SJC – Small Java Compiler

User Manual

Stefan Frenz
Translation: Jonathan Brase

original version: 2011/06/08
latest document update: 2011/08/24
Compiler Version: 182

Table of Contents

1 Quick introduction

For a quick introduction, download the executable package (see [sjc] in the References section), which contains all the necessary files for the development of 32 and 64-bit operating systems:

File	Description
b64_dsk.bin	Floppy bootloader for 64-bit target systems
bootconf.txt	Configuration file for the boot image writer
bts_dsk.bin	Floppy bootloader for 32-bit target systems
compile	Compiler executable for Linux
compile.exe	Compiler executable for Windows
hello.java	Example "Hello World" program
rte.java	Basic runtime environment framework

No programs outside this package are necessary for the compilation of an operating system. However, testing the compiled system requires a PC emulator such as QEmu (see [qemu]). This introduction assumes that a QEmu installation is available.

The next four sections of this chapter demonstrate the invocation of the compiler under Windows, Linux, and a JVM, as well as booting the resulting system in QEmu. These instructions will compile the included "Hello World" program (discussed in chapter 2) for a 32-bit target system.

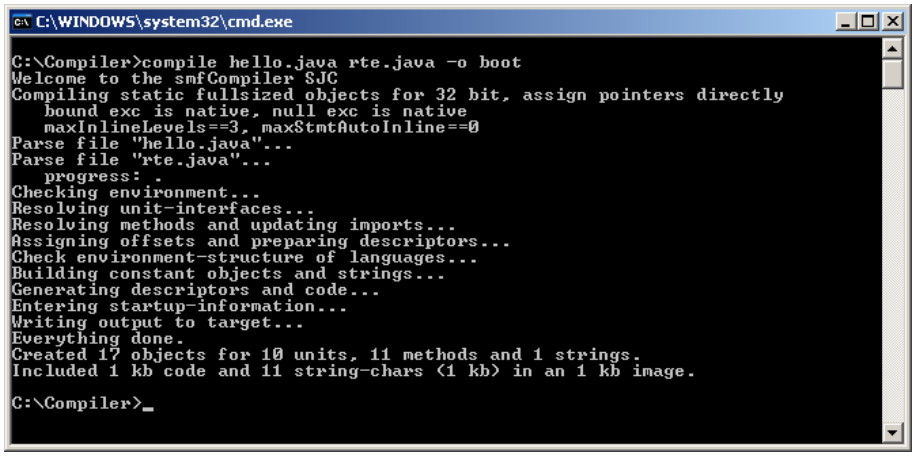
In developing a new system with the compiler, it is reccomended that the user read at least chapters 2, 3.4, 3.8, 5.1, 6.5 and 7. Examples can be found in chapter 12. Furthermore, an example system supporting 32-bit protected mode and 64-bit long mode can be found in the reference section under [picos].

1.1 Windows

If the executable package has been unpacked into C:\Compiler, the installation can be tested with this command line:

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

This should produce approximately the following output (shown here with a few extra status messages):



The above command line will also produce the files BOOT_FLP.IMG and syminfo.txt. The first contains a floppy image of the example program, the second contains symbol information, described in detail in chapter 5.2.

1.2 Linux

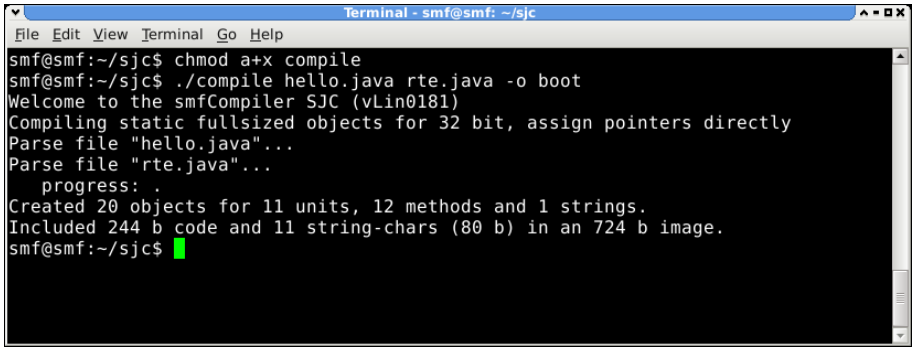
If the executable package has been unpacked into ~/compiler, it may be necessary to set the executable flag:

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

The installation can then be tested with the following command line:

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

This should produce approximately the following output:



As under Windows (see the previous section), the files BOOT_FLP.IMG (a floppy image) and syminfo.txt (containing symbol information) will be produced.

1.3 JVM Environment

In addition to the Windows and Linux versions, there is also, since version 181, a precompiled JAR version of the compiler. Instead of the native compilers, the JVM is executed with sjc.jar as a parameter. All other parameters, as well as the output, correspond to the Linux and Windows versions:

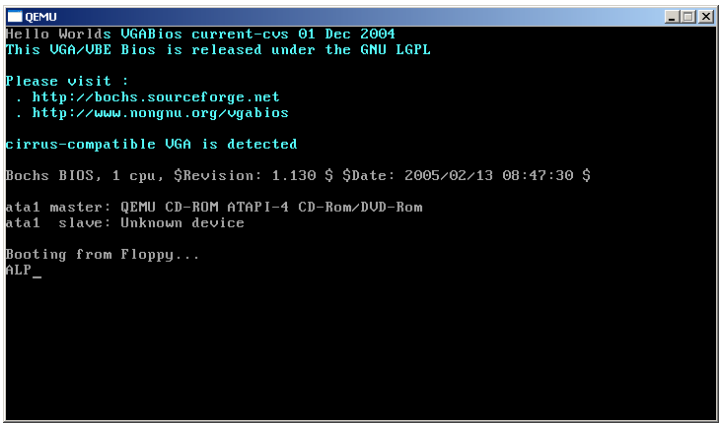
```
C:\Compiler>compile hello.java rte.java -o boot
```

1.4 QEmu

The following command lines can be used to boot the compiled floppy image in QEmu:

```
"C:\Compiler>\Program Files\QEmu\qemu.exe" -m 32 -boot a -fda BOOT_FLP.IMG
smf@linplu:~/compiler> qemu -m 32 -boot a -fda BOOT_FLP.IMG
```

These should (depending on the versions of QEmu and its BIOS) result in the following output:



If the system starts, but hangs without any error message, the cause could be a faulty KVM module. This can be deactivated with **-no-kvm**, the resulting command line would then be:

```
> qemu -no-kvm -m 32 -boot a -fda BOOT_FLP.IMG
```

2 Hello World

This chapter will introduce the source code for a simple "Hello World" program, and the command lines to compile it (all necessary files are in the executable package, see [sjc]), and to boot it in QEmu. Afterwards, the individual steps in this process and the default options assumed by the compiler will be explained.

2.1 Source Code

Program (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);
  }
}
```

Runtime Environment (rte.java)

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

```
  @SJC.Inline
  public char charAt(int i) {
    return value[i];
  }
}
package rte;
public class SArray {
  public final int length=0, _r_dim=0, _r_stdType=0;
  public final Object _r_unitType=null;
}
package rte;
public class SClassDesc {
  public SClassDesc parent;
  public SIntfMap implementations;
}
package rte;
public class SIntfDesc { /*optional: public SIntfDesc[] parents;*/ }
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, Object unitType) { while(true); }
  public static void newMultArray(SArray[] parent, int curLevel,
      int destLevel, int length, int arrDim, int entrySize, int stdType,
      Object unitType) { 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 isInstance(Object o, SClassDesc dest,
      Object unitType, int arrDim, boolean asCast) { while(true); }
  public static void checkArrayStore(SArray dest, Object newEntry) {
      while (true); }
}
```

Bootloader configuration (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 Compilation and Emulation

If the files designated in the last subchapter have been placed in the same directory, along with a compiler executable appropriate to the platform and the file bts_dsk.bin (required to setup the bootsector), the "Hello World" program can be compiled with:

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

bts_dsk.bin is contained in the executable package (see [sjc]) or can be assembled with yasm (see [yasm]) from the source code found in chapter 13. The output of the compiler should look something like this (for a screenshot see chapter 1):

```
Welcome to the smfCompiler SJC (v178)
Compiling static fullsized objects for 32-bit, assign pointers directly
Parse file "rte.java"...
   progress: .
Parse file "hello.java"...
Created 20 objects for 11 units, 12 methods and 1 strings.
Included 244 b code and 11 string-chars (80 b) in an 724 b image.
```

A successful compilation will create the file BOOT_FLP.IMG, which can either be written to a floppy (on Windows this can be accomplished with programs such as rawrite (see [rawrite]), on Linux it can be done with dd), or can be used directly with an emulator such as QEmu. If QEmu (see [qemu]) has been properly installed to C:\Program Files\QEmu, the floppy image can be booted with:

```
"C:\Program Files\Qemu\qemu.exe" -m 32 -boot a -fda BOOT_FLP.IMG
```

A screenshot can be found in chapter 1. Under some circumstances, QEmu's KVM module may be defective. The module can be deactivated with the option `-no-kvm`.

2.3 Details

Program

The static variable **vidMem** is created by the compiler as a class variable. Its value in the disk image is initialized to 0xB8000, the address of the beginning of video memory for VGA compatible graphics cards.

The method **main()** is called by the code in the bootsector and is the entry point to the Java code. It calls a method to print a string, and then puts the processor into an infinite loop.

String output is implemented in the method **print(String str)**, which, for each character in the given string, calls a character output method.

The method **print(char c)** copies the lower 8 bits of the given character, with the help of **MAGIC.wMem8(.)**, to the address given by the current value of vidMem. The graphics card then, in text mode, interprets the copied value as an ASCII character. The post-increment of **vidMem** causes it to point to the next byte, which determines the color of the character that has just been written. A second use of **MAGIC.wMem8(.)** sets the color byte to 0x07, which signifies "gray on black", and **vidMem** is then post-incremented to point to the next character in video memory.

Runtime Environment

The root of the type hierarchy is the class **java.lang.Object**, which requires the instance variables **_r_type**, **_r_next**, **_r_relocEntries** and **_r_scalarSize** (in this order). However, the given "Hello World" program does not use **_r_next**, **_r_relocEntries** und **_r_scalarSize** so these can be omitted by means of the compiler option **-l**.

The implementation of String for this example conforms to the Java standard, so the array and length variables cannot be directly accessed. The array type used, **char[]** also conforms to the Java standard, but in this example no use of Unicode characters is made and thus a **byte[]** implementation (compiler option **-y**) would be sufficient.

The **@SJC.Inline** annotation is used to mark the methods **length()** and **charAt(int)** so that the compiler tries to insert the body of the method in question directly into the calling method, instead of making a subroutine call (See section 7.2).

The runtime environment must provide those routines which chapter 6 specifies as required. The compiler has been written to require these, for the sake of simplicity and speed, even when only a few of them are actually used, as in this example.

Bootloader configuration

- The **floppy32** section starts at the declaration of its name and ends with the keyword **endsection**.
- The section is given the alias **default**, and the compiler looks for this alias if no other section is specified on the command line.
- The name of the file to be output is **BOOT_FLP.IMG**. If necessary, a path to the file can be designated (With **** as a delimiter on Windows or **/** on Linux).
- The size of a block is 512 bytes, i.e, one sector.
- The maximum size of the memory image is specified as 1,474,048 bytes. This limit corresponds to the 2880 sectors on a floppy, minus the boot sector.
- The bootsector is to be read from **bts_dsk.bin**.
- The offsets 30, 34, 38, 42 and 46 contain various addresses, a checksum, and a block count.

- The bootsector is to be written to the output file, followed by the memory image.

Compilation

The compiler accepts options and filenames in arbitrary order, although parameters associated with specific options must remain adjacent to the corresponding option (such as **`-o boot`** in the example above). Rather than individual filenames, the name of a directory may be designated, in which case the compiler will attempt to use all files in this directory and its subdirectories, recursively, as input. If only the files directly in the specified directory are to be compiled, and its subdirectories should not be recursively searched, **`:`** should be appended to the directory name.

If no other options have been given, the compiler assumes **`-t ia32`** (ia32 architecture), **`-a 1m`** (memory image begins at the 1 MB mark) and **`-s 128k`** (maximum size of the memory image is 128 KB).

Without the parameter **`-o boot`** the compiler places the memory image unaltered in the file raw_out.bin. In the above example the bootsector from bts_dsk.bin is added to this, and, once information required for initialization has been added, the floppy image is written to the file BOOT_FLP.img

As a final step, the symbol information is written as text to the file syminfo.txt (see chapter 5.2). The data output on the console summarizes the results of the process:

```
Created 20 objects for 11 units, 12 methods and 1 strings.
Included 244 b code and 11 string-chars (80 b) in an 724 b image.
```

3 Introduction

3.1 Origins, Legal Matters, Resources, and Acknowledgements

The compiler described here was developed in the programmer's free time and no claims are made as to its completeness, freedom from defects, or fitness for a particular purpose. In particular, the author is not responsible for damage or loss of profit resulting from use of the compiler. This manual is not an introduction to programming in Java or to system programming, but rather is intended as an introduction to the use of the Small Java Compiler for readers already experienced in these fields. The source code, precompiled executables, this manual, as well as a brief overview of the publicly available versions can be found at the web address given under [sjc] in the reference section at the end of this manual.

I wish to thank Prof. Dr. Peter Schulthess and Prof. Dr. Michael Schöttner, who assisted in developing the preliminary ideas for this compiler in numerous discussions. I also wish to thank Dipl. Inf. Patrick Schmidt, who wrote the emulator for the pseudo-SSA instruction set developed for the compiler, as well as tests for native Windows programs.

3.2 Basic Principles and Objectives

The Small Java Compiler generates native machine code, which is then put into a memory image (see chapter 3.3) for the target system. The compiler generates all necessary structures and symbol information, so no external linker or loader is necessary. The integrated generation of the output file allows both the compilation of native operating systems as well as programs runable under Windows or Linux (see chapter 11). During development, the following goals were central:

- To be largely source-code compatible with SunJava: Source-code compatibility with SunJava (see [java]) allows the reuse of free development tools, which makes development and testing of programs easier. Submodules can be compiled with SunJava and be tested independently from SJC and the target platform.
- Support for diverse target platforms: Support for various platforms with 8, 16, 32, and 64-bit CPU's is visible in the universal interface between front end and back end (see sections 3.6 and 3.7).
- Support for low-level programming: In order to build a full system and do type-safe programming, selected modules of the target system must be able to perform direct hardware access. The compiler provides all necessary routines and extensions for this (see chapter 7).
- Optional optimization of machine code: The generation of native machine code (see section 3.6) allows optimization on the level of individual methods. Instruction generation is done directly by the back end, allowing targeted optimizations to be made for each architecture.
- Ability to self-compile: The compiler can compile its own source code on all architectures for which a complete implementation of the back end is available, without the need for a running system (see section 3.7).
- Understandable subtasks and modular construction: To ensure extendability and ease debugging, the process of compilation is divided into understandable and traceable subtasks. For this purpose, individual packages never depend on subpackages of other packages. The division between front end and back end with a clearly defined interface makes extensions and changes to individual modules independent of the rest of the compiler (see section 3.6).

3.3 Memory Image Format

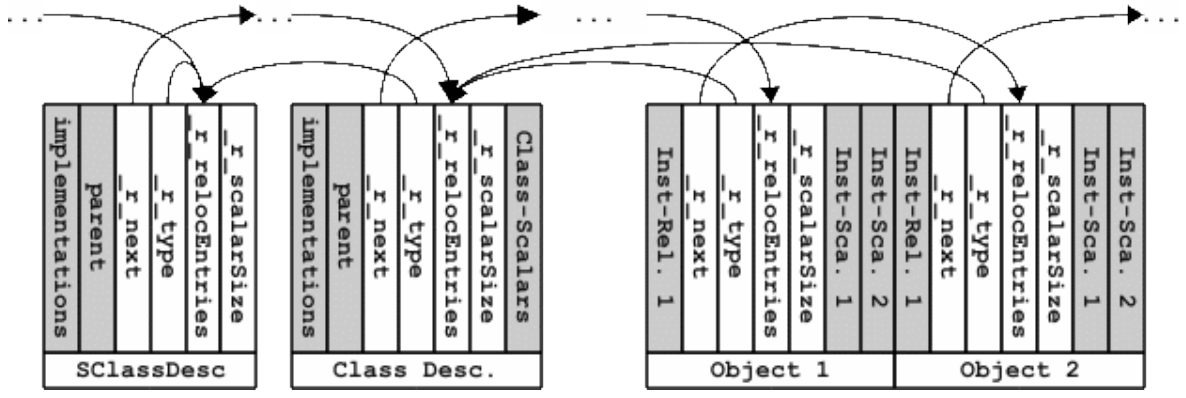
If, as usual, the compiler is not in bootstrap mode (activated with option **`-s 0`**, see section 5.1), it generates the memory image of the target system as follows. The memory image begins with the following header (this can be deactivated with the option **`-P`** or can be replaced with a custom header, see section 5.1):

Offset	Type	Contents
0	int	Base address of the memory image (identical to **`MAGIC.imageBase`**)
4	int	Size of the memory image
8	Pointer	Class descriptor for the class **`kernel.Kernel`**
12	Pointer	First opcode in the method **`kernel.Kernel.main()`**
16	Pointer	First object in the heap
20	Pointer	Memory block for RAM initialization (identical to **`MAGIC.getRamAddr()`**)
24	int	Code offset in methods (Identical to **`MAGIC.codeOffset()`**)
28	int	Architecture parameter

Offset 4 contains the size of the memory image. When added to the base address of the memory image, this gives the address of the first free memory location following the image. The class descriptor for **`kernel.Kernel`** is needed to setup the class context prior to starting the method indicated by the next pointer (at offset 12): **`kernel.Kernel.main()`**, which is the entry point to the Java code. If the memory image is to be searched through as a linked list of objects, the pointer at offset 16 can be used as an entry point. Further objects can be chained together with the **`_r_next`** field of each object, so long as the **`-l`** option was not used in compilation, see section 5.1). The value at offset 28 contains the 32-bit value 0x550000AA logically ORed with the pointer size (bits 8-15) and the stack alignment (bits 16-23). From offset 32 on, the image contains only typed objects, a pointer to the first of these is found, as shown in the table above, at offset 16.

3.4 Object Format

If the **`-M`** option is not given, the compiler generates a memory image containing only typed objects. The pointer to an object points to its first scalar field. Further scalars are placed at higher addresses, the references of the object are placed at lower addresses, with the first pointer pointing to the type of the object. If the option **`-l`** was not used in compilation, two objects of the same type, allocated one after the other, with one class variable of type **`int`** and, for instance variables, one pointer and two **`int`** variables, they will look something like this in the heap of a 32-bit system:



There are usually only pointers to objects, and no pointers to an object's fields.

If, however, the option **`-M`** was used in compilation, the scalars of an object are addressed indirectly. Thus there are two further fields in an object: **`_r_indirScalarAddr`** and **`_r_indirScalarSize`**,

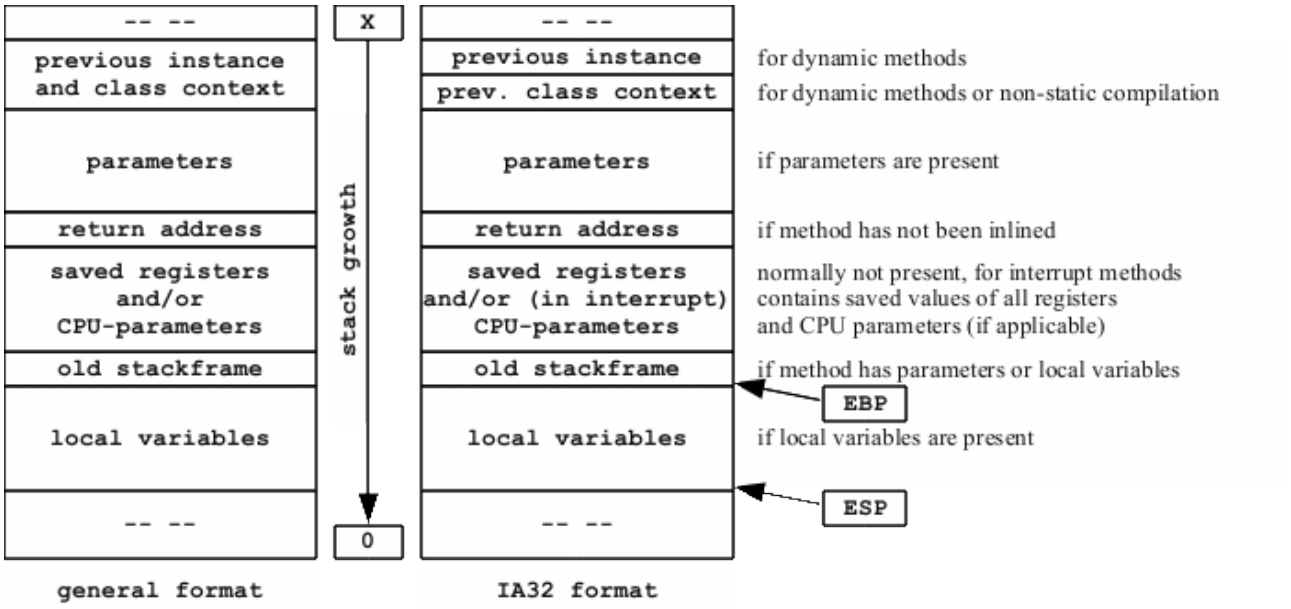
which give, respectively, the base address of the the memory block used to hold the object's scalars and the size of that block. This option is intended for use in a cluster, where, on the one hand, the references of an object, as well as information relevant to the heap structure, need to be kept under a strict consistency model, but, on the other hand, the scalars of an object may under some circumstances need a different underlying consistency model.

This manual will assume, unless otherwise specified, that the **`-M`** option has not been used. An overview of the differences can be found in chapter 9.

3.5 Stack Frame Format

The compiler normally reads in parameters from left to right (with the exception of natively declared C functions) and puts them on the stack going from higher to lower addresses. The minimum size of a stack entry is determined by target architecture: for example, a byte entry on an ATmega system uses one byte, on a IA32 system it uses four bytes.

Different values will be placed on the stack, depending on compilation mode and the properties and requirements of the methods that are called, and this must be allowed for in traversing the stack frame or when accessing the stack with inline assembly code. A stack frame with all possible elements looks like the following:



The correct stack format fundamentally depends on the target architecture. This can produce different apportionment of the stack depending on architecture, so the notes and register use labels given on the right side of the diagram are valid only for the pictured IA32 stack format In particular, the old stack frame pointer is not saved on the IA32 Architecture if the method has neither parameters nor local variables. This optimization can be deactivated with the IA32-specific option **`-T nsop`**. If this option is given the pointer to the previous stack frame will always be saved.

3.6 Compiler Structure

The compiler is divided into separate modules, reflected in the division of its packages. With the exception of very small packages, for example **`memory`** and **`output`**, the interface to a package is in the root package, and its implementations are in the subpackages. Subpackages only import packages in their own package and in the root package, so that individual subpackages are completely independent of one another.

Module	Description
backend	Outputs code for the target architecture

Module	Description
compbase	General settings and base types for the compiler
compression	Performs optional compression of the memory image
debug	Writes debug information
emulation	Emulator (for the moment only for SSA)
frontend	Processes source files
memory	Manages creation of the memory image
osio	Accesses I/O functions of the host operating system
output	Converts the memory image into the desired output format
real	Configurable support for floating point numbers
relations	Management of relationships in methods
symbols	Optional output of symbol information
ui	User interface

Source files are compiled in the following twelve steps, with each step only being started when the previous step has been sucessfully completed:

1. User interface and file access instances appropriate to the host operating system are started, parameters are read in from the command line or from environment variables. All following phases are independent of the host operating system.
2. The parameters for the current compilation run are checked and stored in the current context. The submodule for the chosen target architecture and the memory image manager are instantiated and initialized, setting up the code generator and parameters like pointer size and stack alignment.
3. The designated source files are given to the front end, where, after the appropriate language module has been determined, it's parser is used to convert them into an undecorated parse tree. Although the compiler was primarily developed for Java or similar languages, it can, in principle, be used to compile various languages into a consolidated system.
4. The parse tree is checked, designated references are resolved, and types are checked. Basic information such as what public variables and methods the classes to be generated have is determined and recorded in the parse tree in this step.
5. Method blocks are checked for errors, references resolved, and information necessary for code generation is recorded in the parse tree.
6. The offsets of variables and methods are calculated, based on the chosen memory model and the parameters used. After this step, all information necessary for code generation has been determined.
7. The runtime system is checked for validity, in particular, the existence of the runtime routines to be called is ascertained.
8. All objects, in particular class descriptors and code blocks, are generated and deposited in the memory image.
9. Optionally, the symbol information generated by the compiler can be integrated into the memory image.
10. The memory image can also be compressed or encrypted, followed by another compilation run to generate a decompressor or decoder.

11. The memory image in then further processed according to the chosen output format, for example, as an Intel HEX file for use in creating a ROM, or as a binary file combined with a bootloader for direct execution.
12. The debug symbol information is written (See section 5.2).

If all of the above steps have sucessfully completed, then all output files are valid.

3.7 Implemented Modules

The following table gives a overview of the currently implemented modules of the compiler, with brief descriptions: In any given compilation, all submodules (implementations) from **`front end`**, as well as exactly one **`back end`** and one **`output`** submodule are active (symbol information and compression are optional).

Module	Implementation	Description
`frontend`	`binimp.*`	Imports binary files (see section 4.1
	`clist.*`	Source file lists (see section 4.2)
	`sjava.*`	Java source files
`backend`	`x86.AMD64`	64-bit code for AMD64 processors in long mode
	`x86.IA32`	32-bit code for IA32 processors in protected mode
	`x86.IA32Opti`	Optimized 32-bit code for IA32 in protected mode
	`x86.IA32RM`	32-bit code for IA32 in real mode
	`ssa.*`	Pseudo SSA code
	`atmel.ATmega`	8 bit code for Atmel ATmega microcontrollers
	`atmel.ATmegaOpti`	Optimized 8 bit code for Atmel ATmega
	`arm.ARM7`	32-bit code for ARM7-TDMI Microcontrollers
`memory`	`*`	Memory image manager
`compression`	`BZL`	BWT/LZW compression
	`LZW`	LZW compression
`osio`	`*`	Access to the host system (files, display, etc.)
`output`	`BootOut`	Parametrized output
	`RawOut`	Unprocessed output
`real`	`EmulReal`	Floating point emulation using integer arithmetic
	`NativeReal`	Use of native floating point support by the compiler
`symbols`	`RawSymbols`	Symbol information as a byte stream
	`RTESymbols`	Symbol information in the runtime system
`debug`	`GccInfo`	Debug information as gcc compatible assembly files
	`MthdInfo`	Debug information as lists for all generated methods
	`SymInfo`	Standard debug information with all textual information

The interface defined for each module can be found in the root package with the same name, the choice between the individual implementations takes place in an "Admin" or "Factory" in the same root package.

3.8 Differences from Sun Java

Although broad source-code compatibility between SJC and Sun Java has been achieved, some features of SunJava are not available, or else have been intentionally omitted in order to indicate to the system programmer that certain constructs are error-prone when programming close to the hardware. The primary differences are:

- The compiler has supported Java exceptions since version 165 (not to be confused with CPU exceptions, which are supported in the same way as hardware and software interrupts). The use of the keywords **`try`**, **`catch`**, **`finally`**, **`throw`** and **`throws`** imposes extra requirements on the runtime system (see section 6.6).
- The compiler has supported interface arrays since version 177, although these must be type checked repeatedly at run time, as well as when objects are read from them (equivalent to a simple **`Object`** array with an interface cast on every access to an array element).
- The compiler supports neither enumerations (which can easily be converted into classes "by hand") nor generic types (no run time advantages can be gained by Sun's chosen method of Type Erasure other than explicit conversion).
- Static class initialization must be triggered by calling **`MAGIC.doStaticInit()`** (see section 7.1).
- The SunJava runtime library is not a component of the SJC. This allows targeted runtime environments with specialized functionality to be built for any system, for example, embedded processors, Windows, Linux, or distributed operating systems like Plurix.
- There is no reflection API. With the help of the generated symbol information (see section 5.2) the type of every object can be determined via the memory address of its class descriptor. Through the analysis of in-image symbol information (see Chapter 10) similar functionality can be provided.
- With the use of the **`-x`** option (see chapter 5.1, available since version 175) implicit conversion of numerical types in expressions will not be carried out, in order to avoid unintended behavior, especially for devices prone to sign extension errors and their drivers.
- All designated files (or all files in the designated directories) will always be compiled.
- Multiple classes may be declared in a file, so long as they do not depend on each other.

4 Front-end Modules for non-Java Files

4.1 Importing Binary Files

Files with the extensions **.bib** and **.bim** are imported by the binary import module as a binary byte array or as executable code. Access to files imported in this way can be achieved by means of their filename (minus the extension) for byte arrays through the automatically allocated variable

```
binimp.ByteData.DATEINAME
```

and through

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

for code blocks.

An alternative to the import of byte arrays is the function

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

discussed in sections 4.2 and 7.1.

4.2 Source-File Lists

The compiler supports the designation of source files to be compiled on the command line, and also through files with the extension **.clt**. Files or directories to be compiled may be specified in **clt** files, with the entirety of each line being used as a file or directory name. Any special characters or whitespace will be interpreted literally and not left out.

Source files and directories are to be designated individually as on the command line (see section 5.1). Directories are normally processed recursively, and all files found with extensions known to the compiler are compiled. If a directory is not to be processed recursively, a colon **:** can be used on the end of the directory name.

All directly designated names (files or directories) must exist, otherwise the compilation will be aborted. When a directory is designated, any files with extensions unrecognized by the compiler or that contain errors will be ignored. If the compiler is run without parameters, a list of recognized filename extensions will be output under the heading **Possible input types**.

Morover, files designated by their full filename (including the extension and an optional path) through **MAGIC.getNamedString(.)** will be imported as constant Strings. Files referenced in this way can have any extension and can be stored at any location in the file system or naming service that can be reached at compile time.

5 Compiler Functions

5.1 Compile Options

All files and directories to be compiled can be designated either through a compile-list file (see section 4.2), or can be designated with identical syntax on the command line.

The following table lists the currently available options, also listed in the usage directive that is given when the compiler is called without parameters.

Option	Description
`-i FILE`	"get options from FILE" Reads in command line options from a file.
`-V`	"verbose timing information" Outputs hexidecimal timing information on the compilation process, either in nanoseconds or CPU ticks, depending on compiler variant.
`-v`	"verbose mode" Output notifications and warnings, for example, about uncompiled methods, automatically set inline flags, etc.
`-L`	"flat runtime environment" This option implies **`-l`**. In addition, objects in the runtime environment (for instance, method blocks) are generated without object information, as a result, the memory image no longer consists exclusively of objects.
`-l`	"streamline objects" The variables **`_r_relocEntries`**, **`_r_scalarSize`** and **`_r_next`** are removed. Cannot be combined with **`-m`** or **`-M`**.
`-w`	"use alternate Object and newInstance" Use an alternative structure for **`java.lang.Object`** and the shortened **`rte.DynamicRuntime.newInstance`** adapted for it.
`-m`	"generate movable code and descriptors" Where possible, instead of static addressing of class descriptors, code, and variables, carry out all accesses exclusively through the references stored in the current context, so that all objects can be moved without changes to the code. Cannot be combined with **`-l`** or **`-L`**.
`-M`	"generate movable scalars" The **`-m`** option is implied. Additionally, scalars are indirectly addressed, in order to allow the memory block holding an objects scalars to be relocated. (See sections 3.4 and 9.
`-h FILE`	"output file-listing of compiled files" Write the names of all compiled files to FILE.
`-h`	"call for heap pointer assign" Make all allocations for references in the heap through a runtime routine, instead of doing so directly. Make stack allocations directly.
`-c`	"call for every pointer assign" Make all allocations for references through a runtime routine, instead of doing so directly.

Option	Description
-C	"skip array store checks" Don't generate code to check pointer allocations in an array. As a result, illegal array elements are no longer detected and type safety can no longer be guaranteed.
-B	"skip array bound checks" Don't generate code to check array bounds. As a result, invalid addressing is no longer detected and type safety cannot be guaranteed.
-b	"generate runtime call on bound exception" Call a runtime routine on out of bounds array accesses instead of using native handling (for example with an exception). This option is required for the detection of invalid addressing on machines without exceptions or interrupts.
-n	"generate runtime call on nullpointer use" Test for null pointer dereferencing with a call to a runtime routine on errors. This option is required for the detection of null pointer dereferences on machines without MMU support.
-x	"generate stack limit check for methods" Call a runtime routine in case of possible overrun of the stack limit pointer for all methods (individual methods can be checked with **@SJC.StackExtreme** (see chapter 7). See also section 6.8.
-X	"force explicit conversion for base types" Do not implicitly convert base types, in particular do not convert from **byte** and **short** to **int**. When **-X** is used, all base type conversions must be made explicitly.
-r	"generate assert checks (enable globally)" Activate assert code globally (instead of for individual methods or units through **@SJC.CheckAssert**). See also section 6.9.
-R	"do not encode rte-calls for synchronized" Deactivate synchronized code.
-N PK.U. MT	"use method MT in unit PK.U as entry method" Instead of the usual **kernel.Kernel.main**, use the method given as a parameter to -N as the entry method.
-g	"generate all unit-descriptors (don't skip)" Generate unit descriptors, even if not needed. This is required, for example, for debugging or for TextualCall (see chapter 10).
-G	"generate all mthd-code-bodies (don't skip)" Generate code even for uncalled methods. This is, for example, helpful for TextualCall (see chapter 10), should the needed method not be marked with **@SJC.GenCode** (see chapter 7).
-j	"generate array with parents for interfaces" All interface descriptors will contain an array with the parent interfaces of the interface in question. This is only necessary when interface arrays are to be converted in a controlled fashion.
-d	"create binary debug output for each method" Write the code for every compiled method to a binary file (one per method).
-D DW FILE	"debug writer DW with FILE (sym is default)" The standard SymWriter will not be used as the debug info writer, rather the designated writer DW with the target file FILE. Multiple writers can be activated with multiple applications of the **-D** option.

Option	Description
-q	"enable relation manager to watch relations" Activates the logging of class, method, and variable relationships, which can be evaluated by an applicable debug info writer (option **-D**) or in image symbol generator (option **-u**).
-Q	"enable global source line hint integrating" Globally activates the logging of source line numbers with reference to the generated code. Local, per method, activation can be done with **@SJC.SourceLines** (see chapter 7). This data can be evaluated with an appropriate DebugWriter (option **-D**) or SymbolInformer (option **-U**).
-W	"print all method's instructions if possible" If the target architecture supports this, the generated code for every method will be output (as assembly code).
-F	"insert calls to runtime-profiler at each method" At the entry and exit points to each method a call to the runtime environment will be made, if it is not desired to mark every method with **@SJC.Profile** (see chapter 7).
-f	"do not generate code for throw-frames (no catch)" Remove code needed for Java conformant execution of try-catch-finally blocks (see section 6.6).
-Y	"prefer native float/double usage inside compiler" Use the NativeReal module instead of the EmulReal module when the architecture selection has not automatically chosen a specific floating point module.
-y	"use byte-value instead of char-value in String" Use 8 bit ASCII instead of the usual 16 bit Unicode encoding in Strings.
-K MASK	"align blocks of allocated objects with mask MASK" Align memory blocks with different content to addresses with the bits specified by MASK set to zero.
-P FILE	"special (or no) header instead of standard header" The usual 32 byte header at the start of the memory image can be left out with **-P none** or can be replaced with an arbitrary binary file **FILE** with **-P FILE**.
-p PREF	"naming prefix for all debug files to be written" Filename prefix for debug information, such as code generated for methods or symbol information. A path may be used as part of the prefix.
-I LEVL	"specify maximum level of method inlining" Maximum recursion depth for method inlining. The default value is 3.
-S AMNT	"maximum statement amount for auto-inline" Maximum number of statements a method can contain if it is to be automatically inlined. Default value is zero.
-s SIZE **-s 0**	"specify maximum size of memory image" "do not use image, allocate in bootstrap mode" Size of the memory image during compilation (may not be smaller than the final size of the memory image, memory not needed will be left out of the final memory image). **SIZE==0** is a special case: an isolated memory image is not used, rather objects are directly allocated in the current environment (only possible for compilation in a running system compiled with SJC).

Option	Description
-e ADDR	"embedded mode, set static vars' RAM address" Relocate memory accesses to class variables to another memory block, typically in order to place code and class descriptors in ROM on the one hand and variables in RAM on the other. Initialization of variables is supported through a initialization image stored in ROM.
-E	"constant objects through RAM" Access constant objects (Strings and initialized arrays) through RAM, may only be used with **-e**.
-a ADDR	"specify maximum size of memory image" Base address of the memory image. May not be used in combination with "-s 0".
-Z GB	"relocate image address by GB gb" Used to relocate the memory image for 64-bit architectures.
-t ARCH	"specify target architecture" Specify the target platform. The supported architectures are listed in the compiler's usage directive (shown by running the compiler without parameters) under "Possible architectures".
-T APRM	"parameter for architecture" Specify a parameter for the chosen target platform. The supported parameters are listed in the compiler's usage directive directly after the applicable architecture.
-o OFMT	"specify output format" Choose an output format. The supported formats are listed in the compiler's usage directive under "Possible output formats".
-O OPRM	"parameter for output" Specify a parameter for the chosen output format. Different parameters are available for different formats, these are given in the compiler's usage directive directly after the applicable output format.
-u UFMT	"specify symbol generator" Choose a generator for in-image symbol information (see chapter 10). The supported generators are listed in the compiler's usage directive under "Possible sybol generators".
-U UPRM	"parameter for symbol generator" Specify a parameter for the chosen generator for in-image symbol information. Different parameters are available for different formats, these are listed in the compiler's usage directive directly after the applicable output format.
-z CALG	"compression" Compress the memory image. All following arguments will be used for the compilation of the decompressor. In addition to the usual parameters **-A r** or **-A a** may be given, which chooses the alignment for the target address specified by **-a** (with **-A r** the low order bits of the compressed image are used, with **-A a** the low order bits remain unaltered).

5.2 Symbol Information

Symbol information containing all relevant information about the generated objects is placed in the ASCII text file syminfo.txt on the host system after a succesful compilation (see section 3.6, if a specific debug or SymInfo writer was not specified on the command line with **-D**. This symbol information is broader in scope than the optional in-image symbol information (see chapter 10), and

is also human readable. Alternative debug writers (see section 3.7) write a subset of this information in a different format.

The basic structure is always identical, with extensions based on complier options. The beginning of the file contains general information about the generated memory image, in particular the parameters active during compilation and size information for each component. The example information below corresponds to the example in chapter 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
```

From this, we can glean the following information (this is from a compilation for a 32-bit architecture):

- No compiler options were used.
- The base address of the generated memory image is 0x00100000.
- The size of the memory image is 656 bytes, rounded off to 1 kilobyte.
- The 11 generated methods contain 241 bytes of code (rounded to 1 kilobyte). This is only the actually executable code: the memory taken up by method object headers is not taken into account.
- The image contains one String constant, which contains 11 characters and takes up 80 bytes (including all object headers): The 11 characters, taking up 2 bytes each for a total of 22 bytes, are contained in an array with a header containing 8 entries of 4 bytes each (4 entries from SArray and 4 from Object). After alignment, this takes up 22+32+2 = 56 bytes. This is contained in a String object with 6 header entries (2 from String and 4 from Object, including a pointer to the associated array), taking up 24 bytes. This overhead can be reduced with the options **`-l`** and **`-y`** (see section 5.1 and chapter 6), as well as by embedding the array (see chapter 8). With all of the above options the same string would only take up 20 bytes (11 for the characters, 1 for alignment, 8 for the String header).
- The size of the RAM block to be initialized is 0 (this is always the case when embedded mode (option **`-e`**) has not been used in compilation. The constant objects (strings and initialized arrays) take up 80 bytes.
- No in-image symbol information has been generated (see chapter 10), so it does not take up any space. The total amount of memory used is the sum of that taken up by the objects generated specifically for symbol information and the variables required for already initialized objects.

Furthermore, the following text will be output to syminfo.txt for the classes **`Object`** and **`String`**, when the example program from chapter 2 is compiled.

```
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:
```

From this, we can glean the following information:

- The class descriptor for objects of type **`Object`** lies at 0x00100030.
- The class **`Object`** is the root object (does not extend any other class).
- The class descriptor for **`Object`** specifies 4 class references (on a 32-bit architecture 4*4=16 bytes) and 8 bytes of class scalars, which are part of the class descriptor of every derived class. These elements are the instance variables of the type **`Object`** and the type **`SClassDesc`**.
- The class descriptor for **`Object`** specifies no static references or scalars. These would not be available in derived classes.
- Instances of type **`Object`** have 2 references (2*4=8 bytes on a 32-bit architecture) and 8 bytes of scalars.
- The class **`Object`** has no methods.
- The class **`Object`** has four declared variables:
  - The instance reference **`_r_type`** at a position of -4 relative to the pointer.
  - The instance reference **`_r_next`** at -8 relative to the pointer.
  - The instance scalar **`_r_relocEntries`** at position 0 relative to the pointer.
  - The instance scalar **`_r_scalarSize`** at +4 relative to the pointer.

- The class **Object** contains no constant Strings or initialized arrays, imports no classes (this would only be necessary for relocatable code (option **-m**)) and implements no interfaces.
- The class descriptor for objects of type **String** lies at 0x001000A8.
- The class **String** does not explicitly extend any other class, and thus extends the root object **java.lang.Object** (signified with **LANGROOT** for the sake of abstraction).
- The class descriptor for **String** specifies 8 class references (8*4=32 bytes on a 32-bit architecture) and 8 bytes of class scalars, which are part of the class descriptor of every derived class. These elements are, as in all class descriptors, the instance variables of the type **Object** and the type **SClassDesc**, but with the addition of two method pointers, which account for two references each.
- The class descriptor for **String** specifies no static references or scalars. These would not be available in derived classes.
- Instances of type **String** have three references and 12 bytes of scalars. These include the two references and 8 bytes of scalars for the type **Object**, as well as new variables: a reference for the **value** array and 4 bytes for **count**.
- The class **String** offers two new methods:
  - The method **length()**, at a position relative to the pointer of -24; The code object belonging to this method lies at 0x00100284 and contains 4 bytes of code.
  - The method **charAt(int)** at a position relative to the pointer of -32; The code object belonging to this method lies at 0x00100298 and contains 27 bytes of code.
- The class **String** contains no constant Strings or initialized arrays, imports no classes (this would only be necessary for relocatable code (option **-m**)) and implements no interfaces.

For pointer relative positions, one must pay attention to whether the effective address is calculated according to the instance (for instance variables), the current class (for class variables), or the declaring class (for static class variables).

The pointers to code objects for methods can be read at runtime with the **MAGIC** function **rMem32(.)**, **cast2Ref(.)** and **mthdOff(.)** (see chapter 12. In calculating the entry point, the address of the scalar block for the class **SMthdBlock** is necessary. It can be obtained directly with **MAGIC.getCodeOff()**.

6 Runtime Environment

6.1 Root Object

```
package java.lang;
import rte.SClassDesc;
public class Object {
  //required entries must exist in the order given here
  //obligatory
  public final SClassDesc _r_type;
  //The following entries are dropped if the "-l" option is active.
  public final Object _r_next;
  public final int _r_relocEntries, _r_scalarSize;
}
```

Every object in the heap requires an unambiguous type, in order to ensure type safety in casts and to make the **instanceof** operator available. This is implemented through a pointer **_r_type**, which points to the relevant class descriptor. Without the **-l** option, a record is made in every object of the address of the next object (**_r_next**) and the size of the object. The latter can be calculated from the number of references (**_r_relocEntries**, the size of the memory area used for these references is thus **_r_relocEntries*MAGIC.ptrSize**) and the size of the scalar block (**_r_scalarSize**).

6.2 Strings

```
package java.lang;
public class String {
  //Obligatory, Java conformant default, alternative byte array
  //  implementation with "-y" option
  private char[] value;
  Optional, filled out by the compiler for constant strings when present.
  private int count;
}
```

The class String specifies the variables **value** and **count**, which are both initialized automatically by the compiler for constant Strings. The first points to an array containing the characters making up the String, which are normally Java conformant 16 bit characters, but can be 8 bit (ASCII) characters if the **-y** option is used. The type declaration of **value** (**char[]** or **byte[]**) must be made by the programmer, **count** (**int**) is optional, depending on compilation model (for size differences see chapter 5.2).

6.3 Type system and Code Related Classes

```
package rte;
public class SArray { //Base type for arrays
   public final int length, _r_dim, _r_stdType; //Length, dimensionality,
primitive type, if applicable
  public final Object _r_unitType; //Object type, if applicable
}
```

```
package rte;
public class SClassDesc { //Base type for class descriptors
  public SClassDesc parent; //Parent class
  public SIntfMap implementations; //List of implemented interfaces
}
package rte;
public class SIntfDesc { //Marker for interfaces
  //optional: public SIntfDesc[] parents;
}
package rte;
public class SIntfMap { //Method map for a class onto an interface
  public SIntfDesc owner; //corresponding interface
  public SIntfMap next; //Next map for the current class
}
package rte;
public class SMthdBlock { //Base type for code blocks
}
```

The classes **SArray**, **SClassDesc**, **SInfDesc**, **SIntfMap** and **SMthdBlock** are used by the compiler in the construction of the initial heap.

6.4 Definition of the Runtime Environment for Allocation and Type Checking

The class DynamicRuntime in the package rte must contain the following methods:

Obligatory, used in the generation of new objects:

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

Obligatory, used to generate new one dimensional arrays:

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

Obligatory, used to generate multidimensional arrays:

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

Obligatory, for cast checking (**asCast==true**) and for the **instanceof** test (**asCast==false**), different methods for checking for classes, interfaces, and arrays, as well as for pointer storage checking.

```
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,
    Object unitType, int arrDim, boolean asCast) { ... }
```

```
    public static void checkArrayStore(SArray dest, Object newEntry) { ... }
}
```

The class DynamicRuntime can also contain the following methods, which are checked and used with specific compiler options.

For pointer allocation through the runtime system (`-c` or `-h`), for example for reference tracking or reference counting; The last three parameters must be adapted for architectures with different pointer sizes:

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

Runtime routine for array index errors (`-b`):

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

Runtime routine for illegal dereferencing of null pointers (`-n`):

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

Runtime routine for the use of profiling (`@SJC.Profile` or compiler option `-F`):

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

All runtime routines are only checked for their availability. It is the programmers responsibility to ensure the correct number and type of parameters.

6.5 Example Implementation of Required 32-bit Runtime Routines

```
public class DynamicRuntime {
   //The following variables are to be allocated before the first object allocation, for example
  //  with (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; //temporary variables
    Object me; //Object to be created
    rs=rlE<<2; //4 bytes needed per reloc
    scS=(scS+3)&~3; //Scalar allignment
    start=nextFreeAddress; //Beginning of the object
    nextFreeAddress+=rs+scS; //Place the next object after the current one
     for (i=start; i<nextFreeAddress; i+=4) MAGIC.wMem32(i, 0); //Initialize memory with 0
    me=MAGIC.cast2Obj(start+rs); //Place the object
    me._r_relocEntries=rlE; //Number of relocs in the object
     me._r_scalarSize=scS; //Record the amount of space used up by scalars in the object.
    me._r_type=type; //Record the type of the object in the object.
    return me; //Return the object to the caller
  }
  public static SArray newArray(int length, int arrDim, int entrySize,
      int stdType, Object unitType) {
    int scS, rlE; //temporary variables
    SArray me; //Array to be created
```

```
    scS=MAGIC.getInstScalarSize("SArray"); //Scalar size for an empty array
     rlE=MAGIC.getInstRelocEntries("SArray"); //Number of relocs in an empty
array
    if (arrDim>1 || entrySize<0) rlE+=length; //Array has reloc elements
    else scS+=length*entrySize; //Array has scalar elements
    me=(SArray)newInstance(scS, rlE, MAGIC.clssDesc("SArray")); //allocate
    me.length=length; //Record array length in the array
    me._r_dim=arrDim; //Record the dimension of the array in the array
    me._r_stdType=stdType; //Record the array's type
    me._r_unitType=unitType; //Record the reloc type
    return me; //Return the array to the caller
  }
  public static void newMultArray(SArray[] parent,
      int curLevel, int destLevel, int length, int arrDim,
      int entrySize, int stdType, Object unitType) {
    int i; //temporary variable
     if (curLevel+1<destLevel) { //If there is more than one more dimension
still needed
      curLevel++; //increment the current dimension
      for (i=0; i<parent.length; i++) //Fill every element with an array
        newMultArray((SArray[])((Object)parent[i]), curLevel, destLevel,
            length, arrDim, entrySize, stdType, unitType);
    }
    else { //If only one more dimension is needed
      destLevel=arrDim-curLevel; //Target dimension of an element
       for (i=0; i<parent.length; i++) //Fill every element with the target
type
        parent[i]=newArray(length, destLevel, entrySize, stdType, unitType);
    }
  }
  public static boolean isInstance(Object o, SClassDesc dest,
      boolean asCast) {
    SClassDesc check; //temporary variable
    if (o==null) { //Check for null
      if (asCast) return true; //null can always be converted
      return false; //null is not an instance
    }
    check=o._r_type; //Identify object type for further comparisons
    while (check!=null) { //Find the appropriate class
      if (check==dest) return true; //Matching class found
      check=check.parent; //Try parent class
    }
    if (asCast) while(true); //Conversion error
```

```
  return false; //Object does not match class
}
public static SIntfMap isImplementation(Object o, SIntfDesc dest,
    boolean asCast) {
  SIntfMap check; //temporary variable
  if (o==null) return null; //null implements nothing
  check=o._r_type.implementations; //Establish a list of interface maps
  while (check!=null) { //Find matching interface
    if (check.owner==dest) return check; //Interface found, return map
    check=check.next; //try next interface map
  }
  if (asCast) while(true); //Conversion error
  return null; //Object does not match interface
}
public static boolean isArray(SArray o, int stdType, Object unitType,
    int dim, boolean asCast) { //o is actually an object, check made below!
  SClassDesc clss; //temporary variable
  if (o==null) { //Check for null
    if (asCast) return true; //null can always be converted
    return false; //null is not an instance
  }
  if (o._r_type!=MAGIC.clssDesc("SArray")) { //Check if object is an array
    if (asCast) while(true); //Conversion error
    return false; //not an array
  }
  if (unitType==MAGIC.clssDesc("SArray")) { //Special handling for SArray
    if (o._r_unitType==MAGIC.clssDesc("SArray")) dim--; //Array of SArrays
    if (o._r_dim>dim) return true; //Sufficient remaining depth
    if (asCast) while(true); //Conversion error
    return false; //Not an SArray
  }
  if (o._r_stdType!=stdType || o._r_dim<dim) { //necessary constraints
    if (asCast) while(true); //Conversion error
    return false; //Array with non-matching element type
  }
  if (stdType!=0) { //Array of primitives
    if (o._r_dim==arrDim) return true; //matching depth
    if (asCast) while(true); //Conversion error
    return false; //Array with non-matching element type
  }
  //Type check necessary
  if (o._r_unitType._r_type==MAGIC.clssDesc("SClassDesc")) { //Instances
```

```
      SClassDesc clss=(SClassDesc)o._r_unitType;
      while (clss!=null) {
        if (clss==unitType) return true;
        clss=clss.parent;
      }
    }
    else { //Interfaces not supported
      while(true); //not supported
    }
    if (asCast) while(true); //Conversion error
    return false; //Array with non-matching element type
  }
  public static void checkArrayStore(SArray dest, SArray newEntry) {
    //newEntry ist actually an object, must be checked with isArray!
    if (dest._r_dim>1) isArray(newEntry, dest._r_stdType, dest._r_unitType,
      dest._r_dim-1, true); //Check the array with isArray
                            //  If the dimension of the array is > 1
    else if (dest._r_unitType==null) while (true); //Allocation error
                              //  if the destination array doesn't have reloc
elements
    else {                  //  Instance check in all other cases
      if (dest._r_unitType._r_type==MAGIC.clssDesc("SClassDesc"))
        isInstance(newEntry, (SClassDesc)dest._r_unitType, true);
      else isImplementation(newEntry, (SIntfDesc)dest._r_unitType, true);
    }
  }
}
```

The individual methods must allway be present, but unused methods (for instance **`newMultArray`**, if no multidimensional arrays are constructed) can by all means be left empty, as in the example in chapter 2.

6.6 Runtime Extensions for the Use of Java Exceptions

If Java exceptions are to be used, that is, if at least one of the keywords **`try`**, **`catch`**, **`finally`**, **`throw`** or **`throws`** is used, the compiler will expect the variable

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

as well as the method

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

in **`rte.DynamicRuntime`**. The variable **`currentThrowFrame`** is used to store the address of the current throw frame, as well as those constructed by the back end for **`try`**, **`catch`**, and **`finally`** statements. These blocks, created on the stack, are chained together one under the other and contain all information necessary for the processing of Java exceptions. **`Throw`** statements, as well as exceptions not handled by **`catch`** and **`finally`** blocks, cause a call to the **`doThrow`** method. After the verification of **`currentThrowFrame`** (if this variable does not hold a valid pointer, the current exception was not handled anywhere) this method should perform all tasks required for the current

architecture and appropriate to the constructed throw frame and then end with a jump to the address given in the throw frame.

An implementation for the IA-32 environment could look like the following:

```
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, 0x45);MAGIC.inlineOffset(1, frame); //eax<-frame
  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]
}
```

On the AMD 64 architecture, the following implementation could be used:

```
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]
}
```

Optimizations are possible, depending on target architecture (see section 5.1, and different target architectures require completely different implementations. Notes on the structure of the throw frame can be found in the compiler source code in the back end for a particular architecture.

6.7 Runtime Extensions for the Use of Synchronized Blocks

If the keyword **`synchronized`** is used, the compiler will expect the method

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

in **`rte.DynamicRuntime`**, where **`o`** is the object, on which synchronization is to be carried out, and **`leave`** differentiates whether the synchronized block is being entered or exited (**`false`** on entry, **`true`** on exit).

The implementation is strongly dependent on the rest of the runtime environment and operating system, so no universally valid implementation exists. However, in creating an implementation, the following points should be kept in mind:

- Every object can serve as a synchronization object, so an entry/exit counter would need to be provided in **`java.lang.Object`**.
- A unique thread ID from the operating system must be deliverable to the currently running thread, in order to be stored as a label for the synchronization variable in **`java.lang.Object`**
- On entry to a synchronized block that is already marked with a different thread ID, it should be possible to deactivate the current thread and activate the marked thread.
- The code of the method **`doSynchronize`** should run with interrupts disabled.

In an unthreaded environment the method **`doSynchronize`** can be left empty (for example, if the same source code is being developed for multiple operating systems and now concurrency is to be expected in the SJC variant) or can be used for debugging (for example to record entries and exits to a block of code under observation and to monitor the data stored in the synchronization object).

6.8 Runtime Extensions for Stack Limit Checking

If a method is marked with **`@SJC.StackExtreme`** or the **`-x`** option is used, the compiler will expect the variable

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

as well as the method

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

in **`rte.DynamicRuntime`**. The method **`stackExtremeError()`** will be called if and only if a method appears likely to reach a stack pointer value below the value of **`stackExtreme`** during the course of its execution.

6.9 Runtime Extensions for the Use of Assert Statements

If a method is marked with **`@SJC.StackExtreme`** or the **`-x`** option is used, the compiler will expect the method

```
static void assertFailed(String message) { ... }
```

in **`rte.DynamicRuntime`**. This method will be called if the condition in the **`assert`** statement evaluates to **`false`**.

6.10 Definition of the Optional Arithmetic Library

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

An optional arithmetic library can be included, depending on the architecture, in order to centrally supply operations that are not directly implementable, or not implementable with reasonable complexity. Execution is slowed down by the addition of a method call, but the code need only be available on a per-operator basis rather than a per-operation basis.

For the IA-32 backend **`binLong`** can be activated with the parameter **`-T rtlc`**, for the ATmega backend the analogically constructed methods for **`binShort`**, **`binInt`** and **`binLong`** will be used when the corresponding operators are used.

7 Special Classes

7.1 MAGIC

The following list covers the currently available special variables and methods of the compiler internal class **MAGIC**, examples can be found in chapter 12. Depending on compiler mode (see the compiler options in section 5.1) individual variables or methods may not be available.

- **static int ptrSize;**

  The pointer size of the current architecture (e.g. 4 for IA32, 8 for AMD64)

- **static boolean movable, indirScalars;**

  Flags indicating the relocatability of code and scalars (i.e. whether the code was compiled with **-m** or **-M**).

- **static boolean streamline;**

  Flag indicating a streamlined object structure (i.e. the use of the **-l** option in compilation).

- **static boolean assignCall, assignHeapCall;**

  Flags indicating whether runtime calls are made for all allocations or allocations on the heap, respectively (i.e, whether **-c** or **-h** were used in compilation).

- **static boolean runtimeBoundException;**

  Flag indicating if runtime calls are made for array index out of bounds errors (i.e, if **-b** was used in compilation).

- **static boolean runtimeNullException;**

  Flag that indicates whether runtime tests are made for null pointers and if runtime calls are made when they are used illegally (i.e, whether **-n** was used in compilation).

- **static int imageBase;**

  Base address of the current memory image (the parameter of the **-a** option).

- **static int compressedImageBase;**

  Target address of the memory image to be decompressed by the decompressor (the parameter of the **-a** option for a compressed memory image). The decompressor must ensure that the decompressed memory image is stored at the designated address (or else relocated with the **-Z** option, see **relocation** and **comprRelocation** below).

- **static boolean embedded;**

  static boolean embConstRAM;

  Flags respectively indicating whether class variables have been stored outside of class descriptors in a special memory region (i.e, **-e** was used during compilation) and whether objects have been stored in a special memory region (the **-E** option was used during compilation), see section 12.5.

- **static int relocation;**

  static int comprRelocation;

  Preset relocation (compiler option **-Z XXX**, see section 5.1) of the current memory image, or, in the decompressor, of the memory image to be decompressed. This option only makes sense for 64-bit architectures, as the relocation is always given in gigabytes.

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

  static void inline16(.);

  static void inline32(.);

Pseudo-methods for the direct insertion of machine code. The number of parameters is variable. The parameters must be constants, and for each parameter only the lower 8, 16, or 32-bits (respectively) will be evaluated. The following two statements both produce the same code on little endian architectures:

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

- `static void inlineBlock(String name);`

  Pseudo-method for the insertion of a block of machine code with the name given in the parameter. This must be available as a constant string and must contain no dereferences. The data block normally comes from an imported binary file with a *.bim extension and carries the name of the file minus the path and extension.

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

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

  Pseudo-methods to insert variable offsets into inline code. These method are typically used to access variables in inline code.

- `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 methods to write directly to a given memory address. The address must be given in the format appropriate to the architecture or as an `int`.

- `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-methods to read directly from a given memory address. The address must be given in the format appropriate to the architecture or as an `int`.

- `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-methods to write directly to a given I/O port. The address must be given in the format appropriate to the architecture or as an `int`. Some architectures may not support all bit widths under all circumstances.

- `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-methods to read directly from a given I/O port. The address must be given in the format appropriate to the architecture or as an `int`. Some architectures may not support all bit widths under all circumstances.

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

  Pseudo-method which “returns” the address of the referenced object. Not to be confused with the method `addr(.)`, which “returns” the address of the variable `o` instead of the object referenced through it. Examples can be found under `MAGIC.addr()`.

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

  Pseudo-method which converts an address into an object reference. The address must be that of the first scalar field of the object (see section 3.4 and **`MAGIC.addr()`**).

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

  Pseudo-method which "returns" the address of the variable given in the parameter. Not to be confused with **`cast2Ref(.)`**, which "returns" the address of the passed object. On a 32-bit system, for a given object reference o, the following identities are valid:

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

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

- **`static SClassDesc clssDesc(String clss);`**

  `static SIntfDesc intfDesc(String intf);`

  Pseudo-method which "returns" the descriptor for the class or interface given as a parameter. For Java compatibility, the class or interface name must be in the form of a constant string and may contain no dereferences. This annotation will cause the "returned" descriptor to be generated during compilation, whether or not it is used in the code being compiled.

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

  Pseudo-method which "returns" the offset of the method given in the second parameter within the class given in the first parameter. Both parameters must, for Java compatibility, be in the form of constant Strings and may contain no dereferences. In cooperation with **`clssDesc`** the addresses, for example, of method objects can be obtained.

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

  Pseudo-method which "returns" the offset of the code within a method object. The first opcode can be found this many bytes beyond the target of the method pointer. The beginning of code for a method Cls.Mtd can be calculated as follows:

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

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

  Pseudo-method that "returns" the size of the scalar block for an instance of the class passed as a parameter. For Java compatibility, the identifier must be in the form of a constant String and may contain no dereferences.

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

  Pseudo-method which "returns" the number of references contained in an instance of the class passed as a parameter. The size of the reference block can be calculated by multiplying this number by the pointer size. For Java compatibility, the identifier must be in the form of a constant String and may contain no dereferences.

- **`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-method to set or clear (chosen by the **`set`** parameter) the bits indicated by **`mask`** in memory. The address must be given in the format appropriate to the architecture or as an **`int`**.

- **`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-method which sets or clears (chosen by the **set** parameter) the I/O port bits indicated by **mask**. The address must be given in the format appropriate to the architecture or as an **int**. Some architectures may not support all bit widths under all circumstances.

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

  Pseudo-method that converts an address into a struct reference. The address must be given in the format appropriate to the architecture or as an **int** (see also **cast2Obj(.)**). A further conversion to another struct type will be carried out without any type checking or call to the runtime environment.

- **static int getRamAddr();**

  static int getRamInitAddr();

  static int getRamSize();

  Pseudo-methods which "return" the base address of the RAM block used to store class variables in embedded mode (given as a parameter to the compiler option **-e**), its size, and the address of the initialization routines, see section 12.5.

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

  Pseudo-method which explicitly assigns an object in a constructor as the current instance. The primary use is constructors for classes with inline Arrays, which will be discussed in more detail in chapter 8.

- **static void getConstMemorySize();**

  Pseudo-method which "returns" the amount of memory needed for constant objects. This is of interest, for example, in embedded mode, as constant objects are stored directly after the RAM initialization block. They can thus, for tests with Harvard architectures, easily be copied into RAM in order to test "dynamic" objects.

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

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

  These two pseudo-methods "return" initialized constant arrays corresponding to constant strings, containing the characters of the passed Strings as well as an optional terminating null character. In this way, optionally null-terminated character arrays can be formed from strings without tedious hand initialization or runtime conversion costs.

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

  Pseudo-method which will be replaced by the constant string whose name has been passed as **stringName**. If the object **stringName** is not found by the compiler, the result is **null**. In substance this function is comparable with the import of binary files as byte arrays (see chapter 4.1), however, this function can be used when it is not certain if the target object exists (the result is then **null** and compilaiton is still possible; for references to an object as in chapter 4.1 the referenced object must exist). For more information on importing files see chapter 4.2.

- **static void doStaticInit();**

  Pseudo-method which incorporates the statements used in static class initialization. For this, the target architecture must support method-inlining, as the statements of the target methods are inserted instead of the **doStaticInit()** call.

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

  All parameters passed as parameters to this function will be ignored (although the given parameters must be syntactically correct). For example, if a variable is only read in inline machine code, Sun Java will display a warning, which can be avoided by using **MAGIC.ignore(.)**, which SunJava interprets as an access to the variable.

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

  Code generation for the current block is stopped. For example, this can be used when an inline machine code block has already stored a return value in the register used on the current architecture for passing return values, but a **`return`** statment is necessary for source code compatibility with standard Java.

- **`static void assign(VAR, value);`**

  Forces a write to the variable **`VAR`**, regardless of whether it is marked with the **`final`** flag. By this means a Java conformant initialization of, for example, newly allocated Arrays is possible, even if **`SArray.length`** has been declared as **`final`**.

7.2 SJC Annotations

The following list contains the currently available compiler-internal annotations, which are marked with **`@SJC`**. Examples can be found in chapter 12.

- **`@SJC.Interrupt`**

  Marks a method as an interrupt handler called directly by the hardware. Supporting back-ends generally generate a special prologue and epilogue. A method so marked may not be called by "normal" Java code.

- **`@SJC.Debug`**

  The code generated for the marked method will be stored in a binary file (see also chapter 5.1: compiler option **`-d`**).

- **`@SJC.Inline`**

  `@SJC.NoInline`

  @SJC.Inine marks the code for a method to be inserted inline, if possible. @SJC.NoInline marks a method to be excluded from automatic inlining (see also chapter 5.1: compiler option **`-s`**).

- **`@SJC(enterCodeAddr=ADDR)`**

  Inserts the code for the marked method into the memory image beginning at address **`ADDR`**.

- **`@SJC.GenCode`**

  `@SJC.GenDesc`

  Always generate the code of the marked method or descriptor for the marked class (see also chapter 5.1: compiler option **`-G`**).

- **`@SJC.Profile`**

  `@SJC.NoProfile`

  Generate code for the marked method with or without (respectively) calls to profiler methods (see also chapter 5.1: compiler option **`-F`**). No profiler calls will generally be generated for the profiler methods themselves. If a method is marked with both **`Profile`** and **`NoProfile`** no profiler calls will be inserted.

- **`@SJC.CheckAssert`**

  Code is generated for assert statements (see also compiler option **`-r`** and chapter 6.9.

- **`@SJC.StackExtreme`**

  `@SJC.NoStackExtrme`

  Generate code for the marked method with or without (respectively) stack limit checks (see chapter 6.8: Runtime extensions and chapter 5.1: compiler option **`-x`**). No limit checking calls will generally be generated for error handling methods.

- **`@SJC.PrintCode`**

  If possible, the generated assembly code for the method will be output.

- **`@SJC.SourceLines`**

  Activates the logging of source line numbers with reference to the generated code. This data can be evaluated with an appropriate DebugWriter (option **`-D`**) or SymbolInformer (option **`-U`**).

- **`@SJC.NoOptimization`**

  If possible, code optimization will be deactivated.

- **`@SJC.ExplicitConversion`**

  Within this unit or method conversions of primitive types will only succeed if done explicitly. In system classes, this allows easy detection of unwanted sign extensions.

- **`@SJC.WinDLL`**

  Marked native methods will be called with Windows DLL stackframes rather than Linux library stackframes.

- **`@SJC.NativeCallback`**

  Marked methods will not remove their parameters from the stack when they return, in order to allow them to be called from standard C functions (structured approximately as in Linux). The parameters must be reversed in order by the programmer. Under windows this annotation is not necessary, as cleanup of the stack is, as in SJC, the responsibility of the callee. A method so marked may not called from "normal" Java code.

- **`@SJC.IgnoreUnit`**

  Marked units will be checked for syntax, but not further processed.

- **`@SJC.Flash`**

  In embedded mode, the compiler will try to hold instances of the marked class in flash memory. Initialized array variables so marked will be held in flash in embedded mode.

- **`@SJC(offset=OFF)`**

  Allows variable offsets in **`STRUCT`** classes (see chapter 7.3) to be explicitly set.

- **`@SJC(count=CNT)`**

  Allows otherwise uncheckable array lengths in **`STRUCT`** classes (see chapter 7.3) to be fixed. If the length is to be deliberately left unchecked, this annotation should be used with the parameter -1.

- **`@SJC.Ref`**

  In **`STRUCT`** classes (see chapter 7.3) other **`STRUCT`**s are typically integrated inline. The **`@SJC.Ref`** annotation can be used to force a variable marked with it not to be integrated, but rather to be handled as a reference to a separate **`STRUCT`**.

7.3 STRUCT

Classes derived from **`STRUCT`** (for examples see chapter 12.4) can be used to create memory structures independent of heap and compiler structures in a type safe manner. For example, device registers for memory mapped I/O devices can be specified relative to a start address and named, so that the compiler can compute the offsets for and check accesses to those registers. Done properly, this will also improve the readability and maintainability of device drivers. At the same time, the syntax is built upon instance variables in a fully Java conformant manner, so that all Java tools can be used alongside **`STRUCT`**. Because of missing type information, however, it must be noted that

references to **STRUCT**s cannot be created with the **new** operator, nor can they be cast in a type safe manner.

If **STRUCT** declarations reference other **STRUCT** classes, the fields of the embedded STRUCT will be incorporated into the embedding STRUCT. Should references to the embedded **STRUCT** be used instead, the variable must be marked with **@SJC.Ref** (see chapter 7.2).

7.4 FLASH

Instances of classes that extend **FLASH** can be addressed through the flash memory instead of the RAM of a microcontroller. Support for this is still experimental.

8 Inline Arrays

For classes that are not extended by other classes, it is possible to embed arrays accessed through an instance variable directly within the instance. This gives considerable advantages in storage space on the one hand, but on the other, internally allocated arrays are not compatible with normal arrays. Because no reference exists for these arrays, they cannot be replaced by reallocation. There is thus a compromise between low memory requirements for constant objects on the one hand and reusability or replacibility on the other.

The compiler supports this mode only in static operation, that is, not in combination with the options **`-m`** or **`-M`**. Classes whose instances should contain an integrated array must have an array variable (with the annotation **`@SJC.InlineArrayVar`**), a length variable (type **`int`** with the annotation **`@SJC.InlineArrayCount`**) and must be declared as **`final`**. If instances of this class are to be created at runtime, explicit constructors must be used, which must declare the current **`this`** instance with **`MAGIC.useAsThis(object)`**. No type checking will be done, and furthermore, until this call is made the use of instance variables is forbidden (even in a possibly called **`super`** method).

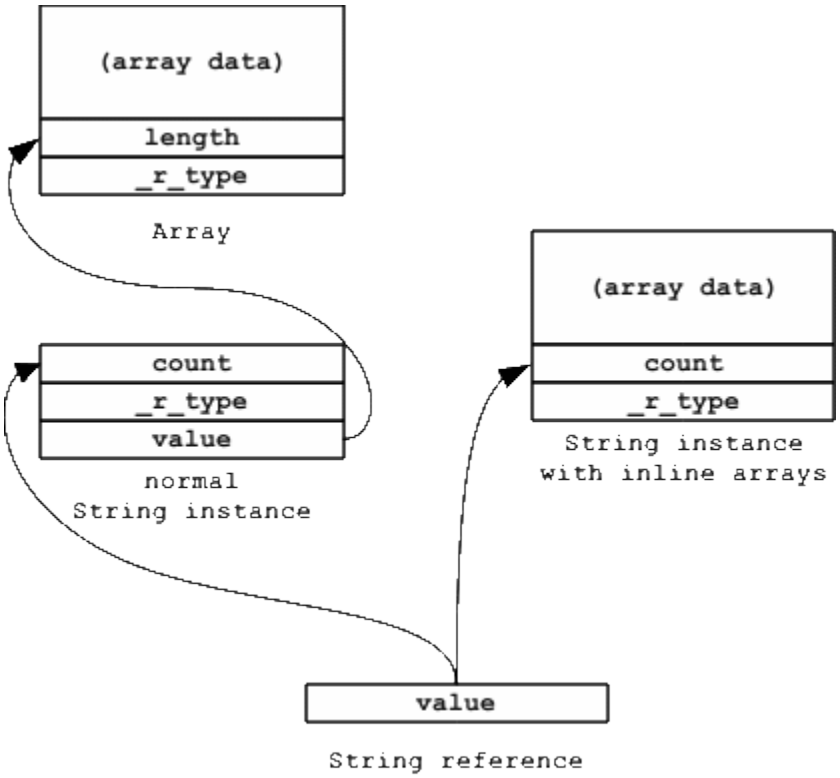
8.1 Changes to the Object Structure

```
public final class String {
  @SJC.InlineArrayVar
  private char[] value; //Name for access to the inline array
  @SJC.InlineArrayCount
  private int count; //Length of the inline array

  public String(char[] arr) {
    int i;
    //Create new instance, only then will instance variables be valid!
    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;
  }
  @SJC.Inline
  public int length() { //Returns length as usual
```

```
        return count;
    }
}
```

In the example above the character array in instances of the **`String`** class is integrated into the object, so no extra array object is necessary for **`value`**. The following graphic clarifies the difference between conventional Strings (left) and Strings with embedded arrays (right) when the `-l` option is used (otherwise there would be three more fields per object, see chapter 6):



9 Indirect Scalars

This chapter covers the features of the **-M** option (see also section 3.4). When this option is used, the scalars of an instance are accessed indirectly through the address variable **_r_indirScalarAddr** in the object header. This makes it possible for instances to have their references and object information on the one hand separated from their scalars on the other hand, which can be advantageous in the implementation of several consistency models.

Objects, so far two-headed and self contained, are extended by the addition of an extra memory block, which contains only the scalars of the object. Type pointers and the like are not intended for this block. (The use of pointers in this block is indeed possible, through the conversion of a scalar to a pointer with the help of **MAGIC**, but not recommended.)

Adaptions to the root object and runtime structure are necessary in order to provide this functionality, which will be highlighted in the following two sections.

9.1 Changes to the Object Structure

```
package java.lang;
import rte.SClassDesc;
public class Object {
  public final SClassDesc _r_type;
  public final Object _r_next;
  @SJC.Head public final int _r_relocEntries;
  @SJC.Head public final int _r_scalarSize;
  @SJC.Head public final int _r_indirScalarAddr;
  @SJC.Head public final int _r_indirScalarSize;
}
package java.lang;
public class String {
  public char[] value;
  @SJC.Head public int count;
}
package rte;
public class SArray {
  @SJC.Head public final int length;
  @SJC.Head public final int _r_dim;
  @SJC.Head public final int _r_stdType;
  public final SClassDesc _r_unitType;
}
```

The annotation **@SJC.Head** anchors the marked variables as part of the object header, instead of allocating them in the indirectly addressed block as is normal when the **-M** option is in effect. The compiler expects this for the three variables marked above, other variables can be moved by the programmer from the indirectly addressed block to the directly addressed block by the same mechanism.

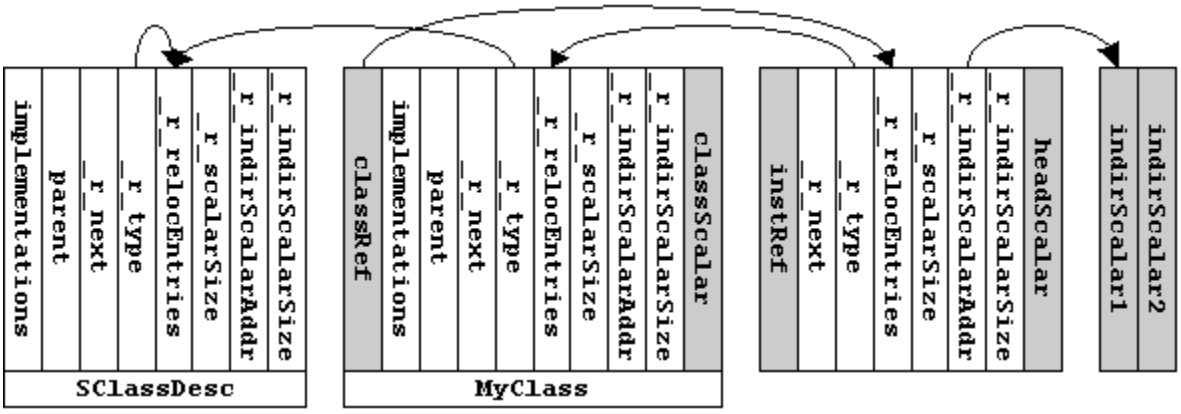
For a class with a class reference, a class scalar, as well as an instance reference and three instance scalars, the delcaration of these variables would look like the following, if one of the instance scalars was to be integrated into the Object itself and two were to be indirectly addressed:

```
public class MyClass {
  public static Object classRef;
  public static int classScalar;
  public Object instRef;
  @SJC.Head public int headScalar;
  public int indirScalar1, indirScalar2;
}
```

An allocation could be performed by the following code:

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

After the execution of the `main()` method, the following memory structure would have been constructed:



The class variables as well as the instance references are integrated into the heap objects. The instance scalars, however, are by default stored in an indirectly addressable memory block. If an instance scalar is to be incorporated into the main object block, this can be achieved by marking it with `@SJC.Head`, as is mandatory for the variables that are required by the runtime structure.

The compiler places the indirectly addressed scalars in the initial memory image directly after the objects with which they are associated, so that an unambiguous mapping from scalar blocks to their objects is possible. At runtime the position is freely selectable, and scalar blocks can be moved after allocation.

9.2 Changes to the Runtime Environment

To support separate storage of scalars, adaptions to the runtime environment are necessary (see chapter 6). In addition to the existing size fields a new one is necessary for the allocation of an object:

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

For the use of arrays, the calls to **newInstance** in **newArray** and **newMultArray** must be adapted. All in all, the changes to the example implementation from section 6.5 are as follows:

```
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, Object unitType) {
    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_unitType=unitType;
    return me;
  }
}
```

10 In-Image Symbol Information

The current version of SJC supports three generators for symbol information, discussed in the following sections. They generate symbol information for packages, units, and methods, which can in turn be used for debugging and serve as the basis for TextualCall (see section 12.7) or for runtime compilation.

10.1 raw Symbol Information

Output of raw symbol information can be selected with the compiler option (see section 5.1) **`-u raw`**, which requires no parameters. After compilation has finished, this module generates a byte-Array containing the symbol information for the generated packages, units, and methods. It can be accessed with the identifier **`info`** in the automatically constructed class **`RawInfo`** in the package **`symbols`** and consists of blocks beginning with identification characters (**`P`** for packages, **`U`** for units, and **`M`** for methods), containing a variable number of ASCII characters, and terminated with an exclamation point (**`!`**). Associated blocks follow directly one after the other, so the blocks for the methods of a unit are in direct contact with the unit in question. The last character of the whole array is also an exclamation point (**`!`**), indicating a completely empty block.

Block structure	Description
`P`(fullname)**`!`**	The only field for a package holds its fully qualified name, for example **`java.lang`**.
`U`(name)**`#`**(modf)(addr)(exts)**`!`**	The first field for a unit contains its basic name, for example **`String`**, and is terminated with the **`#`** sign. This is followed by two 32-bit values, each represented in hexadecimal by 8 ASCII characters. The first is the modifier of the unit (see compiler class **`compbase.Modifier`** or **`compbase.Unit`**) and the second is the address of the unit descriptor. If the unit is a class and not an interface, and does not directly extend the root object, a further text field follows with the fully qualified name of the class, for example **`java.lang.String`**.
`M`(name)**`#`**(modf)(addr)**`!`	The first field of the method contains its simple name with fully qualified parameters, for example **`equals(java.lang.String)`**, and is terminated with the **`#`** sign. This is followed by two 32-bit values, each represented in hexadecimal by 8 ASCII characters. The first is the modifier of the method (see compiler class **`compbase.Modifier`** or **`compbase.Unit`**) and the second is the address of the method's code object.

If an image consisted entirely of the class String and the method length(), the resulting byte array would look something like the following:

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

The first block (1) belongs to the package **`java`**, the second to the package **`java.lang`**. The third (3) defines the class **`String`** with the modifier (4) **`0x00000001`** at the address (5) **`0x00100090`**. The fourth block (6) describes the method **`length()`** with the modifier (7) **`0x00580001`** at the address (8) **`0x00100F5C`**. The last block (9) contains the end marker **`!`**.

10.2 rte Symbol Information

Output of rte Symbol information can be selected with the compiler option **-u rte** (see section 5.1), the parameters for which are clarified below. After compilation has finished, this module generates an object of type **rte.SPackage** and fills in its fields, the root package is recorded in the static variable **rte.SPackage.root**. Furthermore, additional fields are filled in in already existing instances of the classes **rte.SClssDesc** and **rte.SMthdBlock**. The source code for a minimal correct declaration of these three classes is as follows:

```
package rte;
public class SPackage { //Extra class
  public static SPackage root; //static field for root package
  public String name; //Basic name of the package
  public SPackage outer; //Parent package, noPackOuter deactivated*
  public SPackage subPacks; //first child package
   public SPackage nextPack; //next package at the same level of the package
hierarchy
  public SClassDesc units; //first unit of the current package
}
package rte;
public class SClassDesc { //Instance variables have been added to this class
   public SClassDesc parent; //Already part of the class descriptor: class
extended by this one
   public SIntfMap implementations; //Already part of the class descriptor:
implemented interfaces
  public SClassDesc nextUnit; //next unit of the current package
  public String name; //basic name of the unit
    public SPackage pack; //Package this unit belongs to, noClassPack
deactivated
  public SMthdBlock mthds; //first method of this unit
  public int modifier; //Modifier of this unit, noClassMod deactivated*
}
package rte;
public class SMthdBlock { //Instance variables have been added to this class
  public String namePar; //basic name, fully qualified parameter types
  public SMthdBlock nextMthd; //Next method of the current class
  public int modifier; //Modfier for this method
  public int[] lineInCodeOffset; //Optional mapping of source line numbers to
machine code**
}
```

The instances at a given level of the hierarchy form a linked list, the next element of which can be accessed through the **next*** field. For packages, the basic name (for instance **lang**) is recorded in the **name** field. The parent package can be reached through the **outer** field, the first subpackage through **subPacks**, and the first class through **units**. For classes, the **parent** and **implemenations** fields, which are needed even without symbol information, remain. Added fields include **name** for the basic name of the class (for instance **String**), a reference **pack** to the package the class belongs to (which in this case would be **lang**), a reference **mthd** to the first method of the class, and **modifier** which contains the scalar value of the modifier (see **compbas.Modifier** bzw.

compbase.Unit). Method code objects have the fields **namePar**, which contains the basic name of the method with fully qualified parameter types (for example **equals(java.lang.String)**), and **modifier**, which contains the scalar value of the method modifier (see **compbase.Modifier** or **compbase.Mthd**).

The use of the fields marked with * can be deactivated with the correpsonding parameters **-U noPackOuter**, **-U noClassPack** and **-U noClassMod**. If this is done, they must not be declared. Special note should be made of the field marked with **, **lineInCodeOffset**, which can either be deactivated with the parameter **-U noSLHI** or by leaving the field out. If line number integration is needed, either the compiler option **-Q** (see section 5.1) (globally) or the per-method annotation **@SJC.SourceLines** (locally) (see section 7.2) must be used, otherwise the field will remain empty.

10.3 Mthd Symbol Information

Output of Mthd Symbol information can be selected with the compiler option **-u mthd** (see section 5.1), the only parameter it can take is **-U chain** for the chaining of method blocks. After compilation has finished this module stores a string with the name of the method and the implementing class in every method block. Source code for a minimal correct declaration of **rte.SMthdBlock**:

```
package rte;
public class SMthdBlock {
  public String namePar; //Name and fully qualified parameter types
  public SMthdBlock nextMthd; //optional: next method
}
```

11 Native Linux and Windows Programs

11.1 Basic Functionality

In principle, the code generated by the compiler can also be executed under Microsoft Windows or Linux. However, depending on the operating system in use, certain general conditions must be fulfilled and various functions outside of the usual runtime environment must be called for system and user interaction. These system calls must, naturally, be made according to the conventions of the system in use, for which either an interrupt interface (Linux) or function calls through shared memory (Linux and Windows) can be used.

The interrupt interface in Linux is clearly defined by the kernel specification. Depending on the Linux kernel in use the system call parameters can be passed in various ways: In registers, on the stack, or a combination of the two. At the moment the compiler has no direct support for the automatic generation of interrupt calls, so the user must hand code them with inline machine code (see section 12.6). An example can be found in section 11.2, and under [nat].

The function call interface under Linux and Windows is built on dynamic libraries and the basic functionality is very similar for the SJC user. In either case, with the help of a library name, a system function returns a handle to or address of the library that contains the needed function. Another system function can search through the loaded library by name for the needed function, and, if successful, return the entry point of this function. Sections 11.2 and 11.3 contain examples for Linux and Windows. Executable demonstrations can be found under [nat].

Before the functions that resolve library and function requests can be used, they would normally have to be called themselves to determine their entry points, but this would create a circular dependency. To avoid this, the Windows header, in addition to the extended Linux header, contain, in addition to system compliant headers and runtime initialization, the entry points of these basic system functions, so that Java code can always fall back on the Windows or Linux headers to call these functions. All three headers are available under [nat].

If the entry point and signatures of the library function to be called are known, SJC can, with the help of the **`native`** keyword, automatically generate the stack frame required for a call. To accomplish this, a native method must be declared in an (arbitrary) class as **`static`**. A static **`int`** variable of the same name, containing the address of the entry point of this method, must be made available in the same class. For calls to native methods, a stack frame appropriate to the system is generated, and the function at the address pointed to by the **`int`** variable is called. Examples can be found in the next two sections, 11.2 and 11.3, as well as under [nat].

The heap structure used by SJC (see sections 3.3 and 3.4) is totally unknown to both Linux and Windows. For this reason care should always be taken that, in communicating with the operating system, only primitive types or pointers to operating system compliant memory blocks (for example by means of STRUCT, see sections 7.3 and 12.4, or MAGIC.addr, see section 7.1) should be used.

11.2 Native Linux Programs

Under Linux, if only kernel functions are to be used, then the basic Linux header, without library support, can be used. The functions for loading a library and resolving names are then not available. Calls to kernel functions are made by interrupts and must be hand coded. The following example prints a character to standard output:

```
public static void printChar(int c) {
  //set eax to 4: syscall number for write
  MAGIC.inline(0xB8, 0x04, 0x00, 0x00, 0x00); //mov eax,4
  //set ebx to 1: file handle for standard output
```

```
    MAGIC.inline(0xBB, 0x01, 0x00, 0x00, 0x00); //mov ebx,1
    //set ecx to the address of the string to be designated
    MAGIC.inline(0x8D, 0x4D); MAGIC.inlineOffset(1, c); //lea ecx,[ebp+8]
    Set edx to 1: The length of the string to be designated
    MAGIC.inline(0x89, 0xDA); //mov edx,ebx
    //call to the kernel
    MAGIC.inline(0xCD, 0x80); //int 0x80
  }
```

All calls to kernel functions can be handcoded analogously to this example. Depending on the kernel options and kernel version used, there may be circumstances under which some or all parameters are passed on the stack instead of in registers.

If, in addition to kernel functions, library functions, for example, to connect to an X11 system, are used, then an extended Linux header with integrated library support must be used. This gives the entry points to the Linux functions dlopen and dlsym, which resolve library and function names, at **`MAGIC.imagBase-12`** and **`MAGIC.imageBase-8`**. As an example, the function XOpenDisplay can be declared, initialized, and used as follows:

```
public native static int XOpenDisplay(int no); //native function
public static int XOpenDisplay; //Address of the native function
...
public static int dlopen(byte[] libNameZ, int flags) { //handcoded dlopen
wrapper
  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 count;
}
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; //Error, library not found
```

```
    }
    if ((XOpenDisplay=dlsym(i, MAGIC.toByteArray("XOpenDisplay", true)))==0) {
      return; //Error: function XOpenDispaly not found
    }
  }
  ...
  public static void example() {
    int display;
    if ((display=XOpenDisplay(0))==0) { //Accesses native function
      return; //Error: could not open display 0
    }
    ... //Do something with the opened display
  }
```

The functions dlopen and dlsym could also be represented with the keyword **native**, if, instead of a null terminated SJC compliant byte array, a pointer to the first byte of the array were passed. The complexity required for hand coding is not needed for the declared-as-native function **XOpenDisplay**, as the steps required for a valid API call are automatically carried out by the compiler. Furthermore, with declared-as-native API calls the wrapper can be dispensed with, as the compiler generates the native call directly at the point of use.

Access to the parameters given at program startup can be made through both the basic and the extended header through the initial stack pointer stored at **MAGIC.imageBase-4**. The number of parameters passed is stored at this address. Above this a pointer to a null terminated character string can be found for every parameter. The code to evaluate the parameters passed to a program can look something like the following:

```
  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);
  }
```

If the parameters are to be used within the program as SJC-compliant strings an explicit conversion must be made.

11.3 Native Microsoft Windows Programs

As the definition for executable file headers under Windows leaves copious empty space for the resolution of library functions, the addresses of the following runtime variables and system functions are resolved before the execution of Java code begins:

Handle for standard input	**MAGIC.imageBase-512+0**

Handle for standard output	`MAGIC.imageBase-512+4`
Handle for standard error	`MAGIC.imageBase-512+8`
Pointer to parameter list	`MAGIC.imageBase-512+12`
Number of parameters passed	`MAGIC.imageBase-512+16`
Kernel.GetStdHandle function	`MAGIC.imageBase-512+20`
Kernel.GetCommandLineW function	`MAGIC.imageBase-512+24`
Kernel.ExitProcess function	`MAGIC.imageBase-512+28`
Kernel.WriteFile function	`MAGIC.imageBase-512+32`
Kernel.ReadFile function	`MAGIC.imageBase-512+36`
Kernel.CreateFileA function	`MAGIC.imageBase-512+40`
Kernel.GetFileSize function	`MAGIC.imageBase-512+44`
Kernel.CloseHandle function	`MAGIC.imageBase-512+48`
Kernel.LoadLibraryA function	`MAGIC.imageBase-512+52`
Kernel.GetProcAddress function	`MAGIC.imageBase-512+56`
Kernel.VirtualAlloc function	`MAGIC.imageBase-512+60`
Kernel.VirtualFree function	`MAGIC.imageBase-512+0`
Shell.CommandLineToArgvW function	`MAGIC.imageBase-512+72`

Printing a character to standard output can be implemented something like the following, which does not evaluate the `result` value set by the system:

```
public static void printChar(int c) {
  int result=0;
  MAGIC.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 (push pointer to result to the stack)
  MAGIC.inline(0x6A, 0x01); //push byte 1 (Number of characters to output)
  MAGIC.inline(0x8D, 0x45); MAGIC.inlineOffset(1, c); //lea eax,c
  MAGIC.inline(0x50); //push eax (Push pointer to character)
  //Push handle for standard output
  MAGIC.inline(0xFF, 0x35); MAGIC.inline32(MAGIC.imageBase-512+4);
  //Call WriteFile function
  MAGIC.inline(0xFF, 0x15); MAGIC.inline32(MAGIC.imageBase-512+32);
}
```

A call to the LoadLibraryA function can be programmed as follows:

```
int addr=MAGIC.addr(libraryNameZ[0]), handle;
//Push pointer to library name
MAGIC.inline(0xFF, 0x75); MAGIC.inlineOffset(1, addr);
//Call LoadLibraryA
MAGIC.inline(0xFF, 0x15); MAGIC.inline32(MAGIC.imageBase-512+52);
//Store returned handle in variable handle
```

```
MAGIC.inline(0x89, 0x45); MAGIC.inlineOffset(1, handle);
... //From this point on the library handle, if valid, is useable
```

Just as under Linux, Windows functions can also be declared as native. Because, however, the stack frame required by Windows differs from that used by Linux, and the latter is used as the standard stackframe. The corresponding functions must be marked as Windows functions with the help of `@SJC.WinDLL`.

For example:

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

Access to the parameters passed to a program can be made through the parameter count stored at `MAGIC.imageBase-512+16` and the pointer stored at `MAGIC.imageBase-512+12`. Evaluation of the parameters could look something like the following:

```
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);
}
```

If the parameters are to be used within the program as SJC-compliant strings an explicit conversion must be made. Examples for all of the possibilities shown here can be found under [nat].

12 Examples

All of the following examples are designed for static compilation and 32-bit Intel code. Further examples can be found in the example system, PicOS (see [picos]), which supports 32- and 64-bit processors.

12.1 Use of Inlining

Example for `@SJC.Inline`:

```
public class String {
  private char[] value;
  private int count;
  @SJC.Inline
  public int length() {
    return count;
  }
}
```

If the `length()` method of a String is called, then, if possible, no call will be generated. Rather, the required opcodes to read the instance variable `count` will be inserted directly. With the optimizer these accesses are then further optimized, so that the programmed method "call" is somewhat comparable with a direct access to the variable `count`.

12.2 Access to I/O Ports

Example for `MAGIC.wIOs8(.)`, `MAGIC.rIOs8(.)`:

```
public class RTC {
  public int getCurrentHour() {
    MAGIC.wIOs8(0x70, (byte)4); //Select RTC/CMOS register 4
    return (int)MAGIC.rIOs8(0x71)&0xFF; //Read value
  }
}
```

The realtime clock on the PC is accessed through I/O ports beginning with 0x70. I/O port 0x70 is first used to select an internal register on the RTC, then port 0x71 can be used to access the selected register. In this example RTC register 4 is selected, which contains the current hour. A byte is then read out of this register, converted to an `int`, and, after the sign extension done by Java has been undone, returned.

12.3 Writing Interrupt Handlers

Example for `MAGIC.wMem32(.)`, `MAGIC.rMem32(.)`, `MAGIC.cast2Ref(.)`, `MAGIC.clssDesc(.)`, `MAGIC.mthdOff(.)`, `MAGIC.getCodeOff()`, `MAGIC.inline(.)`, `MAGIC.inline32(.)`, `@SJC.Interrupt`:

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

```
    @SJC.Interrupt
    private static void intrHandler() {
      //Only non-static: MAGIC.inline(0x8B, 0x3D); MAGIC.inline32(ADDR);
      //The code of the interrupt handler follows from here
    }
    protected void initNotStatic() { //only non-static compilation
      MAGIC.wMem32(ADDR, MAGIC.cast2Ref(MAGIC.clssDesc("Interrupts")));
    }
  }
```

The **intrHandler** is marked as an interrupt handler by **@SJC.Interrupt**. Because of this, instead of the normal, lightweight method framework, the compiler generates a framework appropriate for an interrupt handler, including the saving and restoration of all integer registers (if **floats** are used both inside and outside interrupt routines, saving and restoration of FPU registers and FPU state must be carried out by the programmer). For dynamically relocateable code (compiler option **-m**) initialization of class context (for example, as in the code in the lines that have been commented out) would also be necessary. Furthermore, in this case, before the occurence of the first interrupt, the address of the class descriptor for the Interrupts class would need to be stored at the address **ADDR**, in a manner similar to that shown in the method **initNotStatic(.)**

12.4 Use of STRUCTs

Example for **MAGIC.cast2Struct(.)**, **MAGIC.wMem16(.)**:

```
  public class VidChar extends STRUCT {
    public byte ascii, color;
  }
  public class VidMem extends STRUCT {
    @SJC(offset=0,count=2000)
    public VidChar[] expl;
    @SJC(offset=0,count=2000)
    public short[] chars;
  }
  public class Tester {
    public void test() {
      VidMem m; //declare STRUCT
      m=(VidMem)MAGIC.cast2Struct(0xB8000); //Define STRUCT's position
        MAGIC.wMem16(0xB8000+15*2, (char)0x0748); //'H' with the appropriate
color at position 15
      m.chars[16]=(char)0x0769; //'i' with the appropriate color at position 16
      m.expl[17].ascii=(byte)'!'; //output '!' at position 17
      m.expl[17].color=(byte)0x07; //Set color for position 17
    }
  }
```

The structure **VidChar** represents the structure of a character in text-mode video memory. The first byte, at offset 0, is interpreted by the graphics card as an ASCII code, the second byte gives its color. All in all, text-mode video memory comprises 2000 such characters in 80x25 mode, which is

represented in the structure **VidMem** once as an array of VidChar's and once as an unstructured array of 16-bit values. Both arrays begin at offset 0 in video memory, which is assumed to begin at 0xb8000, thus the method **Tester.test()** places a VidMem structure at this address with **MAGIC.cast2Struct(.)**. After this, three different possible methods of writing a character to the screen are demonstrated. The first is a direct access to video memory with **MAGIC.wMem16(.)**, with this method, the base address and structure of video must be explicitly taken into consideration. The second method is writing to the VidMem structure with a precalculated value. By this method, after the **STRUCT** has been successfully initialized, the desired postion can be used directly as an array index. The last is a semantically structured access, where the character and its color are accessed by array position and name.

12.5 Initialization in Embedded Mode

Example for **MAGIC.rMem32(.)**, **MAGIC.wMem32(.)**, **MAGIC.embedded**, **MAGIC.getRamAddr()**, **MAGIC.getRamSize()**, **MAGIC.getRamInitAddr()**, **MAGIC.getConstMemorySize()**:

If initialized class variables are to be used in embedded mode, the memory block designated for class variables must be initialized before any class variables are used. For this purpose, a memory block is provided in the image generated by the compiler. This block must be copied linearly into RAM. The following code can accomplish this:

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

If, in addition to the initialized class variables, constant objects are to be copied (for example, for Harvard Architectures for uniform "dynamic" objects with the **-E** compiler option, see section 5.1), this can easily be accomplished by further copying of **MAGIC.getConstMemorySize()** bytes, as constant objects are allocated directly after the initialized class variables:

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

12.6 Handcoded Method Calls

Example for **MAGIC.inline(.)**, **MAGIC.getCodeOff()**:

In an object oriented system, as generated by SJC, references to a method to be called and the associated class descriptor may be required on a case by case basis. The following source code calls an arbitrary method when dynamically relocatable code is used:

```
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
  }
```

As type is not relevant in inline code, references to the instance and class descriptor can be used instead of the instance address and class descriptor address. Thus, the alternative signature

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

can also be used with identical inline code.

If a dynamic method is called, **clssAddr** must contain the type of the instance. If a static method is called, **clssAddr** must contain the declaring class (not any class that extends it). For static methods, all commands affecting the esi register (the second byte on the line with **push**, the line after **push** and the first byte on the line with **pop**) can be omitted. When static compilation is being used (that is, when **-m** is not being used, see chapter 5.1) all commands affecting the edi register (the first byte on the line with **push**, the line before **call**, and the second byte on the line with **pop**) can also be omitted. If objects have been streamlined by the use of the **-l** option, and if **SMthdBlock** does not contain any scalar variables, then **MAGIC.getCodeOff()** always returns 0, corresponding to the value **instScalarTableSize** for **SMthdBlock** in syminfo.txt (see section 5.2) and the first statement in the above source code can be removed. Thus, if only static methods are to be called in statically compiled code, this can be done with

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

12.7 TextualCall

Calling a method by means of text instead of by compiled code is used in various systems (see [oberon] or [plurix]), in order to offer a simple and lightweight user interface. For this, the method calling code from 12.6 is needed. The in-image symbol information generators described in chapter 10 deliver the information needed at run time to locate classes and methods.

As an example, given a static method **toBeCalled()** in the class **Kernel** in the package **kernel**, the method would normally be called in Java source code through a statement of the form:

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

With TextualCall, this call could be represented as

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

with the parameters specifying package, class, and method. As the parameters are normal strings, they can also be altered at runtime and thus the same code could call various methods. Using the rte-generator, (see section 10.2) the method **callByName** could be implemented as follows:

```
  private static void callByName(String pack, String unit, String mthd) {
    SClassDesc u;
    int mthdAddr;
    SPackage p=SPackage.root.subPacks;
    SMthdBlock m;
    while (p!=null) { //search through all packages
      if (pack.equals(p.name)) { //package found
        u=p.units; //select the first unit of the package
        while (u!=null) { //search through all units
```

```
        if (unit.equals(u.name)) { //class found
          m=u.mthds; //select first method of the unit
          while (m!=null) { //search through all methods
            if (mthd.equals(m.namePar)) { //method found
              if ((m.modifier&0x0010)==0)
                return; //error: method is not static
              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 was successful
            }
            m=m.nextMthd; //try next method in the current unit
          }
          return; //error, method not found
        }
        u=u.nextUnit; //try next unit in the current package
      }
      return; //error: class not found
    }
    p=p.nextPack; //try next package
  }
  //error: package not found
}
```

If the raw generator is being used (see section 10.1) then, instead of a search through the runtime objects, an analogous search through `symbols.RawInfo.info` should be carried out.

13 Bootloader

To aid comprehension and experimentation on the part of the user, a bootloader for self-contained programs (without an underlying operating system such as Windows, Linux, etc.) operating in 32-bit protected mode is provided here as commented source code. It can be assembled with, for example, yasm, which can be found under [yasm]. However, it is available already assembled in the executable package for SJC (see [sjc]). An appropriate bootconf.txt is given in chapter 2.

```
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 "read 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 References

[java] http://java.sun.com bzw. http://www.oracle.com/technetwork/java/index.html

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

[nat] http://www.fam-frenz.de/stefan/native.html

[picos] http://www.fam-frenz.de/stefan/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.fam-frenz.de/stefan/compiler.html

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