SJC – Small Java Compiler

Handbuch für Anwender

Stefan Frenz

Stand 2009/10/17
Compiler-Version: 173

Java ist eingetragener Name von Sun Microsystems.

Inhaltsverzeichnis

1 Schnelleinstieg

Für einen Schnelleinstieg bietet sich das exec-Paket (siehe [sjc]) an, in dem alle erforderlichen Dateien zur Erzeugung von 32 Bit und 64 Bit Betriebssystemen enthalten sind:

Datei	Verwendung
b64_dsk.bin	Disketten-Bootlader für 64 Bit Zielsysteme
bootconf.txt	Konfigurationsdatei für den Bootschreiber
bts_dsk.bin	Disketten-Bootlader für 32 Bit Zielsysteme
compile	Ausführbare Datei des Compilers für Linux
compile.exe	Ausführbare Datei des Compilers für Windows
hello.java	Hello-World Beispiel-Programm
rte.java	Grundgerüst für Laufzeitumgebung

Unter Verwendung dieses Pakets werden zur Übersetzung eines Systems keine weiteren Programme benötigt, für den Test des erzeugten Systems bietet sich QEmu (siehe [qemu]) an, dessen Installation in für dieses Kapitel als vorhanden vorausgesetzt wird.

Die nächsten drei Unterkapitel erläutern den Aufruf des Compilers unter Windows und Linux sowie den Start des erzeugten Systems in QEmu. Dabei wird das mitgelieferte und in Kapitel 2 besprochene "Hello World"-Programm für ein 32 Bit Zielsystem übersetzt.

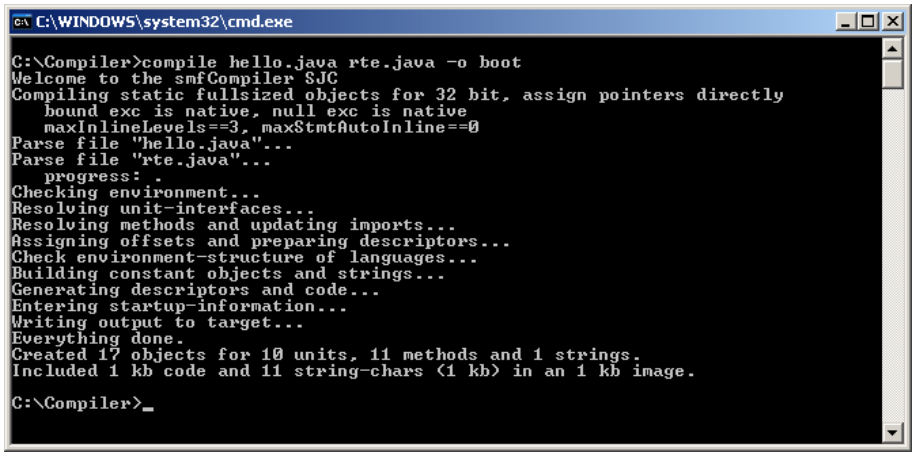
Zur Erstellung eines eigenen Systems wird zumindest die Lektüre der Kapitel 2, 3.4, 3.8, 5.1, 6.5 und 7 empfohlen, Beispiele finden sich in Kapitel 12. Des weiteren ist unter [picos] ein Beispielsystem verfügbar, das den 32 Bit Protected Mode und den 64 Bit Long Mode unterstützt.

1.1 Windows

Ist das exec-Paket im Verzeichnis C:\Compiler entpackt, kann eine erste Überprüfung der Installation über diese Kommandozeile vorgenommen werden:

```
C:\Compiler>compile hello.java rte.java -o boot
```

Folgende Ausgabe ist zu erwarten:



Dadurch werden die beiden Dateien boot_flp.img und syminfo.txt erzeugt, ersteres enthält ein Diskettenabbild des Beispielprogramms, letzteres die in Kapitel 5.2 näher erläuterten Symbolinformationen.

1.2 Linux

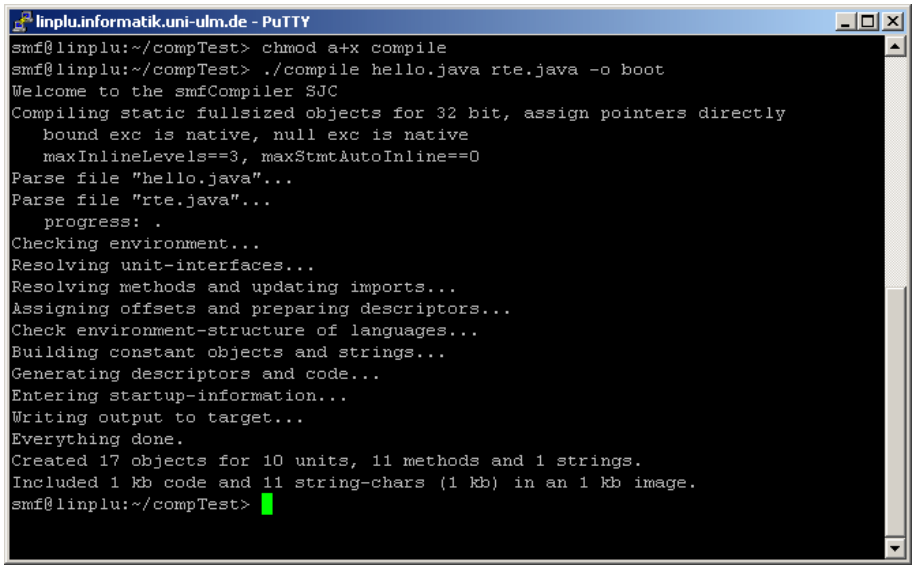
Ist das exec-Paket im Verzeichnis ~/compiler entpackt, muss unter Umständen das executable-Flag noch gesetzt werden:

```
smf@linplu:~/compiler> chmod a+x compile
```

Eine erste Überprüfung der Installation kann über diese Kommandozeile vorgenommen werden:

```
smf@linplu:~/compiler> ./compile hello.java rte.java -o boot
```

Folgende Ausgabe ist zu erwarten:



Wie unter Windows (siehe voriges Unterkapitel) werden dadurch die beiden Dateien boot_flp.img (Diskettenabbild) und syminfo.txt (Symbolinformationen) erzeugt.

1.3 QEmu

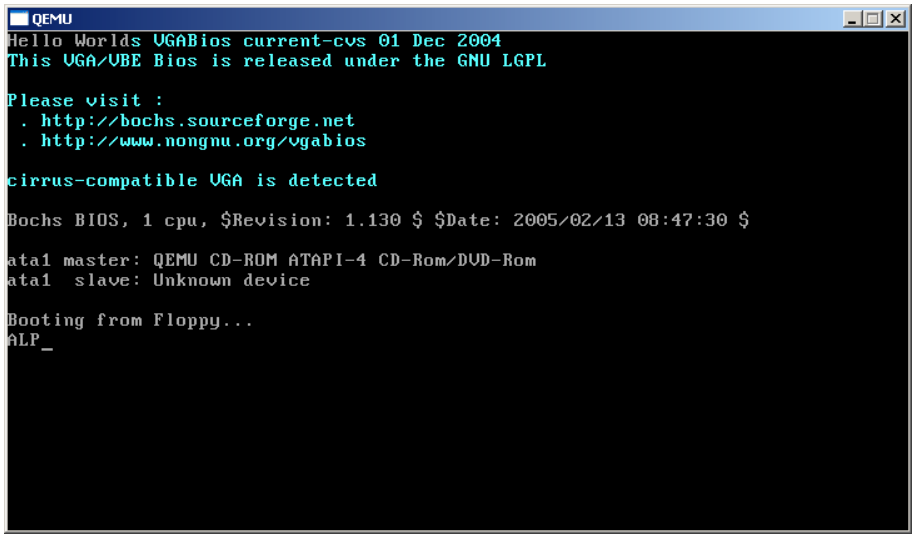
Der Start des erzeugten Diskettenabbilds in QEmu erfolgt unter Windows über die Kommandozeile

```
C:\Compiler>\Programme\QEmu\qemu.exe -m 32 -boot a -fda boot_flp.img
```

bzw. unter Linux mittels

```
smf@linplu:~/compiler> qemu -m 32 -boot a -fda boot_flp.img
```

und sollte (je nach QEmu- und BIOS-Version) etwa folgende Ausgabe erzeugen:



2 Hello-World

Im diesem Kapitel wird der Quellcode für ein einfaches "Hello World"-Programm, die Kommandozeilen-Anweisung zur Übersetzung (alle dafür benötigten Dateien sind im exec-Paket, siehe [sjc] enthalten) sowie das Starten in QEmu vorgestellt. Nachfolgend werden die einzelnen Schritte erklärt und die automatisch von Compiler vorgenommenen Einstellungen erläutert.

2.1 Quellcode

Programm (hello.java)

```
package kernel;
public class Kernel {
  private static int vidMem=0xB8000;
  public static void main() {
    print("Hello World");
    while(true);
  }
  public static void print(String str) {
    int i;
    for (i=0; i<str.length(); i++) print(str.charAt(i));
  }
  public static void print(char c) {
    MAGIC.wMem8(vidMem++, (byte)c);
    MAGIC.wMem8(vidMem++, (byte)0x07);
  }
}
```

Laufzeitumgebung (rte.java)

```
package java.lang;
import rte.SClassDesc;
public class Object {
  public SClassDesc _r_type;
  public Object _r_next;
  public int _r_relocEntries, _r_scalarSize;
}
package java.lang;
public class String {
  private char[] value;
  private int count;
  public int length() {
    MARKER.inline();
    return count;
  }
```

```
  public char charAt(int i) {
    MARKER.inline();
    return value[i];
  }
}
package rte;
public class SArray {
  public int length, _r_dim, _r_stdType;
  public SClassDesc _r_clssType;
}
package rte;
public class SClassDesc {
  public SClassDesc parent;
  public SIntfMap implementations;
}
package rte;
public class SIntfDesc { }
package rte;
public class SIntfMap {
  public SIntfDesc owner;
  public SIntfMap next;
}
package rte;
public class SMthdBlock { }
package rte;
public class DynamicRuntime {
  public static Object newInstance(int scalarSize, int relocEntries,
      SClassDesc type) { while(true); }
  public static SArray newArray(int length, int arrDim, int entrySize,
      int stdType, SClassDesc clssType) { while(true); }
  public static void newMultArray(SArray[] parent, int curLevel,
      int destLevel, int length, int arrDim, int entrySize, int stdType,
      SClassDesc clssType) { while(true); }
  public static boolean isInstance(Object o, SClassDesc dest,
      boolean asCast) { while(true); }
  public static SIntfMap isImplementation(Object o, SIntfDesc dest,
      boolean asCast) { while(true); }
  public static boolean isArray(SArray o, int stdType,
      SClassDesc clssType, int arrDim, boolean asCast) { while(true); }
  public static void checkArrayStore(SArray dest, Object newEntry) {
      while (true); }
}
```

Bootkonfiguration (bootconf.txt)

```
section floppy32
section default
destfile boot_flp.img
blocksize 512
maximagesize 1474048
readbuf bts_dsk.bin
offset 30.l4 value imageaddr
offset 34.l4 value unitaddr
offset 38.l4 value codeaddr
offset 42.l4 crc 0x82608EDB
offset 46.l2 value blockcnt
writebuf
appendimage
endsection
```

2.2 Übersetzen und Emulieren

Sind die im letzten Unterkapitel angegebenen Dateien, eine nativ ausführbare Version des Compilers sowie die für die Erstellung des Bootsektors erforderliche Datei bts_dsk.bin in einem Verzeichnis abgelegt, kann das "Hello World"-Programm mit dem Befehl

```
compile hello.java rte.java -o boot
```

übersetzt werden. Die Datei bts_dsk.bin ist im exec-Paket (siehe [sjc]) enthalten oder kann anhand des Quellcodes aus Kapitel 13 mit yasm (siehe [yasm]) erstellt werden. Die Ausgabe des Compilers sollte folgender entsprechen (Screenshot siehe Kapitel 1):

```
Welcome to the smfCompiler SJC
Compiling static fullsized objects for 32 bit, assign pointers directly
   bound exc is native, null exc is native
   maxInlineLevels==3, maxStmtAutoInline==0
Parse file "hello.java"...
Parse file "rte.java"...
   progress: .
Checking environment...
Resolving unit-interfaces...
Resolving methods and updating imports...
Assigning offsets and preparing descriptors...
Check environment-structure of languages...
Building constant objects and strings...
Generating descriptors and code...
Entering startup-information...
Writing output to target...
Everything done.
Created 17 objects for 10 units, 11 methods and 1 strings.
Included 1 kb code and 11 characters (1 kb) in an 1 kb image.
```

Nach erfolgreicher Übersetzung existiert die Datei boot_flp.img, die entweder auf eine Diskette geschrieben (zum Beispiel unter Windows mit rawrite, siehe [rawrite], unter Linux mit dd) oder direkt in einem Emulator wie QEmu verwendet werden kann. Wurde QEmu (siehe [qemu]) ordnungsgemäß im Verzeichnis C:\Programme\QEmu installiert, kann das erstellte Programm mit

```
C:\Programme\Qemu\qemu.exe -m 32 -boot a -fda boot_flp.img
```

gestartet werden, ein Screenshot findet sich in Kapitel 1.

2.3 Erläuterungen

Programm

Die statische Variable **vidMem** wird vom Compiler als Klassenvariable angelegt. Somit kann ihr Wert im Diskettenabbild mit dem Wert 0xB8000, dem Start des Bildschirmspeichers bei VGA-kompatiblen Grafikkarten, initialisiert werden.

Die Methode **main()** wird vom Code im Bootsektor gerufen und bildet somit den Einstieg in den Java-Code. Hier wird die Methode zur Ausgabe eines Strings gerufen und danach der Prozessor in einer Endlosschleife gefangen.

Die Ausgabe eines Strings ist in der Methode **print(String str)** implementiert. Hier wird für jeden Buchstaben des Strings die Methode zur Ausgabe eines Zeichens gerufen. Im Gegensatz zu SunJava können Variablendeklarationen ausschließlich zu Beginn einer Methode erfolgen, insbesondere gibt es also keine Block-lokalen Variablen.

In der Methode **print(char c)** wird das übergebene Zeichen, vielmehr die unteren acht Bit des Zeichens, mit Hilfe von **MAGIC.wMem8(.)** an die aktuelle Bildschirmposition kopiert, die dort von der Grafikkarte im Textmodus als ASCII-Zeichen interpretiert wird. Durch das Postinkrement der Variable **vidMem** wird die Bildschirmposition auf das nächste Byte gesetzt, also auf die Adresse der Farbe des soeben gesetzten Zeichens. Die Farbe wird über den zweiten **MAGIC.wMem8(.)** Befehl auf 0x07 entsprechend „grau auf schwarz“ gesetzt, durch das Postinkrement zeigt die Variable **vidMem** nun auf das nächste Zeichen im Videospeicher.

Laufzeitumgebung

Die Wurzel der Typhierarchie ist die Klasse **java.lang.Object**, benötigt werden in dieser Klasse zumindest die Instanzvariablen **_r_type**, **_r_next**, **_r_relocEntries** und **_r_scalarSize** (in dieser Reihenfolge). Im angegebenen „Hello World“-Programm werden die Eintrage **_r_next**, **_r_relocEntries** und **_r_scalarSize** nicht verwendet, so dass auf diese auch verzichtet werden könnte (Compiler-Option **-l**).

Die Implementierung von String ist in diesem Beispiel Java-konform gewählt, so dass auf die Array- und Längen-Variable nicht direkt zugegriffen werden kann. Der Array-Typ **char[]** ist ebenfalls Java-konform, wobei in diesem Beispiel kein Gebrauch von Unicode-Zeichen gemacht wird und somit auch eine **byte[]**-Implementierung (Compiler-Option **-y**) ausreichen würde.

Die Anweisung **MARKER.inline()** markiert die Methoden **length()** und **charAt(int)** derart, dass der Compiler versucht, statt eines tatsächlichen Unterprozeduraufrufs den Code der jeweiligen Methode direkt beim Aufrufer "einzukleben" (siehe Kapitel 7.2).

Die Laufzeitumgebung muß die in Kapitel 6 als erforderlich beschriebenen Typen und Routinen bereitstellen, da diese zur Vereinfachung und Beschleunigung des Compilers vorausgesetzt werden, auch wenn wie in diesem Beispiel nur die wenigsten tatsächlich gerufen werden.

Bootkonfiguration

- Die **floppy32**-Section erstreckt sich von der Deklaration des Namens bis zum Schlüsselwort **endsection**.
- Die **floppy32**-Section hat den Alias **default**, der vom Compiler voreingestellt ist. Somit ist beim Aufruf des Compilers keine Option zur Angabe der Section erforderlich.
- Der Name der auszugebenden Datei ist **boot_flp.img**. Hier kann bei Bedarf auch ein Pfad angegeben werden (unter Windows mit **** und unter Linux mit **/** als Trenner).
- Die Größe einer Dateieinheit beträgt 512 Bytes entsprechend einem Diskettensektor.
- Die maximale Größe des Speicherabbilds darf 1474048 Bytes nicht überschreiten. Diese Obergrenze ergibt sich aus den 2880 Sektoren entsprechend 1474560 Bytes einer Diskette abzüglich des zusätzlich benötigten Bootsektors mit 512 Bytes.
- Der Bootsektor soll aus Datei **bts_dsk.bin** gelesen werden.
- An den Offsets 30, 34, 38, 42 und 46 werden die für den Bootvorgang relevanten Größen sowie eine Prüfsumme abgelegt.
- Der Bootsektor soll in die Zieldatei geschrieben werden.
- Das compilierte Speicherabbild soll in die Zieldatei geschrieben werden.

Übersetzen

Der Compiler akzeptiert Optionen und Dateinamen in beliebiger Reihenfolge. Optionen mit Parameter müssen dabei jedoch zusammenbleiben (im obigen Beispiel **-o boot**). Anstelle von einzelnen Dateinamen kann auch der Name eines Verzeichnisses angegeben werden, der Compiler versucht dann, alle Dateien in diesem Verzeichnis und dessen Unterverzeichnissen als Eingabe zu verwenden. Sollen die vorhandenen Unterverzeichnisse nicht rekursiv durchsucht, sondern nur die Dateien im angegeben Verzeichnis übersetzt werden, muss ein **:** an das Verzeichnis angehängt werden.

Falls nichts anderes eingestellt wurde, verwendet der Compiler die Grundeinstellungen **-t ia32** (ia32-Architektur), **-a 1m** (Adresse des Speicherabbildes bei 1 MB) und **-s 128k** (maximale Größe des Speicherabbildes ist 128 KB).

Ohne den Parameter **-o boot** legt der Compiler das erzeugte Speicherabbild unverändert in der Datei raw_out.bin ab. Im Beispiel oben erweitert er es hingegen um den aus der Datei bts_dsk.bin geladenen Bootsektor und legt, nachdem er die zur Initialisierung erforderlichen Informationen dort eingetragen hat, das gesamte Diskettenabbild in der Datei floppy.img ab.

Im letzten Arbeitsschritt werden die Symbolinformationen als Text in der Datei syminfo.txt (siehe Kapitel 5.2) gespeichert. Die auf der Konsole ausgegebenen Daten

```
Created 18 objects for 10 units, 11 methods and 1 strings.

Included 1 kb code and 11 characters (1 kb) in an 1 kb image.
```

stellen eine Zusammenfassung mit aufgerundeten Größenangaben dar (0 Bytes sind 0 Kilobyte, 1 bis 1024 Bytes sind 1 Kilobyte, 1025 bis 2048 Bytes sind 2 Kilobyte etc.).

3 Einführung

3.1 Entstehung, Rechtliches, Bezugsquelle und Danksagung

Der hier beschriebene Compiler wurde in der Freizeit entwickelt und erhebt keinerlei Anspruch auf Vollständigkeit, Fehlerfreiheit oder Verwendbarkeit für einen bestimmten Zweck. Insbesondere wird jede Haftung für entstandene Schäden oder Gewinnausfälle abgelehnt. Dieses Handbuch ist keine Einführung in die Programmierung mit Java oder in die Systemprogrammierung, sondern soll dem bereits versierten Leser den Einstieg in die Verwendung des Small Java Compiler erleichtern. Der Quelltext, vorübersetzte Executables für 32 Bit Windows und Linux, dieses Handbuch sowie eine kurze Übersicht über die öffentlich verfügbaren Versionen befinden sich unter [sjc].

Bedanken möchte ich mich bei den Herren Prof. Dr. Peter Schulthess und Prof. Dr. Michael Schöttner, die in zahlreichen Diskussionen an den Vorüberlegungen zu diesem Compiler mitwirkten. Mein Dank gilt auch Herrn Dipl. Inf. Patrick Schmidt, der den Emulator für den entwickelten Pseudo-SSA-Befehlssatz und Tests für native Windows-Programme schrieb.

3.2 Grundlagen und Zielsetzung

Der entwickelte Small Java Compiler erzeugt Maschinencode, der direkt in einem Speicherabbild (siehe Kapitel 3.3) des Zielsystems abgelegt wird. Der Compiler erzeugt alle notwendigen Strukturen und Symbolinformationen, es wird also kein zusätzlicher Binder oder Lader benötigt. Durch die integrierte Erzeugung der Ausgabedatei lassen sich native Betriebssysteme ebenso erstellen wie unter Linux oder Windows lauffähige Programme (siehe dazu auch Kapitel 11). Bei der Entwicklung standen dabei folgende Ziele im Mittelpunkt:

- Weitgehende Quellcode-Kompatibilität mit SunJava: Die Quellcode-Kompatibilität mit SunJava (siehe [java]) zur Wiederverwendung freier Tools und Entwicklungswerkzeuge erleichtert die Erstellung sowie das Testen von Programmen. Teilmodule können mit SunJava übersetzt und unabhängig vom SJC und der Zielplattform getestet werden.
- Unterstützung unterschiedlicher Zielplattformen: Die Unterstützung unterschiedlicher Zielplattformen mit 8, 16, 32 und 64 Bit CPUs zeigt sich in der allgemein gehaltenen Schnittstelle zwischen Frontend und Backend (siehe Kapitel 3.6 und 3.7).
- Unterstützung hardwarenaher Programmierung: Um ein vollständiges System einfach erstellen und typsicher programmieren zu können, müssen ausgewählte Module des Zielsystems direkt auf die Hardware zugreifen können. Der Compiler stellt dafür alle notwendigen Routinen und Erweiterungen bereit (siehe Kapitel 7).
- Optionale Optimierung des Maschinencodes: Die Erzeugung des Maschinencodes (siehe Kapitel 3.6) ermöglicht eine Optimierung auf Basis einzelner Methoden. Die Instruktionen werden durch das Backend selbst verwaltet, so dass für jede Architektur zielgerichtet optimiert werden kann.
- Fähigkeit zur Selbstübersetzung: Der Compiler kann den eigenen Quellcode unabhängig vom laufenden System für alle Architekturen übersetzen, für die eine vollständige Implementierung des Backends zur Verfügung steht (siehe Kapitel 3.7).
- Verständliche Teilschritte und modularer Aufbau: Um die Erweiterbarkeit sicherstellen zu können und die potenzielle Fehlersuche zu erleichtern, ist der Vorgang der Übersetzung in verständliche und nachvollziehbare Teilschritte unterteilt. Die einzelnen Packages greifen dabei niemals auf Subpackages anderer Packages zu. Durch die genaue Trennung zwischen Frontend und Backend mit klar definierten Schnittstellen ist die Erweiterung und Änderung einzelner Module unabhängig vom restlichen Compiler (siehe Kapitel 3.6).

3.3 Aufbau des Speicherabbilds

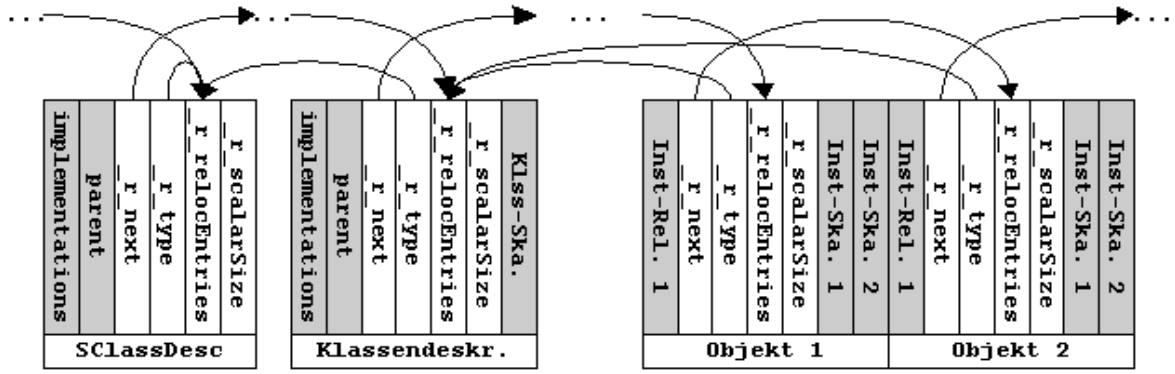
Befindet sich der Compiler wie üblich nicht im Bootstrap-Mode (Option **`-s 0`**, siehe Kapitel 5.1), so erzeugt er ein Speicherabbild des Zielsystems. Am Beginn dieses Speicherabbilds wird folgender Header eingebaut (kann mit der Option **`-P`** deaktiviert oder durch einen eigenen Header ersetzt werden, siehe Kapitel 5.1):

Offset	Typ	Inhalt
0	Integer	Startadresse des Speicherblocks (identisch zu **`MAGIC.imageBase`**)
4	Integer	Größe des verwendeten Speichers
8	Zeiger	Klassendeskriptor der Klasse **`kernel.Kernel`**
12	Zeiger	Erster Opcode der Methode **`kernel.Kernel.main()`**
16	Zeiger	Erstes Objekt im Heap
20	Zeiger	Speicherbereich zur RAM-Initialisierung (identisch zu **`MAGIC.getRamAddr()`**)
24	Integer	Code-Offset in Methoden (identisch zu **`MAGIC.codeOffset()`**)
28	Integer	Architektur-Parameter

An Offset 4 ist die Größe des verwendeten Speichers abgelegt, durch Addition mit der Startadresse des Speicherblocks ergibt sich die erste freie Speicherstelle nach dem Speicherblock. Der Klassendeskriptor der Klasse **`kernel.Kernel`** wird zur Bereitstellung des Klassenkontextes beim Start der nachfolgend angegebenen Methode **`kernel.Kernel.main()`** benötigt, bei der die Ausführung des Java-Codes beginnt. Soll der Speicherblock nach Objekten durchsucht werden, kann der bei Offset 16 abgelegte Zeiger als Einstieg verwendet werden (nachfolgende Objekte sind über das **`_r_next`**-Feld verkettet, falls nicht mit Option **`-l`** übersetzt wurde, siehe Kapitel 5.1). Der Wert bei Offset 28 enthält den 32-Bit-Wert 0x550000AA, der mit der Zeigergröße (Bits 15-8) und der Stackalignierung (Bits 23-16) verodert wurde. Ab Offset 32 folgen dann ausschließlich typisierte Objekte, ein Zeiger auf das erste Objekt befindet sich wie in obiger Tabelle aufgeführt an Offset 16 des Headers.

3.4 Objektaufbau

Der Compiler erzeugt (ohne die Option **`-M`**) ausschließlich typisierte Objekte für das Speicherabbild. Der Zeiger auf ein Objekt zeigt auf das erste skalare Feld. Weitere Skalare befinden sich an höheren Adressen, die Referenzen des Objekts sind relativ zum Zeiger auf das Objekt an niedrigeren Adressen untergebracht, wobei der erste Zeiger einen Zeiger auf den Typ des Objekts enthält. Wird ohne die Option **`-l`** übersetzt (also mit vollen Objektinformationen), sehen zwei nacheinander allozierte Objekte des gleichen Typs, welcher an Klassenvariablen einen **`int`**-Wert sowie an Instanz-Variablen einen Zeiger und zwei **`int`**-Werte besitzt, im Heap für 32 Bit Systeme so aus:



Es existieren üblicherweise also nur Zeiger auf Objekte, es gibt keine Zeiger auf Objektfelder.

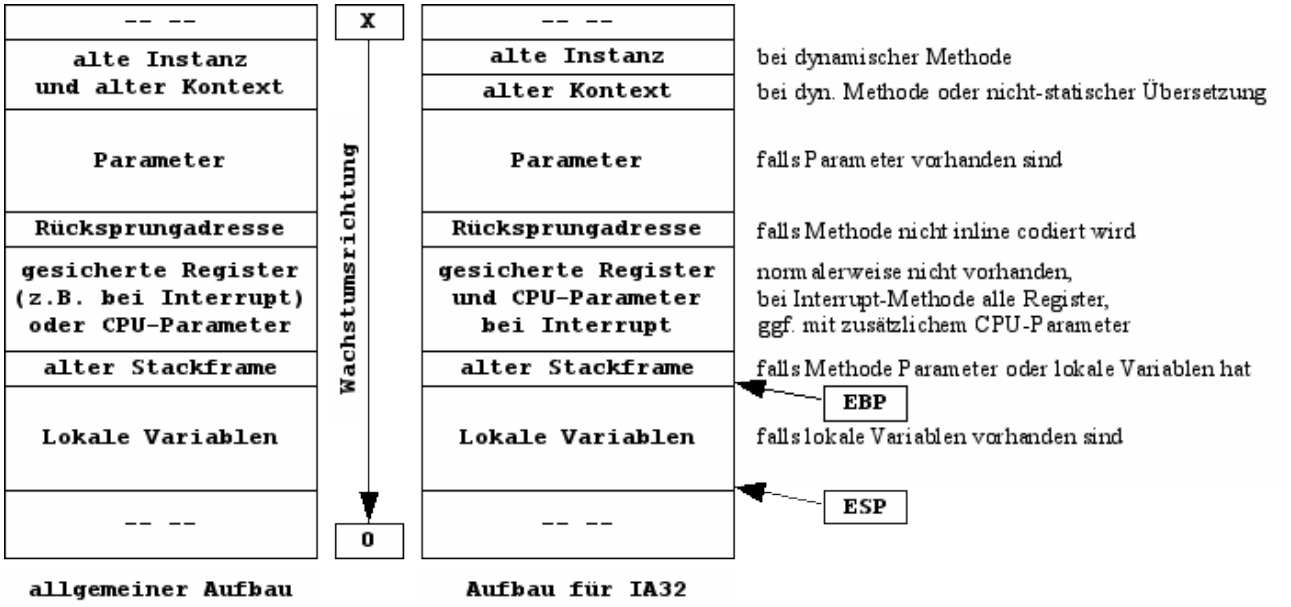
Wird hingegen mit der Option **-M** übersetzt, werden die Skalare eines Objekts indirekt adressiert. Dazu existieren zwei weitere Felder im Objekt: **_r_indirScalarAddr** und **_r_indirScalarSize**, wobei ersteres die Startadresse des Skalarbereichs und letzteres deren Größe angibt. Diese Option ist für einen Betrieb im Cluster gedacht, wo die Referenzen eines Objekts sowie die für den Heapaufbau relevanten Informationen strikt konsistent sein sollen, die Skalare eines Objekts aber unter Umständen einem anderen Konsistenzmodell unterliegen sollen.

In diesem Handbuch wird, sofern nicht anders angegeben, davon ausgegangen, dass nicht mit der Option **-M** übersetzt wird. Eine Übersicht über die Unterschiede findet sich in Kapitel 9.

3.5 Stack-Aufbau

Der Compiler erzeugt im Normalfall (Ausnahme: nativ deklarierte C-Funktionen) alle Parameter von links nach rechts und legt diese von höheren zu niedrigeren Adressen auf dem Stack ab. Dabei wird entsprechend der Zielarchitektur die Mindestgröße eines Stackeintrags berücksichtigt, zum Beispiel belegt ein Byte-Eintrag auf einem ATmega-System genau ein Byte, auf einem IA32-System hingegen vier Bytes.

Je nach Übersetzungsmodus, Eigenschaften der aufgerufenen Methode und Abhängigkeiten der aufrufenden Methode werden unterschiedliche Werte auf dem Stack gesichert, die bei einer Traversierung des Stackframes oder bei einem Zugriff über Inline-Assembler berücksichtigt werden müssen. Der maximal erzeugte Stackframe sieht wiefolgt aus:



Grundsätzlich obliegt der Zielarchitektur der korrekte Aufbau des Stacks. Dies kann je nach Architektur unterschiedliche Aufteilungen ergeben, die rechts angegebenen Erklärungen und Register-Hinweise gelten nur für den dargestellte IA32-Aufbau. Insbesondere wird bei der IA32-Architektur der alte Stackframe-Pointer nicht gesichert, wenn die Methode weder Parameter noch lokale Variablen besitzt. Diese Optimierung kann über die IA32-spezifische Option **-T nsop** deaktiviert werden, der Zeiger auf den vorangehenden Stackframe wird dann immer gesichert.

3.6 Compiler-Struktur

Der Compiler ist in einzelne Module zerlegt, deren Aufteilung sich in den Packages widerspiegelt. Mit Ausnahme sehr kleiner Packages wie zum Beispiel **memory** und **output** ist die Schnittstelle zu einem Modul im Rootpackage abgelegt, die dazugehörenden Implementierungen in den darunterliegenden Subpackages. In Subpackages wird ausschließlich auf Klassen des eigenen Packages und auf Rootpackages zugegriffen, so dass einzelne Subpackages vollständig voneinander unabhängig sind.

Modul	Beschreibung
backend	Architektur des Zielsystems
compbase	Allgemeine Einstellungen und Basistypen des Compilers
compression	Optionale Kompression des Speicherabbilds
debug	Schreiber von Debug-Informationen
frontend	Verarbeitung von Quelldateien
memory	Speicherabbildverwaltung
osio	Zugriff auf I/O-Funktionen des laufenden Systems
output	Konvertierung des Speicherabbilds in das gewünschte Ausgabeformat
real	Wählbare Unterstützung für Fließkomma-Zahlen
symbols	Optionale Ausgabe von Symbolinformationen
ui	Benutzerschnittstelle (Start des Compilers)

Die Quelldateien werden vom Compiler in den folgenden zwölf aufeinander aufbauenden Schritten übersetzt, wobei eine Phase nur gestartet wird, wenn die vorhergehende vollständig erfolgreich abgeschlossen wurde:

1. In Abhängigkeit vom laufenden System werden Instanzen zum Zugriff auf das Dateisystem und zur Benutzerinteraktion erzeugt, die Parameter werden von der Kommandozeile oder aus Umgebungsvariablen ausgelesen. Alle nachfolgenden Phasen sind vom laufenden System unabhängig.
2. Die für den aktuellen Übersetzungslauf eingestellten Parameter werden geprüft und im aktuellen Kontext gespeichert. Die Implementierung der ausgewählten Zielarchitektur sowie die Verwaltung des Speicherabbilds wird instanziert und initialisiert. Somit stehen Codegenerator und Parameter wie Zeigergröße oder Stackalignierung fest.
3. Die angegebenen Quelldateien werden an das Frontend gegeben, wo sie nach Ermittlung des zuständigen Sprachmoduls in dessen Parser zu einem undekorierten Parse-Tree verarbeitet werden. Obwohl der Compiler hauptsächlich für eine Java-ähnliche Sprache entwickelt wurde, können unterschiedlichste Sprachen zu einem gemeinsamen System übersetzt werden.
4. Der erzeugte undekorierte Parse-Tree wird geprüft, angegebene Referenzen aufgelöst und Typen geprüft. Grundlegende Informationen wie öffentliche Variablen oder Methoden der zu erzeugenden Klassen werden in diesem Schritt ermittelt und im Parse-Tree vermerkt.
5. Die Methodenblöcke werden geprüft, Referenzen werden aufgelöst, zur Code-Erzeugung erforderliche Informationen werden im Parse-Tree vermerkt.
6. Die Offsets von Variablen und Methoden werden in Abhängigkeit vom gewählten Speichermodell und den eingestellten Parametern berechnet. Nach diesem Schritt stehen alle zur Code-Erzeugung erforderlichen Informationen bereit.
7. Das Laufzeitsystem wird auf Gültigkeit geprüft, insbesondere wird das Vorhandensein gerufener Laufzeitroutinen sichergestellt.
8. Alle Objekte, insbesondere also Klassendeskriptoren und Codeblöcke, werden erzeugt und im Speicherabbild abgelegt.
9. Optional können die vom Compiler erzeugten Symbolinformationen in das Speicherabbild integriert werden.
10. Optional kann das Speicherabbild komprimiert oder verschlüsselt werden, es folgt ein weiterer Übersetzungsdurchlauf zur Erzeugung eines Dekompressors bzw. Entschlüsselers.

11. Das Speicherabbild wird entsprechend des gewählten Ausgabeformats weiterverarbeitet, zum Beispiel als Hex-Datei zur Speicherung in einem ROM oder als Binär-Datei kombiniert mit einem Bootlader zur direkten Ausführung.
12. Die Debug-Symbolinformationen (siehe Kapitel 5.2) werden gespeichert.

Wurden alle Schritte erfolgreich abgeschlossen, sind die ausgegebenen Dateien gültig.

3.7 Implementierte Module

Die folgende Tabelle bietet eine Übersicht derzeit implementierter Module des Compilers mit kurzer Beschreibung. Bei einer Übersetzung sind immer alle Implementierungen in **`frontend`** und jeweils genau eine Implementierung in **`backend`** und **`output`** aktiv (Symbolinformationen und Kompression sind optional).

Modul	Implementierung	Beschreibung
`frontend`	`binimp.*`	Import von Binärdaten (siehe Kapitel 4.1)
	`clist.*`	Textliste mit Quelldateien (siehe Kapitel 4.2)
	`sjava.*`	Java-Quelldateien
`backend`	`x86.AMD64`	64 Bit Code für AMD64 im Long Mode
	`x86.IA32`	32 Bit Code für IA32 im Protected Mode
	`x86.IA32Opti`	Optimierter 32 Bit Code für IA32 im Protected Mode
	`x86.IA32RM`	32 Bit Code für IA32 im Real Mode
	`ssa.*`	Pseudo-SSA-Code
	`atmel.ATmega`	8 Bit Code für Atmel ATmega Mikrocontroller
	`atmel.ATmegaOpti`	Optimierter 8 Bit Code für Atmel ATmega
`memory`	`*`	Verwaltung des Speicherabbilds
`compression`	`BZL`	BWT/LZW-Kompression
	`LZW`	LZW-Kompression
`osio`	`*`	Zugriff auf das Wirtssystem (Dateien, Bildschirm...)
`output`	`BootOut`	Parametrisierte Ausgabe
	`RawOut`	Unverarbeitete Ausgabe
`real`	`EmulReal`	Fließkomma-Emulation auf Basis von Ganzzahlen
	`NativeReal`	Native Fließkomma-Unterstützung im Compiler
`symbols`	`BootStrapSymbols`	Compiler-konforme Symbolinformationen
	`RawSymbols`	Symbolinformationen als Byte-Stream
	`RTESymbols`	Symbolinformationen im Laufzeit-System
`debug`	`GccInfo`	Debuginfo als gcc-taugliche Assembler-Datei
	`MthdInfo`	Debuginfo als Liste für alle erzeugten Methoden
	`SymInfo`	Standard-Debuginfo mit allen textuellen Infos

Die für die jeweiligen Module definierten Schnittstellen befinden sich im gleichnamigen Rootpackage, die Auswahl der einzelnen Implementierungen erfolgt über einen Admin oder eine Factory im selben Rootpackage.

3.8 Unterschiede zu SunJava

Obwohl eine weitgehende Quellcode-Kompatibilität zwischen SJC und SunJava erreicht wurde, sind einige Spezialitäten von SunJava nicht verfügbar bzw. werden bewusst nicht angeboten, um dem Systemprogrammierer bei Hardware-naher Programmierung fehlerträchtige Konstrukte anzuzeigen. Die hauptsächlichen Unterschiede sind:

- Der Compiler unterstützt ab Version 165 Java-Exceptions (nicht zu verwechseln mit CPU-Exceptions, die wie Hardware- und Software-Interrupts unterstützt werden). Bei Verwendung der Schlüsselworte **`try`**, **`catch`**, **`finally`**, **`throw`** und **`throws`** werden erweiterte Anforderungen an das Laufzeitsystem gestellt (siehe Kapitel 6.6).
- Der Compiler unterstützt keine Interface-Arrays. Diese müssten zur Laufzeit mehrfach und auch beim Auslesen eines Objektes auf ihren Typ geprüft werden, so dass ein einfaches **`Object`**-Array mit Interface-Cast beim Zugriff auf ein Array-Element schlanker im Speicher und schneller in der Ausführung ist.
- Statische Klasseninitialisierungen müssen durch den Aufruf von **`MAGIC.doStaticInit()`** (siehe Kapitel 7.1) angestoßen werden.
- Die Laufzeitbibliothek von SunJava ist nicht Bestandteil des SJC. Dafür besteht hier die Möglichkeit, für sämtliche Systeme wie zum Beispiel Embedded Prozessoren, Windows, Linux oder PC-Clustersysteme wie Plurix gezielt Laufzeitumgebungen mit spezialisierter Funktionsweise zu erstellen.
- Es gibt kein Reflection-API. Anhand der erzeugten Symbolinformationen (siehe Kapitel 5.2) lässt sich jedoch der Typ jedes Objekts über die Speicheradresse des Klassendeskriptors ermitteln. Durch Auswertung von In-Image Symbolinformationen (siehe Kapitel 10) lässt sich eine ähnliche Funktionalität bereitstellen.
- Konvertierungen von Ausdrücken werden bei Zahlen nicht implizit vorgenommen, um unbeabsichtigtes Verhalten zu vermeiden, insbesondere bei für Vorzeichenerweiterungen anfälligen Geräten und ihren Treibern.
- Es werden immer alle angegeben Dateien bzw. alle Dateien in angegebenen Verzeichnissen übersetzt.
- In einer Datei dürfen auch mehrere voneinander unabhängige Klassen deklariert sein.

4 Frontend-Module zusätzlich zu Java-Dateien

4.1 Import von Binärdaten

Dateien mit der Endung **`.bib`** und **`.bim`** werden vom Binärimport-Modul als binäres Byte-Array bzw. als ausführbarer Code importiert. Der Zugriff auf derart importierte Dateien erfolgt anhand ihres Dateinamens (ohne Endung) für Byte-Arrays über die jeweils automatisch angelegte Variable

```
binimp.ByteData.DATEINAME
```

und über

```
MAGIC.inlineBlock(DATEINAME)
```

für Codeblöcke.

Als Alternative zum binären Import von Byte-Arrays bietet sich die Funktion

```
MAGIC.getNamedString(DATEINAME)
```

an, die in Kapitel 4.2 und 7.1 diskutiert wird.

4.2 Textliste mit Quelldateien

Der Compiler unterstützt die Benennung von Quelldateien und Quellverzeichnissen über die Kommandozeile oder auch über Dateien mit der Endung **`.clt`**. Die zu übersetzenden Dateien oder Verzeichnisse sind in **`clt`**-Dateien zeilenweise anzugeben, wobei immer die gesamte Zeile als Datei- oder Verzeichnisname verwendet wird. Eventuell vorhandene Sonderzeichen oder Leerzeichen werden direkt übernommen und nicht entfernt.

Quelldateien und Quellverzeichnisse werden wie auch auf der Kommandozeile (siehe Kapitel 5.1) einzeln angegeben. Verzeichnisse werden im Normalfall rekursiv durchlaufen, wobei alle vorhandenen Dateien mit für den Compiler bekannten Endungen übersetzt werden. Soll ein Verzeichnis nicht rekursiv auf Unterverzeichnisse durchsucht werden, kann ein Doppelpunkt **`:`** am Ende des Namens angegeben werden.

Alle direkt angegebenen Namen (Dateien oder Verzeichnisse) müssen existieren, ansonsten wird der Übersetzungsvorgang abgebrochen. Bei Angabe von Verzeichnissen werden die darin enthaltenen Dateien mit für den Compiler unbekannten Endungen ohne Fehler ignoriert. Eine Liste der im Compiler registrierten Dateiendungen wird durch Aufruf des Compilers ohne Parameter unter der Rubrik **`Possible input types`** ausgegeben.

Darüber hinaus werden Dateien, die über **`MAGIC.getNamedString(.)`** inklusive ihrer Dateiendung und eines optionalen Pfades angegeben sind, als konstante Strings importiert. Diese auf diese Art referenzierten Dateien können jegliche Dateiendung tragen und an einer beliebigen zum Zeitpunkt des Übersetzens erreichbaren Stelle im Dateisystem oder im Namensdienst abgelegt sein.

5 Compiler-Funktionen

5.1 Compiler-Optionen

Alle zu übersetzenden Dateien und Verzeichnisse können über eine Compile-List-Datei (siehe Kapitel 4.2) oder mit identischer Syntax direkt auf der Kommandozeile angegeben werden.

In der folgenden Tabelle sind die derzeit verfügbaren Optionen aufgelistet, die auch in der Direkthilfe des Compilers (Aufruf ohne Parameter) angezeigt werden.

Option	Beschreibung
`-i FILE`	"get options from FILE" Einlesen von Compiler-Optionen aus einer Datei.
`-V`	"verbose timing information" Ausgabe von hexadezimal codierten Zeitinformationen zum Compile-Vorgang, je nach Compiler-Variante in Nanosekunden oder CPU-Ticks.
`-v`	"verbose mode" Ausgabe von Hinweisen und Warnungen, zum Beispiel über nicht übersetzte Methoden, automatisch gesetzte inline-Flags etc.
`-L`	"flat runtime environment" Zusätzlich zur implizit gesetzten Option `-l` werden Objekte der Laufzeitumgebung (zum Beispiel Methodenblöcke) ohne Objekt-Informationen erzeugt, das erzeugte Speicherabbild enthält dann also nicht mehr nur Objekte.
`-l`	"streamline objects" Entfernen der Variablen `_r_relocEntries`, `_r_scalarSize` und `_r_next`. Nicht kombinierbar mit `-m` oder `-M`.
`-w`	"use alternate Object and newInstance" Alternativen Aufbau von `java.lang.Object` und das dazu passende, verkürzte `rte.DynamicRuntime.newInstance` verwenden.
`-m`	"generate movable code and descriptors" Statt möglichst statischer Adressierung von Klassendeskriptoren, Code und Variablen werden alle Zugriffe ausschließlich über die im aktuellen Kontext hinterlegten Referenzen abgewickelt, so dass alle Objekte ohne Anpassung von Code verschoben werden können. Nicht kombinierbar mit `-l` oder `-L`.
`-M`	"generate movable scalars" Zusätzlich zu der implizit gesetzten Option `-m` werden Skalare indirekt adressiert, um eine Verschiebung des Skalaren Bereichs zu ermöglichen (siehe Kapitel 3.4 und 9).
`-h FILE`	"output file-listing of compiled files" Die Namen aller übersetzten Dateien in die Textdatei FILE schreiben.
`-h`	"call for heap pointer assign" Alle Zuweisungen für Referenzen im Heap werden nicht direkt, sondern über eine Laufzeitroutine vorgenommen, Referenzen im Stack werden direkt zugewiesen.
`-c`	"call for every pointer assign" Alle Zuweisungen für Referenzen werden nicht direkt, sondern über eine Laufzeitroutine vorgenommen.

Option	Beschreibung
`-C`	"skip array store checks" Keine Code-Erzeugung für Prüfung bei Zeiger-Zuweisungen in ein Array. Dadurch werden unzulässige Array-Elemente nicht mehr erkannt und die Typsicherheit kann nicht mehr gewährleistet werden.
`-B`	"skip array bound checks" Keine Code-Erzeugung für Prüfung der Array-Grenzen. Dadurch werden Fehladressierungen nicht mehr erkannt und die Typsicherheit kann nicht mehr gewährleistet werden.
`-b`	"generate runtime call on bound exception" Aufruf einer Laufzeitroutine bei einer Array-Grenz-Überschreitung anstelle einer nativen Behandlung (zum Beispiel mittels Exception). Bei Maschinen ohne Exceptions oder Interrupts ist diese Option zur Erkennung von Fehladressierungen erforderlich.
`-x`	"generate stack extreme check for methods" Aufruf einer Laufzeitroutine bei möglichem Überschreiten des Stack-Extreme-Pointers für alle Methoden (einzelne Methoden können über **`MARKER.stackExtreme()`** (siehe Kapitel 7) geprüft werden). Siehe dazu auch Kapitel 6.8.
`-n`	"generate runtime call on nullpointer use" Test auf Null-Zeigerdereferenzierung mit Aufruf einer Laufzeitroutine im Fehlerfall. Bei Maschinen ohne MMU-Unterstützung ist diese Option zur Erkennung von Null-Zeigerfehlern erforderlich.
`-N PK.U. MT`	"use method MT in unit PK.U as entry method" Statt der standardmäßig verwendeten Methode **`kernel.Kernel.main`** wird die als Parameter angegebene Methode als Startmethode verwendet.
`-g`	"generate all unit-descriptors (don't skip)" Auch nicht benötigte Unit-Deskriptoren werden erzeugt. Dies ist zum Beispiel beim Debugging oder für TextualCall (siehe Kapitel 10) notwendig.
`-G`	"generate all mthd-code-bodies (don't skip)" Auch der Code für nicht gerufene Methoden wird erzeugt. Dies ist zum Beispiel für TextualCall (siehe Kapitel 10) hilfreich, wenn die benötigten Methoden nicht mittels **`MARKER.genCode()`** (siehe Kapitel 7) markiert werden sollen.
`-d`	"create binary debug output for each method" Der Code zu jeder compilierten Methode wird in einer Binärdatei im Dateisystem abgelegt.
`-D DW FILE`	"debug writer DW with FILE (sym is default)" Als Debug-Info-Schreiber wird nicht der standardmäßige SymWriter verwendet, sondern der angegebene Schreiber DW mit der Zieldatei FILE. Mehrere Schreiber können durch mehrfache Angabe von Optionen `-D` aktiviert werden.
`-q`	"enable relation manager to watch relations" Aktiviert die Protokollierung von Klassen-, Methoden- und Variablenbeziehungen, die von einem geeigneten Debug-Info-Schreiber (Option `-D`) oder In-Image-Symbol-Generator (Option `-u`) ausgewertet werden können.
`-W`	"print all method's instructions if possible" Falls die aktuelle Architektur dies unterstützt, wird zu jeder Methode der erzeugte Code ausgegeben.
`-r`	"treat structs as objects, store at references" Zeiger auf Structs nicht wie üblich bei den Skalaren, sondern bei den Referenzen ablegen.

Option	Beschreibung
-R WAY	"insert return missing call in WAY=nat or =rte" Einfügen einer zusätzlichen Instruktion am Ende von Methoden mit Rückgabewert, die eine interne Exception wirft, um ein fehlendes „return“ erkennen zu können. Bei WAY=nat wird eine „native“ Exception geworfen, bei WAY=rte wird die Laufzeitroutine **`rte.DynamicRuntime.returnMissing()`** aufgerufen.
-F	"insert calls to runtime-profiler at each method" Bei allen Methoden wird bei Ein- und Austritt ein Aufruf an die Laufzeitumgebung durchgeführt, wenn nicht alle Methoden über **`MARKER.profile()`** (siehe Kapitel 7) markiert werden sollen.
-f	"do not generate code for throw-frames (no catch)" Entfernung der für Java-konforme Abarbeitung von try-catch-finally-Blöcken erforderlichen Codeteile (siehe Kapitel 6.6).
-Y	"prefer native float/double usage inside compiler" Auswahl des NativeReal-Moduls anstelle des EmulReal-Moduls für den Fall, dass die Architektur kein spezielles Fließkomma-Modul ausgewählt hat.
-Y	"use byte-value instead of char-value in String" 8-Bit-ASCII- statt der in üblichen 16-Bit-Unicode-Codierung in Strings verwenden.
-K MASK	"align blocks of allocated objects with mask MASK" Speicherblöcke mit unterschiedlichem Inhalt werden auf Adressen mit gelöschtem MASK-Bits aligniert.
-P FILE	"special (or none) instead of standard header" Der übliche 32-Byte-Header am Anfang des Speicherabbilds kann mit Angabe von **`-P none`** entfernt oder mit Angabe von **`-P FILE`** durch eine beliebige Binärdatei namens **`FILE`** ersetzt werden.
-p PREF	"naming prefix for all debug files to be written" Präfix für Debug-Informationen wie in Methoden generierter Code oder für Symbolinformationen, auch Pfadangaben sind erlaubt.
-I LEVL	"specify maximum level of method inlining" Maximale Rekursionstiefe für Methoden-Inlining, Standardwert ist 3.
-S AMNT	"maximum statement amount for auto-inline" Maximale Anzahl an Statements, bis zu der Methoden automatisch das Inline-Flag erhalten sollen, Standardwert ist 0.
-s SIZE **-s 0**	"specify maximum size of memory image" "do not use image, allocate in bootstrap mode" Größe des Speicherabbilds während der Übersetzung (darf nicht kleiner sein als die tatsächlich benötigte Speichermenge, zu viel allozierter Speicher wird im Ergebnis-Abbild verworfen). Spezialfall für **`SIZE==0`**: Keine Verwendung eines isolierten Speicherabbilds, sondern direkte Allozierung von Objekten in der aktuellen Umgebung (nur möglich für eine Übersetzung in einem von SJC übersetzten, laufenden System).
-e ADDR	"embedded mode, set static vars' RAM address" Verschiebung der Speicherzugriffe auf Klassenvariablen in einen anderen Speicherbereich, typischerweise zur Trennung von Code und Klassendeskriptoren im ROM einerseits und Variablen im RAM andererseits. Die Initialisierung von Variablen wird über ein im ROM abgelegtes Initialisierungsabbild unterstützt.
-E	"constant objects through RAM" Zugriff auf konstante Objekte (Strings und initialisierte Arrays) über RAM, nur in Verbindung mit Option **`-e`**.

Option	Beschreibung
-a ADDR	"specify start address of memory image" Startadresse des Speicherabbilds. Nicht in Kombination mit "-s 0".
-Z GB	"relocate image address by GB gb" Verschiebung des Speicherabbilds für 64-Bit-Architekturen.
-t ARCH	"specify target architecture" Auswahl der Zielplattform, die unterstützten Architekturen sind in der Kurzhilfe des Compilers (Aufruf ohne Parameter) unter "Possible architectures" aufgelistet.
-T APRM	"parameter for architecture" Parameter für die ausgewählte Zielplattform, die unterstützten Parameter sind in der Kurzhilfe des Compilers (Aufruf ohne Parameter) direkt nach der jeweiligen Plattform angegeben.
-o OFMT	"specify output format" Auswahl des Ausgabeformats, die unterstützten Formate sind in der Kurzhife des Compilers (Aufruf ohne Parameter) unter "Possible output formats" aufgelistet.
-O OPRM	"parameter for output" Parameter für das ausgewählte Ausgabeformat, je nach Format sind unterschiedliche Parameter möglich, diese sind in der Kurzhilfe des Compilers (Aufruf ohne Parameter) direkt nach beim jeweiligen Ausgabeformat angegeben.
-u UFMT	"specify symbol generator" Auswahl eines Generators für In-Image Symbolinformationen (siehe Kapitel 10). Die unterstützten Generatoren sind in der Kurzhilfe des Compilers (Aufruf ohne Parameter) unter "Possible symbol generators" aufgelistet.
-U UPRM	"parameter for symbol generator" Parameter für den ausgewählten Generator zur Erzeugung von In-Image Symbolinformationen, je nach Format sind unterschiedliche Parameter möglich, diese sind in der Kurzhilfe des Compilers (Aufruf ohne Parameter) angegeben.
-z CALG	"compression" Speicherabbild komprimieren, alle nachfolgende Argumente werden während der Compilierung des Dekompressors verwendet. Zusätzlich zu den üblichen Parametern ist **-A r** bzw. **-A a** möglich, welches die unter **-a** gemachte Zieladresse aligniert (bei **r** werden nach der Alignierung die unteren Bits aus dem komprimierten Abbild übernommen, bei **a** bleiben die unteren Bits nach der Alignierung unverändert).

5.2 Symbolinformationen

Die Symbolinformationen mit allen relevanten Informationen über erzeugte Objekte werden nach einer erfolgreichen Übersetzung (siehe Kapitel 3.6) in der ASCII-Textdatei syminfo.txt des Wirtssystems abgelegt, falls kein spezieller Debug-Schreiber oder der SymInfo-Schreiber über die Option **-D** ausgewählt wurde. Sie enthalten weitergehende und menschenlesbare Informationen als die potenziell erzeugten In-Image Symbolinformationen (siehe Kapitel 10). Alternative Debug-Schreiber (siehe Kapitel 3.7) enthalten ein Subset dieser Informationen in anderer Darstellung.

Der grundsätzliche Aufbau ist immer identisch, je nach Compiler-Option sind jedoch Erweiterungen vorhanden. Zu Beginn der Datei sind allgemeine Informationen zum erzeugten Speicherabbild angegeben, insbesondere die während der Übersetzung aktiven Parameter sowie eine Größenangabe der einzelnen Bestandteile. Diese sind für das Beispiel in Kapitel 2:

```
Options:

BaseAddress: 0x00100000

Image size: 656 b = 1 kb
```

```
Code size: 241 b = 1 kb in 11 methods
Strings: 1 with 11 characters using 80 b = 1 kb
ramInitAddr: 0x00000000, ramInitSize: 0, constMemorySize: 80
In-image symbol size: 0 b = 0 kb
```

Daraus sind folgende Informationen ablesbar (übersetzt wurde für eine 32 Bit Architektur):

- Es wurden keine Compiler-Optionen gewählt.
- Die Basisadresse des erzeugten Speicherabbildes beginnt bei 0x00100000.
- Die Größe des erzeugten Speicherabbildes beträgt 656 Bytes bzw. aufgerundet 1 Kilobyte.
- Die 11 erzeugten Methoden enthalten insgesamt 241 Bytes (aufgerundet 1 Kilobyte) Code. Hier ist ausschließlich der tatsächlich ausführbare Bereich erfasst, zusätzlicher Speicher für den Header von Methodenobjekten ist nicht berücksichtigt.
- Das Image enthält eine Stringkonstante, wobei für die 11 Zeichen insgesamt (inklusive aller Objektheader) 80 Bytes alloziert werden: 11 Zeichen zu je 2 Bytes benötigen 22 Bytes in einem Array, das einen Header mit 8 Einträgen (4 von SArray und 4 von Object) zu je 4 Bytes enthält, woraus sich nach dem Alignment eine Objektgröße von 22+32+2 gleich 56 Bytes ergibt; zusätzlich wird ein String-Objekt mit 6 Einträgen (2 von String und 4 von Object) mit einem Verweis auf das zugehörige Array benötigt, dieses belegt 24 Bytes im Speicher. Möglichkeiten zur Reduzierung des Overheads bieten die Option `-l` und `-y` (siehe Kapitel 5.1 und 6) sowie das Einbetten des Array (siehe Kapitel 8). Bei Kombination aller Optionen wäre für denselben String nur ein Speicherbedarf von insgesamt 20 Bytes erforderlich (11 für die Zeichen, 1 für das Alignment, 8 für den String-Header).
- Die Größe des zu initialisierenden RAM-Bereichs ist 0 (dist ist immer der Fall, wenn nicht im Embedded Mode (Option `-e`) übersetzt wird). Die konstanten Objekte (Strings und initialisierte Arrays) belegen 80 Bytes.
- Da keine In-Image Symbolinformationen erzeugt wurden (siehe Kapitel 10), belegen diese keine Platz. Die Speichermenge ist die Summe der speziell für die Symbolinformationen erzeugten Objekte sowie den in bereits allozierten Objekten benötigten Variablen.

Des weiteren wird für die Klassen `Object` und `String` aus Kapitel 2 folgender Text ausgegeben:

```
class java.lang.Object at 00100030
clssRelocTableEntries: 4, clssScalarTableSize: 8
statRelocTableEntries: 0, statScalarTableSize: 0
instRelocTableEntries: 2, instScalarTableSize: 8
- added/overwritten methods:
- added vars:
FFFFFFFC->inst-r/1: SClassDesc _r_type (skip candidate)
FFFFFFF8->inst-r/1: Object _r_next (skip candidate)
00000000->inst-s/4: int _r_relocEntries (skip candidate)
00000004->inst-s/4: int _r_scalarSize (skip candidate)
- added strings:
- added constant arrays:
- added imports:
- added interface-maps:
----------
class java.lang.String at 001000A8 extends LANGROOT
```

```
clssRelocTableEntries: 8, clssScalarTableSize: 8
statRelocTableEntries: 0, statScalarTableSize: 0
instRelocTableEntries: 3, instScalarTableSize: 12
- added/overwritten methods:
FFFFFFE8->00100284: int length() (statCall) (stmtCnt==2, 4 bytes)
FFFFFFE0->00100298: char charAt(int) (statCall) (stmtCnt==2, 27 bytes)
- added vars:
FFFFFFF4->inst-r/1: char[] value
00000008->inst-s/4: int count
- added strings:
- added constant arrays:
- added imports:
- added interface-maps:
```

Daraus sind folgende Informationen ablesbar:

- Der Klassendeskriptor für Objekte vom Typ **`Object`** liegt an Adresse 0x00100030.
- Die Klasse **`Object`** ist Wurzelobjekt (leitet sich von keiner andere Klasse ab).
- Der Klassendeskriptor für **`Object`** verfügt über 4 Klassen-Referenzen (bei einer 32 Bit Architektur also 4*4=16 Bytes) und 8 Bytes Klassen-Skalare, die in den Klassendeskriptoren jeder abgeleiteten Klasse ebenfalls vorhanden sind. Diese Elemente sind die Instanz-Variablen des Typs **`Object`** und die des Typs **`SClassDesc`**.
- Der Klassendeskriptor für **`Object`** verfügt weder über statische Referenzen noch über statische Skalare. Diese wären bei abgeleiteten Klassen nicht vorhanden.
- Instanzen von Typ **`Object`** haben 2 Referenzen (bei einer 32 Bit Architektur also 2*4=8 Bytes) und 8 Bytes Skalare.
- Die Klasse **`Object`** hat keine Methoden.
- Die Klasse **`Object`** hat vier Variablen deklariert:
  - Die Instanz-Referenz **`_r_type`** an der Zeiger-relativen Position -4.
  - Die Instanz-Referenz **`_r_next`** an der Zeiger-relativen Position -8.
  - Der Instanz-Skalar **`_r_relocEntries`** an der Zeiger-relativen Position 0.
  - Der Instanz-Skalar **`_r_scalarSize`** an der Zeiger-relativen Position +4.
- Die Klasse **`Object`** enthält keine konstanten Strings und keine initialisierten Arrays, importiert keine Klasse (wird nur bei verschiebbarem Code (Option **`-m`**) benötigt) und implementiert kein Interface.
- Der Klassendeskriptor für Objekte vom Typ **`String`** liegt an Adresse 0x001000A8.
- Die Klasse **`String`** erweitert explizit keine andere Klasse, leitet sich also vom Wurzelobjekt **`java.lang.Object`** ab (der Übersicht wegen als **`LANGROOT`** bezeichnet).
- Der Klassendeskriptor für **`String`** verfügt über 8 Klassen-Referenzen (bei einer 32 Bit Architektur also 8*4=32 Bytes) und 8 Bytes Klassen-Skalare, die in den Klassendeskriptoren jeder abgeleiteten Klasse ebenfalls vorhanden sind. Diese Elemente sind wie bei allen Klassendeskriptoren die Instanz-Variablen des Typs **`Object`** und die des Typs **`SClassDesc`**, zusätzlich jedoch noch 2 Methoden-Zeiger, die jeweils 2 Referenzen belegen.

- Der Klassendeskriptor für **`String`** verfügt weder über statische Referenzen noch über statische Skalare. Diese wären bei abgeleiteten Klassen nicht vorhanden.
- Instanzen vom Typ **`String`** haben 3 Referenzenund 12 Bytes Skalare. Diese setzen sich zusammen aus den 2 Referenzen und 8 Bytes Skalare des Typs **`Object`** sowie den neu hinzugekommenen Variablen: eine Referenz für das **`value`**-Array und 4 Bytes für **`count`**.
- Die Klasse **`String`** bietet zwei neue Methoden:
  - Die Methode **`length()`** an der Zeiger-relativen Position -24; das zu dieser Methode gehörende Code-Objekt liegt bei 0x00100284 und enthält 4 Bytes Code.
  - Die Methode **`charAt(int)`** an der Zeiger-relativen Position -32; das zu dieser Methode gehörende Code-Objekt liegt bei 0x00100298 und enthält 27 Bytes Code.
- Die Klasse **`Object`** enthält keine konstanten Strings und keine initialisierten Arrays, importiert keine Klasse (wird nur bei verschiebbarem Code (Option **`-m`**) benötigt) und implementiert kein Interface.

Bei Zeiger-relativen Positionen gilt zu beachten, ob die effektive Adresse über die Instanz (Instanzvariablen), die aktuelle Klasse (Klassenvariablen) oder die deklarierende Klasse (statische Klassenvariablen) berechnet wird.

Die Zeiger auf Code-Objekte für Methoden sind dieselben, die auch zur Laufzeit über die **`MAGIC`**-Funktionen **`rMem32(.)`**, **`cast2Ref(.)`** und **`mthdOff(.)`** (siehe Kapitel 12) ausgelesen werden können. Wichtig bei der Berechnung der Einsprungadresse ist, den Bereich der Skalare der Klasse **`SMthdBlock`** zu berücksichtigen. Er kann über **`MAGIC.getCodeOff()`** direkt ermittelt werden.

6 Laufzeitumgebung

6.1 Wurzelobjekt

```
package java.lang;
import rte.SClassDesc;
public class Object {
  //benötigte Einträge müssen in der hier angegebenen Reihenfolge vorliegen
  //obligatorisch
  public SClassDesc _r_type;
  //folgende Einträge entfallen, falls die Compiler-Option "-l" aktiv ist
  public Object _r_next;
  public int _r_relocEntries, _r_scalarSize;
}
```

Jedes Objekt im Heap benötigt zur Sicherstellung der Typsicherheit bei Casts sowie zur Bereitstellung des **instanceof**-Operators einen eindeutigen Typ. Dies wird durch einen Zeiger **_r_type** implementiert, der auf den jeweiligen Klassendeskriptor zeigt. Ohne die Compiler-Option **-l** ist in jedem Objekt darüber hinaus vermerkt, an welcher Adresse sich das nächste Objekt befindet (**_r_next**) und welche Ausmaße das Objekt hat. Letztere lassen sich aus der Zahl der Referenzen (**_r_relocEntries**, die Größe des dafür belegten Speicherbereichs ist also **_r_relocEntries*MAGIC.ptrSize**) und der Skalarbereichsgröße (**_r_scalarSize**) berechnen.

6.2 Strings

```
package java.lang;
public class String {
  //obligatorisch, Java-konforme Grundeinstellung, alternativ
  //  mit Compiler-Option "-y" als Byte-Array
  private char[] value;
  //optional, bei Vorhandensein vom Compiler für konstante String ausgefüllt
  private int count;
}
```

Die Klasse String setzt die beiden Variablen **value** und **count** voraus, die beide vom Compiler für konstante Strings automatisch gesetzt werden. Ersteres zeigt auf ein Array mit den enthaltenen Zeichen, die üblicherweise Java-konform 16 Bit breit (Unicode) sind, aber mit der Compiler-Option **-y** auf 8 Bit (ASCII) umgestellt werden können. Die Typendeklaration der Variablen **value** (**char[]** oder **byte[]**) muss vom Programmierer vorgenommen werden, je nach Übersetzungsmodell ist **count** (**int**) optional (Größenverhältnisse siehe Kapitel 5.2).

6.3 Klassen für Typsystem und Code

```
package rte;
public class SArray { //Basistyp für Arrays
  public int length, _r_dim, _r_stdType; //Länge, Dimension, ggf. Basistyp
  public SClassDesc _r_clssType; //ggf. Objekttyp
}
```

```
package rte;
public class SClassDesc { //Basistyp für Klassendeskriptoren
  public SClassDesc parent; //Elter-Klasse
  public SIntfMap implementations; //Liste der implementierten Interfaces
}
package rte;
public class SIntfDesc { //Marker für Interfaces
}
package rte;
public class SIntfMap { //Methoden-Mapping für eine Klasse auf ein Interface
  public SIntfDesc owner; //zugehöriges Interface
  public SIntfMap next; //nächstes Mapping für die aktuelle Klasse
}
package rte;
public class SMthdBlock { //Basistyp für Codeblöcke
}
```

Die Klassen **SArray**, **SClassDesc**, **SIntfDesc**, **SIntfMap** und **SMthdBlock** werden vom Compiler zum Aufbau des initialen Heaps verwendet.

6.4 Definition der Laufzeitumgebung für Allozierung und Typprüfung

```
package rte;
public class DynamicRuntime {
```

Obligatorisch, wird bei Erzeugung eines neuen Objektes gerufen:

```
  public static Object newInstance(int scalarSize, int relocEntries,
      SClassDesc type) { ... }
```

Obligatorisch, zur Erzeugung eines neuen eindimensionalen Arrays:

```
  public static SArray newArray(int length, int arrDim, int entrySize,
      int stdType, SClassDesc clssType) { ... }
```

Obligatorisch, zur automatischen Erzeugung mehrdimensionaler Arrays:

```
  public static void newMultArray(SArray[] parent, int curLevel,
      int destLevel, int length, int arrDim, int entrySize, int stdType,
      SClassDesc clssType) { ... }
```

Obligatorisch, zur Cast-Prüfung (**asCast==true**) und bei **instanceof**-Test (**asCast==false**), Getrennt für die Prüfung auf Klasse, Interface und Array, sowie Zeiger-Ablage-Prüfung:

```
  public static boolean isInstance(Object o, SClassDesc dest,
      boolean asCast) { ... }
  public static SIntfMap isImplementation(Object o, SIntfDesc dest,
      boolean asCast) { ... }
  public static boolean isArray(Object o, int stdType,
      SClassDesc clssType, int arrDim, boolean asCast) { ... }
  public static void checkArrayStore(SArray dest, Object newEntry) { ... }
}
```

Optional kann die Klasse DynamicRuntime auch noch die folgenden vier Methoden enthalten, die in Verbindung mit speziellen Compiler-Optionen geprüft und verwendet werden.

Zeigerzuweisung über Laufzeitsystem (Compiler-Option **-c** oder **-h**), beispielsweise für Reference-Tracking oder Reference-Counting; die letzten drei Parameter müssen für Architekturen mit anderer Zeigergröße angepaßt werden:

```
public static void assign(boolean intf, int addr, int map, int obj) { ... }
```

Laufzeit-Routine bei Array-Fehlindizierungen (Compiler-Option **-b**):

```
public static void boundException(SArray arr, int ind) { ... }
```

Laufzeit-Routine bei unzulässiger Dereferenzierung eines Null-Zeigers (Compiler-Option **-n**):

```
public static void nullException() { ... }
```

Laufzeit-Routine bei Verwendung von Profiling (**MARKER.profile()** oder Compiler-Option **-F**):

```
public static void profile(int mthdID, byte enterLeave) { ... }
```

Alle Laufzeit-Routinen werden nur auf ihr Vorhandensein geprüft. Die Zahl der Parameter sowie deren Typ liegen vollständig in der Verantwortung des Programmierers.

6.5 Beispielimplementierung obligatorischer Laufzeitroutinen für 32 Bit

```
public class DynamicRuntime {
  //folgende Variable vor der ersten Objektallozierung initialisieren, z.B.
  //  mit (MAGIC.imageBase+MAGIC.rMem32(MAGIC.imageBase+4)+0xFFF)&~0xFFF
  private static int nextFreeAddress;
  public static Object newInstance(int scS, int rlE, SClassDesc type) {
    int start, rs, i; //temporäre Variablen
    Object me; //zukünftiges Objekt
    rs=rlE<<2; //pro Reloc werden 4 Bytes benötigt
    scS=(scS+3)&~3; //Alignierung der Skalare
    start=nextFreeAddress; //Start des Objektes
    nextFreeAddress+=rs+scS; //nächstes Objekt hinter aktuellem platzieren
    for (i=start; i<nextFreeAddress; i+=4) MAGIC.wMem32(i, 0); //0-Init
    me=MAGIC.cast2Obj(start+rs); //Objekt platzieren
    me._r_relocEntries=rlE; //Zahl der Relocs in Objekt eintragen
    me._r_scalarSize=scS; //Größe der Skalare in Objekt eintragen
    me._r_type=type; //Typ des Objekts in Objekt eintragen
    return me; //Objekt an Aufrufer zurückgeben
  }
  public static SArray newArray(int length, int arrDim, int entrySize,
      int stdType, SClassDesc clssType) {
    int scS, rlE; //temporäre Variablen
    SArray me; //zukünftiges Array
    scS=MAGIC.getInstScalarSize("SArray"); //Skalargröße eines leeren Arrays
    rlE=MAGIC.getInstRelocEntries("SArray"); //Reloczahl eines leeren Arrays
    if (arrDim>1 || entrySize<0) rlE+=length; //Array mit Reloc-Elementen
    else scS+=length*entrySize; //Array mit skalaren Elementen
```

```
  me=(SArray)newInstance(scS, rlE, MAGIC.clssDesc("SArray")); //allozieren
  me.length=length; //Länge in Array ablegen
  me._r_dim=arrDim; //Dimension in Array ablegen
  me._r_stdType=stdType; //Skalar-Typ in Array ablegen
  me._r_clssType=clssType; //Reloc-Typ in Array ablegen
  return me; //Array an Aufrufer zurückgeben
}
public static void newMultArray(SArray[] parent,
    int curLevel, int destLevel, int length, int arrDim,
    int entrySize, int stdType, SClassDesc clssType) {
  int i; //temporäre Variable
  if (curLevel+1<destLevel) { //es folgt noch mehr als eine Dimension
    curLevel++; //aktuelle Dimension erhöhen
    for (i=0; i<parent.length; i++) //jedes Element mit Array befüllen
      newMultArray((SArray[])((Object)parent[i]), curLevel, destLevel,
          length, arrDim, entrySize, stdType, clssType);
  }
  else { //letzte anzulegende Dimension
    destLevel=arrDim-curLevel; //Zieldimension eines Elementes
    for (i=0; i<parent.length; i++) //jedes Element mit Zieltyp befüllen
      parent[i]=newArray(length, destLevel, entrySize, stdType, clssType);
  }
}
public static boolean isInstance(Object o, SClassDesc dest,
    boolean asCast) {
  SClassDesc check; //temporäre Variable
  if (o==null) { //Prüfung auf null
    if (asCast) return true; //null darf immer konvertiert werden
    return false; //null ist keine Instanz
  }
  check=o._r_type; //für weitere Vergleiche Objekttyp ermitteln
  while (check!=null) { //suche passende Klasse
    if (check==dest) return true; //passende Klasse gefunden
    check=check.parent; //Elternklasse versuchen
  }
  if (asCast) while(true); //Konvertierungsfehler
  return false; //Objekt passt nicht zu Klasse
}
public static SIntfMap isImplementation(Object o, SIntfDesc dest,
    boolean asCast) {
  SIntfMap check; //temporäre Variable
  if (o==null) return null; //null implementiert nichts
```

```
  check=o._r_type.implementations; //Liste der Interface-Maps ermitteln
  while (check!=null) { //suche passendes Interface
    if (check.owner==dest) return check; //Interface gefunden, Map liefern
    check=check.next; //nächste Interface-Map versuchen
  }
  if (asCast) while(true); //Konvertierungsfehler
  return null; //Objekt passt nicht zu Interface
}
public static boolean isArray(SArray o, int stdType, SClassDesc clssType,
    int dim, boolean asCast) { //o ist eigentlich Object, Prüfung unten!
  SClassDesc clss; //temporäre Variable
  if (o==null) { //Prüfung auf null
    if (asCast) return true; //null darf immer konvertiert werden
    return false; //null ist keine Instanz
  }
  if (o._r_type!=MAGIC.clssDesc("SArray")) { //Array-Prüfung
    if (asCast) while(true); //Konvertierungsfehler
    return false; //kein Array
  }
  if (clssType==MAGIC.clssDesc("SArray")) { //Sonderbehandlung für SArray
    if (o._r_clssType==MAGIC.clssDesc("SArray")) dim--; //Array aus SArray
    if (o._r_dim>dim) return true; //ausreichende Resttiefe
    if (asCast) while(true); //Konvertierungsfehler
    return false; //kein SArray
  }
  if (o._r_stdType!=stdType || o._r_dim<dim) { //notwendige Bedingungen
    if (asCast) while(true); //Konvertierungsfehler
    return false; //Array mit nicht passenden Elementen
  }
  if (stdType!=0) { //Array aus Basistypen
    if (o._r_dim==arrDim) return true; //passende Tiefe
    if (asCast) while(true); //Konvertierungsfehler
    return false; //Array nicht mit passenden Elementen
  }
  clss=o._r_clssType; //für weitere Vergleiche Klassen-Typ ermitteln
  while (clss!=null) { //suche passende Klasse
    if (clss==clssType) return true; //passende Klasse gefunden
    clss=clss.parent; //Elternklasse versuchen
  }
  if (asCast) while(true); //Konvertierungsfehler
  return false; //Array mit nicht passenden Elementen
}
```

```
  public static void checkArrayStore(SArray dest, SArray newEntry) {
    //newEntry ist eigentlich Object", die Prüfung muss in isArray erfolgen!
    if (dest._r_dim>1) isArray(newEntry, dest._r_stdType, dest._r_clssType,
      dest._r_dim-1, true); //Prüfung des Arrays über isArray,
                            //  falls Dimension des Zielarrays größer 1 ist
    else if (dest._r_clssType==null) while (true); //Zuweisungsfehler,
                            //  falls Zielarray aus keine Reloc-Elemente hat
    else isInstance(newEntry, dest._r_clssType, true); //Instanz-Prüfung
                            //  in allen anderen Fällen
  }
}
```

Die einzelnen Methoden müssen zwar immer vorhanden sein, die nicht verwendeten Methoden (zum Beispiel **newMultArray**, wenn keine mehrdimensionalen Arrays angelegt werden) können jedoch durchaus wie auch im Beispiel in Kapitel 2 leer sein.

6.6 Laufzeiterweiterung bei Verwendung von Java-Exceptions

Bei Verwendung von Java-Exceptions, also mindestens einem der Schlüsselworte **try**, **catch**, **finally**, **throw** oder **throws**, wird vom Compiler die Variable

```
static short/int/long currentThrowFrame;
```

sowie die Methode

```
static void doThrow(Throwable thrown) { ... }
```

in **rte.DynamicRuntime** erwartet. In **currentThrowFrame** wird die Adresse des aktuellen und vom Backend für **try-**, **catch-** und **finally**-Statements aufgebauten Throw-Frames abgelegt. Diese auf dem Stack angelegten Blöcke sind untereinander verkettet und enthalten alle für die Abarbeitung von Java-Exceptions erforderlichen Informationen. Für das **throw**-Statement sowie für durch **catch-** und **finally**-Blöcke unbehandelte Exceptions wird ein Aufruf der **doThrow**-Methode eingefügt. Dort sollten nach Überprüfung der **currentThrowFrame**-Variable (sollte diese keinen gültigen Zeiger enthalten, wurde die aktuelle Exception nirgends behandelt) alle für die aktuelle Architektur erforderlichen und zum aufgebauten Throw-Frame passenden Arbeiten durchgeführt werden und durch einen Sprung an die im Throw-Frame angegebene Adresse beendet werden.

Eine Implementierung für eine IA32-Umgebung könnte folgendermaßen aussehen:

```
private static int currentThrowFrame;
public static void doThrow(Throwable thrown) {
  int frame=currentThrowFrame;
  if (frame==0) { /* unhandled exception ! */ while(true); }
  MAGIC.inline(0x8B, 0x45);MAGIC.inlineOffset(1, frame); //eax<-frame
  MAGIC.inline(0x8B, 0x5D);MAGIC.inlineOffset(1, thrown); //ebx<-thrown
  MAGIC.inline(0x8B, 0x78, MAGIC.ptrSize*2); //mov edi,[eax+8]
  MAGIC.inline(0x8B, 0x70, MAGIC.ptrSize*3); //mov esi,[eax+12]
  MAGIC.inline(0x8B, 0x68, MAGIC.ptrSize*4); //mov ebp,[eax+16]
  MAGIC.inline(0x8B, 0x60, MAGIC.ptrSize*5); //mov esp,[eax+20]
  MAGIC.inline(0x89, 0x58, MAGIC.ptrSize*6); //mov [eax+24],ebx
  MAGIC.inline(0xFF, 0x60, MAGIC.ptrSize); //jmp [eax+4]
}
```

Bei Verwendung der AMD64-Architektur kann folgende Implementierung verwendet werden:

```
private static long currentThrowFrame;
public static void doThrow(Throwable thrown) {
  long frame=currentThrowFrame;
  if (frame==0l) { /* unhandled exception ! */ while(true); }
  MAGIC.inline(0x48, 0x8B, 0x45);MAGIC.inlineOffset(1, frame); //rax<-frame
  MAGIC.inline(0x48, 0x8B, 0x5D);MAGIC.inlineOffset(1, thrown); //rbx<-thrown
  MAGIC.inline(0x48, 0x8B, 0x78, MAGIC.ptrSize*2); //mov rdi,[eax+16]
  MAGIC.inline(0x48, 0x8B, 0x70, MAGIC.ptrSize*3); //mov rsi,[eax+24]
  MAGIC.inline(0x48, 0x8B, 0x68, MAGIC.ptrSize*4); //mov rbp,[eax+32]
  MAGIC.inline(0x48, 0x8B, 0x60, MAGIC.ptrSize*5); //mov rsp,[eax+40]
  MAGIC.inline(0x48, 0x89, 0x58, MAGIC.ptrSize*6); //mov [rax+48],rbx
  MAGIC.inline(0xFF, 0x60, MAGIC.ptrSize); //jmp [rax+8]
}
```

Je nach Übersetzungmodus (siehe Kapitel 5.1) sind unter Umständen Optimierungen möglich, bei anderen Zielarchitekturen (ATmega, SSA) sind vollständig andere Implementierungen erforderlich. Hinweise zum Aufbau des Throw-Frames finden sich im Compiler-Quelltext im Backend der jeweiligen Architektur.

6.7 Laufzeiterweiterung bei Verwendung von synchronized-Blöcken

Bei Verwendung des Schlüsselworts **`synchronized`** wird vom Compiler die Methode

```
static void doSynchronize(Object o, boolean leave) { ... }
```

in **`rte.DynamicRuntime`** erwartet. Dabei enthält **`o`** das Objekt, auf das synchronisiert werden soll, und **`leave`** eine Unterscheidung zwischen Ein- und Austritt des synchronisierten Blocks (**`false`** beim Eintritt, **`true`** beim Austritt).

Die Implementierung ist stark von der weiteren Laufzeitumgebung und dem Betriebssystem abhängig, weshalb keine allgemeingültige Implementierung existiert. Für eine Implementierung sollten aber zumindest die folgenden Punkte beachtet werden:

- Jedes Objekt kann als Synchronisierungsobjekt dienen, es wird also ein Ein-/Austrittszähler in **`java.lang.Object`** vorgesehen werden müssen.
- Vom Betriebssystem muss eine eindeutige Thread-ID zum aktuell ausgeführten Thread geliefert werden können, die als Markierung im Synchronisierungsmarke in **`java.lang.Object`** abgelegt werden sollte.
- Beim Eintritt in einen synchronized-Block, der bereits mit einer fremden Thread-ID markiert wurde, sollte der aktuelle Thread deaktiviert und der markierte Thread aktiviert werden können.
- Der Code der Methode **`doSynchronize`** sollte vor Unterbrechung geschützt werden.

In Umgebungen ohne Threads kann die Methode **`doSynchronize`** auch leer gelassen werden (zum Beispiel, wenn derselbe Quellcode für mehrere Betriebssysteme entwickelt wird und in der SJC-Variante keine Nebenläufigkeit zu erwarten ist) oder zum Debuggen verwendet werden (zum Beispiel zur Anzeige der Ein- und Austritte aus einem zu beobachtenden Block, wobei die im Synchronisationsobjekt vorhandenen Daten kontrolliert werden können).

6.8 Laufzeiterweiterung bei Stack-Extreme-Überprüfung

Bei Markierung einer Methode mit **`MARKER.stackExtreme()`** oder der Compiler-Option **`-x`** wird vom Compiler die Variable

```
static short/int/long stackExtreme;
```

sowie die Methode

```
static void stackExtremeError() { ... }
```

in **`rte.DynamicRuntime`** erwartet. Die Methode **`stackExtremeError()`** wird genau dann gerufen, wenn eine Methode während ihres Ablaufs voraussichtlich einen Stack-Zeiger erreichen wird, der unterhalb des Wertes von **`stackExtreme`** liegt.

6.9 Definition der optionalen arithmetischen Bibliothek

```
package rte;
public class DynamicAri {
  public long binLong(long a, long b, char op) {
    long res=0l;
    switch (op) {
      case '/': /* fill code here */ break; //Division
      case '%': /* fill code here */ break; //Modulo
      default: while(true); //unbekannte Operation
    }
    return res;
  }
}
```

Je nach Architektur kann eine optionale arithmetische Bibliothek verwendet werden, um nicht direkt oder nur mit unverhältnismäßigem Aufwand implementierbare Operationen zentral bereitzustellen. Die Ausführung ist somit zwar durch einen zusätzlichen Methodenaufruf langsamer, der Code ist jedoch pro Operator und nicht einmal pro Operation vorhanden.

Für das IA32-Backend kann die Verwendung von **`binLong`** durch den Parameter **`-T rtlc`** aktiviert werden, für das ATmega-Backend werden bei Verwendung der entsprechenden Operatoren grundsätzlich die entsprechenden Routinen für **`binShort`**, **`binInt`** und **`binLong`** verwendet.

7 Spezialklassen

7.1 MAGIC

In der folgenden Liste sind die derzeit verfügbaren Spezialvariablen und Spezialmethoden der Compiler-internen Klasse **MAGIC** angegeben, Beispiele finden sich in Kapitel 12. Je nach Compiler-Modus (siehe Compiler-Optionen in Kapitel 5.1) sind einzelne Werte oder Methoden nicht verwendbar.

- **static int ptrSize;**

  Zeigergröße der aktuellen Architektur (zum Beispiel 4 für IA32, 8 für AMD64).

- **static boolean movable, indirScalars;**

  Flag, welches die Verschiebbarkeit des Codes und der Skalare anzeigt (Code wurde mit der Compiler-Option **-m** bzw. **-M** übersetzt).

- **static boolean streamline;**

  Flag, welches einen schlanken Objektaufbau anzeigt (Code wurde mit der Compiler-Option **-l** übersetzt).

- **static boolean assignCall, assignHeapCall;**

  Flag, welches einen Laufzeitaufruf bei Referenzenzuweisung für alle Referenzen bzw. für Referenzen im Heap anzeigt (Code wurde mit der Compiler-Option **-c** bzw. **-h** übersetzt).

- **static boolean runtimeBoundException;**

  Flag, welches einen Laufzeitaufruf im Falle einer Arraygrenzüberschreitung anzeigt (Code wurde mit der Compiler-Option **-b** übersetzt).

- **static boolean runtimeNullException;**

  Flag, welches einen Laufzeittest und Laufzeitaufruf im Falle einer ungültigen Nullzeigerverwendung anzeigt (Code wurde mit der Compiler-Option **-n** übersetzt).

- **static int imageBase;**

  Startadresse des aktuellen Speicherabbilds (Parameter der Compiler-Option **-a**).

- **static int compressedImageBase;**

  Zieladresse des zu dekomprimierenden Speicherabbilds im Dekompressor (Parameter der Compiler-Option **-a** für das komprimierte Abbild).

- **static boolean embedded;**

  static boolean embConstRAM;

  Flag, welches aus den Klassendeskriptoren ausgelagerte Klassenvariablen (Code wurde mit der Compiler-Option **-e** übersetzt) bzw. auch ausgelagerte Objekte (Code wurde mit der Copmiler-Option **-E** übersetzt) anzeigt, siehe Kapitel 12.5.

- **static int relocation;**

  static int comprRelocation;

  Eingestellte Relozierung des aktuellen Speicherabbilds bzw. im Dekompressor des zu dekomprimierenden Speicherabbilds (Compiler-Option **-Z XXX**).

- **static void inline(.);**

  static void inline16(.);

  static void inline32(.);

Pseudo-Methoden zur direkten Angabe von Maschinenbefehlen, variable Anzahl von Parametern ist möglich. Die Parameter müssen konstant sein, pro Parameter werden nur die unteren 8 Bit bzw. die unteren 16 Bit bzw. die gesamten 32 Bit gewertet. Seit Version 160 steht die am Ende dieses Unterkapitels beschriebene Funktion **`inlineOffset`** zur Verfügung. Die folgenden je zwei Statements haben für Little-Endian-Architekturen identischen Inline-Code zur Folge:

```
MAGIC.inline(0x12, 0x34); <==> MAGIC.inline16(0x3412);
```

- **`static void inlineBlock(String name);`**

  Pseudo-Methoden zum Einfügen eines Blocks von Maschinenbefehlen mit dem im Parameter angegebenen Namen. Dieser muss als konstanter String vorliegen und darf keine Dereferenzierung enthalten. Der Datenblock entstammt typischerweise aus einer binär importierten Datei mit der Endung .bim und trägt den Namen der Datei ohne Pfad und Dateiendung.

- **`static void inlineOffset(int inlineMode, VAR);`**

  `static void inlineOffset(int inlineMode, VAR, int baseValue);`

  Pseudo-Methoden zum Einfügen von Variablenoffsets als Maschinencode. Diese Methoden werden typischerweise verwendet, um in Inline-Code auf Variablen zugreifen zu können.

- **`static void wMem64(short/int/long addr, long value);`**

  `static void wMem32(short/int/long addr, int value);`

  `static void wMem16(short/int/long addr, short value);`

  `static void wMem8(short/int/long addr, byte value);`

  Pseudo-Methoden zum direkten Beschreiben des Hauptspeichers. Die Angabe der Adresse muß in dem zur Architektur passenden Format oder als **`int`** vorliegen.

- **`static long rMem64(short/int/long addr);`**

  `static int rMem32(short/int/long addr);`

  `static short rMem16(short/int/long addr);`

  `static byte rMem8(short/int/long addr);`

  Pseudo-Methoden zum direkten Auslesen des Hauptspeichers. Die Angabe der Adresse muß in dem zur Architektur passenden Format oder als in vorliegen.

- **`static void wIOs64(short/int/long addr, long value);`**

  `static void wIOs32(short/int/long addr, int value);`

  `static void wIOs16(short/int/long addr, short value);`

  `static void wIOs8(short/int/long addr, byte value);`

  Pseudo-Methoden zum direkten Beschreiben des I/O-Speichers. Die Angabe der Adresse muß in dem zur Architektur passenden Format oder als **`int`** vorliegen. Unter Umständen werden von manchen Architekturen nicht alle Bitbreiten unterstützt.

- **`static long rIOs64(short/int/long addr);`**

  `static int rIOs32(short/int/long addr);`

  `static short rIOs16(short/int/long addr);`

  `static byte rIOs8(short/int/long addr);`

  Pseudo-Methoden zum direkten Auslesen des I/O-Speichers. Die Angabe der Adresse muß in dem zur Architektur passenden Format oder als **`int`** vorliegen. Unter Umständen werden von manchen Architekturen nicht alle Bitbreiten unterstützt.

- **`static short/int/long cast2Ref(Object o);`**

Pseudo-Methode zur Ermittlung der Adresse des referenzierten Objekts. Nicht zu verwechseln mit der Methode **`addr(.)`**, die zur Ermittlung der Adresse der Variablen **`o`** statt des dadurch referenzierten Objekts dient. Beispiele siehe auch unter **`MAGIC.addr()`**.

- **`static Object cast2Obj(short/int/long addr);`**

  Pseudo-Methode zur Konvertierung einer Adresse in eine Objektreferenz. Die Adresse muss auf das erste Skalare Feld des Objektes zeigen (siehe Kapitel 3.4 und **`MAGIC.addr()`**).

- **`static short/int/long addr(VAR);`**

  Pseudo-Methode zur Ermittlung der Adresse der im Parameter angegebenen Variablen. Nicht zu verwechseln mit **`cast2Ref(.)`**, welches die Zieladresse des übergebenen Objektes liefern würde. In einem 32-Bit System gelten mit der Objektreferenz o die Identitäten:

  `MAGIC.rMem32(MAGIC.addr(o)) <==> MAGIC.cast2Ref(o)`

  `o <==> MAGIC.cast2Obj(MAGIC.cast2Ref(o))`

- **`static short/int/long clssDesc(String clss);`**

  Pseudo-Methode zur Ermittlung der Adresse des im Parameter angegebenen Klassendeskriptors. Der Klassenname muß zur Einhaltung der Java-Kompatibilität in Form eines konstanten Strings vorliegen und darf keine Dereferenzierung enthalten.

- **`static int mthdOff(String clss, String mthd);`**

  Pseudo-Methode zur Ermittlung des Methodenoffsets der im zweiten Parameter angegebenen Methode innerhalb der im ersten Parameter angegebenen Klasse. Beide Parameter müssen zur Einhaltung der Java-Kompatibilität in Form von konstanten Strings vorliegen und dürfen keine Dereferenzierung enthalten.

- **`static int getCodeOff();`**

  Pseudo-Methode zur Offset-Ermittlung des Codes innerhalb eines Methodenobjektes. Der erste ausführbare Opcode befindet sich diese Anzahl an Bytes nach dem Ziel des Methodenzeigers. Der Codestart einer Methode Cls.Mtd läßt sich wiefolgt bestimmen:

  `rMem32(cast2Ref(clssDesc("Cls"))+MAGIC.mthdOff("Cls", "Mtd"))+getCodeOff()`

- **`static int getInstScalarSize(String clss) / getInstIndirScalarSize(.);`**

  Pseudo-Methode zur Ermittlung der Größe des Skalarbereiches einer Instanz der im Parameter angegebenen Klasse. Der Bezeichner muß zur Einhaltung der Java-Kompatibilität in Form eines konstanten Strings vorliegen und darf keine Dereferenzierung enthalten.

- **`static int getInstRelocEntries(String clss);`**

  Pseudo-Methode zur Ermittlung der Anzahl der Referenzen einer Instanz der im Parameter angegebenen Klasse. Die Größe des Referenzbereichs ergibt sich durch Multiplikation mit der Zeigergröße. Der Bezeichner muß zur Einhaltung der Java-Kompatibilität in Form eines konstanten Strings vorliegen und darf keine Dereferenzierung enthalten.

- **`static void bitMem64(short/int/long addr, long mask, boolean set);`**

  `static void bitMem32(short/int/long addr, int mask, boolean set);`

  `static void bitMem16(short/int/long addr, short mask, boolean set);`

  `static void bitMem8(short/int/long addr, byte mask, boolean set);`

  Pseudo-Methode zum Setzen oder Löschen (Auswahl durch den Parameter **`set`**) der durch **`mask`** angegebenen Bitmaske im Hauptspeicher. Die Angabe der Adresse muß in dem zur Architektur passenden Format oder als **`int`** vorliegen.

- **`static void bitIOs64(short/int/long addr, long mask, boolean set);`**

  `static void bitIOs32(short/int/long addr, int mask, boolean set);`

  `static void bitIOs16(short/int/long addr, short mask, boolean set);`

```
static void bitIOs8(short/int/long addr, byte mask, boolean set);
```

Pseudo-Methode zum Setzen oder Löschen (Auswahl durch den Parameter **set**) der durch **mask** angegebenen Bitmaske im I/O-Speicher. Die Angabe der Adresse muß in dem zur Architektur passenden Format oder als **int** vorliegen. Unter Umständen werden von manchen Architekturen nicht alle Bitbreiten unterstützt.

- **static STRUCT cast2Struct(short/int/long addr);**

  Pseudo-Methode zur Konvertierung einer Adresse zu einer Struct-Referenz. Die Angabe der Adresse muss in dem zur Architektur passenden Format oder als **int** vorliegen (siehe auch **cast2Obj(.)**). Eine vorgelagerte Konvertierung auf einen anderen Struct-Typ wird ohne Prüfung, also ohne Aufruf der Laufzeitumgebung durchgeführt.

- **static int getRamAddr();**

  static int getRamInitAddr();

  Pseudo-Methoden zur Ermittlung der RAM-Adresse für Klassenvariablen im Embedded-Mode (Parameter der Compiler-Option **-e**) sowie des Initialisierungsspeichers, siehe Kapitel 12.5.

- **static int getRamSize();**

  Pseudo-Methode zur Ermittlung der Größe des RAM-Bereichs für Klassenvariablen im Embedded-Mode, siehe Kapitel 12.5.

- **static void useAsThis(Object o);**

  Pseudo-Methode zur expliziten Zuweisung eines Objektes im Konstruktor als aktuelle Instanz. Hauptanwendung sind Konstrkutoren von Klassen mit inline-Arrays, die in Kapitel 8 genauer besprochen werden.

- **static void getConstMemorySize();**

  Pseudo-Methode zur Ermittlung des für konstante Objekte verbrauchten Speicherplatzes. Dies ist beispielsweise im Embedded-Mode interessant, da die konstanten Objekte direkt nach dem RAM-Init-Bereich abgelegt werden. Somit können sie für Tests mit Harvard-Architekturen leicht ins RAM kopiert werden, um „dynamische“ Objekte zu testen.

- **static byte[] toByteArray(String what, boolean trailingZero);**

  static char[] toCharArray(String what, boolean trailingZero);

  Die beiden Pseudo-Methoden liefern zu konstanten Strings entsprechend initialisierte konstante Arrays, die aus den Zeichen des übergebenen Strings sowie optional einem abschließenden 0-Zeichen bestehen. Auf diese Weise können sonst umständlich von Hand initialisierte Arrays elegant angelegt werden, ohne zur Laufzeit Konvertierungsaufwand bei der Umwandlung zu einem 0-terminierten Array zu erfordern.

- **static String getNamedString(stringName);**

  Pseudo-Methode, die durch den konstanten String ersetzt wird, der durch **stringName** benenant ist. Sollte das Objekt **stringName** nicht vom Übersetzungssystem gefunden werden, ist das Resultat **null**. Inhaltlich ist diese Funktion vergleichbar mit dem Import von Binärdaten als Byte-Array (siehe Kapitel 4.1), jedoch kann diese Funktion auch verwendet werden, wenn unsicher ist, ob das Zielobjekt existiert (das Ergebnis ist dann **null**, die Übersetzung ist dennoch möglich; bei Referenzierung eines Objekts wie in Kapitel 4.1 muss das referenzierte Objekt vorhanden sein). Für weitere Informationen zum Import von Dateien siehe auch Kapitel 4.2.

- **static void doStaticInit();**

  Pseudo-Methode zum Einbau der in statischen Klassen-Initialisierungen vorgenommenen Statements. Die Zielarchitektur muß dafür Methoden-Inlining unterstützen, da die Statements der Zielmethoden anstelle des **doStaticInit()**-Aufrufs eingefügt werden.

7.2 MARKER

In der folgenden Liste sind die derzeit verfügbaren Spezialvariablen und Spezialmethoden der Compiler-internen Klasse **MARKER** für den allgemeinen Einsatz (außer **DEFINE**-Blöcke) angegeben. Für **MARKER**-Anweisungen werden grundsätzlich keine Maschinenbefehle erzeugt, sie dienen ausschließlich als Ersatz für Java-inkompatible Methoden-Attribute, zur Herstellung von Java-Kompatibilität und zur Modifizierung des Speicherabbilds. Beispiele finden sich in Kapitel 12.

- **static void interrupt();**

  Methode wird als direkt von der Hardware gerufener Interrupt-Handler markiert. Unterstützende Architekturen erzeugen üblicherweise einen speziellen Pro- und Epilog.

- **static void debug();**

  Der erzeugte Code der markierten Methode wird in einer Binärdatei im Dateisystem abgelegt (siehe auch Kapitel 5.1: Compiler-Option **-d**).

- **static void ignore(.);**

  Alle als Parameter übergebenen Parameter werden ignoriert (dennoch müssen die angegebenen Parameter syntaktisch korrekt sein). Werden beispielsweise Variablen ausschließlich in Inline-Code gelesen, zeigt SunJava eine Warnung an, was durch den für SunJava als Auslesen interpretierten **MARKER.ignore(.)**-Befehl verhindert werden kann.

- **static void inline();**

  static void noInline();

  Code der Methode nach Möglichkeit inline in aufrufende Methoden einbauen bzw. von der inline-Automatik (siehe auch Kapitel 5.1: Compiler-Option **-S**) ausschließen.

- **static void enterCodeAddr(int addr, String clss, String mthd);**

  Start des Codes der in Parameter zwei und drei angegebenen Methode im Speicherabbild an Adresse **addr** eintragen. Die beiden letzten Parameter müssen zur Einhaltung der Java-Kompatibilität in Form von konstanten Strings vorliegen und dürfen keine Dereferenzierung enthalten.

- **static void genCode();**

  Code der Methode in jedem Fall erzeugen (siehe auch Kapitel 5.1: Compiler-Option **-G**).

- **static void profile();**

  static void noProfile();

  Code der Methode mit bzw. ohne Aufrufe an die Profiler-Methoden versehen (siehe auch Kapitel 5.1: Compiler-Option **-F**). Für die Profiler-Methoden wird generell kein Profiler-Aufruf erzeugt. Bei gleichzeitigem Markieren einer Methode sowohl mit **profile** und **noProfile** werden keine Aufrufe eingefügt.

- **static void stackExtreme();**

  static void noStackExtrme();

  Code der Methode mit bzw. ohne Überprüfung des extremen Stack-Zeigers generieren (siehe Kapitel 6.8: Laufzeiterweiterung und Kapitel 5.1: Compiler-Option **-x**). Für die Fehler-Methode wird generell kein Prüf-Aufruf erzeugt. Bei gleichzeitigem Markieren einer Methode sowohl mit **stackExtreme** und **noStackExtreme** werden keine Überprüfungen eingefügt.

- **static void stopBlockCoding();**

  Die Code-Erzeugung für den aktuellen Block wird gestoppt. Dies kann zum Beispiel verwendet werden, wenn ein Inline-Block bereits einen Rückgabewert im für die aktuelle Architektur passenden Primär-Register erzeugt, zur Erfüllung der Java-Quellcode-Kompatibilität jedoch ein **return**-Statement erforderlich wäre.

- **`static void printCode();`**

  Falls möglich, wird der Code der Methode ausgegeben.

- **`static void noOptimization();`**

  Falls möglich, wird die Code-Optimierung verhindert.

- **`static void treatAsUnwritten(VAR);`**

  Die Variable **`VAR`** wird einmalig unbeschrieben behandelt. Dieser Markierung sollte ein schreibender Zugriff auf die Variable folgen, um wieder einen definierten Zustand zu erreichen. Verwendung findet diese Markierung, um z.B. auf fremde final-Variablen schreibend zuzugreifen (benötigt wird dies unter anderem bei dem final-deklariertem length-Feld eines vom Laufzeitsystem soeben angelegten Arrays).

7.3 STRUCT

Mit Klassen, die sich von **`STRUCT`** ableiten, können Speicherstrukturen unabhängig von Heap- und Compilerstrukturen typsicher abgebildet werden. Zum Beispiel können die Geräteregister bei Geräten mit Memory-Mapped-I/O relativ zu einer Startadresse spezifiziert und benannt werden, so dass die Offsets durch den Compiler berechnet und der Zugriff durch den Compiler geprüft werden kann. Die Lesbarkeit und Wartbarkeit des Gerätetreibers wird bei sinnvoller Verwendung ebenfalls erhöht, gleichzeitig ist die Syntax vollständig Java-konform auf Instanzvariablen abgebildet, so dass auch bei **`STRUCT`**-Verwendung alle Java-Tools verwendet werden können. Aufgrund der fehlenden Typinformationen muss jedoch beachtet werden, dass Referenzen auf **`STRUCT`**-Bereiche nicht über den **`new`**-Operator angelegt oder typsicher ineinander überführt werden können.

In jedem **`STRUCT`** kann eine **`DEFINE`**-Methode vorhanden sein, in der die Offsets der deklarierten Variablen mit der Pseudo-Methode **`MARKER.offset(varName, offset)`** und bei Arrays zusätzlich die zu überprüfende Länge mit **`MARKER.count(varName, elements)`** festgelegt werden können. Soll die Array-Bereichsüberprüfung abgeschaltet und die Warnung bei der Compilierung unterdrückt werden, kann für die Anzahl der Elemente der Wert 0 eingetragen werden. In Kapitel 12.4 ist ein Beispiel zur Abbildung des Bildschirmspeichers im Textmodus angegeben.

8 Inline Arrays

Für nicht weiter abgeleitete Klassen besteht die Möglichkeit, die Daten eines Arrays – zugegriffen über eine Instanzvariable – direkt innerhalb der Instanz abzulegen. Dadurch ergeben sich zum Teil deutliche Speichervorteile, allerdings sind die innerhalb einer Instanz allozierten Arrays nicht mit üblichen Arrays kompatibel. Da für diese Arrays auch keine weiterführende Referenz besteht, können sie nicht durch Neuallozierung ausgetauscht werden. Es ist also ein Kompromiss zwischen geringem Speicherbedarf für konstante Objekte einerseits und Wiederverwendbarkeit sowie Austauschbarkeit andererseits zu schließen.

Der Compiler unterstützt diesen Modus ausschließlich im statischen Betrieb, also nicht in Kombination mit der Option **`-m`** oder **`-M`**. Klassen, deren Instanzen über ein integriertes Array verfügen sollen, benötigen einen **`DEFINE`**-Block, in dem die Variablennamen für Array und Array-Länge über die Pseudo-Methode **`MARKER.inlineArray(arrayName, lengthName)`** definiert werden. Sollen zur Laufzeit Instanzen dieser Klassen angelegt werden, müssen explizite Konstruktoren verwendet werden, die die aktuelle **`this`**-Instanz über **`MAGIC.useAsThis(object)`** deklarieren müssen. Dabei erfolgt keine Typprüfung, des weiteren ist bis zu diesem Aufruf die Verwendung von Instanzvariablen unzulässig (auch in einer eventuell gerufenen **`super`**-Methode).

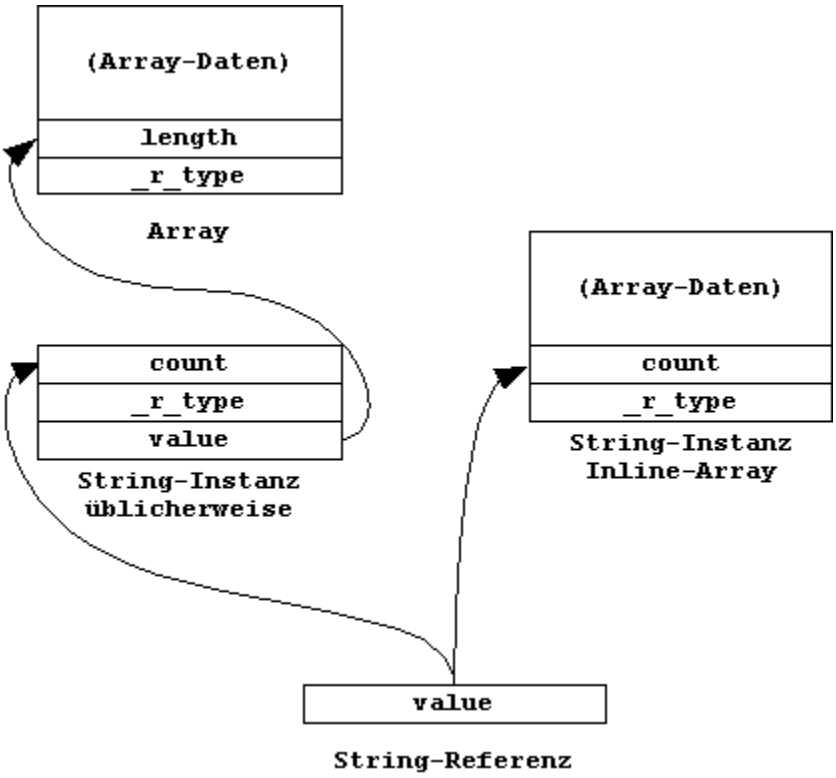
8.1 Veränderte Objektstruktur

```
public final class String {
  private char[] value; //Name für den Zugriff auf das Inline-Array
  private int count; //Länge des Inline-Arrays

  private void DEFINE() {
    MARKER.inlineArray(value, count); //value und count markieren
  }
  public String(char[] arr) {
    int i;
    //neue Instanz anlegen - erst dann sind Instanz-Variablen gültig!
    MAGIC.useAsThis(rte.DynamicRuntime.newInstance(
      MAGIC.getInstScalarSize("String")+(arr.length<<1),
      MAGIC.getInstRelocEntries("String"), MAGIC.clssDesc("String")));
    count=arr.length;
    for (i=0; i<arr.length; i++) value[i]=arr[i];
  }
public String copy() {
    int i;
    String ret;
    ret=(String)rte.DynamicRuntime.newInstance(
      MAGIC.getInstScalarSize("String")+count,
      MAGIC.getInstRelocEntries("String"), MAGIC.clssDesc("String"));
    ret.count=count;
    for (i=0; i<count; i++) ret.value[i]=value[i];
    return ret;
  }
```

```
    public int length() { //Zugriff auf Länge wie gewohnt
      MARKER.inline();
      return count;
    }
  }
```

Im obigen Beispiel wird das Buchstaben-Array bei Instanzen der Klasse **`String`** in das Objekt integriert, es wird also kein zusätzliches Array-Objekt für **`value`** benötigt. Folgende Graphik verdeutlicht den Unterschied zwischen herkömmlichen Strings (links) und Strings mit eingebetteten Arrays (rechts) bei Verwendung der Option **`-l`** (ansonsten sind pro Objekt weitere drei Felder vorhanden, siehe Kapitel 6):



9 Indirekte Skalare

In diesem Kapitel werden die Besonderheiten der Compiler-Option **-M** (siehe auch Kapitel 3.4) behandelt. Bei Verwendung dieser Option werden Skalare einer Instanz indirekt über das Adressfeld **_r_indirScalarAddr** im Objekt-Header zugegriffen. So ist für Instanzen eine Entkopplung der Referenzen und Objektinformationen einerseits und den Skalaren andererseits möglich, was beim Einsatz mehrerer Konsistenzmodelle vorteilhaft sein kann.

Die bisher zweiköpfigen und in sich abgeschlossenen Objekte werden also um einen zusätzlichen Speicherbereich erweitert, der ausschließlich Skalare des Objekts enthält. Typzeiger oder ähnliches sind in diesem Bereich nicht vorgesehen (eine Verwendung von Zeigern über eine Konvertierung eines Skalars in einen Zeiger mit Hilfe von **MAGIC** ist zwar möglich, aber nicht empfohlen).

Um diese Funktionalität bereitstellen zu können, sind Anpassungen am Wurzelobjekt sowie an der Laufzeitstruktur erforderlich, welche in den folgenden beiden Unterkapiteln beleuchtet werden.

9.1 Veränderte Objektstruktur

```
package java.lang;
import rte.SClassDesc;
public class Object {
  public SClassDesc _r_type;
  public Object _r_next;
  public int _r_relocEntries, _r_scalarSize;
  public int _r_indirScalarAddr, _r_indirScalarSize;
  private void DEFINE() {
    MARKER.head(_r_relocEntries); MARKER.head(_r_scalarSize);
    MARKER.head(_r_indirScalarAddr); MARKER.head(_r_indirScalarSize);
  }
}
package java.lang;
public class String {
  public char[] value;
  public int count;
  private void DEFINE() {
    MARKER.head(count);
  }
}
package rte;
public class SArray {
  public int length, _r_dim, _r_stdType;
  public SClassDesc _r_clssType;
  private void DEFINE() {
    MARKER.head(length); MARKER.head(_r_dim); MARKER.head(_r_stdType);
  }
}
```

Durch die Angabe von **`MARKER.head(.)`** in den **`DEFINE()`**-Methoden werden die übergebenen Variablen im Objektheader verankert, anstatt sie wie im Standardfall bei der Option **`-M`** im indirekt adressierten Bereich zu allozieren. Der Compiler erwartet dies bei den oben aufgeführten Variablen, weitere Variablen können vom Programmierer über den gleichen Mechanismus aus dem indirekt adressierten in den direkt adressierten Bereich verschoben werden.

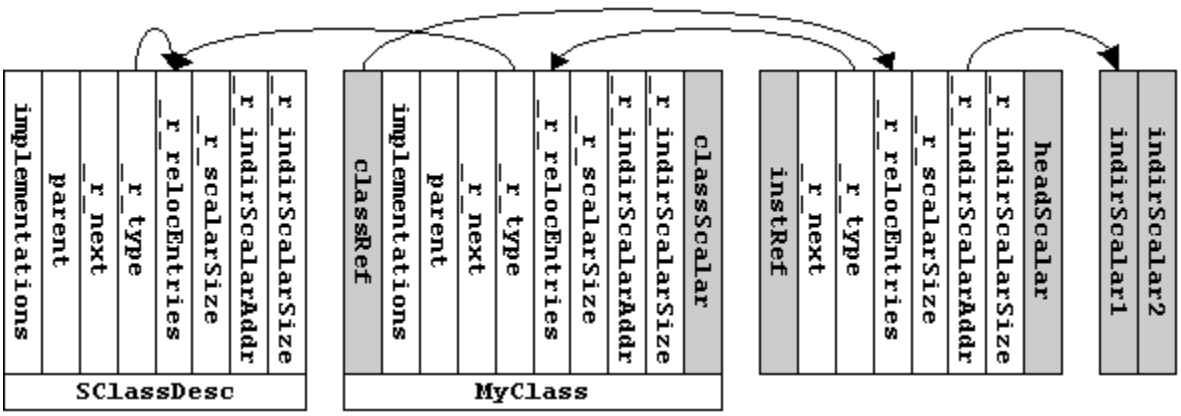
Für eine Klasse mit einer Klassenreferenz und einem Klassenskalar sowie einer Instanzreferenz und drei Instanzskalaren sähe die Deklaration wie folgt aus, wenn ein Skalar im Objekt selbst und zwei Skalare indirekt adressiert werden sollen:

```
public class MyClass {
  public static Object classRef;
  public static int classScalar;
  public Object instRef;
  public int headScalar;
  public int indirScalar1, indirScalar2;
  private void DEFINE() {
    MARKER.head(headScalar);
  }
}
```

Eine Allozierung könnte durch folgenden Code vorgenommen werden:

```
public class Kernel {
  public void main() {
    MyClass.classRef=new MyClass();
  }
}
```

Nach der Ausführung der **`main()`**-Methode wäre folgende Speicherstruktur aufgebaut:



Die Klassenvariablen sowie die Instanzreferenzen sind also in den Heap-Objekten integriert, die Instanzskalare sind jedoch standardmäßig in einen indirekt adressierbaren Speicherbereich ausgelagert. Soll ein Instanzskalar in den Objektbereich eingegliedert werden, wird dies über eine Markierung über **`MARKER.head(.)`** in der **`DEFINE()`**-Methode erreicht, wie dies bei den für die Laufzeitstrukturen erforderlichen Variablen vorgeschrieben ist.

Der Compiler legt die indirekt adressierten Skalare im initialen Speicherabbild direkt hinter das Eigentümerobjekt, so dass eine eindeutige Zuordnung der Bereiche möglich ist. Zur Laufzeit ist die Position frei wählbar, auch nach der Allozierung kann der Skalarbeich verschoben werden.

9.2 Veränderte Laufzeitumgebung

Zur Unterstützung von ausgelagerten Skalaren ist eine Anpassung der Laufzeitumgebung (siehe Kapitel 6) erforderlich. Zusätzlich zu den bisher vorhandenen Größenfeldern ist ein weiteres Größenfeld bei der Allozierung eines Objektes erforderlich:

```
public static Object newInstance(int scalarSize, int indirScalarSize,
    int relocEntries, SClassDesc type) { ... }
```

Bei Nutzung von Arrays sind die Aufrufe an **newInstance** in **newArray** und **newMultArray** anzupassen, die Änderungen an der Beispielimplementierung von Kapitel 6.5 sind also insgesamt:

```
package rte;
public class DynamicRuntime {
  private static int nextFreeAddress;
  public static Object newInstance(int scalarSize, int indirScalarSize,
      int relocEntries, SClassDesc type) {
    int start=nextFreeAddress, rs, i; Object me;
    rs=relocEntries<<2;
    scalarSize=(scalarSize+3)&~3;
    indirScalarSize=(indirScalarSize+3)&~3;
    nextFreeAddress+=rs+scalarSize+indirScalarSize;
    for (i=start; i<nextFreeAddress; i+=4) MAGIC.wMem32(i, 0);
    me=MAGIC.cast2Obj(start+rs);
    me._r_indirScalarAddr=start+rs+scalarSize;
    me._r_scalarSize=scalarSize; me._r_indirScalarSize=indirScalarSize;
    me._r_relocEntries=relocEntries;
    me._r_type=type;
    return me;
  }
  public static SArray newArray(int length, int arrDim, int entrySize,
      int stdType, SClassDesc clssType) {
    int iSS, rlE; SArray me;
    rlE=MAGIC.getInstRelocEntries("SArray");
    iSS=MAGIC.getInstIndirScalarSize("SArray");
    if (arrDim>1 || entrySize<0) rlE+=length;
    else iSS=length*entrySize;
    me=(SArray)newInstance(MAGIC.getInstScalarSize("SArray"), iSS, rlE,
      MAGIC.clssDesc("SArray"));
    me.length=length;
    me._r_dim=arrDim;
    me._r_stdType=stdType;
    me._r_clssType=clssType;
    return me;
  }
}
```

10 In-Image Symbolinformationen

In der aktuellen Version werden drei Generatoren für Symbolinformationen unterstützt, die in den folgenden Unterkapiteln diskutiert werden. Sie erzeugen Symbolinformationen für Packages, Units und Methoden, die wiederum zum Debugging verwendet werden können und als Grundlage für den TextualCall (siehe Kapitel 12.7) oder für Laufzeit-Compilierung dienen.

10.1 raw-Symbolinformationen

Die Auswahl von raw-Symbolinformationen geschieht über die Compiler-Option (siehe Kapitel 5.1) **`-u raw`**, er werden keine Parameter benötigt. Nach Abschluss der Übersetzung erzeugt dieses Modul ein byte-Array, in dem sich die Symbolinformationen für erzeugte Packages, Units und Methoden befinden. Es ist über den Bezeichner **`info`** in der automatisch angelegten Klasse **`RawInfo`** im Package **`symbols`** zugreifbar und besteht aus Blöcken, die mit einem Identifikationsbuchstaben (**`P`** für Package, **`U`** für Unit oder **`M`** für Methode) beginnen, eine variable Zahl an ASCII-Zeichen enthalten und mit dem Ausrufezeichen **`!`** terminiert sind. Zusammengehörende Blöcke folgen direkt aufeinander, die Blöcke für die Methoden einer Unit stehen also im unmittelbaren Anschluss an den Block dieser Unit. Das letzte Zeichen des Gesamtarrays ist ebenfalls ein Ausrufezeichen **`!`** entsprechend einem vollständig leeren Block.

Blockaufbau	Beschreibung
`P`(fullname)**`!`**	Das einzige Feld eines Packages enthält dessen voll qualifizierten Namen, also zum Beispiel **`java.lang`**.
`U`(name)**`#`**(modf)(addr)(exts)**`!`**	Das erste Feld einer Unit enthält deren einfachen Namen, also zum Beispiel **`String`**, und wird durch das Zeichen **`#`** abgeschlossen. Danach folgen zwei hexadezimal-ASCII-codierte 32-Bit-Werte mit folglich jeweils acht Zeichen Länge, wobei ersteres die Modifier der Unit (siehe Compiler-Klasse **`compbase.Modifier`** bzw. **`compbase.Unit`**) und letzteres die Adresse des Deskriptors enthält. Sollte die Unit eine Klasse und kein Interface sein und nicht direkt vom Wurzelobjekt abgeleitet sein, folgt ein weiteres Textfeld mit dem voll qualifizierten Namen dieser Klasse, also zum Beispiel **`java.lang.String`**.
`M`(name)**`#`**(modf)(addr)**`!`**	Das erste Feld einer Methode enthält deren einfachen Namen mit voll qualifizierten Parametern, also zum Beispiel **`equals(java.lang.String)`**, und wird durch das Zeichen **`#`** abgeschlossen. Danach folgen zwei hexadezimal-ASCII-codierte 32-Bit-Werte mit folglich jeweils acht Zeichen Länge, wobei ersteres die Modifier der Methode (siehe Compiler-Klasse **`compbase.Modifier`** bzw. **`compbase.Mthd`**) und letzteres die Adresse des Code-Objekts enthält.

Bestünde ein Image ausschließlich aus der Klasse String mit einer Methode length(), würde das resultierende Byte-Array in etwa folgendermaßen aussehen:

```
Pjava!Pjava.lang!UString#0000000100100090!Mlength()#0058000100100F5C!!
1-----2----------3-------4-------5--------6---------7-------8--------9
```

Der erste Block (1) gehört zum Package **`java`**, der zweite Block (2) zum Package **`java.lang`**. Der dritte Block (3) definiert die Klasse **`String`** mit Modifier (4) **`0x00000001`** an Adresse (5) **`0x00100090`**. Der vierte Block (6) beschreibt die Methode **`length()`** mit Modifier (7) **`0x00580001`** an Adresse (8) **`0x00100F5C`**. Der letzte Block (9) enthält die Ende-Markierung **`!`**.

10.2 rte-Symbolinformationen

Die Auswahl von rte-Symbolinformationen geschieht über die Compiler-Option (siehe Kapitel 5.1) **`-u rte`**, er werden keine Parameter benötigt. Nach Abschluss der Übersetzung erzeugt dieses Modul für jedes Package ein Objekt vom Typ **`rte.SPackage`** und füllt dessen Felder, das root-Package wird in die statische Variable **`rte.SPackage.root`** eingetragen. Darüberhinaus werden in den bereits erzeugten Instanzen der Klassen **`rte.SClssDesc`** und **`rte.SMthdBlock`** weitere Felder ausgefüllt. Die Quellcode für eine korrekte minimale Deklaration dieser drei Klassen ist wiefolgt:

```
package rte;
public class SPackage { //zusätzliche Klasse, alle Felder werden benötigt
  public static SPackage root; //statisches Feld für root-Package
  public String name; //einfacher Name des Packages
  public SPackage outer; //höhergelegenes Package
  public SPackage subPacks; //erstes tiefergelegenes Package
  public SPackage nextPack; //nächstes Package auf gleicher Höhe
  public SClassDesc units; //erste Unit des aktuellen Packages
}
package rte;
public class SClassDesc { //Klasse mit zusätzlichen Instanz-Variablen
  public SClassDesc parent; //bereits bisher vorhanden: erweiterte Klasse
  public SIntfMap implementations; //bereits bisher vorhanden: Interfaces
  public SClassDesc nextUnit; //nächste Unit des aktuellen Packages
  public String name; //einfacher Name der Unit
  public SPackage pack; //besitzendes Package
  public SMthdBlock mthds; //erste Methode der Unit
  public int modifier; //Modifier der Unit
}
package rte;
public class SMthdBlock { //Klasse mit zusätzlichen Instanz-Variablen
  public String namePar; //einfacher Name, vollqualifizierte Parametertypen
  public SMthdBlock nextMthd; //nächste Methode der aktuellen Klasse
  public int modifier; //Modifier der Methode
}
```

Die Instanzen auf einer Ebene sind jeweils durch lineare Listen miteinander verbunden, die über die **`next*`**-Felder zugreifbar sind. Bei Packages wird der einfache Name (zum Beispiel **`lang`**) im Feld **`name`** eingetragen, das umschließende Package über das Feld **`outer`** erreicht werden, das erste enthaltene Subpackage über das Feld **`subPacks`**, die erste Klasse über das Feld **`units`**. Bei Klassen bleiben die auch ohne Symbolinformationen benötigten Felder **`parent`** und **`implementations`** bestehen, hinzu kommt ein Feld **`name`** für den einfachen Namen (zum Beispiel **`String`**), eine Referenz **`pack`** auf das enthaltene Package (welches in diesem Fall den Namen **`lang`** hätte), eine Referenz **`mthd`** auf die erste Methode sowie in **`modifier`** den skalaren Wert der Modifier (siehe **`compbase.Modifier`** bzw. **`compbase.Unit`**). Bei Methoden-Code-Objekten enthält das Feld **`namePar`** den einfachen Namen mit voll qualifizierten Parametertypen (zum Beispiel **`equals(java.lang.String)`**) und in **`modifier`** den skalaren Wert der Modifier (siehe **`compbase.Modifier`** bzw. **`compbase.Mthd`**).

10.3 BootStrap-Symbolinformationen

Die Auswahl von BootStrap-Symbolinformationen geschieht über die Compiler-Option (siehe Kapitel 5.1) **`-u strap`**, der optionale Parameter **`noCompilerPacks`** schließt die Packages des Compilers von der Symbolgenerierung aus. Ähnlich wie bei rte-Symbolinformationen (siehe voriges Unterkapitel 10.2) werden die Symbolinformationen in Objekten des Heaps abgelegt, wobei diese jedoch kompatibel zu den Compiler-Instanzen sind und zusätzlich auch Informationen zu Variablen enthalten. Wenn der Compiler im Zielsystem integriert ist, kann auf diese Weise nicht nur unabhängiger Quelltext im laufenden System übersetzt werden. Stattdessen kann auf alle Klassen, Methoden und Variablen zugegriffen werden, auch wenn der Quelltext dieser bereits übersetzten Module nicht mehr zur Verfügung steht. Das root-Objekt der Symbolinformationen wird im statischen Feld **`root`** der Klasse **`symbols.BootStrap`** eingetragen, der minimal erforderliche Quelltext dazu kann wie folgt aussehen:

```
package symbols;
public class BootStrap {
  public static compbase.Pack root; //statisches Feld für root-Package
}
```

Darüberhinaus werden die Klassen **`Token`**, **`Unit`**, **`UnitDummy`**, **`Pack`**, **`QualID`**, **`QualIDList`**, **`StringList`**, **`TypeRef`**, **`AccVar`**, **`Vrbl`**, **`Mthd`**, **`MthdDummy`** und **`Param`** aus dem Package **`compbase`** benötigt. Typischerweise werden diese direkt aus den Compilerquellen stammen und müssen infolgedessen nicht gesondert angelegt werden. Falls zum Beispiel für Tests oder andere Aufgaben das BootStrap-Modul verwendet werden soll, ohne dass der Compiler im Zielsystem integriert ist, können die jeweils benötigten Felder in der Klasse **`symbols.BootStrapSymbols`** ermittelt werden.

Der Aufruf eines integrierten Compilers könnte wie folgt implementiert werden:

```
public class Compiler {
  public static void compileTest() {
    String[] argv; //Parameter für die zu startende Übersetzung
    Context ctx; //Kontext des Compilers, Erzeugung wie folgt:
    ctx=new Context(DeviceList.out, new SBinWriter(),
        new STextReader(), new SBinReader(), new SDirLister());
    ctx.mem=new NativeMemory(); //kein Abbild, sondern Wirtssystem nutzen
    ctx.dynaMem=MAGIC.movable; //verschiebbare Objekte wie im Wirtssystem
    ctx.assignCall=MAGIC.assignCall; //...weitere Optionen nach Wunsch
    ctx.root=symbols.BootStrap.root; //root-Package setzen
    argv=new String[7]; //neues Parameter-Array anlegen
    argv[0]="-s"; argv[1]="0"; //nativen Speicher statt Abbild nutzen
    argv[2]="-t"; argv[3]=MAGIC.ptrSize==4 ? "ia32": "amd64"; //Architektur
    argv[4]="-G"; argv[5]="-g"; //keine auslassenden Optimierungen
    argv[6]="CompileTest.java"; //Quelltext
    if (ctx.compile(argv)==0) { /* ok */ }
    else { /* Fehler */ }
  }
}
```

Eine Implementierung der für den Compiler erforderlichen Klassen (siehe Konstruktor der Klasse **`Context`**) ist vom Wirtssystem abhängig und daher nicht Bestandteil dieses Compiler-Handbuchs.

11 Native Linux- und Windows-Programme

11.1 Grundlegende Funktionsweise

Grundsätzlich ist der vom Compiler erzeugte Code auch unter Microsoft Windows oder Linux ausführbar. Dennoch müssen je nach verwendetem Betriebssystem bestimmte Rahmenbedingungen erfüllt werden und für System- oder Benutzer-Interaktion verschiedene Funktionen außerhalb der üblichen Laufzeitumgebung gerufen werden. Diese Systemaufrufe müssen sich dabei natürlich nach den Konventionen des verwendeten Systems richten, wobei dafür einerseits eine Interrupt-Schnittstelle (Linux) und andererseits Funktionsaufrufe mittels Shared-Memory (Linux und Windows) in Frage kommen.

Die Interrupt-Schnittstelle unter Linux ist durch die Kernel-Spezifikation klar definiert. Je nach verwendetem Linux-Kern werden die Funktionsparameter unterschiedlich übergeben: in Registern, über den Stack oder in Kombination von Registern und Stack. Derzeit bietet der Compiler keine direkte Unterstützung für die automatische Generierung von Interrupt-Aufrufen, so dass der Benutzer auf die Handcodierung mittels Inline-Code angewiesen ist (siehe Kapitel 12.6). Ein Beispiel dazu findet sich in Kapitel 11.2 und unter [nat].

Die Funktionsaufruf-Schnittstelle unter Linux und Windows ist auf dynamische Bibliotheken aufgebaut und von der grundsätzlichen Funktionsweise für den SJC-Benutzer recht ähnlich. In beiden Fällen wird anhand des Bibliotheksnamens über eine Systemfunktion ein Handle oder eine Adresse der Bibliothek ermittelt, die die gewünschte Funktion enthält. Durch eine weitere Systemfunktion kann innerhalb der dann geladenen Bibliothek die gewünschte Funktion über ihren Namen gesucht werden, was im Erfolgsfall die Einsprungadresse in diese Funktion liefert. Die Kapitel 11.2 und 11.3 enthalten Beispiele für Linux und Windows, unter [nat] findet sich eine ausführliche Demonstration.

Um die Systemfunktion rufen zu können, die die Bibliotheks- und Funktionsanfragen auflöst, ist bereits eine Systemfunktion erforderlich. Aus diesem Grund enthält der Windows-Lader ebenso wie der erweiterte Linux-Lader neben einem systemkonformen Header und der Laufzeitinitialisierung auch die Auflösung dieser grundlegenden Systemfunktionen, so dass im Java-Code einfach auf diese zurückgegriffen werden kann. Alle drei Header sind unter [nat] verfügbar.

Sind die Einsprungsadressen und Signaturen der zu rufenden Bibliotheksfunktionen bekannt, so kann SJC mit Hilfe des Schlüsselworts **`native`** die für einen Aufruf erforderlichen Stackframes automatisch erzeugen. Dazu muss in einer (beliebigen) Klasse eine native Methode als **`static`** deklariert werden und eine gleichnamige statische **`int`**-Variable mit der Einsprungsadresse dieser Methode in derselben Klasse zur Verfügung gestellt werden. Bei Aufrufen dieser nativen Methoden werden die für das System passenden Stackframes automatisch erzeugt und die Funktion an der Adresse gerufen, auf die die **`int`**-Variable zeigt. Beispiele finden sich in den nächsten Unterkapiteln 11.2 und 11.3 sowie unter [nat].

Die von SJC verwendete Heap-Struktur (siehe Kapitel 3.3 und 3.4) ist sowohl Linux als auch Microsoft Windows gänzlich unbekannt. Aus diesem Grund sollte stets darauf geachtet werden, dass bei der Kommunikation mit dem Betriebssystem nur Basistypen oder Zeiger auf betriebssystemkonforme Speicherbereiche (zum Beispiel mittels STRUCT, siehe Kapitel 7.3 und 12.4, oder MAGIC.addr, siehe Kapitel 7.1) verwendet werden.

11.2 Native Linux Programme

Werden unter Linux nur Kernelfunktionen verwendet, so kann der einfache Linux-Header ohne Bibliotheksunterstützung verwendet werden. Die Funktionen zum Laden einer Bibliothek und zur Namensauflösung sind dann nicht vorhanden. Der Aufruf von Kernel-Funktionen per Interrupt muss handcodiert werden. Das folgende Beispiel gibt ein Zeichen auf Standard-Out aus:

```
  public static void printChar(int c) {
    //setze eax auf Funktion 4: print string
    MAGIC.inline(0xB8, 0x04, 0x00, 0x00, 0x00); //mov eax,4
    //setze ebx auf 1: Handle für Standard-Out
    MAGIC.inline(0xBB, 0x01, 0x00, 0x00, 0x00); //mov ebx,1
    //setze ecx auf Adresse des auszugebenden Strings
    MAGIC.inline(0x8D, 0x4D); MAGIC.inlineOffset(1, c); //lea ecx,[ebp+8]
    //setze edx auf 1: Länge des auszugebenden Strings
    MAGIC.inline(0x89, 0xDA); //mov edx,ebx
    //Aufruf des Kernels
    MAGIC.inline(0xCD, 0x80); //int 0x80
  }
```

Entsprechend diesem Beispiel können alle Kernelfunktionen handcodiert werden. Je nach verwendeten Kerneloptionen und verwendeter Kernelversion sind unter Umständen weniger oder sogar keine Parameter über Register zu übergeben, stattdessen ist in diesem Fall eine Übergabe per Stack erforderlich.

Sollen neben Kernelfunktionen auch Bibltiotheksfunktionen, zum Beispiel zur Anbindung an ein X11-System, verwendet werden, so ist der erweiterte Linux-Header mit integrierter Bibliotheksunterstützung zu verwenden. Dieser liefert an den Adresse **`MAGIC.imageBase-12`** und **`MAGIC.imageBase-8`** die Einsprungadressen der Linuxfunktionen dlopen und dlsym zur Auflösung von Bibliotheks- und Funktionsnamen. Die Funktion XOpenDisplay kann beispielsweise folgendermaßen deklariert, initialisiert und verwendet werden:

```
  public native static int XOpenDisplay(int no); //native Funktion
  public static int XOpenDisplay; //Adresse der nativen Funktion
  ...
  public static int dlopen(byte[] libNameZ, int flags) { //handcodiertes dlopen
    int libAddr=MAGIC.addr(libNameZ[0]);
    MAGIC.inline(0xFF, 0x75); MAGIC.inlineOffset(1, flags);
    MAGIC.inline(0xFF, 0x75); MAGIC.inlineOffset(1, libAddr);
    MAGIC.inline(0xFF, 0x15); MAGIC.inline32(MAGIC.imageBase-12);
    MAGIC.inline(0x89, 0x45); MAGIC.inlineOffset(1, libAddr);
    MAGIC.inline(0x5B, 0x5B);
    return libAddr;
  }
  public static int dlsym(int libHandle, byte[] funcNameZ) {
    int funcAddr=MAGIC.addr(funcNameZ[0]);
    MAGIC.inline(0xFF, 0x75); MAGIC.inlineOffset(1, funcAddr);
    MAGIC.inline(0xFF, 0x75); MAGIC.inlineOffset(1, libHandle);
    MAGIC.inline(0xFF, 0x15); MAGIC.inline32(MAGIC.imageBase-8);
    MAGIC.inline(0x89, 0x45); MAGIC.inlineOffset(1, funcAddr);
    MAGIC.inline(0x5B, 0x5B);
    return funcAddr;
  }
```

```
...
public static void init() {
  int bib;
  if ((bib=dlopen(MAGIC.toByteArray("libX11.so", true), 0x0102))==0) {
    return; //Fehler: Bibliothek nicht gefunden
  }
  if ((XOpenDisplay=dlsym(i, MAGIC.toByteArray("XOpenDisplay", true)))==0) {
    return; //Fehler: Funktion XOpenDisplay nicht gefunden
  }
}
...
public static void example() {
  int display;
  if ((display=XOpenDisplay(0))==0) { //Zugriff auf native-Funktion
    return; //Fehler: konnte Display 0 nicht öffnen
  }
  ... //Verwendung des geöffneten Displays
}
```

Die Funktionen dlopen und dlsym könnten auch über native Funktionen abgebildet werden, wenn man statt des Null-terminierten SJC-konformen Byte-Arrays einen Zeiger auf das erste Byte des Null-terminierten Byte-Arrays übergeben würde. Der für die Handcodierung erforderliche Aufwand ist bei der nativ deklarierten Funktion **`XOpenDisplay`** nicht erforderlich, die für einen gütigen Systemaufruf erforderlichen Maßnahmen werden vom Compiler automatisch durchgeführt. Des weiteren entfällt bei nativ deklarierten Systemaufrufen der Code des Wrappers sowie der Aufruf des Wrappers, da der Compiler den nativen Systemaufruf direkt an der verwendeten Stelle erzeugt.

Der Zugriff auf die beim Programmstart übergebenen Parameter erfolgt sowohl beim einfachem wie auch beim erweiterten Header über den an **`MAGIC.imageBase-4`** abgelegten originalen Stackpointer. An dieser Adresse ist die Zahl der übergebenen Parametern abgelegt, darüber befindet sich für jeden Parameter ein Zeiger auf eine Null-terminierte Zeichenkette. Eine Auswertung der an ein Programm übergebenen Parameter könnte also folgendermaßen aussehen:

```
public static void showParam() {
  int esp, argc, i;
  esp=MAGIC.rMem32(MAGIC.imageBase-4);
  argc=MAGIC.rMem32(esp);
  for (i=1; i<=argc; i++) printArg(MAGIC.rMem32(i<<2));
}
public static void printArg(int addr) {
  int c;
  while ((c=(int)MAGIC.rMem8(addr++)&0xFF)!=0) Viewer.printChar(c);
  Viewer.printChar(13);
}
```

Sollen die Parameter innerhalb des Programms als SJC-konforme Strings verwendet werden, muss eine explizite Konvertierung vorgenommen werden.

11.3 Native Microsoft Windows Programme

Da unter Microsoft Windows die definitionskonformen Header für ausführbare Dateien reichlich leeren Platz bieten, um darin das Auflösen von Bibliotheksfuntionen vozusehen, werden die Adressen folgender Laufzeitvariablen und Systemfunktionen bereits vor dem Start des Java-Codes aufgelöst:

Handle für Standard-In	`MAGIC.imageBase-512+0`
Handle für Standard-Out	`MAGIC.imageBase-512+4`
Handle für Error-Out	`MAGIC.imageBase-512+8`
Zeiger auf übergebene Parameter	`MAGIC.imageBase-512+12`
Anzahl der übergebenen Parameter	`MAGIC.imageBase-512+16`
Funktion Kernel.GetStdHandle	`MAGIC.imageBase-512+20`
Funktion Kernel.GetCommandLineW	`MAGIC.imageBase-512+24`
Funktion Kernel.ExitProcess	`MAGIC.imageBase-512+28`
Funktion Kernel.WriteFile	`MAGIC.imageBase-512+32`
Funktion Kernel.ReadFile	`MAGIC.imageBase-512+36`
Funktion Kernel.CreateFileA	`MAGIC.imageBase-512+40`
Funktion Kernel.GetFileSize	`MAGIC.imageBase-512+44`
Funktion Kernel.CloseHandle	`MAGIC.imageBase-512+48`
Funktion Kernel.LoadLibraryA	`MAGIC.imageBase-512+52`
Funktion Kernel.GetProcAddress	`MAGIC.imageBase-512+56`
Funktion Kernel.VirtualAlloc	`MAGIC.imageBase-512+60`
Funktion Kernel.VirtualFree	`MAGIC.imageBase-512+64`
Funktion Shell.CommandLineToArgvW	`MAGIC.imageBase-512+72`

Entsprechend kann die Ausgabe eines Zeichens auf Standard-Out umgesetzt werden, wobei der vom System gesetzte `result`-Wert nicht ausgewertet wird:

```
public static void printChar(int c) {
  int result=0;
  MARKER.ignore(result);
  MAGIC.inline(0x6A, 0x00); //push byte 0 (no overlap)
  MAGIC.inline(0x8D, 0x45); MAGIC.inlineOffset(1, result); //lea eax,result
  MAGIC.inline(0x50); //push eax (Zeiger auf Ergebnis auf den Stack legen)
  MAGIC.inline(0x6A, 0x01); //push byte 1 (Zahl der auszugebenden Zeichen)
  MAGIC.inline(0x8D, 0x45); MAGIC.inlineOffset(1, c); //lea eax,c
  MAGIC.inline(0x50); //push eax (Zeiger auf Zeichen auf den Stack legen)
  //Handle für Standard-Out auf den Stack legen
  MAGIC.inline(0xFF, 0x35); MAGIC.inline32(MAGIC.imageBase-512+4);
  //Systemfunktion WriteFile aufrufen
  MAGIC.inline(0xFF, 0x15); MAGIC.inline32(MAGIC.imageBase-512+32);
}
```

Der Aufruf der Funktion LoadLibraryA kann wie folgt programmiert werden:

```
int addr=MAGIC.addr(libraryNameZ[0]), handle;
//Zeiger auf Bibliotheksnamen auf den Stack legen
MAGIC.inline(0xFF, 0x75); MAGIC.inlineOffset(1, addr);
//Funktion LoadLibraryA aufrufen
MAGIC.inline(0xFF, 0x15); MAGIC.inline32(MAGIC.imageBase-512+52);
//zurückgegebenes Handle sichern
MAGIC.inline(0x89, 0x45); MAGIC.inlineOffset(1, handle);
... //ab hier ist das Handle für die Bibliothek, falls gültig, verwendbar
```

Ebenso wie unter Linux können Funktionen auch für Microsoft Windows Programme als nativ deklariert werden. Da sich jedoch der von Windows geforderte Stackframe von demjenigen unter Linux unterscheidet und letzterer als Standard-Stackframe verwendet wird, müssen die entsprechenden Funktionen zusätzlich im **DEFINE**-Block der deklarierenden Klasse mit Hilfe von **MARKER.winDLL(.)** als Windows-Funktion markiert werden.

Zum Beispiel:

```
public class Win32Lib {
  private native static int getDC(int hWindow);
  private static int getDC;
  private static void DEFINE() { MARKER.winDLL("getDC"); }
  private static void initFunctions() {
    int handleDLL=loadLibrary(MAGIC.toByteArray("user32.dll", true));
    getDC=loadFunction(handleDLL, MAGIC.toByteArray("GetDC", true));
  }
}
```

Der Zugriff auf die einem Programm übergebenen Parameter erfolgt über die bei **MAGIC.imageBase-512+16** abgelegte Parameteranzahl sowie den an **MAGIC.imageBase-512+12** abgelegten Zeiger. Eine Auswertung könnte also folgendermaßen aussehen:

```
public static void showParam() {
  int argc=MAGIC.rMem32(MAGIC.imageBase-512+16);
  int base=MAGIC.rMem32(MAGIC.imageBAse-512+12);
  for (int i=0; i<argc; i++) printArg(MAGIC.rMem32(base+(i<<2)));
}
public static void printArg(int addr) {
  int c;
  while ((c=(int)MAGIC.rMem16(addr))!=0) {
    Viewer.printChar(c);
    addr+=2;
  }
  Viewer.printChar(13); Viewer.printChar(10);
}
```

Sollen die Parameter innerhalb des Programms als SJC-konforme Strings verwendet werden, muss eine explizite Konvertierung vorgenommen werden. Beispiele zu allen gezeigten Möglichkeiten finden sich unter [nat].

12 Beispiele

Alle Beispiele sind für statische Compilierung und 32 Bit Intel-Code ausgelegt. Weitere Beispiele finden sich auch in dem Beispielsystem PicOS (siehe [picos]), das 32 und 64 Bit unterstützt.

12.1 Verwendung von Inlining

Beispiel für: **MARKER.inline()**.

```
public class String {
  private char[] value;
  private int count;
  public int length() {
    MARKER.inline();
    return count;
  }
}
```

Bei Aufrufen der Methode **length()** eines Strings wird nach Möglichkeit kein Call erzeugt, sondern die erforderlichen Opcodes zum Auslesen der Instanz-Variable **count** direkt eingefügt. Mit dem Optimierer werden dann die Zugriffe weiter optimiert, so dass der programmierte Methodenaufruf in etwa vergleichbar mit einem direkten Zugriff auf die Variable **count** ist.

12.2 Zugriff auf I/O-Ports

Beispiel für: **MAGIC.wIOs8(.)**, **MAGIC.rIOs8(.)**.

```
public class RTC {
  public int getCurrentHour() {
    MAGIC.wIOs8(0x70, (byte)4); //Register 4 auswählen
    return (int)MAGIC.rIOs8(0x71)&0xFF; //Wert auslesen
  }
}
```

Die Echtzeituhr im PC belegt den I/O-Adreßraum ab 0x70. Das I/O-Register 0x70 wird zur Auswahl des RTC-internen Speichers verwendet, über I/O-Register 0x71 kann nachfolgend auf die ausgewählte RTC-Speicherstelle zugegriffen werden. Hier im Beispiel wird das RTC-Register 4 ausgewählt, das die aktuelle Stunde enthält. Das ausgelesene Byte wird nach einer Konvertierung in ein **int** und dem Löschen der von Java erzwungenen Vorzeichenerweiterung zurückgegeben.

12.3 Programmierung von Interrupt-Handlern

Beispiel für: **MAGIC.wMem32(.)**, **MAGIC.rMem32(.)**, **MAGIC.cast2Ref(.)**, **MAGIC.clssDesc(.)**, **MAGIC.mthdOff(.)**, **MAGIC.getCodeOff()**, **MAGIC.inline(.)**, **MAGIC.inline32(.)**, **MARKER.interrupt()**.

```
public class Interrupts {
  protected int getHandlerAddr() {
    return MAGIC.rMem32(MAGIC.cast2Ref(MAGIC.clssDesc("Interrupts"))
      +MAGIC.mthdOff("Interrupts", "intrHandler"))+MAGIC.getCodeOff();
  }
```

```
  private static void intrHandler() {
    MARKER.interrupt();
    //nur nicht-statisch: MAGIC.inline(0x8B, 0x3D); MAGIC.inline32(ADDR);
    //ab hier folgt der Code des Handlers
  }
  protected void initNotStatic() { //nur nicht-statische Compilierung
    MAGIC.wMem32(ADDR, MAGIC.cast2Ref(MAGIC.clssDesc("Interrupts")));
  }
}
```

Die Methode **intrHandler** ist durch **MARKER.interrupt(.)** als Interrupt-Handler markiert, der Compiler erzeugt in diesem Fall statt des üblichen, schlanken Methodenrahmens einen für Interrupts tauglichen Rahmen mit Sicherung und Wiederherstellung aller Ganzzahl-Register (bei Verwendung von **float** sowohl innerhalb als auch außerhalb von Interrupt-Routinen muß vom Programmierer eine entsprechende Sicherung der FPU-Register und des FPU-Zustands vorgenommen werden). Bei dynamisch verschiebbarem Code (Compiler-Option **-m**) wäre des weiteren die Initialisierung des Klassenkontextes erforderlich, zum Beispiel wie in der auskommentierten Zeile angegeben. In diesem Fall wäre vor Auftreten des ersten Interrupts außerdem an Adresse **ADDR** die Adresse des Klassendeskriptors für die Klasse Interrupts abzulegen, etwa wie in der Methode **initNotStatic(.)** angegeben.

12.4 Verwendung von STRUCTs

Beispiel für: **MAGIC.cast2Struct(.)**, **MAGIC.wMem16(.)**, **DEFINE()**.

```
public class VidChar extends STRUCT {
  public byte ascii, color;
}
public class VidMem extends STRUCT {
  public VidChar[] expl;
  public short[] chars;
  private void DEFINE() {
    //Offset und Element-Anzahl für expl und chars setzen
    MARKER.offset(expl, 0); MARKER.count(expl, 2000);
    MARKER.offset(chars, 0); MARKER.count(chars, 2000);
  }
}
public class Tester {
  public void test() {
    VidMem m; //STRUCT deklarieren
    m=(VidMem)MAGIC.cast2Struct(0xB8000); //Position des Structs definieren
    MAGIC.wMem16(0xB8000+15*2, (char)0x0748); //'H' mit Farbe an Position 15
    m.chars[16]=(char)0x0769; //'i' mit Farbe an Position 16
    m.expl[17].ascii=(byte)'!'; //'!' an Position 17 ausgeben
    m.expl[17].color=(byte)0x07; //Farbe der Position 17 setzen
  }
}
```

Die Struktur **VidChar** bildet das Speichermuster für ein angezeigtes Zeichen im Textbildschirmspeicher ab. Das erste Byte an Offset 0 wird von der Grafikkarte als ASCII-Code interpretiert, das zweite Byte gibt dessen Farbe an. Der gesamte Textbildschirmspeicher setzt sich im 80x25-Modus aus 2000 solcher Zeichen zusammen, die in der Struktur **VidMem** einmal strukturiert als Array aus Bildschirmzeichen und einmal unstrukturiert als 16-Bit-Werte deklariert werden. Beide Arrays beginnen an Offset 0 des Bildschirmspeichers. Dessen Beginn wird bei 0xB8000 angenommen, passend dazu wird in der Methode **Tester.test()** eine Struktur des Bildschirmspeichers mit **MAGIC.cast2Struct(.)** an diese Adresse gesetzt. Nachfolgend sind drei unterschiedliche Möglichkeiten angegeben, wie ein Zeichen auf den Bildschirm geschrieben werden kann. Zuerst über einen direkten Zugriff auf den Videospeicher mit **MAGIC.wMem16(.)**, hier muß einerseits die Startadresse als auch die interne Organisation des Bildschirmspeichers explizit berücksichtigt werden. Als zweites ist ein Zugriff über die Struktur-Variable mit vorberechnetem Wert angegeben, hier ist nach erfolgreicher Initialisierung des **STRUCT** direkt die Zielposition als Array-Index verwendbar. Als letztes ist ein semantisch-strukturierter Zugriff angegeben, bei dem Zeichen und Farbe über Zielposition und Namen zugreifbar sind.

12.5 Initialisierung im Embedded-Mode

Beispiel für: **MAGIC.rMem32(.)**, **MAGIC.wMem32(.)**, **MAGIC.embedded**, **MAGIC.getRamAddr()**, **MAGIC.getRamSize()**, **MAGIC.getRamInitAddr()**, **MAGIC.getConstMemorySize()**.

Werden initialisierte Klassenvariablen im Embedded-Mode verwendet, muss der für Klassenvariablen vorgesehene Bereich vor einer Verwendung initialisiert werden. Dazu ist in dem vom Compiler erzeugten Speicherabbild ein Speicherbereich vorgesehen, der linear ins RAM kopiert werden muss. Folgender Code kann dies bewerkstelligen:

```
int m, src, dst;
if (MAGIC.getRamSize()>0) { //nur möglich, wenn MAGIC.embedded==true ist
  m=(dst=MAGIC.getRamAddr())+MAGIC.getRamSize();
  for (src=MAGIC.getRamInitAddr(); dst<m; dst+=4, src+=4)
      MAGIC.wMem32(dst, MAGIC.rMem32(src));
}
```

Sollen außer den initialisierten Klassenvariablen auch konstante Objekte kopiert werden (zum Beispiel für Harvard-Architekturen für einheitliche „dynamische“ Objekte mit Compiler-Option **-E**, siehe Kapitel 5.1), kann dies einfach durch weiteres Kopieren von **MAGIC.getConstMemorySize()** Bytes erreicht werden, da die konstanten Objekte direkt hinter den initialisierten Klassenvariablen alloziert werden:

```
int m, src, dst;
if (MAGIC.getRamSize()>0) { //nur möglich, wenn MAGIC.embedded==true ist
  src=MAGIC.getRamInitAddr();
  m=(dst=MAGIC.getRamAddr())+MAGIC.getRamSize()+MAGIC.getConstMemorySize();
  while (dst<m) MAGIC.wMem8(dst++, MAGIC.rMem8(src++));
}
```

12.6 Handcodierter Methodenaufruf

Beispiel für: **MAGIC.inline(.)**, **MAGIC.getCodeOff()**.

In einem objektorientierten System, wie es SJC erzeugt, sind dafür je nach vorliegendem Fall eine Instanzreferenz und eine Referenz auf die zu rufende Methode sowie der dazugehörende Klassendeskriptor erforderlich. Folgender Quelltext ruft beliebige Methoden bei Verwendung von dynamisch verschiebbarem Code:

```
static void call(int instAddr, int clssAddr, int mthdAddr) {
  mthdAddr+=MAGIC.getCodeOff();
  MAGIC.inline(0x57, 0x56); //push edi, esi
  MAGIC.inline(0x8B, 0x75, 0x10); //mov esi,[ebp+16]
  MAGIC.inline(0x8B, 0x7D, 0x0C); //mov edi,[ebp+12]
  MAGIC.inline(0xFF, 0x55, 0x08); //call dword [ebp+8]
  MAGIC.inline(0x5E, 0x5F); //pop esi, edi
}
```

Da der Typ im Inline-Code nicht relevant ist, kann statt der Instanz-Adresse auch eine Referenz auf die Instanz und statt der Klassendeskriptor-Adresse auch eine Referenz auf den Klassendeskriptor übergeben werden. Alternativ kann also auch die Signatur

```
static void call(Object inst, SClassDesc desc, int mthdAddr) { . }
```

mit identischem Inline-Code verwendet werden.

Wird eine dynamische Methode gerufen, muss **clssAddr** den Typ der Instanz enthalten. Wird eine statische Methode gerufen, muss **clssAddr** die deklarierende Klasse (und keine davon abgeleitete Klasse) enthalten. Für statische Methoden kann auf alle Befehle mit esi-Register (letztes Byte bei **push**, Zeile nach **push** sowie erstes Byte bei **pop**) verzichtet werden, in diesem Fall kann bei statischer Übersetzung (keine Option **-m**, siehe Kapitel 5.1) zusätzlich auf alle Befehle mit edi-Register (erstes Byte bei **push**, Zeile vor **call** sowie letztes Byte bei **pop**) verzichtet werden. Werden streamline-Objekte durch Auswahl der Option **-l** erzeugt und enthält **SMthdBlock** keine skalaren Variablen, liefert **MAGIC.getCodeOff()** immer 0 entsprechend dem Wert **instScalarTableSize** für **SMthdBlock** in syminfo.txt (siehe Kapitel 5.2), so dass auf das erste Statement in obigem Quellcode verzichtet werden kann. Sollen also nur statische Methoden bei statischer Übersetzung gerufen werden, kann dies mittels

```
static void call(int mthdAddr) {
  mthdAddr+=MAGIC.getCodeOff(); //mit SMthdBlock-Skalaren oder !streamline
  MAGIC.inline(0xFF, 0x55, 0x08); //call dword [ebp+8]
}
```

erreicht werden.

12.7 TextualCall

Der Aufruf einer Methode anhand eines Textes anstelle von compiliertem Code wird in diversen Systemen (siehe [oberon] oder [plurix]) benutzt, um ein einfaches und schlankes Benutzer-Interface anbieten zu können. Dazu wird der in Kapitel 12.6 vorgestellte Code zum Aufruf einer Methode benötigt, die in Kapitel 10 beschriebenen Generatoren für In-Image Symbolinformationen liefern die während der Laufzeit benötigten Informationen zum Auffinden von Klassen und Methoden.

Existiert beispielsweise eine statische Methode **toBeCalled()** in der Klasse **Kernel** im Package **kernel**, so wird diese im Java-Quelltext üblicherweise über ein Statement der Form

```
kernel.Kernel.toBeCalled();
```

gerufen. Beim TextualCall könnte dieser Aufruf abgebildet werden über

```
callByName("kernel", "Kernel", "toBeCalled()");
```

mit den Parametern für Package, Klasse und Methode. Da die Parameter übliche Strings sind, können diese auch zur Laufzeit verändert werden und somit ein und derselbe Code zum Aufruf verschiedenster Methoden genutzt werden. Unter Verwendung des rte-Generators (siehe Kapitel 10.2) könnte die Methode **callByName** wiefolgt implementiert werden:

```
private static void callByName(String pack, String unit, String mthd) {
  SClassDesc u;
  int mthdAddr;
  SPackage p=SPackage.root.subPacks;
  SMthdBlock m;
  while (p!=null) { //alle Packages durchsuchen
    if (pack.equals(p.name)) { //Package gefunden
      u=p.units; //erste Unit des Packages auswählen
      while (u!=null) { //alle Units durchsuchen
        if (unit.equals(u.name)) { //Klasse gefunden
          m=u.mthds; /erste Methode der Unit auswählen
          while (m!=null) { //alle Methoden durchsuchen
            if (mthd.equals(m.namePar)) { //Method gefunden
              if ((m.modifier&0x0010)==0)
                return; //Fehler: Methode ist nicht statisch
              mthdAddr=MAGIC.cast2Ref(m)+MAGIC.getCodeOff();
              MAGIC.inline(0x57); //push edi
              MAGIC.inline(0x8B, 0x7D, 0xFC); //mov edi,[ebp-4]
              MAGIC.inline(0xFF, 0x55, 0xF8); //call dword [ebp-8]
              MAGIC.inline(0x5F); //pop edi
              return; //call war erfolgreich
            }
            m=m.nextMthd; //nächste Methode in aktueller Unit versuchen
          }
          return; //Fehler: Methode nicht gefunden
        }
        u=u.nextUnit; //nächste Unit im aktuellen Package versuchen
      }
      return; //Fehler: Klasse nicht gefunden
    }
    p=p.nextPack; //nächstes Package versuchen
  }
  //Fehler: Package nicht gefunden
}
```

Bei Verwendung des raw-Generators (siehe Kapitel 10.1) ist statt der Suche in den Laufzeitobjekten eine dazu analoge Suche im `symbols.RawInfo.info`-Array durchzuführen.

13 Bootlader

Zum Verständnis oder für eigene Experimente ist hier der Bootlader für eigenständige Programme (ohne zugrundeliegendes Betriebssystem wie zum Beispiel Windows, Linux etc.) im 32 Bit Protected Mode im Quelltext mit Kommentaren abgedruckt. Zur Übersetzung eignet sich beispielsweise yasm, der unter [yasm] erhältlich ist, er ist aber im exec-Paket von SJC (siehe [sjc]) bereits übersetzt vorhanden. Eine passende bootconf.txt ist in Kapitel 2 angegeben.

```
INITSTACK   EQU   00009BFFCh ;Stack in protected mode starts here
CRCPOLY     EQU   082608EDBh ;CRC-polynom
ORG 07C00h
BITS 16
begin:
    jmp    start             ;jump over header
    times 003h-($-$$) db 0   ;first byte of header is at pos 3
    times 01Eh-($-$$) db 0   ;fill up header(27) with zero
; *** variables for primary image
PIdest:    dd 0              ;destination for image
PICDAddr:  dd 0              ;class-descriptor for pi
PIExAddr:  dd 0              ;method-offset for pi
; *** variables for bootloader
CRC32:     dd 0              ;CRC over image
RDsectors: dw 0              ;sectors to read
ScCnt:     db 18             ;sectors to read per head, init: floppy
HdMax:     db 1              ;highest usable headnumber, init: floppy
InitCX:    dw 2              ;start read with this cx, init: floppy
InitDX:    dw 0              ;start read with this dx, init: floppy
MxReadErr: db 5              ;maximum read errors
; *** prepared gdt
ALIGN 4, db 000h             ;align gdt to 4-byte-address
mygdt:
    dw 00000h, 00000h, 00000h, 00000h ;null descriptor
    dw 0FFFFh, 00000h, 09A00h, 000CFh ;4GB 32-bit code at 0x0
    dw 0FFFFh, 00000h, 09200h, 000CFh ;4GB 32-bit R/W at 0x0 (cf)
    dw 0FFFFh, 00000h, 09A06h, 0008Fh ;4GB 16-bit code at 0x60000
endgdt:
ptrgdt:                      ;use this offset for lgdt
    dw endgdt-mygdt          ;length
    dd mygdt                 ;linear physical address (segment is 0)
PrintChar: ;print character in al, destroys es
    push   si                ;save si (destroyed by BIOS)
    push   di                ;save di (destroyed by BIOS)
    push   bx                ;save bx (destroyed for color)
    mov    bl,007h           ;color 007h: gray on black
```

```
    mov    ah,00Eh           ;function 00Eh: print character
    int    10h               ;BIOS-call: graphics adapter
    pop    bx                ;restore bx
    pop    di                ;restore di
    pop    si                ;restore si
    ret                      ;return to caller
updCRC32: ;update value in ebp, update from dword in eax
    push   cx                ;save cx
    xor    ebp,eax           ;xor new dword
    mov    cx,32             ;32 bits
uC3NextBit:
    test   ebp,000000001h    ;lowest bit set?
    pushf                    ;save result
    shr    ebp,1             ;shift value
    popf                     ;restore result
    jz     uC3NoPOLY         ;lowest bit was clear -> jump
    xor    ebp,CRCPOLY       ;xor with CRCPOLY
uC3NoPOLY:
    loop   uC3NextBit        ;handle next bit if existing
    pop    cx                ;restore cx
    ret
waitKeyboard: ;wait until inputbuffer empty
    in     al,064h           ;read status register
    test   al,002h           ;inputbuffer empty?
    jnz    waitKeyboard      ;  no->retry
    ret
start:
;_________________ Initialize stack and ds _____________________________
    cli                      ;disble interrupts while setting ss:sp
    xor    ax,ax             ;helper for ss, ds, es
    mov    ss,ax             ;initialize stack
    mov    sp,07BFCh         ;highest unused byte
    mov    ds,ax             ;our address segment
;_________________ Enable A20 gate ____________________________________
    mov    al,'A'            ;character to print
    call   PrintChar         ;say "A20"
    call   waitKeyboard
    mov    al,0D1h           ;command: write output
    out    064h,al           ;write command to command register
    call   waitKeyboard
    mov    al,0DFh           ;everything to normal (A20 then enabled)
    out    060h,al           ;write new value
```

```
    call   waitKeyboard
;_________________ Switch to protected and back to real mode _________
;now to protected mode (interrupts still cleared!)
    lgdt   [ptrgdt]          ;load gdt (pointer to six-byte-mem-loc)
    mov    eax,cr0           ;read machine-status
    or     al,1              ;set bit: protected mode enabled
    mov    cr0,eax           ;write machine-status
    mov    dx,010h           ;helper for segment registers
    mov    fs,dx             ;prepare fs for big flat segment model
    dec    ax                ;clear lowest bit: protected mode disabled
    mov    cr0,eax           ;write machine-status
;now back with large segment-limits
    sti
;_________________ Load block from Disk ___________________________
    xor    ebp,ebp           ;helper for ds/fs
    mov    ds,bp             ;our address segment
    mov    fs,bp             ;destination with flat segment
    mov    al,'L'            ;character to print
    call   PrintChar         ;say "loading"
    mov    cx,[InitCX]       ;first sector / first cylinder
    mov    dx,[InitDX]       ;drive / first head
    mov    edi,[PIdest]      ;linear destiniation address
    dec    ebp               ;initialize internal CRC to 0FFFFFFFFh
    mov    si,[RDsectors]    ;get sectors to read
Readloop:
    mov    bx,01000h         ;helper for es
    mov    es,bx             ;destination segment
    xor    bx,bx             ;destination is 1000:0000
    mov    ax,00201h         ;function 002h par 001h: read 1 sector
    push   dx                ;some BIOS-versions destroy dx
    stc                      ;some BIOS-versions don't set CF correctly
    int    013h              ;BIOS-call: DISK
    sti                      ;some BIOS-version don't restore IF
    pop    dx                ;restore dx
    jc     ReadError         ;CF set on error
    push   cx                ;save cx (cyl/sec)
    mov    cx,128            ;each sector has 128 dwords
Copyloop:
    mov    eax,[es:bx]       ;load byte
    mov    [fs:edi],eax      ;store byte
    call   updCRC32          ;update CRC for this dword
    add    bx,4              ;increase source address
```

```
    add     edi,byte 4          ;increase destination address
    loop    Copyloop            ;decrease cx and jump if some bytes left
    pop     cx                  ;restore cx (cyl/sec)
    dec     si                  ;decrease sectors to read
    jz      ReadComplete        ;nothing left -> continue
    test    si,0001Fh           ;at least one bit in 4..0 set?
    jnz     NoDots              ;  yes -> don't print dot
    mov     al,'.'              ;character to print
ReadCont:
    call    PrintChar           ;say "read OK"
NoDots:
    mov     al,cl               ;save sector + highest cylinder for calc
    and     al,03Fh             ;extract sector
    cmp     al,[ScCnt]          ;maximum reached?
    je      NextHead            ;  yes -> increase head
    inc     cl                  ;increase sector
    jmp     Readloop            ;read next sector with this head
NextHead:
    and     cl,0C0h             ;extract highest cylinder
    or      cl,1                ;set sector to 1
    cmp     dh,[HdMax]          ;already at end of cylinder?
    je      NextCyl             ;  yes -> reset head and increase cylinder
    inc     dh                  ;next head
    jmp     Readloop            ;read next side in this cylinder
NextCyl:
    xor     dh,dh               ;reset head
    inc     ch                  ;increase cylinder
    jnz     Readloop            ;no overflow, read this cylinder
    add     cl,040h             ;increase highest two bits of cylinder
    jmp     Readloop            ;read next cylinder
ReadError:
    dec     byte [MxReadErr]    ;try to decrease maximum of read errors left
    jz      NRL                 ;  nothing left -> loop forever
    mov     al,'e'              ;character to print
    call    PrintChar           ;say "read error"
    xor     ah,ah               ;function 000h: reset
    int     13h                 ;BIOS-call: DISK
    jmp     Readloop            ;try again
ReadComplete:
    mov     dx,03F2h            ;port adress of disk controler
    in      al,dx               ;get status register of disk controler
    and     al,0EFh             ;switch of motor
```

```
    out    dx,al              ;set status register
    cli                       ;disable interrupts
    cmp    ebp,[CRC32]        ;checksum correct?
    je     CallJava           ;  yes -> continue
    mov    al,"c"             ;character to print
    call   PrintChar          ;say "checksum error"
NRL:
    jmp    NRL                ;stop machine
;_________________ Switch to protected mode and call java ____________
CallJava:
    mov    al,'P'             ;character to print
    call   PrintChar          ;say "Protected Mode"
    lgdt   [ptrgdt]           ;load gdt (pointer to six-byte-mem-loc)
    mov    eax,cr0            ;read machine-status
    or     al,1               ;set bit: protected mode enabled
    mov    cr0,eax            ;write machine-status
    db 0EAh                   ;jump to 32-Bit Code
    dw doit                   ;  offset (linear physical address)
    dw 008h                   ;  selector
BITS 32
;_________________ Initialise segments _________________________________
doit:
    mov    dx,010h            ;helper for data-segment
    mov    ds,dx              ;load data-segment into ds
    mov    es,dx              ;load data-segment into es
    mov    fs,dx              ;load data-segment into fs
    mov    gs,dx              ;load data-segment into gs
    mov    ss,dx              ;load data-segment into ss
    mov    esp,INITSTACK      ;set stackpointer
    mov    edi,[PICDAddr]     ;load address of class descriptor
    mov    eax,[PIExAddr]     ;load method address
    call   eax                ;call java-method
Ready:
    jmp    Ready              ;endless loop
BITS 16
    times 01FEh-($-$$) db 0   ;fill with zero until BIOS-mark of sector
    db 055h, 0AAh             ;BIOS-mark at 510: valid sector for booting
end
```

14 Referenzen

[java] http://java.sun.com

[oberon] http://www.oberon.ethz.ch

[nat] http://www-vs.informatik.uni-ulm.de/dept/staff/frenz/private/native.html

[picos] http://www-vs.informatik.uni-ulm.de/dept/staff/frenz/private/picos.html

[plurix] http://www.plurix.de

[qemu] http://fabrice.bellard.free.fr/qemu/ und http://www.h7.dion.ne.jp/~qemu-win/

[rawrite] http://www.chrysocome.net/rawwrite

[sjc] http://www-vs.informatik.uni-ulm.de/dept/staff/frenz/private/compiler.html

[yasm] http://www.tortall.net/projects/yasm/