ; count<rce_A->reclen ; ++count )
      printf( "%d.Zeichen: %08lX\n", count+1, ptr[count] & 0xFFUL) ;
}


/*
 *    Und mit diesem CE nochmal weiterspielen. Die Abbruchbedingungen
 *    sollen so bleiben und es sollen wieder 25 Zeichen gelesen werden.
 *    Aber diesmal soll so gelesen werden, dass nach dem Abschicken des
 *    Leseauftrags noch ein paar Dinge berechnet werden sollen (so eine
 *    CPU ist eben *viel* schneller als eine popelige Schnittstelle)...
 */
rce_A->reclen =    25   ; // Hier Laenge eintragen
rce_A->mode  &= ~MODMWA ; //  !! Nicht warten !!

rt_transfer_ce( rce_A ) ; // CE an den Treiber schicken.

/*
 *    Jetzt ist das CE unterwegs und wird vom Treiber
 *    betreut --- das Teil gehoert also nicht mehr diesem
 *    Programm und jeder Zugriff auf das CE und den
 *    Buffer des CE's ist streng untersagt.
 */
{
   /*
    * Als Beispiel pure Beschaeftigungstherapie....
    * Hier koennen auch sinnvolle Dinge passieren,
    * solange keiner am abgeschickten CE rumfummelt...
    */
   for ( count=0 ; count<10000 ; ++count )
      ;
}

/*
 * Und jetzt legen wir uns hin und schauen nach, ob der
 * Treiber mit unserem Lese-CE fertig ist.
 */
rt_wait_for_ce( rce_A ) ;

/*
 * Wenn man hier angekommen ist, ist der Puffer entweder
 * gefuellt oder ein Fehler aufgetreten...
 */
if ( rce_A->reclen <= 0 )
{
   /* FEHLER */
}
else
{
   /*
```

```
       * Hier stehen jetzt wieder maximal 25 Datenbytes im Puffer!
       * In 'reclen' ist die wahre Anzahl der gelesenen Daten eingetragen.
       */
      ptr = rce_A->buffer ; /* Pointer auf Puffer */

      for ( count=0 ; count<rce_A->reclen ; ++count )
         printf( "%d.Zeichen: %08lX\n", count+1, ptr[count] & 0xFFUL) ;
   }

   /*
    *    Genug gespielt und weg mit dem CE
    */
   rt_release_ce( rce_A ) ;

   exit( 0 ) ;
}
```

15.6 Speicherverwaltung

Der Umgang mit dynamischem Speicher zählt in C–Programmen zum alltäglichen Handwerk jedes Programmieres. Andere Programmiersprachen (z.B. PEARL) bieten diese Möglichkeit nicht. Als Begründung wird stets der Sicherheitsaspekt angeführt. Wenn ein PEARL–Programm geladen und gestartet werden konnte — so heisst es — kann es später nicht mehr aus Speicherplatzmangel liegenbleiben, während C–Programme eben dies nicht gewährleisten. Deshalb seien dynamische Allokationen gefährlich und nicht in sicherheitsrelevanten Bereichen einzusetzen. Und irgendwie haben die Leute damit auch recht. Dynamische Speicheranforderungen haben in Programmteilen nichts zu suchen, die **immer lauffähig** sein müssen. Alle Funktionen, die in diesem Kapitel abgehandelt werden, besitzen die eingebaute Option des Fehlschlags. Die Lauffähigkeit einer Steuerung oder Regelung von der Verfügbarkeit freien Systemspeichers abhängig zu machen, ist schlechter Programmierstil. Diese Tatsache muss Ihnen klar sein, bevor Sie anfangen, ihre Tasks zu kodieren. Es liegt in ihrer Verantwortung, wann es sinnvoll und vertretbar ist, dynamischen Speicher zu verwenden.

CREST–C bietet zwei Methoden an, mit denen sich der Programmierer dynamisch zur Laufzeit der Task Speicher beschaffen kann. Es stehen alle genormten Funktionen gemäss ANSI–C zur Verfügung. Weiterhin können — unter Verzicht auf portable Programmierung — auch direkt vom Betriebssystem Speicherblöcke angefordert werden.

15.6.1 Genormte Speicheranforderungen

Die ANSI–C–Norm stellt drei Funktionen zur Verfügung, die zur Anforderung dynamischem Speichers dienen. Die Prototypen sind in der Includedatei <stdlib.h> aufgeführt. Die Funktionen selbst sind Bestandteil der Standardbibliothek.

```
void  *malloc( size_t  size ) ;
void  *calloc( size_t  count, size_t  size ) ;
void  *realloc( void *ptr, size_t size ) ;
void   free( void *ptr ) ;
```

Zur Verwaltung der Speicherallokationen mittels der ANSI–C–Funktionen legt CREST–C für jede Subtask einen eigenständigen Ringpuffer an, in dem alle dynamischen Speichersegmente, die während der Lebensdauer einer Task angefordert wurden, verzeichnet sind. Beim Start der Task ist dieser Ring noch leer. Berücksichtigen Sie bitte bei der Verwendung dieser Funktionen, dass Aufrufe nur innerhalb

von C–Tasks und deren Ablegern zulässig sind. In Exception–Handlern, PEARL–Unterprogrammen und allen anderen Fällen, in denen der Aufrufer keine reguläre C–Task ist, haben diese Funktionen (und auch Bibliotheksroutinen, die sich darauf abstützen) nichts verloren, da ein Crash sonst unvermeidlich ist.

Wenn eine Speicheranforderung des Benutzers nicht mittels der freien Segmente im Freispeicherring befriedigt werden kann, so holt sich die aufgerufene Funktion Nachschlag vom RTOS–UH. Dabei hat es sich als praktikabel erwiesen, die Mindestgrösse der Allokationen vom RTOS–UH auf 1kB festzusetzen. Der Aufrufer erhält einen Block der angeforderten Grösse und der überschüssige Speicher wird dem Freispeicherring zugeführt. Bei grösseren Anforderungen wird exakt die verlangte Grösse vom System allokiert. Die Rückgabe des dynamischen Speichers mittels `free()` führt jedoch **niemals** zu einer Rückgabe des betreffenden Speichers an das Betriebssystem. Um es unmissverständlich zu formulieren:

Was Sie einmal mit den ANSI–C**–Allokationsroutinen unter** CREST–C **geholt haben, geht erst mit der Terminierung der anfordernden Task wieder in den Besitz des Betriebssystems über!**

Die aktuelle Grösse des Speicherrings lässt sich mittels der Funktion `rt_allocated_bytes()` überwachen, die die Zahl der allokierten Bytes der aufrufenden Task liefert.

In Hinblick auf Betriebssicherheit wurde die Speicherverwaltung so konzipiert, dass die üblichsten Fehlbedienungen wie Freigabe von `NULL`–Pointern und Doppelfreigabe von Blöcken nicht zu den allseitig geliebten Systemabschüssen führen können. Auch `free()`–Aufrufe mit Pointern auf nicht–allokierte Blöcke führen nicht unbedingt zu Speichersalat.

Die `free()`–Routine bricht bei derartigen Fällen ab, ohne wirre Spuren im System zu hinterlassen — die berühmten Ausnahmen bestätigen allerdings wie üblich die Regel. Wenn also Ihr Programm Sie mit der Meldung `free():_argument_is_NULL` beschimpft, dann haben Sie gerade versucht, einen `NULL`–Pointer in den Speicherring zurückzugeben.

Die lapidare Meldung `free():_argument_is_not_linked` ist eine Runde dramatischer. Sie deutet auf totalen Schwachsinn als Argument der Funktion `free()` hin. Hier sind einige unangenehme Fälle denkbar. Sie sollten zunächst überprüfen, ob der übergebene Pointer überhaupt einen sinnvollen Wert enthält. Ist grober Unfug ausgeschlossen, kann es sich um den Versuch einer erneuten Freigabe eines bereits freigegeben Blockes handeln. Beim Umgang mit mehreren Subtasks kann auch der Fall aufgetreten sein, dass eine Task Speicher einer anderen Task freizugeben versucht — was völlig unzulässig ist. Wenn die Meldung auch dann noch herauskommt, nachdem Sie diese Checkliste abgearbeitet haben, dürfen Sie nun beruhigt in Panik verfallen, denn in dem Fall ist es sicher, dass ihre Task — oder irgendeine andere Task im Rechner — sich damit vergnügt, fremden Speicher vollzumalen.Überschreitungen der angeforderten Blöcke mit schreibenden Operationen sind glänzend dazu geeignet, die laufende Task — oder gar das RTOS–UH selbst — in die ewigen Jagdgründe zu schicken.

Besondere Vorsicht ist bei fehlerhaften Pointern geboten, die in geschützte oder unzulässige Bereiche zeigen. Derlei Zugriffe kann `free()` selbstverständlich im nicht–priviligierten CPU–Modus nicht mehr abfangen und statt eines netten Hinweises der Bibliotheksroutine rennen Sie nun in die `BUSERROR`–Keule des Betriebssystems.

Die Verwendung der ANSI–C–Funktionen hält Ihre Programme kompatibel, gehört aber in Bezug auf die Geschwindigkeit nicht gerade zu den Spitzenreitern der Bibliotheksfunktionen. Bei Terminierung der Task gehen alle bis dahin allokierten Speicherbereiche wieder in den Besitz des Betriebssystem über. Da sich auch die genormten Speicheranforderungen auf den bekannten Systemtraps zur Beschaffung von Procedureworkspace abstützen, räumt RTOS–UH diese Bereiche automatisch ab und gibt sie als Freispeicher ans System zurück.

Sie sollten also der Versuch wiederstehen, Speicher einer beliebigen Subtask in Listen einer anderen Task zu ketten. Wenn eine Subtask terminiert, ist das weitere Verhalten der anderen Task beim Zugriff

auf derartige Listen von zunehmender Stochastik bestimmt. Solche nichtdeterministischen Fehler zu debuggen ist hochgradig ekelig und deshalb sollten Sie immer daran denken, was Sie sich mit einem solchen Speicher–Mischmasch einhandeln können.

Die Funktionsweise der ANSI–C–Routinen ist simpel. Bei fehlendem Speicher antworten sie mit einem `NULL`–Pointer. Ist Speicher vorhanden, so wird ein Pointer auf einen ausreichend grossen Block geliefert. Dieser Speicherblock ist grundsätzlich an einer Wortgrenze des Speichers ausgerichtet.

15.6.2 Direkte Speicheranforderungen

In manchen Fällen ist es lästig oder geradezu unmöglich, sich mit den von mir implementierten ANSI–C–Funktionen herumzuschlagen. Gerade in Programmen, die intensiv mit vielen kleinen Speicherbereichen hantieren, die dynamisch angefordert und wieder freigegeben werden müssen, kann der Verwaltungs–Overhead der ANSI–C–Funktionen störend sein. So führt in der aktuell ausgelieferten Implementierung die Anforderung von einem Byte zu einer Speicherbelegung von achtzehn Bytes im Ringpuffer. Dieses krasse Missverhältnis basiert auf der Grösse des Verwaltungskopfes von sechszehn Bytes pro dynamischem Block und der Forderung der MC68000–CPU, den nächsten Header bitteschön auf einer geraden Adresse zu platzieren. Selbstverständlich verbessert sich das Verhältnis von Brutto– zu Nettospeicher bei grösseren Speicherblöcken. Trotzdem kenne ich eigentlich keinen C–Programmierer, der nicht eigene Speicherverwaltungen über die Standardroutinen legt. Selbst der C–Compiler verwendet die Libraryfunktionen nur zur Beschaffung grösserer Blöcke, die dann durch übergeordnete Funktionen verwaltet werden.

Sicherheit hat eben ihren Preis und die Zahl der Bytes, die die Gültigkeit eines Blockes, seinen Zustand im Speicherring und viele langweilige Dinge mehr kontrollieren, verlangt eben diesen Wasserkopf. Betrachten Sie die Standard–Funktionen deshalb als langsame aber sichere Arbeitsgrundlage, die bei tagtäglich vorkommende Programmierfehlern nicht unbedingt den Griff zum Stromschalter des Computers erzwingt.

Selbstverständlich geht es auch anders. Ich will hier nicht vorschlagen, dass Sie sich eine neue ANSI–C–Speicherverwaltung kodieren. Das ist kein Spielplatz für Sie als Nutzer des Compilers. Es gibt jedoch eine Reihe von Methoden, sich legal vom Betriebssystem Speicher zu beschaffen. Diese Wege sind in der Dokumentation des RTOS–UH beschrieben und deshalb sollte die Nutzung derartiger System–Funktionalitäten wohl gesichert sein. Drei Wege der Speicherbeschaffung sollen hier vorgestellt werden, um Ihnen die Allokation von dynamischem Speicher zur Laufzeit der Task zu ermöglichen.

15.6.2.1 Die Procedureworkspace–Traps

Der unmittelbarste Weg besteht in der Nutzung der Systemtraps des RTOS–UH. Das Betriebssystem verwaltet den zur Verfügung stehenden Gesamtspeicher als eine doppeltverkettete Liste von Speicherblöcken. Jeder Block beginnt mit einem Verwaltungs–Header, der die Verkettungspointer und die aktuelle Verwendung des jeweiligen Segmentes beinhaltet. Wenn eine Task mittels der Traps `.WSBS`, `WSFA` oder `WSFS` (siehe RTOS–UH–Manual F-I-66) Speicher vom System anfordert, liefert RTOS–UH entweder eine Fehlermeldung oder einen Zeiger auf den beschafften Speicherbereich. Der Pointer zeigt dabei auf den Verwaltungs–Header dieses Blockes. In der Includedatei `<rtos.h>` finden Sie die Angaben über den Aufbau solcher Speichersegmente und die Prototypen der Systemanschlüsse auf Hochsprachenebene.

```
typedef struct LinkedWorkspace
{  MemSectionHeader         head  ;
   struct LinkedWorkspace  *fort  ;
   struct LinkedWorkspace  *backt ;
```

```
      Task                    *tido  ;
   }  LinkedWorkspace ;

   typedef struct MemSectionHeader
   {  struct MemSectionHeader  *forward  ;
      struct MemSectionHeader  *backward ;
      UserNumber                owner    ;
      MemSection                typ      ;
   }  MemSectionHeader ;

   LinkedWorkspace  *rt_alloc_memory_backward( size_t  size );
   LinkedWorkspace  *rt_alloc_memory_forward ( size_t  size );
   LinkedWorkspace  *rt_alloc_memory_fixed( void  *start, vo-
id  *end );
```

In CREST–C sind die Anschlüsse an die Speicherbeschaffungs–Traps so implementiert worden, dass bei Speicherplatzmangel `NULL` und sonst ein Pointer auf den Verwaltungs–Header vom Typ `LinkedWorkspace` geliefert wird.

Es stehen drei wesentliche Funktionen zur Speicherverwaltung Verfügung. `rt_alloc_memory_backward()` und `rt_alloc_memory_forward()` besorgen im Freispeicher des Systems einen ausreichenden Block. Dabei durchsucht `rt_alloc_memory_forward()` die Speicherketten von kleinen Adressen beginnend nach oben und teilt nach dem *First Fit*–Verfahren Speicher aus dem ersten passenden `FREE`–Block zu. Entsprechend der RTOS–UH–Konvention sollten Sie diese Funktion für grosse und langlebige Speicheranforderungen verwenden. Für kleine oder kurzfristige Speicherallokationen wird unter RTOS–UH bevorzugt Speicher von den höheren Adressen her zugeteilt. Dazu dient `rt_alloc_memory_backward()`. Wenn Sie sich an diese Zuteilungsregeln halten, erleichtern Sie dem Betriebssystem die Sucherei in den Systemketten ganz erheblich und werden üblicherweise mit besseren Leistungen von RTOS–UH belohnt.

Zur Anforderung eines Speicherblockes an einer festgelegten Position ist zudem die Funktion `rt_alloc_memory_fixed()` vorhanden, die als Argumente die erste Startadresse des zu belegenden Blockes sowie die Startadresse, ab der die nächste Speichersektion beginnen soll, erwartet. Befindet sich in diesem Bereich keine `FREE`–Sektion, so scheitert der Aufruf der Funktion.

Zur korrekten Verwendung der Trapanschlüsse ist es unumgänglich, die Funktionsweise und den Aufbau der Struktur `LinkedWorkspace` zu verstehen. Zu Beginn der Struktur liegen zwei Pointer `head.forward` und `head.backward`, die das Betriebssystem zur Verkettung aller von ihm verwalteten Bereiche benutzt. Darauf folgen noch zwei Speicherplätze, in denen RTOS–UH die aktuelle Verwendung des Blockes (`head.typ`) und die Kennziffer des Benutzers (`head.owner`) eingetragen hat. In `head.typ` ist nach der Speicheranforderung eingetragen, dass es sich bei dem Block um `PWSP` (Procedureworkspace) handelt. Zur Verwaltung eines Moduls würden diese Informationen bereits ausreichen. Diese Speicherköpfe werden vom RTOS–UH angezeigt, wenn Sie auf Kommandoebene `S` eingeben. Bei `PWSP`–Blöcken sind zudem die drei folgenden Pointer sinnvoll vorbesetzt. Mittels `fort` und `backt` wird eine weitere doppeltverkettete Liste verwaltet, in der sämtliche `PWSP`–Anforderungen der Task aufgeführt sind, um bei Terminierung der Task auch aufräumen zu können. Die Task, die den Block nun besitzt, ist zudem noch mittels des Pointers `tido` zu identifizieren.

Bei der Anforderung von Speicher mittels der direkten Trapanschlüsse müssen Sie stets berücksichtigen, dass RTOS–UH die Brutto–Blockgrösse als Argument verlangt. Wenn Sie also mittels der drei Trap–Anschlüsse Speicher allokieren, müssen Sie den Platz für einen Header zusätzlich anfordern und daran denken, dass ihre Nutzdaten erst hinter dem Header beginnen. Die Zerstörung des Headers würde sehr schnell den totalen Absturz des RTOS–UH nach sich ziehen. Der Nutzdatenspeicher — also der Bereich, der für ihre eigenen Daten zur Verfügung steht — wird vom Betriebssystem nicht initialisiert,

enthält also in der Regel noch die Daten der letzten Verwendung — wer auch immer den Speicher vorher im Besitz gehabt haben mag.

15.6.2.2 Procedureworkspace leicht verwaltet

Die Verwendung der Trapanschlüsse ist mühsam und bestraft Fehlbedienungen gnadenlos. Um etwas einfacher an PWSP–Memory heranzukommen, sind ein paar Funktionen entstanden, die sich etwas weniger benutzerfeindlich präsentieren. `rt_pwsp_memory_backward()` und `rt_pwsp_memory_forward()` verlangen nun nur noch die Angabe des vom Anwender gewünschten Netto–Speicherbedarfs und berücksichtigen intern den notwendigen RTOS–UH–Header. Weiterhin zeigt der Rückgabepointer nunmehr direkt auf den Nutzdatenspeicher. `rt_alloc_memory_fixed()` liefert bei Erfolg Nutzdatenspeicher im Bereich von `start` bis `end`. Die Verwaltungsstrukturen befinden sich — für den C–Programmierer unsichtbar — direkt vor `start`.

```
void  *rt_pwsp_memory_backward( size_t  size );
void  *rt_pwsp_memory_forward ( size_t  size );
void  *rt_pwsp_alloc_memory_fixed( void  *start, void  *end );
int    rt_pwsp_free_memory( void  *addr );
```

Eine Freigabe von Procedureworkspace während der Laufzeit der Task ist vom Betriebssystem nicht als Trap vorgesehen. Dazu bedarf es einiger spezieller Aktionen, die von der Funktion `rt_pwsp_free_memory()` ausgeführt werden. Um zu verstehen, wann Sie diese Funktion einsetzen dürfen, ist es hilfreich, das Funktionsprinzip zu kennen.

Der angeforderte PWSP bleibt im Besitz der Task, bis diese beendet wird. Wenn Sie Blöcke während der Laufzeit einer Task ans System zurückgeben wollen, so wird die Sache etwas schwieriger. Es gibt zwar den RTOS–UH–Trap RWSP, der Procedureworkspace wieder als Freispeicher ans Betriebssystem übergibt. Der Name des Trapanschlusses lautet `rt_free_memory()` . Also den Pointer auf den freizugebenden Block an diese Funktion übergeben, die Funktion aufrufen und weg ist der Speicher? So einfach ist es nicht, denn leider zerfetzt es jetzt mit aller Wahrscheinlichkeit in geringem zeitlichen Abstand ihr System. Der Block ist zwar wieder frei und kann anderweitig neu allokiert werden — die Task, die ihn angefordert hatte — kennt ihn jedoch noch und betrachtet ihn als integralen Bestandteil ihrer PWSP–Kette. Wenn zwei Tasks sich um einen Block streiten ohne voneinander zu wissen, ist das meist das Todesurteil für die Systemstabilität.

Darin besteht der ganze Trick. Vor der Freigabe ans System muss der Block aus der PWSP–Kette der Task ausgeklinkt werden, die ihn angefordert hat. Im Prinzip müssen Sie nur die Zeiger `fort` und `backt` benutzen, um dem Vorgänger und Nachfolger zu erklären, dass der fragliche Block nicht mehr zwischen den beiden liegt. Dazu ist ein kurzer Ausflug in den Supervisor–Mode angebracht, da das System sonst durch offene Ketten verärgert werden könnte. Sie befinden sich schliesslich in einer Umgebung, in der Sie nie so genau wissen, wer sonst noch auf den Pointerketten langläuft, die Sie gerade verschrotten wollen. Durch das Umschalten in den Supervisor–Mode erreichen Sie exklusiven Zugriff auf die Speicherverzeigerung.

Erst wenn die Auskettung erfolgreich abgeschlossen wurde, gehört der ausgekettet Block Ihnen alleine und kann gefahrlos mittels `rt_free_memory()` freigegeben werden. Um diese Arbeit zu vereinfachen, können Sie auf die Funktion `rt_pwsp_free_memory()` zurückgreifen, die für Auskettung und Verschrottung von Speicherblöcken sorgt, die mittels der `rt_pwsp_alloc...`–Funktionen (und ausschliesslich diese sind gemeint!) allokiert wurden. Eine dringende Warnung: Verwechseln Sie nie die Funktionen `rt_pwsp_free_memory()` und `rt_free_memory()` — Sie verheizen damit mutwillig wichtige Speicherketten und die Fehlersuche in einem Programm, das den Rechner bei jedem Test ermordert, ist aufwendig.

Um es Ihnen zu ermöglichen, bei Bedarf eigene Allokationsroutinen zu schreiben, liegen hier exemplarisch die Funktionen `rt_pwsp_alloc_memory_forward()` und `rt_pwsp_free_memory()` im Quelltext vor.

```
   void *rt_pwsp_alloc_memory_forward( size_t  netto )
   {
      register LinkedWorkspace  *header ;

      header = rt_alloc_memory_forward( netto + sizeof( Linked-
Workspace ) ) ;

      if ( header )
         return( ++header ) ;
      else
         return( NULL ) ;
   }

   void rt_pwsp_free_memory( void  *data_ptr )
   {
      register LinkedWorkspace  *header = data_ptr ;
      register void             *user_stack        ;

      --header ;

      user_stack = rt_supervisor_mode( 0x2700U ) ;
      {
         header->fort->backt = header->backt ;
         header->backt->fort = header->fort  ;
      }
      rt_user_mode( user_stack ) ;

      rt_free_memory( (void*)header ) ;
   }
```

15.6.2.3 Dauerhafte Speicherblöcke als Module

Die Procedureworkspace–Bereiche verschwinden automatisch bei Terminierung der Besitzertask. Meist ist dieser Automatismus das erwünschte Verhalten. Wenn Speicherbereiche einen ABORT oder das Ende des ersten Besitzers überleben sollen, ist es jedoch unumgänglich, alle verwandtschaftlichen Beziehungen zwischen Task und Block aufzukündigen. Das Verfahren besteht wiederum im Ausklinken aus der PWSP–Kette der anfordernden Task. Auch hier stehen unter CREST–C spezielle Funktionen bereit, die diese Aktionen implizit vornehmen.

```
   void  *rt_named_alloc_memory_fixed   ( char *name, vo-
id *start, void *end );
   void  *rt_named_alloc_memory_backward( char *name, size_t size );
   void  *rt_named_alloc_memory_forward ( char *name, size_t size );
   void   rt_named_free_memory( void  *addr );
```

Die Funktionen liefern keinen PWSP–Speicher sondern Blöcke mit der Kennung MDLE. Als Programmierer sind Sie alleine für die Verwaltung dieser Speicherblöcke zuständig. Über das Argument name kann dem Modul ein Name zugeordnet werden. Da die Module das Ende der anfordernden Task nun überleben — das war schliesslich Sinn der Aktion — müssen diese später explizit gelöscht werden, wenn der Speicher nicht mehr benötigt wird.

Dazu übergeben Sie den Pointer, den die Allokationsroutinen geliefert haben an die Funktion `rt_named_free_memory()`. `Sie` sind dafür verantwortlich, dass das Modul noch unverändert an dieser Stelle vorhanden ist. Im Fehlerfalle dürfen Sie sich schon mal nett von Ihrem Rechner verabschieden, da RTOS–UH Unfug mit seinen Systemspeicherketten recht intollerant gegenübersteht.

Erneut die Warnung! Verwechseln Sie nie die Funktionen `rt_named_free_memory()`, `rt_pwsp_free_memory()` und `rt_free_memory()`. Die letztgenannte Funktion ist der Anschluss eines Systemtraps und erwartet den Pointer auf den `MemSectionHeader` des Speicherblocks, der freigegeben werden soll. `rt_pwsp_free_memory()` erwartet einen eingeketten `PWSP`–Block und schlägt ebenfalls lang hin, wenn Sie die Funktion auf ein Modul loslassen. Die Funktion `rt_named_free_memory()` dient nur der Freigabe von Blöcken, die Sie mit den zugehörigen Allokationsroutinen beschafft haben.

Für Programme, die über gemeinsame Speicherblöcke hinweg Daten austauschen müssen, gibt es im Verwaltungsblock dieser Module einen weiteren Eintrag, der von den `rt_named_alloc...`–Routinen korrekt gesetzt wird. Wenn Sie also wissen, dass eine andere Task ein namentlich bekanntes Modul im Speicher abgelegt hat und möchten nun darauf zugreifen, so können Sie mit der C–Funktion `rt_search_modul()` die Adresse des Modulkopfes bestimmen. Dabei ist jedoch zu beachten, dass RTOS–UH sich darauf versteift, Module, deren Namen mit einem Doppelkreuz # beginnen, nicht finden zu wollen. Es handelt sich dabei um eine Art Abwehrhaltung des Betriebssystems, um zu verhindern, dass Nutzer an systemeigenen Tasks und Modulen herumspielen können und deshalb so tut, als wären sie nicht in der Speicherverwaltung eingetragen.

```
MemSectionHeader *rt_search_modul( char *modulename ) ;
```

Bei Misserfolg liefert `rt_search_modul()` den Pointer `NULL`. Um an die Nutzdaten eines derartig aufgefundenen Moduls heranzukommen, könnten Sie sich z.B. eine Funktion wie `search_memory_modul()` schreiben, die direkt den Pointer auf die Nutzdaten des Moduls liefert.

```
void *rt_search_named_memory( char *name )
{
   register MemSectionHeader  *header ;

   if ( ( header = rt_search_modul( name ) ) != NULL )
      return( ( (UnlinkedWorkspace*)header )->user_data ) ;
   else
      return( NULL ) ;
}
```

Alternativ steht die eben beschriebene Funktion `rt_search_named_memory()` auch direkt in den CREST–C–Bibliotheken zur Verfügung.

Ihr Rückgabewert ist identisch mit dem Zeiger, den Sie mittels der Allokationsroutinen bekommen hätten; ist also auch für Speicherfreigaben mittels `rt_named_free_memory()` tauglich. Da die Möglichkeiten beim Umgang mit derartigen Speichermodulen zu vielseitig sind, um sie mit speziellen Bibliotheksfunktionen vollständig abzudecken, möchte ich zur weiteren Vertiefung des Themas noch exemplarisch den Quelltext der Bibliotheksfunktionen `rt_named_alloc_memory_forward()` und `rt_named_free_memory()` vorstellen und Ihnen viel Spass bei eigenen Experimenten wünschen.

```
void  *rt_named_alloc_memory_forward( char  *str, size_t  size )
{
   void              *header            ;
   size_t             len, brutto       ;
   void              *netto, *user_stack ;
   MemSectionName    *name_ptr          ;
   char              *l_name_entry      ;
```

```
      len  = strlen( str ) ;
      size = ( size + 1 ) & ~1L ;

      brutto = sizeof( UnlinkedWorkspace  )
             + sizeof( UnlinkedWorkspace* )
             + size                        ;

      if ( len > sizeof( MemSectionName ) )
         brutto += ( len + 2 ) & ~1L ;

      if ( ( header = rt_alloc_memory_forward( brutto ) ) == NULL )
         return( NULL ) ;

      user_stack = rt_supervisor_mode( 0x2700U ) ;
      {
         ((LinkedWorkspace*)header)->fort->backt =
         ((LinkedWorkspace*)header)->backt       ;

         ((LinkedWorkspace*)header)->backt->fort =
         ((LinkedWorkspace*)header)->fort        ;

         name_ptr = &((UnlinkedWorkspace*)header)->name ;

         if ( len <= sizeof( MemSectionName ) )
         {
            memset( name_ptr->name, ' ', sizeof( MemSectionName ) ) ;
            strncpy( name_ptr->name, str, len ) ;

            /* Pointer auf die Rueckverkettung bestimmen */
            netto = (char*)( (UnlinkedWorkspace*)header + 1 ) ;
         }
         else
         {
            l_name_entry = (char*)( (UnlinkedWorkspace*)header + 1 ) ;
            name_ptr->lname.name = (char*)( (long)l_name_entry-
(long)name_ptr ) ;
            name_ptr->lname.name_mark = 0 ;
            strncpy( l_name_entry, str, len ) ;
            l_name_entry[ len ] = '\xFF' ;

            /* Pointer auf die Rueckverkettung bestimmen */
            netto = (char*)( ( (long)l_name_entry + len + 2 ) & ~1L ) ;
         }
         /* Rueckverkettung fuer spaetere Freigabe aufbauen */
         *(UnlinkedWorkspace**)netto = header ;
         (char*)netto += sizeof( UnlinkedWorkspace* ) ;

         /* Wir sind ein MODUL und haben Speicher */
         ((UnlinkedWorkspace*)header)->head.typ  = 0x0010 ;
         ((UnlinkedWorkspace*)header)->user_data = netto  ;
      }
      rt_user_mode( user_stack ) ;
      return( netto ) ;
   }

   void rt_named_free_memory( void  *addr )
   {
      /* Pointer auf Rueckverkettung holen */
      addr = ((char*)addr) - sizeof( UnlinkedWorkspace* ) ;
```

```
    /* Pointer auf Modulkopf holen */
    addr = *(UnlinkedWorkspace**)addr ;

    /* Modulkopf rauswerfen */
    rt_free_memory( addr ) ;
}
```

15.6.3 Speicherplatzreservierung beim Systemstart

Bei vielen Programmen besteht die Notwendigkeit, über gemeinsame Datenbereiche mit anderen Prozessen in Verbindung zu bleiben. Dazu ist es bei RTOS–UH möglich, sich bereits beim Hochlaufen des Systems Speicher zu beschaffen. Es muss lediglich eine entsprechende Scheibe im Scanbereich des Systems untergebracht werden. Als Resultat dieser Bemühungen, richtet das RTOS–UH einen gelöschten Speicherbereich als eigenständiges Modul ein.

```
#pragma MEMORY  "MEMORY" 0x100000 0x200000
```

Die beiden Parameter dieses Kommandos bestehen aus der ersten zu reservierenden Adresse und der ersten wieder freien Adresse hinter dem einzurichtenden Modul.

Sie sollten dabei Vorsicht walten lassen. Ist die Angabe des Speicherbereiches fehlerhaft, so kommt das System gar nicht erst hoch. Wenn Sie den entsprechenden Block vor unbeabsichtigtem Entladen bewahren wollen, ist es angebracht, den Namen des Moduls mit einem „#“ als einleitendes Zeichen zu versehen. Damit sind Sie in guter Gesellschaft, da RTOS–UH selbst dieses Verfahren verwendet, um wichtige Module und Tasks vor unachtsamen Nutzern in Deckung zu bringen. Bedenken Sie dabei jedoch, dass Suchaktionen mittels der Funktion `rt_search_modul()` an Modulen mit Doppelkreuz scheitern werden.

Der Name des einzurichtenden Moduls darf maximal sechs Buchstaben umfassen. Diese Festlegung ist von mir willkürlich getroffen worden und erleichert im späteren Umgang mit den erzeugten Modulen den Zugriff auf die Nutzdaten, da Ihnen unter dieser Voraussetzung der angeforderte Speicher stets ab der Position `start+$10` zur Verfügung gestellt werden kann. Bei langen Modulnamen, die erst nach heftigstem Kampf mit den Möglichkeiten der entsprechenden Scheibe erzeugt werden können, würde die Angelegenheit nicht gerade komfortabler für Sie und mich.

15.7 CPU–Status wechseln

Die Motorola–CPU’s besitzen zwei unterschiedliche Betriebszustände. Im normalen Modus, dem User–Mode, steht nur ein eingeschränkter Befehlssatz zur Verfügung, der jedoch für die üblichen Aufgaben von Nutzerprogrammen hinreichend ist. Im privilegierten Betriebszustand, dem Supervisor–Mode stehen weitere Befehle bereit, die für administrative Aufgaben auf Betriebssystemebene benötigt werden. Dazu zählen z.B. schreibende Zugriffe auf bestimmte Prozessorregister, diverse Sonderadressierungsarten der CPU und viele andere Dinge mehr.

Die Zweiteilung dient bei vielen Betriebssystemen dem Datenschutz. Man erreicht damit, dass nicht jeder Anwenderprozess sich durch legale CPU–Anweisungen Privilegien verschaffen kann, die ihm nicht zustehen. Nur der Systemadministrator kann Prozesse starten, bei denen die Hardware–Schutzmechanismen nicht mehr greifen. Unter RTOS–UH gehörten solche Datenschutzüberlegungen nicht zu den Kriterien bei der Auslegung des Betriebssystems. Jede Task kann sich beliebige Systemrechte verschaffen. Diese Freiheit erkauft man — wie überall im Leben — mit einem zwangsläufig höheren Überlegungsaufwand bei der Verwendung dieser Features. Man kann durch Missbrauch des

Supervisor–Modes RTOS–UH beliebig verärgern, das Systemverhalten gravierend beeinträchtigen. Unter RTOS–UH bewirkt ein Wechsel in den privilegierten Modus ein Ausschalten des Dispatchers. Das bedeutet für die Task, die die Funktion `rt_supervisor_mode()` aufruft, dass sie nicht mehr durch eine andere Task vom Besitz des Betriebsmittels CPU getrennt werden kann. Einzig der Interrupt–Mechanismus der CPU funktioniert noch, d.h. dass Hardware–Interrupts noch durchkommen, solange sie eine höhere Priorität haben als der aktuell im Statusregister `SR` der CPU vermerkte Wert.

```
void  *rt_supervisor_mode( StatusReg  sr ) ;
```

Da sich auch das Statusregister beim Aufruf dieser Funktion angeben lässt, ist Vorsicht beim Aufruf unumgänglich. Wenn Sie das Statusregister auf Interrupt–Level 7 schalten(`rt_supervisor_mode(0x2700)`), kommen **keine** Interrupts mehr durch, solange Sie nicht Kumpel sind und wieder auf den User–Mode zurückschalten. Interrupts, die in dieser Phase auflaufen, können verschlafen werden, da sie von der CPU nicht mehr als *unbedingte* Aufforderung zum Kontextwechsel aufgefasst werden und erst nach Abarbeitung der höherprioren Interrupts zur Ausführung gelangen.

Sie können so unteilbare Sequenzen programmieren und auch ungestört im RTOS–UH selbst rumfrieseln. Wenn Sie dieses Spiel allerdings zu oft und zu lange spielen, werden Sie erschreckt feststellen, dass Ihre Systemuhr plötzlich nachgeht (funktioniert eben durch Timer–Interrupts), Datenübertragungen von Schnittstellen während der Ausführung Ihrer Supervisor–Sequenzen zusammenbrechen (funktioniert mittels Schnittstellen–Interrupts) und viele hässliche Scherze mehr.

Um wieder in den User–Mode zu wechseln, dient die Funktion `rt_user_mode()` . Diese erwartet als Parameter den von `rt_supervisor_mode` gelieferten Pointer, um den alten User–Stack zu restaurieren. Dazu noch ein paar erklärende Worte.

```
void rt_user_mode( void  *user_stack ) ;
```

Der Supervisor–Mode der Motorola–CPU's verwendet einen eigenen Stack. Dieser Supervisor–Stack ist unter RTOS–UH verdammt knapp dimensioniert und bei Überläufen kommt es definitiv zum Super–GAU. Deshalb verwendet CREST–C einen Trick und trägt beim Wechsel des CPU–Modus automatisch den User–Stack der aufrufenden Task als Supervisor–Stack ein. Die Dimensionierung des User–Stacks liegt, wie im Abschnitt 3.7 beschrieben, in der Hand des Programmierers. Beim Aufruf von `rt_supervisor_mode()` wird der alte Supervisor–Stackpointer als Resultat geliefert. Beim Wechsel in den User–Mode **muss** dieser Pointer wieder restauriert werden, um den alten Systemzustand wieder herzustellen. Bei der Belastung des Stacks auf Supervisor–Ebene sollten Sie sich dennoch etwas zurückhalten. Das folgende kleine Beispiel stellt eine übliche Sequenz dar, wie man unter CREST–C unteilbare Sequenzen kodieren kann.

```
{
   void  *stack ;

   stack = rt_supervisor_mode( 0x2700 ) ;
   {
      // Hier den eigentlichen Code unterbringen
   }
   rt_user_mode( stack ) ;
}
```

Der Rückfallmechanismus ist bei mittels der Compileroption `-U` übersetzten Programmen bedauerlicherweise auf Supervisor–Ebene paralysiert und deshalb *steht* der Rechner, wenn Sie in diesem Modus einen `Stackoverflow` erzeugen. Sie sollten einer Task, die in den Supervisor–Mode wechselt, immer genugend Stack zur Verfügung stellen, um derartige Abschüsse zu vermeiden.

15.8 Fehlermeldungen

Unter RTOS–UH kann mittels der Funktion `rt_error()` eine wortweise zusammengesetzte Meldung auf das Terminal des verantwortlichen Nutzers geschrieben werden. `errmsg` muss auf **konstanten** Text zeigen, der mit `$FF` endet. Berücksichtigen Sie, dass bei einem Blank (eigentlich sogar alles kleiner oder gleich 0x20) im Text die Ausgabe abgebrochen wird.

Sollten Sie sich schon einmal über die Underlines anstelle von Blanks in den Fehlermeldungen von Filemanagern oder anderen Treibern gewundert haben: das ist der Grund! Da dieses Verhalten RTOS–UH–intern als Feature genutzt wird, macht es wohl wenig Sinn, sich darüber zu beschweren…

Mit `errcod` wird festgelegt, was an Standardtexten zusätzlich ausgegeben wird. `errcod` muss man sich dabei als ein aus vier Nibbles (halben Bytes) zusammengesetztes Wort `ABCD` vorstellen. Die Belegung von `errcod` ist der Tabelle 15.11 zu entnehmen. Von der Verwendung der als *nicht belegt* gekennzeichneten Zahlenkombination ist dabei tunlichst abzusehen, da es sonst zu unsinnigen Ausgaben kommt.

```
void rt_error( char *errmsg, ErrMessage errcod ) ;
```

A: enthält folgende funktionelle Bits:					
2	unterdrücke Text von `errmsg`				
8	suspendiere die aufrufende Task				
B: Auswahl aus folgendem Vorrat:					
0	Blank	1	NOT	2	WRONG
3	ZERO–DIV	4	CHK	5	BLOCKS
6	BREAKPOINT	7	DIRECTORY	8	DISC
9	MEMORY	A	MODULE	B	MISSING
C bis F nicht belegt					
C: Auswahl aus folgendem Vorrat:					
0	Blank	1	BUS–ERROR	2	LDN
3	PRIO	4	LOADED	5	SUSPENDED
6	ACTIVE	7	COMMAND	8	ADDRESS
9	OP–CODE	A	PRIVILEDGED	B	OVERFLOW
C	IN SYSTEM	D	I/O	E	OPERAND
F nicht belegt					
D: Auswahl aus folgendem Vorrat:					
0	Blank	1	(ACT)	2	(TERMI)
3	(CONTINUE)	4	(XIO)	5	(TRAP)
6	(FLOPPY)	7	LOADER–INPUT	8	REC–CHECKSUM
9	LABEL	A	(MODE)	B	TIMING
C	INDEX	D	FPU–68881		
E bis F nicht belegt					

Abbildung 15.11: Aufbau des Errorcodes

Ab dem Nukleus 7.x haben sich die Ausgabetexte des `ERROR`–Traps verändert. Nunmehr werden die meisten Ausgaben in Kleinbuchstaben getätigt — sehr zur Erbauung der Anwender, die in ihren Programmen die bisherigen Fehlermeldungen ausgewertet haben. Programme, die bislang Textanalyse der Fehlermeldungen betrieben haben, sollten deshalb schnellstens umgestellt werden. In Tabelle 15.12 sind die neuen Meldungen aufgelistet.

b: Auswahl aus folgendem Vorrat:					
0	Blank	1	not	2	wrong
3	zero–division	4	CHK	5	blocks
6	breakpoint	7	directory	8	disc
9	memory	A	module	B	missing
C bis F nicht belegt					
c: Auswahl aus folgendem Vorrat:					
0	Blank	1	bus–error	2	device–ldn
3	prio	4	loaded	5	suspended
6	active	7	command	8	address
9	op–code	A	priviledged	B	overflow
C	in system	D	I/O	E	operand
F nicht belegt					
d: Auswahl aus folgendem Vorrat:					
0	Blank	1	(activate)	2	(terminate)
3	(continue)	4	(xio–call)	5	(trap)
6	(floppy/harddisc)	7	loader–input	8	rec–checksum
9	label	A	(mode)	B	timing
C	index	D	FPU–68881		
E bis F nicht belegt					

Abbildung 15.12: Aufbau des Errorcodes ab NUK 7.x

Mittels des folgenden kleinen Programmes lassen sich z.B. alle unterstützten Kombinationen der vordefinierten Fehlertexte ausgeben — ein entnervendes Spiel, wenn Sie den Lautsprecher Ihres Terminals nicht stummschalten können...

```
void main( void )
{
   ErrMessage  message, b, c, d ;
   for ( b=0 ; b<=0xB ; ++b )
      for ( c=0 ; c<=0xE ; ++c )
         for ( d=0 ; d<=0xD ; ++d )
         {
            message = ( b << (ErrMessage)8 )
                    | ( c << (ErrMessage)4 )
                    |   d ;
            printf( "%04lX\n", message ) ;
            rt_error( "MESSAGE-TEST:", message ) ;
         }
}
```

Ebenfalls erst ab Nukleusversion 7.x steht mit `rt_decode_error()` eine Funktion zur Verfügung, die die Dekodierung der Errorcodes in Klartextstrings ermöglicht. Die ersten drei Parameter der Funktion enthalten den Pointer auf den Ausgabepuffer, die Länge des Puffers und den zu übersetzenden Errorcode. Zusätzlich stehen noch die Argumente `ierrmsg` und `ierrlen` zur Verfügung. Wenn Sie dort sinnvolle Pointer vorgeben, so liefert die Funktion dort den Pointer hinter den Ausgabetext und die Restgrösse des Puffers zurück. Sind Sie an einer oder gar beiden Rückgabewerten nicht interessiert, so ist jeweils ein `NULL`–Pointer als Argument anzugeben.

```
void  rt_decode_error( char         *errmsg,
                       ErrLength     errlen,
                       ErrMessage    errcod,
```

```
    char       **ierrmsg,
    ErrLength   *ierrlen
) ;
```

15.8.1 Das Error–Handling von RTOS–UH

Die im letzten Abschnitt beschriebene Funktion `rt_error()` stellt lediglich eine vereinfachte C–Schnittstelle zum `ERROR`–Trap von RTOS–UH dar. Korrekt angewendet, benötigt man eigentlich keinerlei weitere Informationen über das interne Systemverhalten. Da ich mich nach Freigabe einer früheren Version dieses Handbuches mittels vieler Codefragmente und Fax–Schnipsel davon überzeugen durfte, dass diese Funktion von Anwendern ohne Hintergrundinformationen eigentlich nur fehlerhaft oder unsinnig verwendet wird, folgt nun eine knappe Zusammenfassung des Systemverhaltens beim Aufruf des `ERROR`'s.

Der `ERROR`–Trap als Systemaufruf stellt lediglich die erste Stufe bei der Ausgabe von System–Meldungen dar. Er arbeitet eng mit einer Task zusammen, die zu den essentiellen Komponenten von RTOS–UH zählt: der `#ERROR-Task`! Der Aufruf des Traps bewirkt lediglich, dass ein kleiner Ringpuffer innerhalb der undokumentierten Systemzellen des Betriebssystems mit den relevanten Informationen über die auszugebende Meldung gefüllt wird.

- Das Message–Wort (im C–Aufruf `errcod`)
- Der Task–Identifier der aufrufenden Task
- Der Pointer auf den Text

Für jeden User des Systems existiert ein derartiger Ringpuffer mit einer **endlichen** Anzahl von Einträgen. In üblichen System stehen exakt 6 Einträge zur Verfügung! Der `ERROR`–Trap ordnet nun die eingelaufene Meldung dem verursachenden User zu und sortiert sie in dessen Meldungsring ein. Dies kann aus zwei Gründen fehlschlagen.

Einerseits muss es sich beim Verursacher nicht zwangsweise um eine Task handeln. Auch Interruptroutinen sind potentielle Kanditaten für Fehler und entsprechende Meldungen. In diesem Falle kann kein User zugeordnet werden und die Meldung landet auf der Systemconsole — üblicherweise eben auf dem Terminal, das mit der Schnittstelle `/A1/` (oder wie die Betreuungstask für `LDN=0` auch immer heissen mag) verdrahtet ist.

Andererseits besteht bei Ringpuffern auch ständig die Gefahr, dass diese überlaufen. Wenn schneller Meldungen in den Puffer hineingeschrieben werden, als die Ausgabeseite herauslesen kann, gehen schlicht Meldungen verloren.

Womit der Punkt erreicht ist, an dem die Leseseite des Ringpuffers betrachtet werden soll. Es handelt sich dabei um die bereits angesprochene `#ERROR`–Task. In der Systemkette werden Sie diese üblicherweise unter dem Namen `#ERROR` finden. Im hochgelaufenen Systemzustand liegt diese hochpriore Task permanent in Lauerstellung, und wartet geduldig, dass ein `ERROR`–Trap ausgelöst wird, der sie fortsetzt.

In diesem Falle bastelt sie sich aus den Informationen des Ringpuffers einen String zusammen, der mittels eines speziellen Ce's zur Schnittstelle des betreffenden Users geschickt wird. Da der eigentliche Ausgabevorgang nur mit einem gewissen Zeitaufwand zu erledigen ist — über serielle Schnittstellen geht da z.B. bei 9600 Baud pro Zeichen eine Millisekunde ins Land — ist es durchaus realistisch, dass die Informationen im Messagepuffer zum Zeitpunkt ihrer Auswertung von der `#ERROR`–Task, bereits hoffnungslos veraltet sind.

Ein beliebtes Beispiel zu diesem Thema stellt der `COPY`–Befehl dar. Die Abschlussmeldung `CO-`

`PY/xx: (TERMI).` wird von der `COPY/xx`–Subtask nach der Beendigung ihres Tuns mittels des `ERROR`–Trap's auf die Reise geschickt. Üblicherweise — wenn nicht zuviele Meldungen quasi gleichzeitig anfallen oder das Ausgabegerät z.B. durch *Ctrl–S* blockiert wird — schafft es die hochpriore `#ERROR`–Task, aus dem Taskidentifier im Meldungspuffer noch den Namen der Task zu dekodieren und auszugeben. Ist die `#ERROR`–Task allerdings kräftig mit Ausgaben beschäftigt, so greift das Multitasking und der `COPY`–Prozess benutzt die freibleibende CPU–Zeit, um zwischenzeitlich aus dem System zu verschwinden. Kommt die `#ERROR`–Task nun endlich doch noch bei dem Eintrag an, so stellt Sie fest, dass die dort eingetragene TID nicht mehr gültig ist und gibt als Zeichen ihres guten Willens den Pseudo–Tasknamen `--??--` aus.

Dieses Verhalten gilt allerdings nur für den Tasknamen. Bei dem optinalen Ausgabetext besteht für die `#ERROR`–Task keinerlei Chance, festzustellen, ob der Pointer im Messagepuffer noch auf einen sinnvollen Text zeigt. Deshalb auch die (oft ignorierte) Massgabe, dass der `ERROR`–Trap nur mit einem **Pointer auf einen konstanten Text** versorgt werden darf!

Im folgenden Fall wurde das Beispiel des vorausgegangenen Abschnitts so verstümmelt, dass es einem üblichen Anwenderprogramm gleichkommt. Es wird mit einer lokalen Variable als Textpuffer gearbeitet und mit Hochgeschwindigkeit ein Sack voll Error–Meldungen produziert.

```
void main( void )
{
   ErrMessage  message, b, c, d ;
   char        str[ 32 ]        ;
   for ( b=0 ; b<=0xB ; ++b )
      for ( c=0 ; c<=0xE ; ++c )
         for ( d=0 ; d<=0xD ; ++d )
         {
            message = ( b << (ErrMessage)8 )
                    | ( c << (ErrMessage)4 )
                    |   d ;
            sprintf( str, "MESSAGE-TEST:%04lX:\xFF", message ) ;
            rt_error( str, message ) ; // !!! SO NIEMALS !!!
         }
}
```

Dabei gehen dann definitiv zwei Dinge gravierend schief. Erstens kann man beruhigt davon ausgehen, dass ein üblicher RTOS–UH–Rechner heute in der Lage ist, das Zusammenbauen des Message–Wortes und des Ausgabetextes schneller zu bewerkstelligen, als eine serielle Schnittstelle den betreffenden Text ausgeben kann. Das bedeutet: es gehen Meldungen verloren, weil der interne Ringpuffer überläuft! Wurden derartige `rt_error()`–Aufrufe quasi als schnell aufeinanderfolgende Debugausgaben missbraucht, kommt es dann zu verwirrten Rückfragen, weshalb die auf dem Schirm zu beobachteten Ausgaben nicht mit dem erwarteten Programmfluss übereinstimmen. Nun, man kann schlicht und final festhalten, dass der `ERROR`–Trap für diesen Anwendungszweck absolut ungeeignet ist!

Schlimmer als verlorene Ausgaben ist jedoch die Tatsache, dass in dem vorgestellten Negativbeispiel auch Datenmüll bei den Ausgaben erzeugt wird — soll heissen: das ausgebenene Hexmuster hat nur in den seltensten Fällen etwas mit dem vordefinierten Fehlertext zu tun, der dahinter erscheint. Die Begründung ist simpel: alle im Ringpuffer verzeichneten Pointer auf den Ausgabetext verweisen auf den identischen Speicherplatz! Und der wird bei jedem Schleifendurchlauf brav neu belegt...

Wenn das `main()`–Programm es dann auch noch schafft, nach Durchlaufen der Schleifen zu terminieren, bevor die `#ERROR`–Task dazu kommt, die letzten Meldungen auszugeben, verweist der gespeicherte Textpointer sogar auf einen Speicherbereich mit unbekanntem Besitzer. Eine sehr unbefriedigende Lage mit dem Potential sehr interessanter Effekte...

Die Funktionalität der `#ERROR`–Task wurde hier noch ganz rudimentär wiedergegeben. In der Praxis ist diese Task eine Art eierlegende Wollmilchsau im Betriebssystem, die noch ein paar Aufgaben mehr

zu verwalten hat. Wenn Sie ausprobieren möchten, wie nachtragend RTOS–UH reagiert, wenn man seine #ERROR–Task verärgert, dann reicht der Aufruf des ERROR–Traps mit einem illegalen Pointer, der die #ERROR z.B. auf einen BUS ERROR laufen lässt. Die Anwendertask, die den Ärger verursacht hat, wird davon nicht betroffen, aber die #ERROR–Task wird beim Zugriff über den Pointer versenkt. Das System schafft es nicht einmal mehr, die BUS ERROR–Meldung auszugeben und den Gedanken, über Ctrl-A im System nachzuschauen, was da passiert ist, können Sie ebenfalls begraben, weil zur Aktivierung des Users, der dann das Sternchen ausgibt und auf Eingaben wartet, leider die viel zu früh von uns gegangene #ERROR–Task zuständig gewesen wäre.

Kapitel 16

Systemkonfiguration

Wenn ein Computer mit RTOS–UH–EPROM's eingeschaltet wird, so wird der Start–PC auf dem zentralsten Teil des Betriebssystems stehen: dem Nukleus! Der Nukleus übernimmt nun die Aufgabe, den Rest des Betriebssystems im Adressbereich des Rechners zu finden und zu einer funktionellen Einheit zu verbinden. Im Prinzip existieren vier wesentliche Betriebszustände von RTOS–UH:

1. *Pre–Cold*
2. *Kaltstart*
3. *Warmstart*
4. *Normalbetrieb*

Die folgende Abschnitte beschreiben den groben Ablauf der Systemkonfiguration. Sollten manche Dinge etwa nebulös klingen, dann sollten Sie stets bedenken, dass Gottes Tierreich gross ist und viele Eigenschaften von Hardware zu speziell und undurchsichtig sind, um hier abgehandelt zu werden.

16.1 Pre–Cold

Der Zeitpunkt *Power–On* , an dem der Rechner, auf dem RTOS–UH laufen soll, seinen Strom bekommt: Die CPU beginnt auf dem Start–PC loszulaufen.

Je nach Art der CPU und der umgebenden Hardware des Gesamtcomputers kann es nun erstmal notwendig sein, die Hardware zu konfigurieren — gemeint sind damit so essentielle Resourcen wie RAM und EPROM! Diese Hochlaufphase ist oft noch eine echte Ursuppe, in der sich der Rechner für die CPU völlig anders darstellt, als Sie das System später im betriebsbereiten Zustand präsentiert bekommen. Teilweise sieht die CPU beim Aufwachen nur winzige Stückchen von RAM und EPROM und das auch noch auch auf recht seltsamen Adressen.

Diese Phase wird als *Pre–Cold* bezeichnet, weil sie *vor* dem eigentlichen Kaltstart stattfindet und hier Aktionen durchgeführt werden, die es überhaupt erst ermöglichen, dass die CPU in die Lage versetzt wird, auch noch den nächsten Befehl auszuführen. Für Sie als Anwender gibt es hier keinerlei Eingriffsmöglichkeit! Es handelt sich im Implementierungsdetails für die spezielle Hardware. Da die dafür durchzuführenden Aktionen sehr spezifisch ausfallen, erspare ich mir weitere Erläuterungen.

In manchen Fällen ist der *Pre–Cold*–Code obsolet, weil RTOS–UH aus Betriebszuständen heraus gestartet wird, die diesen essentiellen Setup der Hardware bereits durchgeführt haben. Wenn z.B. RTOS–UH aus einem anderen Betriebssystem heraus gestartet wird (Atari, Amiga) oder bereits ein Monitorprogramm auf der Hardware läuft, so sind die wesentlichsten Aktionen logischerweise bereits erledigt.

16.2 Kaltstart

Die Kaltstartphase ist praktisch gesehen der erste Atemzug des hardwareunabhängigen Teiles des Betriebssystems und wird in folgenden Fällen ausgeführt:

- Beim Stromeinschalten.
- Beim Drücken des Reset–Knopfes.
- Bei der Verwendung des Befehles `SYSTEM_RESET` .
- Bei fehlgeschlagenen Versuchen, einen korrekten Warmstart durchzuführen.

In diesem Betriebszustand konfiguriert sich RTOS–UH; sucht, findet und initialisiert Hardware, die diesen Vorgang nur einmal benötigt oder zulässt. RTOS–UH lässt bereits in diesem Zustand Eingriffe von Seiten des System–Programmieres zu, der nicht im Besitz der Betriebssystemquellen ist. Systemdienste sind noch nicht verfügbar. Der Anwender kann jedoch Maschinencode, der ohne Systemunterstützung lauffähig ist, zur Ausführung bringen. Entsprechender Code kann mittels des Kommandos `#pragma COLDSTART` dem Betriebssystem hinzugefügt werden.

In der Kaltstartphase läuft RTOS–UH auf Supervisor–Mode unter Interruptsperre auf Level 7 — Sie sollten nicht versuchen, diesen Zustand in einer Kaltstartscheibe zu verändern...

16.2.1 Initialisieren der Systemvariablen

RTOS–UH kommt — wie fast jedes andere Programm auch — nicht um die Tatsache herum, Variablen benutzen zu müssen. Diese Variablen liegen auf fest vereinbarten Adressen und beginnen auf der Position `0x7FE` im RAM. Eine gewisse Anzahl dieser Variablen ist im RTOS–UH–Manual dokumentiert. Für den Grossteil der Systemvariablen behält sich Professor Gerth jedoch vor, auf eine Dokumentation zu verzichten — eine Einstellung, die ich durchaus verstehen kann, weil es mir a) ebenfalls gegen den Strich geht, wenn jeder Anwender an allem Dingen rumfummeln kann und b) ich auch gelegentlich (oder oft) an undokumentierten Features rumbasteln will, ohne inkompatibel zu irgendwelcher Dokumentation zu werden.

Zum wesentlichen Verständnis von RTOS–UH genügt das Wissen, dass es zwei Bereiche von Variablen gibt:

1. Der statische Anteil, in dem sich die Systemzellen tummeln, die grundsätzlich für jedes RTOS–UH unverzichtbar sind. Dieser Bereich liegt von `0x7FE` bis grob `0xA00` — ist also etwas über ein halbes Kilobyte gross.
2. Der dynamische Bereich, der sich direkt an den statischen Bereich anschliesst und die Daten enthält, die sich bei der Konfiguration des aktuell gestarteten Systems ergeben. Als groben Anhaltspunkt sollten Sie zunächst im Hinterkopf behalten, dass dieser Bereich die Grenze von 32KB nicht überschreiten sollte — ansonsten können böse Dinge geschehen!

Der statische Anteil der Variablen wird beim Kaltstart zunächst gelöscht.

16.2.2 Installieren des Scheiben–Scanners

Zum Verständnis dieses Abschnittes sind ein paar einleitende Worte notwendig. RTOS–UH ist die Summe aus dem Systemkern (Nukleus) und den bei der Implementierung hinzugefügten Komponenten. Der Nukleus ist isoliert betrachtet eine reichlich unnütze Angelegenheit. Der nackte Nukleus kennt nur

eine einzige Task, die Idle–Task `#IDLE`, die nichts weiter tut, als in einer Endlosschleife die CPU bei Laune zu halten.

Der Nukleus besitzt jedoch die Fähigkeit, beim Systemstart nachzuschauen, ob das Betriebssystem mehr können soll, als sinnlos Strom mit der Hardware zu verbrauchen. Dazu existiert das Konzept der Scheiben bzw. neudeutsch Slices. Diese Scheiben bestehen aus jeweils drei 16–Bit–Worten im Speicher. Der Inhalt einer Scheibe beginnt stets mit `0xAEB1 0xBF95`. Es folgt eine verschlüsselte Scheibennummer, die sich nach der Rechenvorschrift `(scheibennr*2+1)*37` ergibt. Warum die Scheibennummer so seltsam kodiert wird?

Die Scheibennummer -1 bis 18 sind aktuell im RTOS–UH–Manual dokumentiert und werden im CREST–C–Handbuch in den folgenden Abschnitten erläutert.

Für RTOS–UH stellt sich zu Beginn das Problem, **wo** nach diesen Scheiben zu suchen ist. Um sich selbst am Schopf aus dem Sumpf zu ziehen, existiert zu Beginn des Nukleus an dokumentierten Position eine Tabelle, der diese Information zu entnehmen ist.

Der folgende Assemblercode ist repräsentativ für jedes RTOS–UH auf Motorola–Prozessoren:

```
NukleusStart:
         .DC.L    0,0                          ; LOADABLE MODULE
         .DC.W    $0010
         .DC.B    'NucX.x'

         .DC.L    $800                         ; INITIAL SYS STACK
         .DC.W    NukleusEnd-NukleusStart      ; LEN OF NUKLEUS
         .IROFF                                ; FOR USER - RESET
         BRA.W    ColdStart                    ; FOR RESET
         BRA.W    EXCEPTION_Abort              ; FOR ABORT

Initial_ScanTable:
         .DC.L    $00000001,$0001FFF1          ; FIRST SCAN-RANGE
         .DC.L    0,0                          ; ADDITIONAL SCAN-RANGE
         .DC.L    0                            ; END-MARKER

ColdStart:
         ; Der Kaltstart-Code
         ; ....

WarmStart:
         ; Der Warmstart-Code
         ; ....

EXCEPTION_Abort:
         ; Eine Testroutine, ob ein Warmstart sinnvoll ist
         ; oder ein Kaltstart ausgelöst werden muss!
         ; ....

         ; Der eigentliche Nukleus !

NukleusEnd:
         .END
```

Die mit dem Label `Initial_ScanTable` markierte Tabelle liegt immer `0x20` Bytes hinter dem

Beginn des Nukleus und besteht aus zwei Langwortpaaren mit einem abschliessenden Langwort, das immer den Wert Null enthält. Der Tabelleninhalt gibt quasi den Suchbereich vor, in dem RTOS–UH nach Scheiben forschen soll, um sich zu konfigurieren.

Das erste Langwort eines Pärchens bestimmt jeweils die Startadresse eines Bereiches und das zweite Langwort die Endadresse. Ist das unterste Bit des Langwortes gesetzt, so ist das betreffende Langwort als relativer Offset zum Nukleusbeginn zu interpretieren — ansonsten handelt es sich um eine absolute Adresse im Speicherbereich.

Die Initial–Tabelle besitzt lediglich die Möglichkeit, zwei Bereiche anzugeben. In den meisten Fällen reicht das völlig aus. Es genügt, um dem Nukleus mitzuteilen, wie gross der zu überscannende Bereich ist, der bei der Implementierung vorgegeben wurde und erlaubt es dem Anwendungsprogrammierer, einen zweiten Bereich frei zu definieren, der zusätzlich zu untersuchen ist.

Im Beispiel ist nur ein Bereich definiert, der vom Nukleusbeginn bis 128KB hinter dem Nukleusstart reicht. Korrekt ausgedrückt wird die Adresse, die durch das zweite Langwort spezifiziert wird, nicht mehr zum Test auf einen Scheiben–Beginn `0xAEB1` herangezogen, womit das Beispiel nur einen Scanbereich von 128KB-16Bytes umfassen würde, um Haarspaltern seitenlange Faxe zu ersparen...

Der Nukleus **muss** stets in einem der zu überscannenden Bereiche liegen, weil in ihm grundsätzlich Scheiben enthalten sind, die zum Betrieb des Systems absolut unverzichtbar sind!

Das folgende Beispiel könnte z.B. für ein System sinnvoll sein, das einen Nukleus von 64KB besitzt und bei dem der Anwender zudem den Bereich von `0xD00000` bis `0xDFFFF0` nach Systemkomponenten durchsucht haben möchte.

```
Initial_ScanTable:
         .DC.L    $00000001,$0000FFF1 ; FIRST SCAN-RANGE
         .DC.L    $00D00000,$00DFFFF0 ; ADDITIONAL SCAN-RANGE
         .DC.L    0                   ; END-MARKER
```

Wenn zwei Scanbereiche nicht ausreichen, so besteht für den Anwender (oder Implementator des Gesamt–Betriebssystems) die Möglichkeit, RTOS–UH auf die Suche nach einer neuen Scan–Tabelle zu schicken. Enthalten die ersten beiden Langworte der Initial–Tabelle den Wert Null, so wird das zweite Langwortpärchen dazu verwendet, nach einer Scheibe mit der Definition einer neuen Scan–Tabelle zu suchen. Diese Option ist mit Vorsicht zu verwenden. Findet RTOS–UH in der Kaltstartphase auf der Suche nach einer neuen Scantabelle keine Scheibe mit der Kennung 0 (`.SLICE 0`) im definierten Scanbereich, dann steht stützt das System kommentarlos ab. Aus Anwendersicht passiert eben nichts: keine Startmeldung und der Schirm bleibt absolut leer!

Eine Scheibe zur Definition einer neuen Scan–Tabelle lässt sich bei Verwendung von CREST–C mittels des Kommandos `#pragma SCAN_RANGES` einrichten. Die *Vernichtung* der Originaltabelle wird von diesem Kommando nicht mit übernommen. Sie müssen zu diesem Zwecke die Initial–Scantabelle selbstständig modifizieren — ohne die zwei Null–Langworte auf den Adressen `NukleusStart+0x20` und `NukleusStart+0x24` wird **nicht** nach einer neuen Scantabelle gesucht. Und ohne den korrekten Scanbereich für die neue Scheibe auf den Langworten `NukleusStart+0x28` und `NukleusStart+0x2C` bleibt RTOS–UH schon aus Prinzip schlicht stehen...

Um mal einen Fall durchzuspielen, sei folgendes Szenario gegeben: die Anwenderhardware bestehe aus zwei EPROM–Bänken und zwei Flash–EPROM's. Die erste EPROM-Bank mit 1MB Umfang läge von `0xA0000` bis `0xAFFFFF` und sei von IEP geliefert. Sie enthielte RTOS–UH und die zugehörigen Treiber für die betreffende Hardware — und das komplette Paket von CREST–C–Executables, um die Grösse zu rechtfertigen. Die zweite EPROM–Bank sei nicht so gigantisch und läge von `0xB0000` bis `0xBFFFF` mit 64KB Platz für Anwendersoftware. Weiterhin sollen zwei Flash-Bänke mit jeweils 1MB Speicher ab `0xD0000` und `0xE00000` vollständig beziehungsweise im zweiten Fall nur zur Hälfte überscannt werden.

Unter CREST–C sähe die Codierung wie folgt aus:

```
#pragma SCAN_RANGES 0x000001 0x00FFFF \
                    0x0B0000 0x0BFFF0 \
                    0x0D0000 0x0DFFF0 \
                    0xE00000 0xE07FF0
```

Das Kommando übernimmt die korrekte Codierung der `.SLICE 0` und terminiert die Tabelle mit dem verlangten Null–Langwort.

Weiterhin müsste die Initial–Scantabelle so gepatcht werden, dass der für dieses Kommando von CREST–C generierte Code im Bereich des zweiten Langwortpaares liegt und das erste Langwortpärchen auf Null gesetzt werden. Sie kommen also nicht umhin, das Originalbetriebssystem zu patchen oder sich eine angepasste Variante bei IEP zu ordern — letzteres dürfte für halbwegs aktuelle Systeme per Download über Modem innerhalb weniger Minuten realisierbar sein.

Der Pointer auf die aktuell aktive Scan–Tabelle des laufenden Systems wird von RTOS–UH als Langwort auf der Systemvariable an Adresse `0x934` eingetragen. Die Position des laufenden Nukleus findet sich auf Adresse `0x930`.

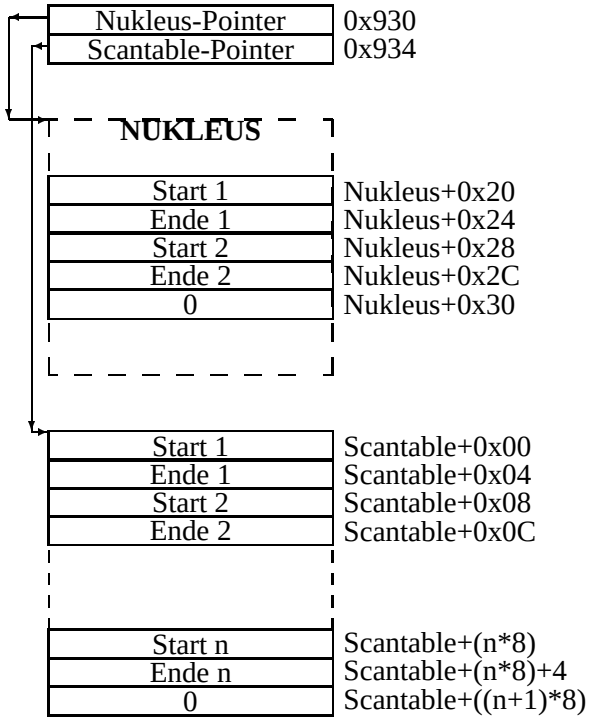


Abbildung 16.1: Scanbereiche

Bei der Angabe eigener Scan–Bereiche sollten tunlichst sinnvolle Adressen verwendet werden. Tritt bei Scheibentests auf den angegebenen Scan–Bereichen ein `BUS ERROR` auf, so läuft das System nicht weiter.

Zu den sinnvollen Bereichen zählt das RAM eines RTOS–UH–Rechners in der Regel eben nicht. Sie sollten sich auf das Überscannen von nichtflüchtigen Bereichen beschränken und dabei möglichst sicherstellen, dass wirklich nur beabsichtigte Systemkomponenten vom Scanner erfasst werden. Irgendwelche Tabellen mit dort gespeicherten Anwenderdaten sind in der Regel unkritisch, weil die Scheibenkennung mit 32 signifikanten Bits `($AEB1BF95)` nicht gerade zu den typischen Speicherinhalten zählt. Gefährlich ist jedoch die Angabe von falsch gewählten RAM–Bereichen, weil dort die Möglichkeit besteht, über gefährliche Artefakte zu stolpern. Wenn eine *gültige* Signalmarke, die noch aus den

Zeiten vor dem letzten Reset stammt, nun als *gültige* Scheibe akzeptiert wird, kann beliebiger Unfug angerichtet werden.

Bereiche zu scannen, die nicht mit sinnvollen Werten besetzt sind, ist eine digitale Abart des russischen Rouletts — wieviele Kammern dabei scharf geladen sind, wird unfairerweise vorher nicht verraten.

Beim Vorgang des Scheibenscannens handelt es sich um einen schlichten Vergleich über den angegebenen Speicherbereich, ob eine Signalmarke (`$AEB1BF95`) auftaucht. Um diesen Vorgang zu beschleunigen, existiert in den meisten RTOS–UH–Implementierungen der sogenannte Quick–Scanner, der diese Aufgabe in der Kaltstartphase einmal vorneweg ausführt und dabei eine Tabelle aus Scheibenadressen aufbaut. Diese Tabelle liegt gleich zu Beginn der dynamischen Systemvariablen und belegt pro aufgefundener Scheibe ein Langwort — eben den Pointer, der hinter die Signalmarke und auf die kodierte Scheibennummer verweist. Der zusätzliche Speicherplatzverbrauch hält sich durch diese Tabelle meist in vernünftigen Grenzen. Ein durchschnittliches Basissystem enthält in der Regel um die 100 Scheiben, was also grob einer Tabellengrösse von einem halben Kilobyte entspricht. Da der Scheibenmechanismus aber auch auf Anwenderebene einsetzbar ist und nicht nur die vom System reservierten Scheibennummern hier abgelegt werden, sind die in Abschnitt 16.3.8 angesprochenen Restriktionen strikt zu beachten!

16.2.3 Ausführen von Kaltstart–Code

Wie Sie unschwer an der führenden Position dieses Abschnittes erkennen können, erfolgt die Ausführung des (benutzerdefinierten) Kaltstart–Codes fast ganz zu Beginn der Systemkonfiguration — oder anders ausgedrückt: es existiert noch kein RTOS–UH! Die Systemvariablen sind grösstenteils uninitialisiert, die Systemtraps noch nicht angeschlossen und Exception–Handler sind auch noch nicht aktiv. Selbstredend existiert auch noch kein Tasking oder gar Tasks. Was man innerhalb einer solchen Kaltstartscheibe kodieren kann, ist nackter Maschinencode. Dem Abschnitt 13.7 sind zusätzliche Erklärungen zu entnehmen, wie und was man in CREST–C unternehmen muss, um derartigen Kaltstartcode erzeugen zu lassen.

In der Regel enthalten ausgelieferte RTOS–UH–Systeme bereits mehrere Kaltstartscheiben, die dazu dienen, die Hardware eines Rechners zu initialisieren. Es sind jedoch auch eine Menge anderer Einsatzmöglichkeiten denkbar. Ein paar Beispiele:

- Das Herunterkopieren des Betriebssystems oder von Anwenderprogrammen ins RAM. Da sich in der Regel der Zugriff auf Eproms viel zeitaufwendiger gestaltet, als ein Lesezugriff auf RAM–Bereiche, wird oft das Verfahren angewendet, die entsprechenden EPROM–Bereiche ins RAM zu kopieren und dort ausführen zu lassen. Geschieht dieser Vorgang allerdings mit dem Betriebssystem selbst, so muss am Ende dieser Kaltstartscheibe ein erneuter Kaltstart ausgelöst werden, um RTOS–UH an seiner neuen Position aufwachen zu lassen.
- In einigen Fällen ist es aus Anwendersicht wünschenswert, eine Überwachung der Kalt– und Warmstarts des Systems mitzuloggen. Die Vorgehensweise besteht dann zumeist darin, im batteriegepufferten RAM eine Verwaltungsstruktur anzulegen, in der bei jedem Kaltstart ein Flag gesetzt wird, dass eine Kaltstartscheibe überlaufen wurde. In einer dazu korrespondierenden Warmstartscheibe — also zu einem Zeitpunkt des Systemstarts, zu dem bereits die meisten Systemdienste zur Verfügung stehen — kann dann in der Verwaltungsstruktur das vorher gesetzte Flag ausgewertet werden. Im Normalbetrieb kann dann mittels einer Anwendertask die Verwaltungsstruktur z.B. auf Platte abgelegt oder über ein Netzwerk weitergeleitet werden.

16.2.4 Scannen nach Systemtasks

16.2.5 Anforderung der Interruptpuffer

16.2.6 Suchen nach Device–Treibern

16.2.7 Installieren der Error–Puffer

16.2.8 Suchen nach Device–Parametern

16.2.9 Scannen nach Shell–Befehlen

16.2.10 Einrichten des verwalteten RAM's

In diesem Betriebszustand sucht RTOS–UH nach Anwendervorgaben bezüglich des zu verwendenden RAM's. *Aber das System arbeitet doch schon mit seinem RAM!?* könnten Sie hier einwenden. Richtig, denn einige Bedingungen müssen prinzipiell erfüllt sein, damit RTOS–UH es bis zu dieser Stelle schafft:

1. Im Bereich von 0x00000008 bis 0x000007FF muss RAM vorhanden sein! Hier liegen Vektoren, die für die CPU/FPU wichtig sind und vom Betriebssystem aufgesetzt werden müssen. Da derartige Zugriffe von RTOS–UH stets im Supervisor–Modus ausgeführt werden, darf dieser Bereich auch von der Hardware gegen User–Mode–Zugriffe geschützt sein.
2. Im Bereich ab 0x0000800 muss User–Mode–taugliches RAM zu finden sein. Die benötigte Grösse hängt stark von der Art des konfigurierenden Systems ab. Mit 8kB dürfte ein Mini–RTOS–UH bereits korrekt anlaufen, bei dem allerdings kaum mehr Anwenderprogramme zu laden sein werden. Realistischer dürfte es sein, von 16 bis 32kB RAM auszugehen, die ein ausgewachsenes RTOS–UH hinter der Adresse 0x00000800 für sich selbst benötigt.

Wo und wieviel RAM ansonsten noch für Applikationsprogramme benötigt wird, ist dagegen konfigurierbar. Dabei ist es nicht notwendig, dass diese Bereiche physikalisch hinter den RTOS–UH–Systemvariablen liegen. Die einzelnen RAM–Kacheln müssen auch keine einheitliche Grösse aufweisen.

Das System sucht beim Hochlaufen nach Scheiben mit der Kennung 12 (`.SLICE 12`) . Es können beliebig viele Scheiben in den Scanbereichen angegeben werden: die **letzte** gefundene Scheibe ist für die Konfiguration relevant! In der Scheibe sind paarweise Start– und Endadressen der einzubindenen RAM–Sektionen aufgeführt, die unter RTOS–UH–Verwaltung gestellt werden sollen. Dabei sind folgende Restriktionen zu beachten:

- Die Startadressen müssen grundsätzlich gerade sein.
- Alle Endadressen (mit Ausnahme der des letzten Bereiches) müssen auf Langwortgrenzen liegen.
- Wird eine ungerade Endadresse vorgegeben, so scannt der Nukleus in 1kB grossen Schritten bis zum Eintritt eines `BUS ERROR` auf das Vorhandensein von RAM. Die letzte Adresse
- Oberhalb der jeweiligen Endadresse müssen 0x20 Bytes RAM für den Nur–Lese–Zugriff verfügbar sein.
- Die Adressbereiche müssen in überlappungsfreier und aufsteigender Reihenfolge notiert werden.

Wie derartige Scheiben mittels CREST–C zu erzeugen sind und was sonst noch alles zu beachten ist, können Sie dem Abschnitt 17.5 entnehmen, der sich mehr mit den praktischen Gesichtspunkten dieser

Scheiben auseinandersetzt.

16.2.11 Aufbau der RTOS–UH–Speicherverwaltung

16.2.12 Einrichten von Modulvariablen–Bereichen

16.2.13 Suchen nach Systemtasks

16.3 Warmstart

Dieser Bereich des Betriebssystem wird in folgenden Fällen ausgeführt:

- Grundsätzlich nach Beendigung der Kaltstartphase.
- Beim Drücken des Abort–Knopfes, wenn die entsprechende Behandlungsroutine zu der Erkenntnis gelangt, dass die Systemvariablem nicht *berbraucht* sind.
- Bei der Verwendung des Befehles `SYSTEM_ABORT` .

Für CREST–C–Programmierer, die das C–Laufzeitsystem nutzen, ist der Abort allerdings reichlich unnütz — man könnte fast sagen: katastrophal! Rechner, auf denen solche CREST–C–Programme exekutiert werden, sollten stets durch einen Reset von ihrem Leiden erlöst werden, wenn gravierende Probleme auftreten. Ein Abort hinterlässt nur noch Fragmente von C–Tasks, die allenfalls zur Erforschung der Umstände dienen können, die zum Crash geführt haben. Ein erneuter Start dieser Taskrümpfe führt **nie** zu erwünschten Erfolg! Eine durch einen Abort abgebrochene C–Task verliert durch den Warmstart wichtige Resourcen — so sind z.B. die allokierten Datenbereiche und der Stack nun wieder als Freispeicher im System eingetragen. Es sei Ihrer Fantasie überlassen, was passiert, wenn eine C–Task erneut gestartet wird, die guten Glaubens mit Speicher rumhantiert, der ihr offiziell nicht mehr gehört...

In der Warmstartphase werden alle Initialisierungen durchgeführt, die zu einem erneuten Aufsetzen eines abgestürzten Betriebssystems sinnvoll erscheinen. Verklemmte Hardeware, Dauerläufer im Tasking, abgeschossene Bedieninterfaces — die Liste der Fehler ist schlicht endlos, die einen Abort sinnvoll erscheinen lassen kann.

Auch hier kann der Anwender Maschinencode, der ohne die meisten Systemunterstützung lauffähig ist, zur Ausführung bringen. Entsprechender Code kann mittels des Kommandos `#pragma WARMSTART` dem Betriebssystem hinzugefügt werden.

Auch in der Warmstartphase läuft RTOS–UH auf Supervisor–Mode unter Interruptsperre auf Level 7 — Sie sollten auch hier nicht versuchen, diesen Zustand in einer Warmstartscheibe zu verändern...

16.3.1 Löschen der Interruptbuffer

16.3.2 Initialisieren der Vektortabellen

16.3.3 Bestimmung der CPU/FPU

16.3.4 Initialisierung der Exceptionhandler

16.3.5 Aufsetzen der Dispatcherkette

16.3.6 Initialisieren der I/O–Queues

16.3.7 Anlegen der residenten Taskworkspaces

16.3.8 Initialisierung der Error–Puffer

16.3.9 Ausführen von Warmstart–Code

16.3.10 Starten des Normalbetriebes

16.4 Normalbetrieb

Wenn RTOS–UH sich komplett konfiguriert hat, sind folgende Minimalvoraussetzungen erfüllt:

- Die Vektortabellen des Prozessors sind initialisiert und verweisen auf entsprechende Behandlungsroutinen im Nukleus.
- Die RTOS–UH–Traps — also die Aufrufmöglichkeit von Systemfunktionen — sind installiert.
- Ein Hardware–Timer liefert äquidistante Interrupts — quasi das Herz von RTOS–UH. Auf MC68K–Plattformen ist dieses Intervall auf 1 Millisekunde eingestellt. Alle Millisekunde wird folglich eine Interruptroutine durchlaufen, die kontrolliert, ob Aktionen bezüglich des Taskings vorzunehmen sind.
- Es existiert mindestens eine Task: die Idle–Task `#IDLE`! Wenn keine anderen Tasks Prozessorzeit benötigen, so lässt das System eben diese Task laufen. Sie exekutiert mit der absolut niedrigsten Priorität eine Endlosschleife.
- Weiterhin existiert eine Error–Task `#ERROR`, die immer dann aufgerufen wird, wenn Fehlermeldungen auszugeben sind. Diese Task läuft sehr hochprior — also mit negativer Priorität — und Anwendertasks sollten es tunlichst unterlassen, mit noch höherer Priorität laufen zu wollen.

Wenn dieser Punkt erreicht ist, dann stehen Ihnen die bekannten Dienste von RTOS–UH zur Verfügung. Erst jetzt wird das Tasking und die Bearbeitung von Interrupts zugelassen. Erst können alle Traps und Systemfunktionen genutzt werden.

Kapitel 17

Der Umgang mit Scheiben

In den folgenden Abschnitten soll erläutert werden, welche zusätzlichen Features von CREST–C–Programmen genutzt werden können bzw. wie Programme aufgebaut sein müssen, die auf das C–Laufzeitsystem verzichten wollen.

17.1 Scheiben–Scannen

Wenn Sie ein kleines CREST–C–Programm geschrieben haben, in dem sich eine Warmstart–Scheibe befindet, werden Sie nach schnellem Laden und einem anschliessenden `ABORT` frustriert feststellen, dass sich eigentlich nichts getan hat. Der Grund ist dann nicht in einem Fehler des RTOS–UH oder des Compilers zu suchen. Der Witz besteht darin, dass nur Scheiben gefunden werden, die sich im Abtastbereich des Scanners befinden — und auf gerade Adressen liegen!

Unter RTOS–UH besteht auch auf Nutzerebene die Möglichkeit, nach Scheiben an Hand von Signalmarken (`$AEB1BF95`) zu fahnden. CREST–C stellt dazu die zwei Funktionen `rt_scan_first()` und `rt_scan_next()` zur Verfügung.

```
typedef struct ScanSave { ULONG _D7, _A1, _A6 ; } ScanSave ;

void  *rt_scan_first( ULONG  slice, ScanSave  *last ) ;
void  *rt_scan_next (               ScanSave  *last ) ;
```

Beim Aufruf von `rt_scan_first()` muss die Scheibennummer und ein Pointer auf einen kleinen Zwischenpuffer angegeben werden. Im Erfolgsfall erhält man einen Zeiger, der hinter den Scheibenkopf auf die jeweiligen Nutzdaten zeigt. Im nachfolgenden Beispiel wird dieses Vorgehen an Hand eines kleinen Scheiben–Scanners demonstriert, der das System nach Headertexten durchsucht und diese zeilenweise ausgibt.

```
void ZeigeHeader( void )
{
   ScanSave     temp ;
   signed char *ptr  ;

   if ( ( ptr = rt_scan_first( 16, &temp ) ) != NULL )
   {
      do
      {
```

```
            while ( *ptr > 0 ) printf( "%c", *ptr++ ) ;
            printf( "\n" ) ;
         }
            while ( ( ptr = rt_scan_next( &temp ) ) != NULL ) ;
      }
   }
```

17.2 Ausblenden von Scanbereichen

Bei der Verwendung von Gerätetreibern ist es in vielen Fällen sehr nützlich, nur diejenigen Treiber bei der Systemkonfiguration vom RTOS–UH einbinden zu lassen, für die auch reale Hardware im Rechner verfügbar ist. Wenn Sie z.B. maximal fünf A/D–Karten in einem Rechner vorgesehen haben, die durch Betreuungstasks verwaltet werden, dann hagelt es `BUS-ERROR`'s, wenn eine dieser Karten gerade nicht im Gehäuse steckt und die Betreuungstask ins Leere greift. Es kann auch nicht Sinn der Angelegenheit sein, jeder Betreuungstask die Aufgabe ans Bein zu binden, das Vorhandensein der zu betreuenden Hardware ständig erneut abzutesten. Kartenschrott, von dem man von Beginn an weiss, dass der Kram stochastisch im laufenden System abschmiert, hat eigentlich nichts in industriellen Anlagen verloren...

Für den einmaligen Vorabtest beim Hochlaufen des Systems existieren unter RTOS–UH die Skip–Slices mit der Nummer `-1`. Der Maschinencode direkt hinter der Signalmarke einer solchen Scheibe wird bei der Systemkonfiguration ausgeführt und in Abhängigkeit des Rückgabewertes wird ein zu definierender Bereich bei der Systemkonfiguration ignoriert oder als gültiger Scanbereich akzeptiert. Auf diese Weise lassen sich z.B. Kaltstartscheiben und Systemtasks, die wegen fehlender Hardware nur Unfug anstellen würden, komplett ausblenden. Sie können beim Systemstart üblicherweise an Hand der ausgegebenen Headertexte feststellen, welcher Treiber vom System eingebunden wurde.

Unter CREST–C kann dieser Mechanismus recht simpel mittels der zwei `#pragma`–Anweisungen `START_SLICE_SKIP` und `END_SLICE_SKIP` ausgenutzt werden. Dabei sind einige Konventionen einzuhalten. Direkt hinter jedem `#pragma START_SLICE_SKIP` wird eine Testerfunktion ohne Parameter und mit Rückgabetyp `int` erwartet. Ähnlich zu den Kalt– und Warmstartscheiben (siehe Abschnitt 13.7) ist das Spielen mit Variablen hier nur sehr eingeschränkt möglich, da das System sich erst konfiguriert und viele Dienste noch nicht zur Verfügung stehen.

Antwortet die Testerfunktion mit Null, so hat die Skip–Slice keinerlei Wirkung. Wird ein Wert ungleich Null geliefert, so überspringt RTOS–UH bei der Suche nach weiteren Scheiben den Bereich, dessen Ende durch die Angabe von `#pragma END_SLICE_SKIP` definiert wird.

```
#pragma START_SLICE_SKIP
int test( void ) { ... }

// beliebige Treiber etc...
#pragma HEADER "Bin drin!!"
#pragma END_SLICE_SKIP
```

CREST–C verwaltet dabei bis zu 16 ineinandergeschachtelte „`START_SLICE_SKIP`/`END_SLICE_SKIP`“–Paare. Sie sollten es sich zur Angewohnheit machen, bei derartig eingeblendeten Bereichen, aussagekräftige Headertexte hinter der Testerfunktion einzutragen, wie es in Abschnitt 17.3 erläutert wird.

17.3 Headertexte beim Systemstart

Sie kennen ohne Zweifel den Effekt, dass RTOS–UH bei jedem Hochlaufen den Bildschirm mit diversen Meldungen vollschreibt. Diese Angewohnheit ist recht sinnvoll, um zu überprüfen, ob die Selbstkonfiguration des Systems auch geklappt hat und welche Versionen der einzelnen Komponenten verfügbar sind. Wenn Sie das Betriebssystem um eigene Treiber und Programme erweitern, sollten Sie ebenso verfahren. Mittels der Anweisung `#pragma HEADER` können Sie die Einschaltmeldungen um die Aufzählung der eigenen Grosstaten bereichern. Die Syntax entspricht der von normalen C–Strings. Sonderzeichen und Escape–Sequences können wie üblich angegeben werden. Den Zeilenumbruch und triviale Formatierungen übernimmt RTOS–UH automatisch, sofern die Meldungen kurz genug gehalten sind. Wenn die Headertexte mehr als 15 druckbare Zeichen enthalten, sollten der Lesbarkeit halber Zeilenumbrüche eingefügt werden.

```
#pragma HEADER  "CCC=2.60-1"
```

Das Beispiel führt allerdings nur zu der erwünschten Ausgabe, wenn Sie die Scheibe im Scanbereich des RTOS–UH plaziert haben und der Ort, an dem die Scheibe auftaucht nicht durch eine explizite Skipperslice ausgeblendet wurde, wie es in Abschnitt 17.2 erläutert ist.

17.4 Modulkopf generieren

Beim Verzicht auf Startup–Dateien im Linkfile eines Projektes ist es zwingend notwendig, zu Beginn des zu erzeugenden S–Records einen geeigneten `MemSectionHeader` einzufügen, wenn man den generierten S–Record später laden möchte.

Um einen Modulkopf entsprechend den RTOS–UH–Konventionen zu generieren, wurde die Anweisung `#pragma MODULE` implementiert. Sinn macht dieses Kommando nur, wenn es **vor dem Beginn der eigentlichen Codegenerierung** verwendet wird. Das bedeutet, dass es so ziemlich in die erste Zeile der C–Datei gehört, die im Linkfile des Projektes ganz vorne steht. Geprüft wird lediglich die Länge des Bezeichners, die 23 Zeichen nicht überschreiten darf.

Der triviale Anwendungszweck besteht darin, z.B. PEARL–Interfaceroutinen ladbar zu machen oder einen Modulkopf für eine *Shared Library* zu generieren, die fürs RAM übersetzt wurde. Bei S–Records, die immer und grundsätzlich im EPROM verschwinden, sind Modulköpfe lediglich Speicherplatzverschwendung und erfüllen keinerlei Zweck.

Als Beispiel soll ein kleines Unterprogramm dienen, dass von PEARL aus aufgerufen werden kann und einen übergebenen Integerwert um Eins inkrementiert und als Funktionsergebnis zurückliefert:

1. Das C–Programm `inc.c` besteht lediglich aus dem Modulkopf und der Funktion selbst.

   ```
   #include <stdio.h>
   #pragma MODULE "INC"
   Fixed15  Inc( Fixed15  value ) { return( value+1 ) ; }
   ```

2. Das Makefile `inc.mak` übersetzt den einzelnen Quelltext und erzeugt einen ladbaren S–Record.

   ```
   inc.sr: inc.obj
      cln inc.lnk inc.sr -0 -M

   inc.obj: inc.c
      ccc inc.c inc.obj -0 -H=$(INCLPATH)
   ```

3. Das Linkfile `inc.lnk` ist ebenfalls nur ein Einzeiler, da in diesem trivialen Falle nun wirklich

keine Funktionen aus Bibliotheken benötigt werden.

`inc.obj`

4. Der erzeugte S–Record `inc.sr` ist nun normal ladbar und stellt sich in der Speicherkette als `MDLE INC` dar.

17.5 RAM–Scheiben generieren

Wie dem Abschnitt 16.2.10 zu entnehmen war, dienen die RAM–Scheiben zur Konfiguration des unter RTOS–UH–Verwaltung befindlichen RAM's — oder um es klar auszudrücken: RAM, das hier **nicht** aufgeführt ist, wird von RTOS–UH weder verwendet, initialisiert noch kaputtgeschrieben!

In CREST–C wurde die Anweisung `#pragma RAM_RANGES` implementiert, um diese Scheiben auch auf Hochsprachenebene generieren zu können. Im einfachsten und üblichsten Fall beginnt der RAM–Bereich eines Rechners bei 0x00000000 und bildet einen linearen Adressraum entsprechend der Kapazität und Anzahl der verwendeten Bausteine — z.B. bei 512 kB hinauf bis 0x0007FFFF. Um den kompletten RAM–Speicher unter RTOS–UH–Verwaltung zu stellen, genügt es, das folgende Kommando in den Quelltext aufzunehmen und das resultierende Programm im Scanbereich des Rechners abzulegen.

```
#pragma RAM_RANGES  0x800 0x7FFE0
```

Achtung: Bevor Sie nun blindwütig und voller Tatendrang eine RAM–Scheibe generieren und sich wundern, weshalb das System damit nicht mehr hochkommt oder sich völlig anders verhält als vor dem Eingriff, sollten Sie sich stets vor Augen halten, dass in einem laufenden RTOS–UH bereits grundsätzlich eine RAM–Scheibe bei der Implementierung vorgegeben wurde. Wenn ihre RAM–Scheibe als **letzte** beim Systemstart gefunden wird, so wird die Original–Scheibe gnadenlos übersteuert!

In vielen ausgelieferten RTOS–UH–Systemen werden die RAM–Scheiben aber bereits dazu verwendet, um bestimmte Bereiche für Spezial–Betriebssysteme auszublenden. Oft genügt schon ein schneller Blick auf die mittels `Ctrl-A S` erzeugte Speicherliste, um Kuriositäten zu identifizieren:

```
00002D42->00002D4C  MARK
00002D4C->00002DAE  ATSK  Resident  #IDLE
00002DAE->00002E10  ATSK  Resident  #ERROR
00002E10->00002E72  TASK  Resident  #EDFMN
00002E72->00002ED4  TASK  Resident  #VDATN
00002ED4->00002F36  TASK  Resident  #XCMMD
00002F36->00002F98  TASK  Resident  #NIL
00002F98->00002FFA  TASK  Resident  #UAR_1
00002FFA->0000305C  TASK  Resident  #UDU_1
0000305C->000030BE  TASK  Resident  #UAR_2
000030BE->00003120  TASK  Resident  #UDU_2
00003120->00003182  TASK  Resident  #USER1
00003182->000031E4  TASK  Resident  #USER2
000031E4->00003246  TASK  Resident  #USERT
00003246->000032A8  TASK  Resident  #V_DUP
000032A8->00003F0A  ATSK  Resident  #VTERM
00003F0A->0007FDF6  FREE
0007FDF6->00000000  MARK
```

Dieser Speicherliste stammt von einem leeren 512kB–Rechner, bei dem offensichtlich 512 Bytes am

oberen Ende des verfügbaren Speichers weggeschnitten wurden — für Profis an der letzten Zeile ablesbar, wo bei einem *normalen* System:

```
0007FFD6->00000000  MARK
```

zu erwarten gewesen wäre. Mittels folgender paar Programmzeilen lässt sich feststellen, wie und welche RAM–Scheiben vom System erkannt wurden.

```
void ZeigeScanRanges( void )
{
   ScanSave   temp ;
   ULONG     *ptr  ;

   if ( ( ptr = rt_scan_first( 12, &temp ) ) != NULL )
   {
      do
      {
         printf( "%08lX:\n", ptr ) ;
         while ( *ptr > 0 )
         {
            printf( "         %08lX %08lX\n", ptr[ 0 ], ptr[ 1 ] ) ;
            ptr += 2 ;
         }
         printf( "\n" ) ;
      }
         while ( ( ptr = rt_scan_next( &temp ) ) != NULL ) ;
   }
}
```

Im vorliegenden Fall liefert das Programm die folgende Ausgabe:

```
00D113AC:
          00000800 0007FE00
```

Soll heissen: an der Adresse 0x00D113AC liegen die Nutzdaten einer 12er–Scheibe und der RAM–Bereich ist von 0x800 bis 0x7FDFF spezifiziert, wie sich mittels `DM` leicht verifizieren lässt:

```
*DM D113AC-6
00D113A6: AEB1 BF95 039D 0000 0800 0007 FE00 0000  ................
```

Bei der Implementierung des hier betrachteten Systems wurden folglich bereits 512 Bytes vor RTOS–UH in Sicherheit gebracht, was z.B. mittels des Kommandos:

```
#pragma RAM_RANGES  0x800 0x7FE00
```

zu erreichen ist. Um weiteren Speicher für eigene Nutzung abzuschneiden, reicht es in der Regel aus, die Endadresse einer bereits im System vorhandenen RAM–Scheibe um die entsprechende Anzahl von Bytes herabzusetzen und eine eigene angepasste Scheibe im Scanbereich unterzubringen.

Es besteht allerdings auch die Möglichkeit, gewollt Löcher in den linearen Speicher zu stanzen. So bewirkt das folgende Kommando, dass der Bereich von 0x40000 bis 0x4FFFF der RTOS–UH–Verwaltung entzogen wird.

```
#pragma RAM_RANGES  0x00800 0x40000  \
                    0x50000 0x7FE00
```

Die resultierende Speicherliste hat dann folgendes Aussehen:

```
00002D4E->00002D58  MARK
00002D58->00002DBA  ATSK  Resident  #IDLE
00002DBA->00002E1C  ATSK  Resident  #ERROR
00002E1C->00002E7E  TASK  Resident  #EDFMN
00002E7E->00002EE0  TASK  Resident  #VDATN
00002EE0->00002F42  TASK  Resident  #XCMMD
00002F42->00002FA4  TASK  Resident  #NIL
00002FA4->00003006  TASK  Resident  #UAR_1
00003006->00003068  TASK  Resident  #UDU_1
00003068->000030CA  TASK  Resident  #UAR_2
000030CA->0000312C  TASK  Resident  #UDU_2
0000312C->0000318E  TASK  Resident  #USER1
0000318E->000031F0  TASK  Resident  #USER2
000031F0->00003252  TASK  Resident  #USERT
00003252->000032B4  TASK  Resident  #V_DUP
000032B4->00003F16  ATSK  Resident  #VTERM
00003F16->0003FFF0  FREE
0003FFF0->00050000  MDLE  #NORAM
00050000->0007FDF6  FREE
0007FDF6->00000000  MARK
```

Für derart ausgeblendete Bereiche richtet RTOS–UH ein Modul mit dem Namen `#NORAM` ein, in dem — wie der Name schon aussagt — kein RAM verwaltet wird. Dieser Speicher ist programmtechnisch von Applikationen über Pointer erreich– und verwaltbar, liegt aber ausserhalb der Verantwortung des Betriebssystems. Auf Grund des Doppelkreuzes im Namen sind derartige `#NORAM`–Module nicht mehr entladbar und die darin enthaltenen Daten überstehen problemlos jeden Reset und sogar — wenn es sich um batteriegepuffertes RAM handelt — einen Stromausfall.

Anhang A

Mathematische Funktionen

Die Erstellung guter mathematischer Funktionen ist harte Arbeit. Auch heute noch ist es in kommerziellen Compilern nahezu üblich, auf grobe Fehler bei der Implementierung von Fliesskommafunktionen zu stossen. Auch der CREST–C–Compiler war und ist davon leider trotz aller Bemühungen sicherlich nicht frei.

Der CREST–C bietet die Möglichkeit zum Einsatz der Arithmetikprozessoren MC68881, MC68882 und der internen FPU im MC68040 bzw. MC68060.

A.1 Fliesskommadarstellung

Sowohl bei den Fliesskommaprozessoren, als auch in der Fliesskomma–Emulation innerhalb des CREST–C–Paketes, werden Zahlendarstellungen entsprechend `IEEE`–754 verwendet. Dies gewährleistet den problemlosen Austausch von Binärdaten zwischen CREST–C–Programmen mit und ohne FPU–Unterstützung. Unterstützt werden drei unterschiedliche Fliesskommaauflösungen:

1. Single Precision
2. Double Precision
3. Extended Precision

Diese unterscheiden sich sowohl in der Mantissengenauigkeit, als auch im Bereich des darzustellenden Exponenten. Um böse Überraschungen bezüglich Rechenzeit, Speicherbedarf und Rechengenauigkeit zu vermeiden, sind in den folgenden Abschnitten einige **wesentliche** und **grundlegende** Informationen zusammengefasst, um durch die richtige Auswahl der Mittel zum Ziel zu gelangen.

A.1.1 Single Precision

Bei dieser Fliesskommauflösung handelt sich um den kleinsten Datentyp. Er belegt 32 Bit im Speicher und ist unter CREST–C als Datentyp `float` anzusprechen. Die interne Darstellung enthält 24 Bit Mantisse (ein Bit davon ist nur implizit enthalten), 8 Bit Exponent und ein Vorzeichenbit.

Die `float`–Mantisse bietet eine maximale Auflösung von etwas mehr als 7 Dezimalstellen (exakt: $ln(2)/ln(10) * 24$). Die grösste darstellbare Zahl beträgt `3.402823E+0038F`; die kleinste darstellbare Zahl entsprechend `2.938736E-0039F`. Die kleinstmögliche Annäherung an die Zahl `1` beträgt `5.960464E-0008F`. Die entsprechenden Konstanten sind in der Includedatei <`float.h`> als Makros abgelegt.

```
#define FLT_MANT_DIG                        24
#define FLT_DIG                              7
#define FLT_EPSILON              5.960464E-0008F
#define FLT_MIN                  2.938736E-0039F
#define FLT_MAX                  3.402823E+0038F
```

A.1.2 Double Precision

Bei dieser Fliesskommauflösung handelt sich um einen Kompromiss aus Speicherplatzbedarf und Rechnengenauigkeit. Er belegt 64 Bit im Speicher und ist unter CREST–C als Datentyp `double` anzusprechen. Gleichzeitig handelt es sich beim Datentyp `double` um den *default*–Typen für Fliesskommaangaben. Die interne Darstellung enthält 53 Bit Mantisse (ein Bit davon ist nur implizit enthalten), 11 Bit Exponent und ein Vorzeichenbit.

Die `double`–Mantisse bietet eine maximale Auflösung von knapp 16 Dezimalstellen (exakt: $ln(2)/ln(10)*53$). Die grösste darstellbare Zahl beträgt `1.797693134862315E+0308`; die kleinste darstellbare Zahl entsprechend `5.562684646268003E-0309`. Die kleinstmögliche Annäherung an die Zahl `1` beträgt `1.110223024625156E-0016`. Die entsprechenden Konstanten sind in der Includedatei <`float.h`> als Makros abgelegt.

```
#define DBL_MANT_DIG                        53
#define DBL_DIG                             16
#define DBL_EPSILON      1.110223024625156E-0016
#define DBL_MIN          5.562684646268003E-0309
#define DBL_MAX          1.797693134862315E+0308
```

A.1.3 Extended Precision

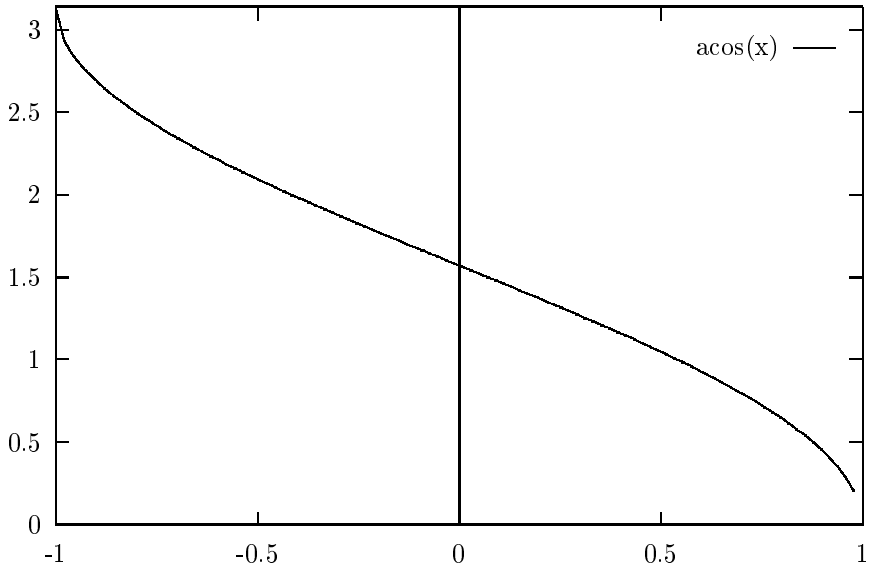
Bei dieser Fliesskommauflösung handelt sich um den grössten Datentyp. Er belegt 96 Bit im Speicher und ist unter CREST–C als Datentyp `long double` anzusprechen. Die interne Darstellung enthält 64 Bit Mantisse, 15 Bit Exponent und ein Vorzeichenbit. Zusätzlich sind 16 Bit in der internen Darstellung vorhanden, die keinerlei Nutzinformationen, sondern stets ein Nullmuster beinhalten.

Eine Mantisse vom Datentyp `long double` bietet eine maximale Auflösung von etwas mehr als 19 Dezimalstellen (exakt: $ln(2)/ln(10) * 64$). Die grösste darstellbare Zahl beträgt `1.189731495357231764E+4932L`; die kleinste darstellbare Zahl entsprechend `1.681051571556046753E-4932L`. Die kleinstmögliche Annäherung an die Zahl `1` beträgt `5.421010862427522170E-0020L`. Die entsprechenden Konstanten sind in der Includedatei <`float.h`> als Makros abgelegt.

```
#define LDBL_MANT_DIG                       64
#define LDBL_DIG                            19
#define LDBL_EPSILON 5.421010862427522170E-0020L
#define LDBL_MIN     1.681051571556046753E-4932L
#define LDBL_MAX     1.189731495357231764E+4932L
```

A.2 Trigonometrische Funktionen

A.2.1 acos()



Syntax:

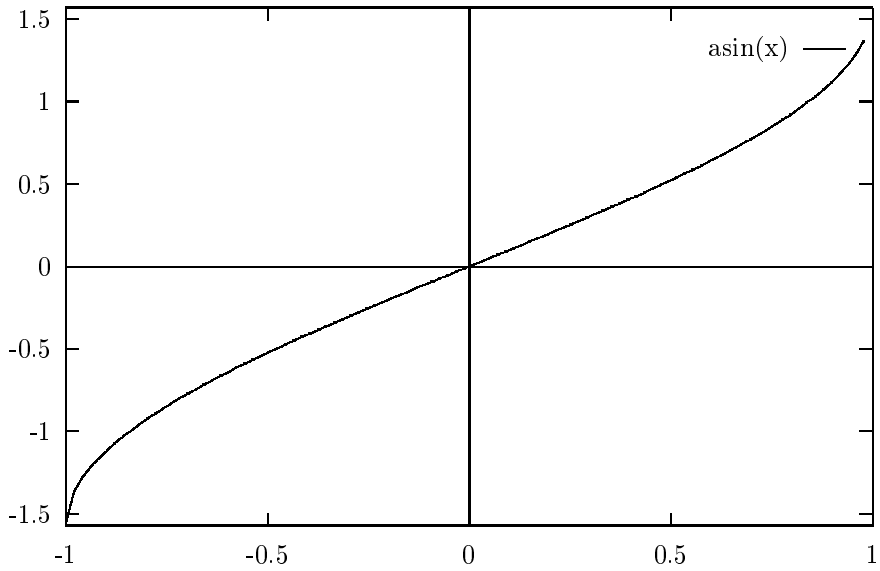
```
     float   facos(      float  x ) ;
     double   acos(      double x ) ;
long double  lacos( long double x ) ;
```

Die Funktion `acos(`*x*`)` berechnet den Arcus–Cosinus (im Bogenmass) des Argumentes *x*.

Wertebereich und Fehler:
Wird als Parameter *x* der Wert *NaN* oder $\pm$*Inf* angegeben, so wird ein *domain error* durch das Setzen der Variable `errno` auf *EDOM* angezeigt. Die Funktion liefert den Parameter *x* zurück.

Liegt das Argument *x* ausserhalb des Wertebereiches von -1 bis +1, so liefert die Funktion eine Null als Funktionsergebnis und setzt die Variable `errno` auf *EDOM*.

A.2.2 asin()



Syntax:

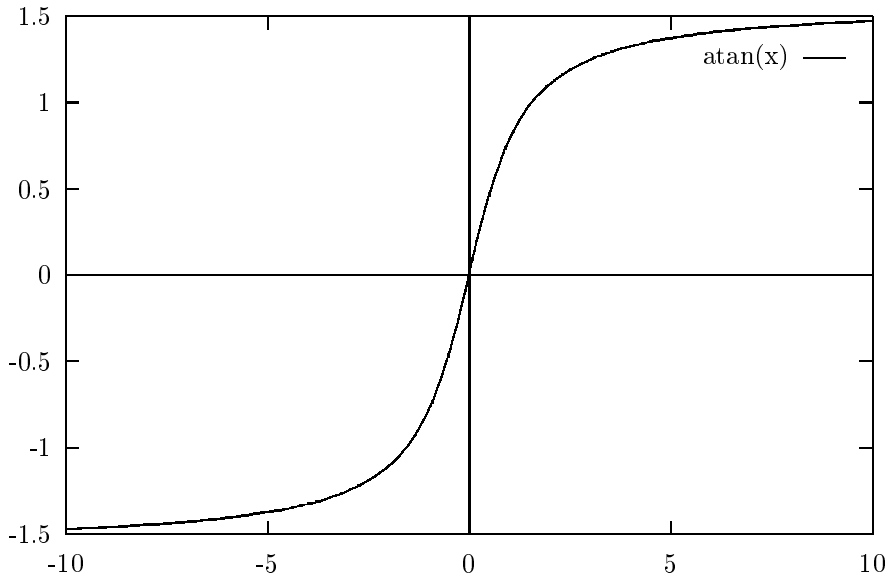
```
     float   fasin(      float  x ) ;
     double   asin(      double x ) ;
long double  lasin( long double x ) ;
```

Die Funktion `asin(`*x*`)` berechnet den Arcus–Sinus (im Bogenmass) des Argumentes *x*.

Wertebereich und Fehler:
Wird als Parameter *x* der Wert *NaN* oder $\pm$*Inf* angegeben, so wird ein *domain error* durch das Setzen der Variable `errno` auf *EDOM* angezeigt. Die Funktion liefert den Parameter *x* zurück.

Liegt das Argument *x* ausserhalb des Wertebereiches von -1 bis +1, so liefert die Funktion eine Null als Funktionsergebnis und setzt die Variable `errno` auf *EDOM*.

A.2.3 atan()



Syntax:

```
     float   fatan(      float  x ) ;
     double   atan(      double x ) ;
long double  latan( long double x ) ;
```

Die Funktion `atan(`*x*`)` berechnet den Arcus–Tangens (im Bogenmass) des Argumentes *x*.

Wertebereich und Fehler:
Wird als Parameter *x* der Wert *NaN* oder $\pm$*Inf* angegeben, so wird ein *domain error* durch das Setzen der Variable `errno` auf *EDOM* angezeigt. Die Funktion liefert den Parameter *x* zurück.

A.2.4 atan2()

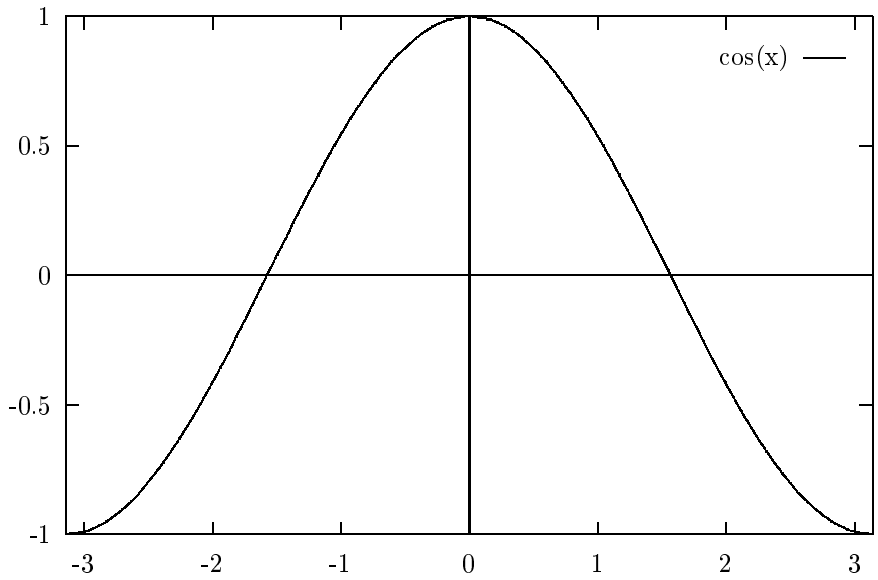
Syntax:

```
     float   fatan2(      float   y,      float  x ) ;
     double   atan2(      double  y,      double x ) ;
long double  latan2( long double  y, long double x ) ;
```

Die Funktion `atan2(`*y*`, `*x*`)` berechnet den Winkel im Bogenmass, dessen Tangens dem Quotienten *y/x* entspricht. Das Resultat liegt im Bereich von $-\pi$ bis $+\pi$. Im Unterschied zur Funktion `atan()` liefert `atan2()` zusätzlich den korrekten Quadranten des Ergebnisses.

Wertebereich und Fehler:
Wird als Parameter *x* oder *y* der Wert *NaN* oder $\pm$*Inf* angegeben, so wird ein *domain error* durch das Setzen der Variable `errno` auf *EDOM* angezeigt. Die Funktion liefert *NaN* zurück.

A.2.5 cos()



Syntax:

```
     float   fcos(      float  x ) ;
     double   cos(      double x ) ;
long double  lcos( long double x ) ;
```

Die Funktion `cos(`*x*`)` berechnet den Cosinus des Argumentes *x*, das im Bogenmass anzugeben ist. Bei

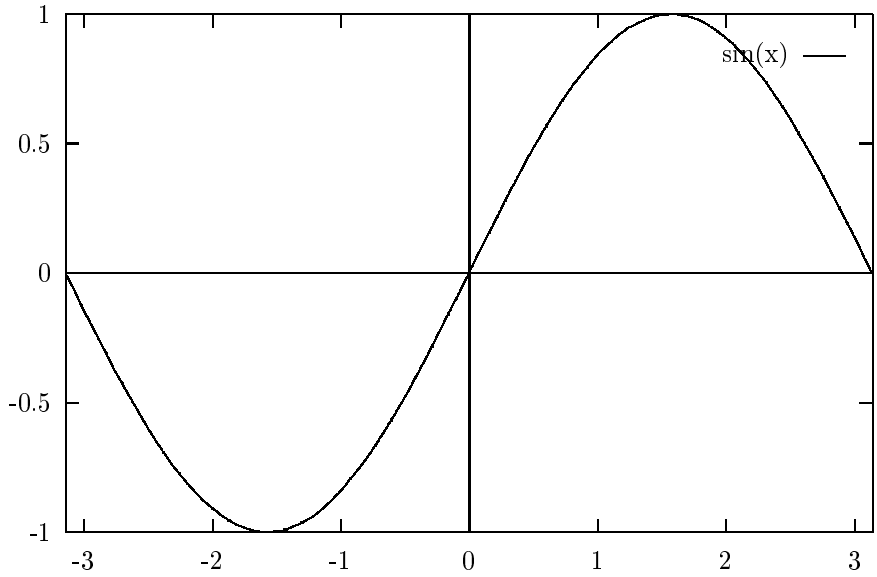
grossen Argumenten verlieren die Resultate an Genauigkeit.

Wertebereich und Fehler:
Wird als Parameter *x* der Wert *NaN* oder $\pm$*Inf* angegeben, so wird ein *domain error* durch das Setzen der Variable `errno` auf *EDOM* angezeigt. Die Funktion liefert den Parameter *x* zurück.

Theoretisch deckt die Funktion `cos(`*x*`)` den gesamten Bereich der Rationalen Zahlen ab. Verursacht durch die begrenzte Auflösung der Fliesskommadarstellungen verliert der Parameter *x* jedoch bereits bei der Reduktion auf den internen Wertebereich von $-\pi \leq x \leq +\pi$ an Genauigkeit. Entsprechend ungenauer werden die Funktionsergebnisse.

Beim Einsatz von `float`–Arithmetik liegt die Grenze, ab der das Resultat definitiv keinerlei erkennbaren Zusammenhang mehr mit dem Aufrufparameter hat, bei 10^7; entsprechend bei `double`–Arithmetik 10^{16} und bei `long double`–Rechnung 10^{19}. Die Funktionen liefern in diesem Fall eine Null als Funktionsergebnis und zeigen den Verlust sämtlicher signifikanter Stellen durch das Setzen von `errno` auf *EDOM*.

A.2.6 sin()



Syntax:

```
     float   fsin(      float  x ) ;
     double   sin(      double x ) ;
long double  lsin( long double x ) ;
```

Die Funktion `sin(`*x*`)` berechnet den Sinus des Argumentes *x*, das im Bogenmass anzugeben ist. Bei grossen Argumenten verlieren die Resultate an Genauigkeit.

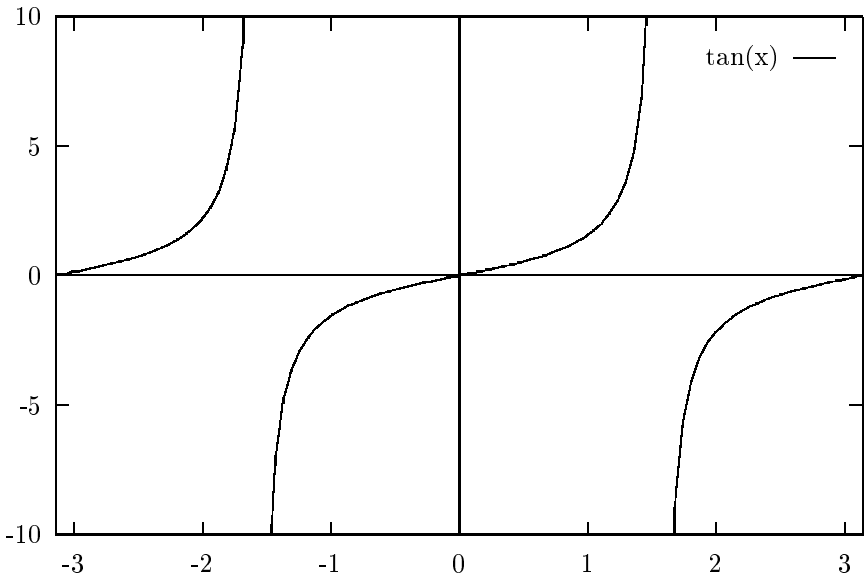
Wertebereich und Fehler:
Wird als Parameter *x* der Wert *NaN* oder $\pm$*Inf* angegeben, so wird ein *domain error* durch das Setzen der Variable `errno` auf *EDOM* angezeigt. Die Funktion liefert den Parameter *x* zurück.

Theoretisch deckt die Funktion `sin(`*x*`)` den gesamten Zahlenbereich der Rationalen Zahlen ab. Verursacht durch die begrenzte Auflösung der Fliesskommadarstellungen verliert der Parameter *x* jedoch bereits bei der Reduktion auf den internen Wertebereich von $0 \leq x \leq 2 * \pi$ an Genauigkeit. Entsprechend ungenauer werden die Funktionsergebnisse.

Beim Einsatz von `float`–Arithmetik liegt die Grenze, ab der das Resultat definitiv keinerlei erkennba-

ren Zusammenhang mehr mit dem Aufrufparameter hat, bei 10^7; entsprechend bei `double`–Arithmetik 10^{16} und bei `long double`–Rechnung 10^{19}. Die Funktionen liefern in diesem Fall eine Null als Funktionsergebnis und zeigen den Verlust sämtlicher signifikanter Stellen durch das Setzen von `errno` auf *EDOM*.

A.2.7 tan()



Syntax:

```
     float   ftan(      float  x ) ;
     double   tan(      double x ) ;
long double  ltan( long double x ) ;
```

Die Funktion `tan(`*x*`)` berechnet den Tangens des Argumentes *x*, das im Bogenmass anzugeben ist. Bei grossen Argumenten verlieren die Resultate an Genauigkeit.

Wertebereich und Fehler:
Wird als Parameter *x* der Wert *NaN* oder $\pm$*Inf* angegeben, so wird ein *domain error* durch das Setzen der Variable `errno` auf *EDOM* angezeigt. Die Funktion liefert den Parameter *x* zurück.

In der Nähe der Polstellen ($\pm\pi/2$, $\pm 3 * \pi/2$,...) kann es zu Überläufen bei der jeweils ausgewählten Fliesskommaauflösung kommen.

Theoretisch deckt die Funktion `tan(`*x*`)` den gesamten Zahlenbereich der Rationalen Zahlen ab. Verursacht durch die begrenzte Auflösung der Fliesskommadarstellungen verliert der Parameter *x* jedoch bereits bei der Reduktion auf den internen Wertebereich von $0 \leq x \leq 2 * \pi$ an Genauigkeit. Entsprechend ungenauer werden die Funktionsergebnisse.

Beim Einsatz von `float`–Arithmetik liegt die Grenze, ab der das Resultat definitiv keinerlei erkennbaren Zusammenhang mehr mit dem Aufrufparameter hat, bei 10^7; entsprechend bei `double`–Arithmetik 10^{16} und bei `long double`–Rechnung 10^{19}. Die Funktionen liefern in diesem Fall eine Null als Funktionsergebnis und zeigen den Verlust sämtlicher signifikanter Stellen durch das Setzen von `errno` auf *EDOM*.

A.3 Hyperbolische Funktionen

A.3.1 acosh()

Syntax:

```
     float    facosh(      float  x ) ;
     double    acosh(      double x ) ;
long double   lacosh( long double x ) ;
```

A.3.2 asinh()

Syntax:

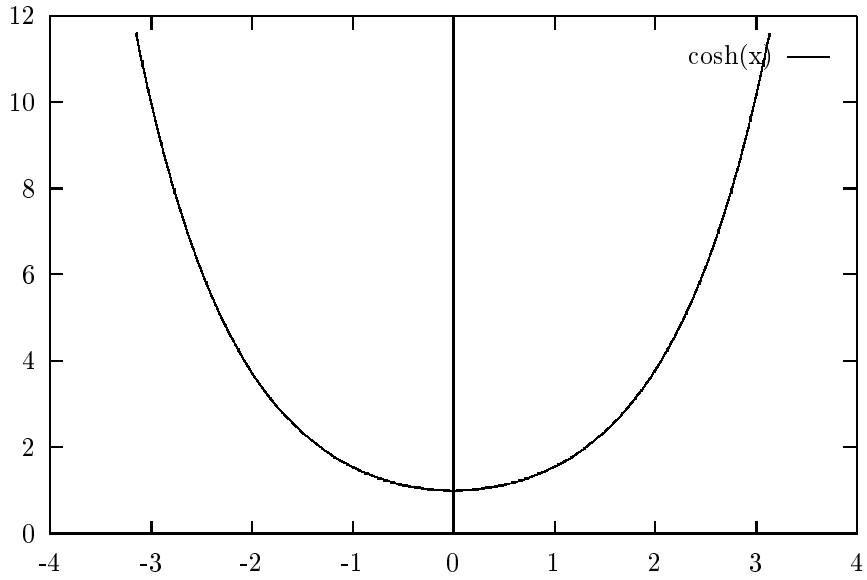
```
     float    fasinh(      float  x ) ;
     double    asinh(      double x ) ;
long double   lasinh( long double x ) ;
```

A.3.3 atanh()

Syntax:

```
     float    fatanh(      float  x ) ;
     double    atanh(      double x ) ;
long double   latanh( long double x ) ;
```

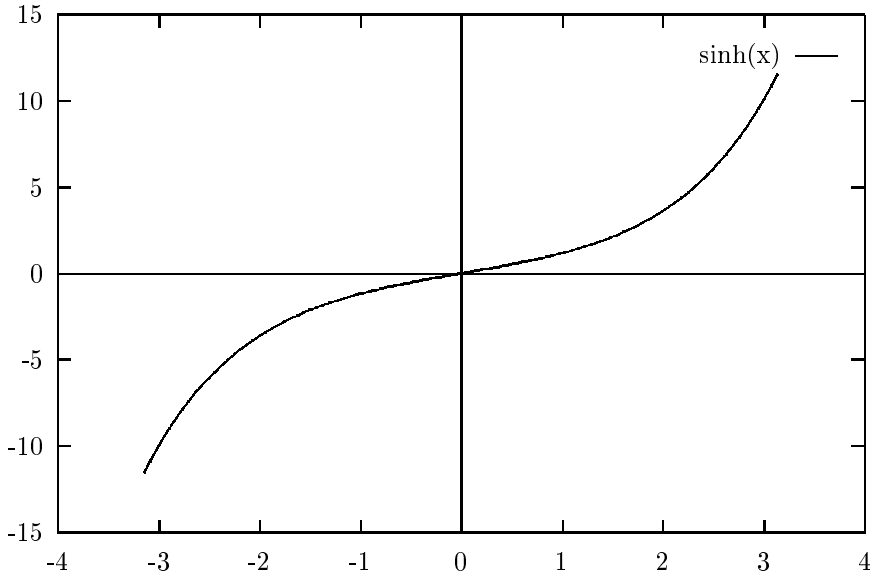
A.3.4 cosh()



Syntax:

```
     float    fcosh(      float  x ) ;
     double    cosh(      double x ) ;
long double   lcosh( long double x ) ;
```

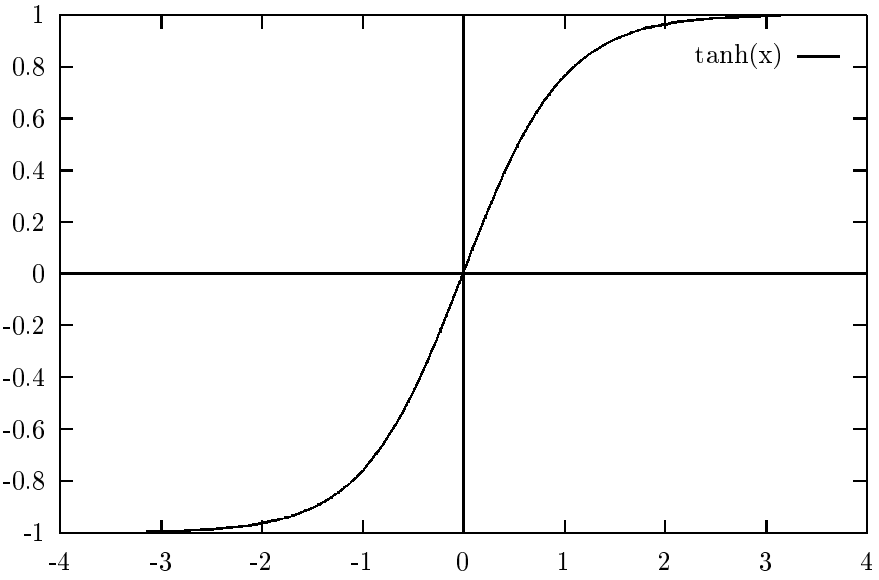
A.3.5 sinh()



Syntax:

```
     float    fsinh(      float  x ) ;
     double    sinh(      double x ) ;
long double   lsinh( long double x ) ;
```

A.3.6 tanh()

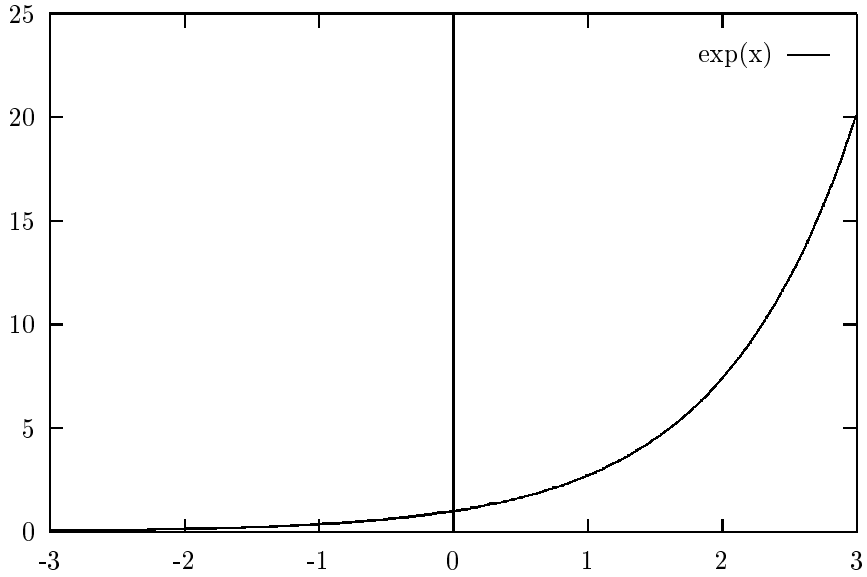


Syntax:

```
     float    ftanh(      float  x ) ;
     double    tanh(      double x ) ;
long double   ltanh( long double x ) ;
```

A.4 Exponential– und logarithmische Funktionen

A.4.1 exp()



Syntax:

```
     float  fexp(      float   x ) ;
     double  exp(      double  x ) ;
long double lexp( long double  x ) ;
```

Die Funktion `exp()` liefert das Resultat der Berechnung e^x.

Wertebereich und Fehler:
Wird als Parameter *x* eine unzulässige Zahl angegeben (*NaN* oder $\pm$*Inf*), so wird ein *domain error* durch das Setzen der Variable `errno` auf *EDOM* angezeigt. Bei *NaN* und *+Inf* wird der Eingabeparameter zurückgegeben; bei *–Inf* liefert die Funktion Null.

Die Wertebereiche, in denen darstellbare Ergebnisse geliefert werden, hängen von der jeweiligen Fliesskommaauflösung ab. Bei `float` liegt der Bereich bei etwa $\pm$87, bei `double` entsprechend $\pm$708 und bei `long double` ungefähr bei $\pm$11355. Bei kleineren Argumenten wird immer Null als Resultat geliefert; grössere Argumente liefern *+Inf* und führen zu einem *range error*, der durch das Setzen von `errno` auf *ERANGE* angezeigt wird.

A.4.2 exp2()

Syntax:

```
     float  fexp2(      float   x ) ;
     double  exp2(      double  x ) ;
long double lexp2( long double  x ) ;
```

Die Funktion `exp2()` liefert das Resultat der Berechnung 2^x.

A.4.3 exp10()

Syntax:

```
     float  fexp10(      float   x ) ;
     double  exp10(      double  x ) ;
long double lexp10( long double  x ) ;
```

Die Funktion `exp10()` liefert das Resultat der Berechnung 10^x.

A.4.4 fmod()

Syntax:

```
     float    ffmod(      float   x,      float   y ) ;
     double    fmod(      double  x,      double  y ) ;
long double   lfmod( long double  x, long double  y ) ;
```

A.4.5 frexp()

Syntax:

```
     float  ffrexp(      float   value, int  *n ) ;
     double  frexp(      double  value, int  *n ) ;
long double lfrexp( long double  value, int  *n ) ;
```

Die Funktion `frexp()` liefert die Mantisse und den binären Exponenten einer Fliesskommazahl in zwei getrennten Werten zurück. Dabei wird aus dem Argument *value* eine Mantisse *x* (absoluter Wert grösser 0.5 und kleiner 1.0) zusammen mit einen Exponenten der Basis 2 gebildet. Es handelt sich dabei um eine (binär) normalisierte Fliesskommazahl. Anders gesagt: `frexp()` erfüllt die Gleichung $value = x * 2^n$. Der Wert *x* wird als Funktionsergebnis zurückgeliefert, der Exponent *n* in dem Integer, auf den der Parameter *n* zeigt.

Wertebereich und Fehler:
Wird als Parameter *value* eine unzulässige Zahl angegeben (*NaN* oder $\pm$*Inf*), so wird ein *domain error* durch das Setzen der Variable `errno` auf *EDOM* angezeigt. Der Eingabewert wird als Funktionsrückgabe zurückgeliefert und der Integerwert, auf den der Parameter *n* zeigt, auf Null gesetzt.

A.4.6 ldexp()

Syntax:

```
     float  fldexp(      float   x, int  n ) ;
     double  ldexp(      double  x, int  n ) ;
long double lldexp( long double  x, int  n ) ;
```

Die Funktion `ldexp()` liefert als Rückgabewert `value` das Resultat der Berechnung $value = x * 2^n$, stellt also das Gegenstück zur Funktion `frexp()` dar.

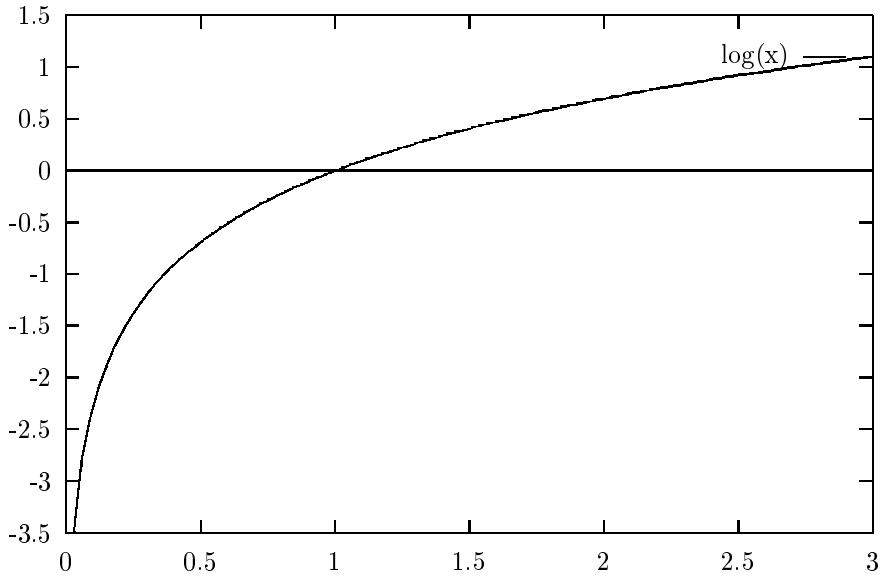
Wertebereich und Fehler:
Wird als Parameter *x* der Wert *NaN* angegeben, so wird ein *domain error* durch das Setzen der Variable `errno` auf *EDOM* angezeigt. Die Funktion liefert den Parameter *x* zurück.

Wird als Parameter *x* der Wert $\pm$*Inf* angegeben, so wird ein *range error* durch das Setzen der Variable `errno` auf *ERANGE* angezeigt. Die Funktion liefert den Parameter *x* zurück.

Überschreitet die Berechnung $value = x * 2^n$ den darstellbaren Wertebereich der gewählten Fliesskommaauflösung, so wird ein *range error* durch das Setzen der Variable `errno` auf *ERANGE* angezeigt

und entsprechend dem Vorzeichen des Parameters *x* als Rückgabewert $\pm$*Inf* geliefert.

A.4.7 log()



Syntax:

```
     float  flog(      float   x ) ;
     double  log(      double  x ) ;
long double llog( long double  x ) ;
```

Die Funktion `log()` liefert den natürlichen Logarithmus des Parameters *x*.

Wertebereich und Fehler:
Wird als Parameter *x* eine unzulässige Zahl angegeben (*NaN* oder $\pm$*Inf*), so wird ein *domain error* durch das Setzen der Variable `errno` auf *EDOM* angezeigt. Bei *NaN* und *+Inf* wird der Eingabeparameter zurückgegeben; bei *–Inf* liefert die Funktion *NaN*.

Bei negativen Argumenten liefert `log()` als Resultat *NaN* und setzt `errno` auf *EDOM*. Ist der Parameter *x* Null, so liefert die Funktion *–Inf* und setzt `errno` ebenfalls auf *EDOM*.

A.4.8 log2()

Syntax:

```
     float  flog2(      float   x ) ;
     double  log2(      double  x ) ;
long double llog2( long double  x ) ;
```

Die Funktion `log2()` liefert den Logarithmus des Parameters *x* zur Basis 2.

A.4.9 log10()

Syntax:

```
     float  flog10(      float   x ) ;
     double  log10(      double  x ) ;
```

```
long double llog10( long double  x ) ;
```

Die Funktion `log10()` liefert den Logarithmus des Parameters *x* zur Basis 10.

A.4.10 modf()

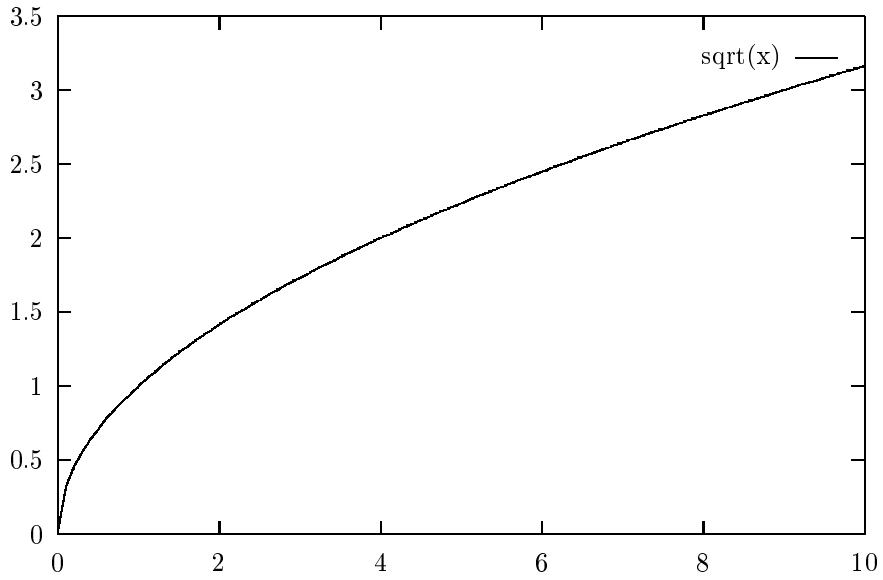
Syntax:

```
     float    fmodf(      float   x,      float  *y ) ;
     double    modf(      double  x,      double *y ) ;
long double   lmodf( long double  x, long double *y ) ;
```

A.5 Potenzfunktionen

A.5.1 pow()

A.5.2 sqrt()



Syntax:

```
     float  fsqrt(      float  x ) ;
     double  sqrt(      double x ) ;
long double lsqrt( long double x ) ;
```

Die Funktion `sqrt(`*x*`)` berechnet die Quadratwurzel des Arguments *x*.

Wertebereich und Fehler:
Bei negativen Argumenten wird ein *domain error* durch das Setzen der Variable `errno` auf *EDOM* angezeigt.

Wird als Parameter *x* der Wert *NaN* oder *–Inf* angegeben, so wird ebenfalls ein *domain error* durch das Setzen der Variable `errno` auf *EDOM* angezeigt. Die Funktion liefert den Parameter *x* zurück.

Wird als Parameter *x* der Wert *+Inf* angegeben, so wird ein *range error* durch das Setzen der Variable `errno` auf *ERANGE* angezeigt. Die Funktion liefert den Parameter *x* zurück.

A.6 Sonstige Funktionen

A.6.1 ceil()

Syntax:

```
     float  fceil(      float  x ) ;
     double  ceil(      double x ) ;
long double lceil( long double x ) ;
```

Die Funktion `ceil(`*x*`)` rundet das übergebene Argument *x* in Richtung auf den nächsthöheren Integerwert auf und liefert das Resultat als Fliesskommazahl zurück.

Wertebereich und Fehler:
Wird als Parameter *x* der Wert *NaN* oder $\pm$*Inf* angegeben, so wird ein *domain error* durch das Setzen der Variable `errno` auf *EDOM* angezeigt. Die Funktion liefert den Parameter *x* zurück.

A.6.2 fabs()

Syntax:

```
     float  ffabs(      float  x ) ;
     double  fabs(      double x ) ;
long double lfabs( long double x ) ;
```

Die Funktion `fabs(`*x*`)` liefert den Absolutwert des Arguments *x*.

Wertebereich und Fehler:
Wird als Parameter *x* der Wert *NaN* oder $\pm$*Inf* angegeben, so wird ein *domain error* durch das Setzen der Variable `errno` auf *EDOM* angezeigt. Bei *NaN* wird der Parameter *x* zurückgeliefert. Bei $\pm$*Inf* lautet das Resultat +Inf.

A.6.3 floor()

Syntax:

```
     float  ffloor(      float  x ) ;
     double  floor(      double x ) ;
long double lfloor( long double x ) ;
```

Die Funktion `floor(`*x*`)` rundet das übergebene Argument *x* in Richtung auf den nächstniedrigeren Integerwert ab und liefert das Resultat als Fliesskommazahl zurück.

Wertebereich und Fehler:
Wird als Parameter *x* der Wert *NaN* oder $\pm$*Inf* angegeben, so wird ein *domain error* durch das Setzen der Variable `errno` auf *EDOM* angezeigt. Die Funktion liefert den Parameter *x* zurück.

A.6.4 round()

Syntax:

```
     float  fround(      float  x ) ;
     double  round(      double x ) ;
long double lround( long double x ) ;
```

Die Funktion `round(`*x*`)` rundet das übergebene Argument *x* in Richtung auf den nächstliegenden Integerwert und liefert das Resultat als Fliesskommazahl zurück. Endet das Argument *x* exakt mit `0.5`, so wird der nächstliegende **gerade** Integerwert ausgewählt.

Wertebereich und Fehler:
Wird als Parameter *x* der Wert *NaN* oder $\pm$*Inf* angegeben, so wird ein *domain error* durch das Setzen der Variable `errno` auf *EDOM* angezeigt. Die Funktion liefert den Parameter *x* zurück.

A.6.5 sign()

Syntax:

```
     float  fsign(      float  x ) ;
     double  sign(      double x ) ;
long double lsign( long double x ) ;
```

Die Funktion `sign(`*x*`)` liefert Vorzeichen des übergebenen Argumentes *x* als Fliesskommazahl zurück.

$$y = \begin{cases} -1.0 & : \quad x < 0 \\ 0.0 & : \quad x = 0 \\ +1.0 & : \quad x > 0 \end{cases}$$

Wertebereich und Fehler:
Wird als Parameter *x* der Wert *NaN* oder $\pm$*Inf* angegeben, so wird ein *domain error* durch das Setzen der Variable `errno` auf *EDOM* angezeigt. Die Funktion liefert den Parameter *x* zurück.

A.6.6 Testroutinen für Fliesskommazahlen

Die folgenden Funktionen dienen zur Abfrage, ob Fliesskommavariablen ungültige Bitmuster enthalten. Die Prototypen befinden sich in <`math.h`>. Es handelt sich allerdings nicht um ANSI–C–Funktionen. Diese Routinen sollten immer dort aufgerufen werden, wo die Möglichkeit besteht, dass die Ergebnisse einer Fliesskommarechnung in den nicht mehr darstellbaren Bereich abgleiten.

1. Kontrolle, ob eine Zahl *NaN* (Not-a-Number) ist.

isnan()	**Rückgabewert**
NaN	-1
nicht *NaN*	0

```
int    fisnan(      float  x ) ;
int     isnan(      double x ) ;
int    lisnan( long double x ) ;
```

2. Kontrolle, ob eine Zahl $\pm$*Inf* (Infinite) ist.

isinf()	**Rückgabewert**
$\pm$*Inf*	-1
nicht $\pm$*Inf*	0

```
int    fisinf(      float  x ) ;
int     isinf(      double x ) ;
int    lisinf( long double x ) ;
```

3. Kontrolle, ob eine Zahl *NaN* oder $\pm$*Inf* ist.

finite()	**Rückgabewert**
NaN oder $\pm$*Inf*	-1
nicht *NaN* oder $\pm$*Inf*	0

```
int   ffinite(      float  x ) ;
int    finite(      double x ) ;
int   lfinite( long double x ) ;
```

Anhang B

Zusammenstellung der Pragma–Kommandos

In der nachfolgenden Auflistung sind die dem CCC bekannten `#pragma`–Kommandos aufgeführt. Bei der Bearbeitung von dieser Kommandos ist auf die korrekte Schreibweise der Kommandobezeichnungen zu achten, da der Compiler unbekannte Kammandos kommentarlos ignoriert. Ein Check auf zulässige Syntax findet gemäss ANSI–C erst dann statt, wenn sich ein Compiler sich für ein spezielles `#pragma`–Kommando als zuständig empfindet.

Normalerweise werden Makros in Kommandozeilen nicht durch den Präprozessor aufgelöst; es findet auch keinerlei Evaluierung von Expressions statt. In Abschnitt 3.1.8 ist erläutert, wie mittels der Compileroption `-q` der Makroprozessor auch für Kommandozeilen aktiviert werden kann.

B.1 ALLOCATE_INTERRUPT_BUFFER

Anfordern eines Interruptpuffers beim Systemstart.

```
#pragma ALLOCATE_INTERRUPT_BUFFER  LEVEL level  SIZE size
```

Parameter:

`LEVEL` *level*	gibt den Interruptlevel (1 bis 7) an.
`SIZE` *size*	gibt die Grösse des einzurichtenden Puffers an (kleiner 32kB).

Siehe auch Abschnitt 13.5.2.

B.2 COLDSTART

Einrichten einer Funktion als Kaltstartcode.

```
#pragma COLDSTART
void Kaltstart( void ){ ... }
```

Parameter: keine

Siehe auch Abschnitt 13.7.

B.3 DISABLE_BIT_MNEMONICS

Unterdrückt bei der Codegenerierung die Verwendung von Einzelbit–Befehlen. Bei Zugriff auf externen Speicherbereich werden keine BSET–, BCLR–, BTST– und BCHG–Mnemonics mehr verwendet.

Parameter: keine

Siehe auch Abschnitt 3.1.18.2.

B.4 DISABLE_CLEAR_MNEMONICS

Unterdrückt bei der Codegenerierung die Verwendung des CLR–Mnemonics bei Speicherzugriffen.

Parameter: keine

Siehe auch Abschnitt 3.1.18.1.

B.5 DYNAMIC_STACK

Einrichten von Prolog– und Epilogcode für die nachfolgende Funktion, um bei Bedarf den Stack zur Laufzeit zu vergrössern.

```
#pragma DYNAMIC_STACK  NEWSTACK size  RANGE range
void RecursiveFunction( void ){ ... }
```

Parameter:

NEWSTACK *size*	gibt die zusätzliche Stackgrösse in Bytes an.
RANGE *range*	gibt den Sicherheitsabstand zum Stackende in Bytes an (kleiner 32kB)

Siehe auch Abschnitt 3.9.

B.6 ENABLE_BIT_MNEMONICS

Hebt die Wirkung von DISABLE_BIT_MNEMONICS auf und erlaubt dem Compiler die Verwendung von BSET–, BCLR–, BTST– und BCHG–Mnemonics.

Parameter: keine

Siehe auch Abschnitt 3.1.18.2.

B.7 ENABLE_CLEAR_MNEMONICS

Hebt die Wirkung von DISABLE_CLEAR_MNEMONICS auf und erlaubt dem Compiler die Verwendung des CLR–Mnemonics.

Parameter: keine

Siehe auch Abschnitt 3.1.18.1.

B.8 END_SLICE_SKIP

Beendet den Bereich einer Skip–Scheibe.

Parameter: keine

Siehe START_SLICE_SKIP...

B.9 EXCEPTION

Richtet die nachfolgende Funktion als Exceptionhandler ein. Die Angabe des für die Exception zuständigen Vektors kann wahlweise in einer der drei folgenden Schreibweisen erfolgen.

```
#pragma EXCEPTION
void IrgendeineException( void ){ ... }

#pragma EXCEPTION VECTOR 0x100
void Exception100( void ){ ... }

#pragma EXCEPTION TRAP 4
void Trap4( void ){ ... }

#pragma EXCEPTION LINE-A A008
void LineA008( void ){ ... }
```

Optionale Parameter:

VECTOR *nr*	gibt als Zahlenwert den Vektor direkt vor.
TRAP *nr*	gibt den Vektor als Trap (0 bis 15) vor.
LINE-A *nr*	gibt den Vector in LINE-A–Schreibweise vor.
NO_DISPATCHER_CALL	Der Exceptionhandler terminiert immer mittels RTE.
IROFF	Der Exceptionhandler wird komplett auf IR–Level 7 ausgeführt.

Siehe auch Abschnitt 13.6.

B.10 HEADER

Systemstartmeldung erzeugen.

```
#pragma HEADER "String in doppelten Hochkommas"
```

Siehe auch Abschnitt 17.3.

B.11 INCLUDE_ONCE

Unterdrückt das mehrfache Includieren einer Headerdatei.

Parameter: keine

Siehe auch Abschnitt 3.1.9.2.

B.12 INTERRUPT

Richtet die nachfolgende Funktion als Interruptroutine ein.

```
#pragma INTERRUPT
void IrgendeineException( void ){ ... }

#pragma INTERRUPT VECTOR 0x100
void Exception100( void ){ ... }

#pragma INTERRUPT LEVEL 2
void IRQ2( void ){ ... }
```

Optionale Parameter:

VECTOR *nr*	gibt als Zahlenwert den Vektor direkt vor.
LEVEL *nr*	gibt als Zahlenwert (1 bis 7) den Vektor als IR–Level vor.
NO_DISPATCHER_CALL	Der Interrupthandler terminiert immer mittels RTE.
IROFF	Der Interrupthandler wird komplett auf IR–Level 7 ausgeführt.
EVENT *mask*	Der Interrupt feuert beim Verlassen des Handlers Events.
EPROM	Es wird eine Scheibe generiert, die den Vektor beim Systemstart setzt.
NO_VECTOR	Es wird kein Vektor vom Prolog–Code eingesetzt.
NO_IID	Der Prolog–Code setzt nicht den Interrupt–Identifier.
RESET_STACK	Der Interrupthandler setzt immer auf dem Initialsystemstack auf.
NO_MALFUNCTION	Der Malfunction–Mechanismus wird nicht unterstützt.

Siehe auch Abschnitt 13.5.

B.13 INTERRUPT_EXIT

Richtet die nächste Funktion so ein, dass sie mit korrektem Epilog–Code eines Interrupthandlers terminiert.

Optionale Parameter:

IROFF	Der Interrupthandler wird komplett auf IR–Level 7 ausgeführt.

Keine weitere Beschreibung im Handbuch verfügbar.

B.14 INTERRUPT_PROCEDURE

Der Code der nachfolgenden Funktion wird entsprechend den Konventionen einer Interruptroutine generiert. Es werden zusätzliche Adressregister bei der Codegenerierung verwendet (A4 und A5).

Parameter: keine

Keine weitere Beschreibung im Handbuch verfügbar.

B.15 KALTSTART

identisch `COLDSTART ...`

B.16 MEMBER_PADDING_OFF

Schaltet das reguläre Padding von Strukturmembers auf Wortgrenzen aus. Die Mitglieder einer Struktur werden ohne Padding direkt hintereinander allokiert.

Parameter: keine

Siehe auch Abschnitte 3.4.1.1 und 3.4.1.2.

B.17 MEMBER_PADDING_ON

Stellt das reguläre Padding von Strukturmembers auf Wortgrenzen wieder her.

Parameter: keine

Siehe auch Abschnitte 3.4.1.1 und 3.4.1.2.

B.18 MEMORY

Richtet eine Modulvariablenscheibe ein.

```
#pragma MEMORY "NAME" 0x10000 0x20000"
```

Parameter:
1. String mit max. 6 Zeichen langem Bezeichner.
2. Startadresse des einzurichtenden Moduls.
3. Endadresse des einzurichtenden Moduls.

Siehe auch Abschnitt 15.6.3.

B.19 MODULE

Richtet einen Modulkopf ein.

```
#pragma MODULE "MeinModulkopp"
```

Siehe auch Abschnitt 17.4.

B.20 PEARL_PROCEDURE

Einrichten von Prolog– und Epilogcode für die nachfolgende Funktion, um den C–Code von PEARL aus aufrufen zu können.

```
#pragma PEARL_PROCEDURE
void  PearlEntry( void ) {...}
```

Optionale Parameter:
STACKSIZE *size*: Grösse des einzurichtenden Stacks in Bytes.

B.21 RAM_RANGES

Richtet eine Scheibe ein, in der die unter RTOS–UH zu verwaltenden RAM–Bereiche anzugeben sind.

```
#pragma RAM_RANGES  0x000800 0x0FFFE0 \
                    0x100000 0x3FFFE0
```

Parameter: n Zahlenpaare jeweils mit Start– und Endadresse.

Siehe auch Abschnitt 16.2.10.

B.22 SCAN_RANGES

Richtet eine Scheibe ein, in der die beim Systemstart zu überscannenden Speicherbereiche anzugeben sind.

```
#pragma SCAN_RANGES 0x00001 0x20001 0x50001 0x70001
```

Parameter: n Zahlenpaare jeweils mit Start– und Endadresse.

Siehe auch Abschnitt 16.2.2.

B.23 SET_VECTOR

Richtet beim Systemstart einen Vektor ein.

```
#pragma SET_VECTOR 0x80E "InterruptTrigger"
#pragma INTERRUPT_EXIT  IROFF
void InterruptTrigger( StoredRegisterSet  *registers ){ ... }
```

Parameter:
1. Adresse, an der bei Systemstart ein Pointer gesetzt werden soll.
2. Symbol im `.text`–Segment, das als String angegeben werden muss.

Keine weitere Beschreibung im Handbuch verfügbar.

B.24 START_SLICE_SKIP

Legt die nächste Funktion als Tester–Funktion für einen Skip–Bereich fest und dient als Startposition des zu skippenden Bereiches.

```
#pragma START_SLICE_SKIP
int test( void ) { ... }
```

```
// beliebige Treiber etc...
#pragma HEADER "Bin drin!!"
#pragma END_SLICE_SKIP
```

Parameter: keine

Siehe auch Abschnitt 17.2.

B.25 STRUCT_PADDING_OFF

Schaltet das reguläre Padding der Strukturgrösse auf die nächste Wortgrenze aus. Die Strukturgrösse wird auf die nächste Zweierpotenz aufgerundet.

Parameter: keine

B.26 STRUCT_PADDING_ON

Schaltet das reguläre Padding der Strukturgrösse auf Wortgrenzen ein.

Parameter: keine

B.27 SUBTASK

Einrichten von Prolog– und Epilogcode für die nachfolgende Funktion, um beim Funktionsauf eine Task zu generieren. Die Task verschwindet nach der Abarbeitung des Funktionscodes aus dem System.

```
#pragma SUBTASK  PRIO 37  STACKSIZE 4096  FILES 3
int KurzlebigeTask( void ) { ... }
```

Optionale Parameter:

PRIORITY *prio*	Priorität der zu generierenden Task.
PRIO *prio*	dito...
STACKSIZE *size*	Stackgrösse der Task in Bytes.
RESIDENT	Der Taskworkspace der Task soll resident bleiben.
USE_FUNCTION_NAME	Task erhält den Namen der nachfolgenden Funktion.
USE_NAME *fkt*	Der Rückgabewert der Funktion *fkt* ist neue Taskname.
NO_TASKSTART	Die Task wird nur erzeugt und nicht aktiviert.
USE_FPU	Die zu generierende Task rettet die FPU–Register.
NO_FPU	Die zu generierende Task rettet keine FPU–Register.
NO_ALLOC	Die C–Speicherringe werden nicht initialisiert.
NO_FILES	Die C–File–I/O wird nicht initialisiert.
NO_SETUP	Es finden keinerlei C–Initialisierungen statt.
FILES	Gibt die Zahl der gleichzeitig zugänglichen FILE–Pointer an.

Siehe auch Abschnitt 13.3.

B.28 SYSTEMTASK

Einrichten der nachfolgenden Funktion als Task beim Systemstart.

```
#pragma SYSTEMTASK  PRIO -2 RESIDENT AUTOSTART  QUEUE 4
int GrundebeneLdn4( void ) { ... }
```

Optionale Parameter:

PRIORITY *prio*	Priorität der zu generierenden Task.
PRIO *prio*	dito...
STACKSIZE *size*	Stackgrösse der Task in Bytes.
RESIDENT	Der Taskworkspace der Task soll resident bleiben.
TASKHEADSIZE *size*	Der Taskkopf der Task wird um *size* Bytes vergrössert.
AUTOSTART	Die Task wird autostartfähig.
QUEUE*ldn*	Die Task wird Betreuungstask für die betreffende *ldn*.
INTERFACE *ldn*	Die Task wird Userinterface für die betreffende *ldn*.
ERROR	Die Task soll als #ERROR–Task arbeiten.

Siehe auch Abschnitt 13.4.

B.29 TAG_COPY_BYTE

Es wird Code der Form generiert, dass bei der Zuweisung von Strukturen oder Unions der Speicherinhalt byteweise (MOVE.B) umkopiert wird.

Parameter: keine

Siehe auch Abschnitt 3.4.1.3.

B.30 TAG_COPY_WORD

Es wird Code der Form generiert, dass bei der Zuweisung von Strukturen oder Unions der Speicherinhalt wortweise (MOVE.W) umkopiert wird.

Parameter: keine

Siehe auch Abschnitt 3.4.1.3.

B.31 TAG_COPY_LONG

Es wird Code der Form generiert, dass bei der Zuweisung von Strukturen oder Unions der Speicherinhalt langwortweise (MOVE.L) umkopiert wird.

Parameter: keine

Siehe auch Abschnitt 3.4.1.3.

B.32 TAG_COPY_SIZE

Gibt die Grösse von Strukturen oder Unions an, ab der statt einzelner MOVE–Befehle bei Zuweisungen zum Umkopieren des Speicherinhaltes eine DBF–Schleife generiert wird.

```
#pragma TAG_COPY_SIZE 20
```

Parameter:
Maximale Grösse eines Tags in Bytes, der mittels einzelner MOVE–Befehle umkopiert werden soll.

Siehe auch Abschnitt 3.4.1.3.

B.33 TAG_PUSH_SIZE

Gibt die Grösse von Strukturen oder Unions an, ab der statt einzelner MOVE–Befehle bei der Parameterübergabe an eine Funktion *per value* eine DBF–Schleife generiert wird.

```
#pragma TAG_PUSH_SIZE 20
```

Parameter:
Maximale Grösse einer Tags in Bytes, der mittels einzelner MOVE–Befehle auf den Stack kopiert werden soll.

Keine weitere Beschreibung im Handbuch verfügbar.

B.34 TASK

Einrichten von Prolog– und Epilogcode für die nachfolgende Funktion, um beim Funktionsauf eine Task zu generieren. Die Task bleibt auch nach Beendigung des Codes der Funktion dauerhaft im System.

```
#pragma TASK  PRIO 37  STACKSIZE 4096  FILES 3
int LanglebigeTask( void ) { ... }
```

Optionale Parameter:

PRIORITY *prio*	Priorität der zu generierenden Task.
PRIO *prio*	dito...
STACKSIZE *size*	Stackgrösse der Task in Bytes.
RESIDENT	Der Taskworkspace der Task soll resident bleiben.
USE_FUNCTION_NAME	Task erhält den Namen der nachfolgenden Funktion.
USE_NAME *fkt*	Der Rückgabewert der Funktion *fkt* ist neue Taskname.
NO_TASKSTART	Die Task wird nur erzeugt und nicht aktiviert.
USE_FPU	Die zu generierende Task rettet die FPU–Register.
NO_FPU	Die zu generierende Task rettet keine FPU–Register.
NO_ALLOC	Die C–Speicherringe werden nicht initialisiert.
NO_FILES	Die C–File–I/O wird nicht initialisiert.
NO_SETUP	Es finden keinerlei C–Initialisierungen statt.
FILES	Gibt die Zahl der gleichzeitig zugänglichen FILE–Pointer an.

Siehe auch Abschnitt 13.3.1.

B.35 WARMSTART

Einrichten einer Funktion als Warmstartcode.

```
#pragma WARMSTART
void Warmstart( void ){ ... }
```

Parameter: keine

Siehe auch Abschnitt 13.7.

Anhang C

Usage–Meldungen der Tools

Auf den folgenden Seiten sind die Usage–Meldungen der Tools der CREST–C–Paketes aufgelistet.

C.1 ccc68k

```
Crest C Compiler CCC                68K-Version for Linux x86
Copyright (C) 1987-1999             by Stephan Litsch
Release 2.597                       Nov  3 1999  16:57:33

Usage:ccc68k infile [outfile] [-options]
  -#macnam[=repl]  Define an argumentfree macro
  @file            Read argument file
  -A=digit         Stop compiler
     0                 never
     1                 after fatals
     2                 after fatals and errors
     3                 after fatals, errors and warnings
  -B               Generate '.err'-File
  -C=digit         Addressing mode for <.data>- and <.bss>-variables
     0                 Word access
     1                 Long access
  -D=digit         Addressing mode for <.text>-constants
     0                 Word access
     1                 Long access
  -E=digit         Addressing mode for <.local>-variables
     0                 Word access
     1                 Long access
  -G=digit         Generate '.dpc'-File
     0                 No dependency file
     1                 list included ".c"-files
     2                 list included ".c"- and ".h"-files
     3                 list included ".c"-, ".h" and <.h>-files
  -H=path          Set headerfile directory
  -J=digit         Search includes in upper|lower case
     0                 only upper case
     1                 only lower case
     2                 upper and lower case
```

```
-L              Add line infos to '.s'-File
-N              Suppress CLR (Ax)-Mnemonics
-P=digit        Enable PEARL-Interface
   0               No
   1               PEARL80
   2               PEARL90
   3               PEARL80/PEARL90
-Q[=digit]      Show source
-R=digit        Addressing mode for function calls
   0               JSR    label
   1               JSR    (label,PC)
   2               BSR.L  label
   3               BSR.W  label
-S              Default character is signed
-T              Informations about compiler modifications
-U              Stack check
-V              Verbose messages
-W=digit        Ignore warnings
-Y=digit        Linetracer
   0               No
   1               Set linecell
   2               Use system trap
-Z              Accept global register variables
-0              Generate 68000 Code
-2              Generate 68020 Code
-3              Generate CPU32 Code
-a=digit        Number of reserved Ax  registers
-d=digit        Number of reserved Dx  registers
-e              Dump internal expression trees
-f=digit        Number of reserved FPx registers
-h              Use of global variables forbidden
-j=digit        Parametertransfer for floatingpoint
   0            ANSI compliant
   1               float
   2               double
   3               long double
-n              Suppress optimization of bit operations
-o              Dump all optimizations to '.opl'-File
-p              Show all floatingpoint casts as warnings
-q              Preprocess #pragma's
-s              Generate '.s'-File
-u              Add stack infos to '.obj'-File
-x              Generate '.lst'-File
-y              No #undef-warnings
-z              Add debug infos to '.obj'-File
-+              Enable some C9X features

--ansi          Force ANSI casting
--fpu           Generate Code for FPU
```

C.2 cccppc

```
Crest C Compiler CCC                PPC-Version for Linux x86
Copyright (C) 1987-1999             by Stephan Litsch
Release 2.597                       Nov  4 1999  12:28:00

Usage:cccppc infile [outfile] [-options]
  -#macnam[=repl]  Define an argumentfree macro
  @file            Read argument file
  -A=digit         Stop compiler
     0                 never
     1                 after fatals
     2                 after fatals and errors
     3                 after fatals, errors and warnings
  -B               Generate '.err'-File
  -C=digit         Addressing mode for <.data>- and <.bss>-variables
     0                 Word access
     1                 Long access
  -D=digit         Addressing mode for <.text>-constants
     0                 Word access
     1                 Long access
  -E=digit         Addressing mode for <.local>-variables
     0                 Word access
     1                 Long access
  -G=digit         Generate '.dpc'-File
     0                 No dependency file
     1                 list included ".c"-files
     2                 list included ".c"- and ".h"-files
     3                 list included ".c"-, ".h" and <.h>-files
  -H=path          Choose headerfile directory
  -J=digit         Search includes in upper|lower case
     0                 only upper case
     1                 only lower case
     2                 upper and lower case
  -L               Add line infos to '.s'-File
  -P=digit         Enable PEARL-Interface
     0                 No
     2                 PEARL90
  -Q[=digit]       Show source
  -R=digit         Addressing mode for function calls
     0                 bl     label
     1                 bla    label
     2                 blrl
  -S               Default character is signed
  -T               Informations about compiler modifications
  -U               Stack check
  -V               Verbose messages
  -W=digit         Ignore warnings
  -Y=digit         Linetracer
     0                 No
     1                 Set linecell
```

```
    2               Use system trap
  -Z              Accept global register variables
  -e              Dump internal expression trees
  -h              Use of global variables forbidden
  -n              Suppress optimization of bit operations
  -o              Dump all optimizations to '.opl'-File
  -p              Show all floatingpoint casts as warnings
  -q              Preprocess #pragma's
  -s              Generate '.s'-File
  -u              Add stack infos to '.obj'-File
  -x              Generate '.lst'-File
  -y              No #undef-warnings
  -z              Add debug infos to '.obj'-File
  -+              Enable some C9X features

  --ansi          Force ANSI casting
  --fpu           Generate Code for FPU
```

C.3 cln68k

```
Crest S-Record-Linker CLN          68K-Version for Linux x86
Copyright (C) 1987-1999            by Stephan Litsch
Release 2.597                      Nov  4 1999  12:25:28

Usage:cln68k linkfile [outputfile] [-options]
  @file             Read option file
  -#macro[=replace] Define argumentfree macro
  -B                No slice for <.common>-section
  -C=address        Set <.common>-section address
  -E                Autostart for task-startup
  -F=files          Number of FILE structures
  -H=size           Extended taskhead
  -L=path           Set library directory
  -M                Dump loader map
  -N=name           Set name of shellmodule or task
  -O                Usable for PROM
  -P=digit          Enable PEARL-interface
    0                  No
    1                  PEARL80
    2                  PEARL90
  -Q=prio           Priority for task-startup
  -R=name           Set RAM module name
  -S=size           Set stacksize
  -T=address        Set <.text>-section address
  -U                Resident for task-startup
  -V                Verbose messages
  -h                Use of global variables forbidden
  -z                Dump debug infos
  -0                Link 68000 Code
  -2                Link 68020 Code
```

```
  -3                Link CPU32 Code

  --fpu             Link Code for FPU
```

C.4 clnppc

```
Crest S-Record-Linker CLN          PPC-Version for Linux x86
Copyright (C) 1987-1999            by Stephan Litsch
Release 2.597                      Nov  4 1999  12:28:00

Usage:clnppc linkfile [outputfile] [-options]
  @file             Read option file
  -#macro[=replace] Define argumentfree macro
  -B                No slice for <.common>-section
  -C=address        Set <.common>-section address
  -E                Autostart for task-startup
  -F=files          Number of FILE structures
  -H=size           Extended taskhead
  -L=path           Set library directory
  -M                Dump loader map
  -N=name           Set name of shellmodule or task
  -O                Usable for PROM
  -P=digit          Enable PEARL-interface
     0                 No
     2                 PEARL90
  -Q=prio           Priority for task-startup
  -R=name           Set RAM module name
  -S=size           Set stacksize
  -T=address        Set <.text>-section address
  -U                Resident for task-startup
  -V                Verbose messages
  -h                Use of global variables forbidden
  -z                Dump debug infos

  --fpu             Link Code for FPU
```

C.5 lnk68k

```
Crest Library-Linker LNK           68K-Version for Linux x86
Copyright (C) 1987-1999            by Stephan Litsch
Release 2.597                      Nov  4 1999  12:25:28

Usage:lnk68k linkfile [outfile] [-options]
  @file             Read option file
  -#macro[=replace] Define argumentfree macro
  -L=path           Set library directory
  -M                Dump loader map
  -V                Verbose messages
  -z                Add debug informations to library
```

```
  -0                Link 68000 Code
  -2                Link 68020 Code
  -3                Link CPU32 Code

  --fpu             Link Code for FPU
```

C.6 lnkppc

```
Crest Library-Linker LNK          PPC-Version for Linux x86
Copyright (C) 1987-1999           by Stephan Litsch
Release 2.597                     Nov  4 1999  12:28:00

Usage:lnkppc linkfile [outfile] [-options]
  @file             Read option file
  -#macro[=replace] Define argumentfree macro
  -L=path           Set library directory
  -M                Dump loader map
  -V                Verbose messages
  -z                Dump debug infos

  --fpu             Link Code for FPU
```

C.7 ssl68k

```
Crest Shared-Library-Linker SSL    68K-Version for Linux x86
Copyright (C) 1987-1999            by Stephan Litsch
Release 2.597                      Nov  4 1999  12:25:28

Usage:ssl68k linkfile [outfile] [-options]
  @file             Read option file
  -#macro[=replace] Define argumentfree macro
  -L=path           Set library directory
  -M                Dump loader map
  -P=digit          Enable PEARL-interface
     0                 No
     1                 PEARL80
     2                 PEARL90
  -T=address        Set <.text>-section address
  -V                Verbose messages
  -z                Add debug informations to library
  -0                Link 68000 Code
  -2                Link 68020 Code
  -3                Link CPU32 Code

  --fpu             Link Code for FPU
```

C.8 sslppc

```
Crest Shared-Library-Linker SSL    PPC-Version for Linux x86
Copyright (C) 1987-1999            by Stephan Litsch
Release 2.597                      Nov  4 1999  12:28:00

Usage:sslppc linkfile [outfile] [-options]
  @file             Read option file
  -#macro[=replace] Define argumentfree macro
  -L=path           Set library directory
  -M                Dump loader map
  -P=digit          Enable PEARL-interface
     0                 No
     1                 PEARL80
     2                 PEARL90
  -T=address        Set <.text>-section address
  -V                Verbose messages
  -z                Add debug informations to library

  --fpu             Link Code for FPU
```

C.9 clm68k

```
Crest Library Manager CLM          68K-Version for Linux x86
Copyright (C) 1987-1999            by Stephan Litsch
Release 2.597                      Nov  4 1999  12:25:28

Usage:clm library [-options]
  @file       Read argument file
  -M          Dump loader map
  -V          Verbose all infos
```

C.10 clmppc

```
Crest Library Manager CLM          PPC-Version for Linux x86
Copyright (C) 1987-1999            by Stephan Litsch
Release 2.597                      Nov  4 1999  12:28:00

Usage:clm library [-options]
  @file       Read argument file
  -M          Dump loader map
  -V          Verbose all infos
```

C.11 cop68k

```
Crest Object Inspector COP         68K-Version for Linux x86
Copyright (C) 1987-1999            by Stephan Litsch
```

```
Release 2.597                    Nov  4 1999  12:25:28

Usage:cop crest-binaryfile [-options]
  @file       Read argument file
  -O          OBJ-Flags
      R          reading of section info
      B          building of section info
      P          .shstrtab
      S          .strtab
      Y          .symtab
      T          .text
      t          .rel.text
      D          .data
      d          .rel.data
      M          .crest_debug_module
      +          additional hexdump
  -L          LIB-Flags
      R          reading of section info
      B          building of section info
      P          .shstrtab
      S          .strtab
      L          .crest_library_dir
      +          additional hexdump
  -R          REF-Flags
      R          reading of section info
      B          building of section info
      P          .shstrtab
      S          .strtab
      L          .crest_reference_dir
      +          additional hexdump
  -D          DBG-Flags
      R          reading of section info
      B          building of section info
      H          Elf32_Ehdr
      P          .shstrtab
      S          .strtab
      Y          .symtab
      T          .text
      t          .rel.text
      D          .data
      d          .rel.data
      G          .crest_debug_program
      +          additional hexdump
  -F          Function flags
      R          reading of section info
      B          building of section info
      P          .shstrtab
      S          .strtab
      O          .crest_objects
        X           Functions
        V           Variables
        G           Registers
```

```
      C          .crest_structures
      U          .crest_unions
      E          .crest_enums
      Y          .crest_typedefs
      +          additional hexdump
  -M          Module flags
      R          reading of section info
      B          building of section info
      P          .shstrtab
      S          .strtab
      A          .crest_macros
      T          .crest_types
      L          .crest_member_table
      O          .crest_objects
        X           Functions
        V           Variables
      C          .crest_structures
      U          .crest_unions
      E          .crest_enums
      Y          .crest_typedefs
      M          .crest_members
      I          .crest_file_dir
      F          .crest_func_dir
      D          .crest_compounds
      K          .crest_breakpoints
      +          additional hexdump
  -P          Program flags
      R          reading of section info
      B          building of section info
      P          .shstrtab
      S          .strtab
      E          .crest_entry_dir
      O          .crest_obj_dir
      F          .crest_ref_dir
      +          additional hexdump
  -B          Generate Breakpoint-Sources
  -V          Verbose all infos
```

C.12 copppc

```
Crest Object Inspector COP         PPC-Version for Linux x86
Copyright (C) 1987-1999            by Stephan Litsch
Release 2.597                      Nov  4 1999  12:28:00

Usage:cop crest-binaryfile [-options]
  @file       Read argument file
  -O          OBJ-Flags
      R          reading of section info
      B          building of section info
      P          .shstrtab
```

```
    S          .strtab
    Y          .symtab
    T          .text
    t          .rel.text
    D          .data
    d          .rel.data
    M          .crest_debug_module
    +          additional hexdump
-L          LIB-Flags
    R          reading of section info
    B          building of section info
    P          .shstrtab
    S          .strtab
    L          .crest_library_dir
    +          additional hexdump
-R          REF-Flags
    R          reading of section info
    B          building of section info
    P          .shstrtab
    S          .strtab
    L          .crest_reference_dir
    +          additional hexdump
-D          DBG-Flags
    R          reading of section info
    B          building of section info
    H          Elf32_Ehdr
    P          .shstrtab
    S          .strtab
    Y          .symtab
    T          .text
    t          .rel.text
    D          .data
    d          .rel.data
    G          .crest_debug_program
    +          additional hexdump
-F          Function flags
    R          reading of section info
    B          building of section info
    P          .shstrtab
    S          .strtab
    O          .crest_objects
      X           Functions
      V           Variables
      G           Registers
    C          .crest_structures
    U          .crest_unions
    E          .crest_enums
    Y          .crest_typedefs
    +          additional hexdump
-M          Module flags
    R          reading of section info
    B          building of section info
```

```
      P          .shstrtab
      S          .strtab
      A          .crest_macros
      T          .crest_types
      L          .crest_member_table
      O          .crest_objects
        X           Functions
        V           Variables
      C          .crest_structures
      U          .crest_unions
      E          .crest_enums
      Y          .crest_typedefs
      M          .crest_members
      I          .crest_file_dir
      F          .crest_func_dir
      D          .crest_compounds
      K          .crest_breakpoints
      +          additional hexdump
  -P          Program flags
      R          reading of section info
      B          building of section info
      P          .shstrtab
      S          .strtab
      E          .crest_entry_dir
      O          .crest_obj_dir
      F          .crest_ref_dir
      +          additional hexdump
  -B          Generate Breakpoint-Sources
  -V          Verbose all infos
```

C.13 cmake

```
Crest CMAKE                          68K-Version for Linux x86
Copyright (C) 1987-1999              by Stephan Litsch
Release 2.598                        Nov  5 1999  12:07:21

Usage:cmake [makefile] [-options]
  @file             Read argument file
  -#macro[=replace] Define argumentfree macro
  -A                Make always
  -I                Ignore errors while processing
  -T                Trace commands
  -V                MAKE echoing commands

  -K                Define macro "__M68K__"
  -P                Define macro "__MPPC__"
  -0                Define macro "__MC68000__"
  -2                Define macro "__MC68020__"
  -3                Define macro "__CPU32__"
  --fpu             Define macro "__FPU__"
```

```
-?                  Dump usage
```

Index