# E Das ELF-Binärformat

ELF steht für Executable and linkable format und wird als Dateiformat für ausführbare Dateien, Objektfiles und Bibliotheken verwendet. Unter Linux hat es sich seit mehreren Jahren als Standardformat etabliert und das in den Anfangszeiten verwendete a.out-Format abgelöst. Der besondere Vorteil von ELF liegt darin, dass auf praktisch allen vom Kernel unterstützten Architekturen das gleiche Dateiformat verwendet werden kann, was nicht nur die Erstellung von Userspace-Tools, sondern auch die Programmierung des Kerns selbst leichter macht – beispielsweise, wenn es darum geht, Laderoutinen für ausführbare Dateien zu erstellen. Die Tatsache, dass sich das Format der Dateien nicht voneinander unterscheidet, bedeutet allerdings nicht, dass Binärkompatibilität zwischen Programmen verschiedener Systemen bestehen würde, die beide ELF als Binärformat verwenden - beispielsweise FreeBSD und Linux: Obwohl beide die Daten in ihren Dateien gleich organisieren, bestehen dennoch Unterschiede beim Systemaufruf-Mechanismus oder auch bei der Semantik der vorhandenen Systemaufrufe, weshalb Programme von FreeBSD nicht ohne eine weitere Emulationsschicht unter Linux ausgeführt werden können (der umgekehrte Weg ist natürlich ebensowenig ohne Emulation möglich). Verständlicherweise können Binär-Programme nicht zwischen verschiedenen Architekturen ausgetauscht werden (beispielsweise können Linux-Binaries, die für Alpha-CPUs kompiliert wurden, nicht auf Sparc-Linux ausgeführt werden), da sich die zugrunde liegenden Architekturen völlig voneinander unterscheiden. Identisch ist in allen Fällen aber - dank des ELF-Formats - die Art und Weise, wie die vorhandenen Informationen über das Programm und seine Komponenten in der Binärdatei kodiert werden.

Linux verwendet das ELF-Format nicht nur für Userspace-Applikationen und -Bibliotheken, sondern auch zur Realisierung von Modulen. Der Kern selbst wird ebenfalls im ELF-Format erzeugt.

Das Elf-Format ist ein offenes Format, dessen Spezifikation frei verfügbar ist (das Dokument ist auch auf der Website zum Buch verfügbar). Der Aufbau dieses Anhangs orientiert sich an der Spezifikation und soll als Zusammenfassung der Aussagen dienen, die für unsere Zwecke relevant sind.

# E.1 Aufbau und Struktur

Wie Abbildung E.1 auf der nächsten Seite zeigt, besteht jede ELF-Datei aus verschiedenen Abschnitten, wobei zwischen Linkobjekten und ausführbaren Dateien unterschieden werden muss:

- Der ELF-Header enthält neben einigen Bytes, die zur eindeutigen Identifikation als ELF-Datei verwendet werden – Informationen über den genauen Typ der Datei, ihre Größe oder den Einsprungpunkt, an dem die Programmausführung nach dem Laden der Datei einsetzt.
- Die Programm-Headertabelle gibt dem System Hinweise, wie die Daten einer ausführbaren Datei im virtuellen Adressraum eines Prozesses angeordnet werden sollen. Außerdem wird geregelt, wie viele Sektionen in der Datei enthalten sind, wo sie sich befinden und welchem Zweck sie dienen.



Abbildung E.1: Grundlegender Aufbau von ELF-Dateien

- Die einzelnen Sektionen nehmen die unterschiedlichen Daten auf, die mit einer Datei verbunden sind, beispielsweise die Symboltabelle, den eigentlichen Binärcode oder feste Werte wie Zeichenketten oder numerische Konstanten, die das Programm verwendet.
- In der Sektions-Headertabelle sind zusätzliche Informationen über die einzelnen Sektionen enthalten.

readelf ist ein nützliches Tool, um die Struktur von ELF-Dateien zu analysieren, was wir anhand eines einfachen Beispielprogramms zeigen wollen:

```
#include<stdio.h>
int add (int a, int b) {
    printf("Zahlen werden addiert\n");
    return a+b;
}
int main() {
    int a,b;
    a = 3;
    b = 4;
    int ret = add(a,b);
    printf("Resultat: %u\n");
    exit(0);
}
```

Natürlich zählt das Programm nicht unbedingt zu den nützlichsten Mitgliedern seiner Gattung, kann aber dennoch als gutes Beispiel verwendet werden, indem sowohl eine ausführbare Datei wie auch ein Objektfile erzeugt werden:

```
\begin{tabular}{ll} wolfgang@meitner> & gcc & test.c & -o & test \\ wolfgang@meitner> & gcc & test.c & -c & -o & test.o \\ \end{tabular}
```

file zeigt, dass der Compiler zwei ELF-Dateien erzeugt hat – eine ausführbare Datei und eine relozierbare Objektdatei:

```
wolfgang@meitner> file test filetest: ELF 32-bit LSB executable, Intel 80386, version 1, dynamically linked (uses shared libs), not stripped wolfgang@meitner> file test.o test.o: ELF 32-bit LSB relocatable, Intel 80386, version 1, not stripped
```

## E.1.1 ELF-Header

Wir verwenden readelf, um die Bestandteile der beiden Dateien zu untersuchen. <sup>1</sup> Zunächst soll der ELF-Header betrachtet werden:

```
wolfgang@meitner> readelf test
ELF Header:
          7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00
 Magic:
 Class:
                                     ELF32
 Data:
                                     2's complement, little endian
 Version:
                                     1 (current)
 OS/ABI:
                                     UNIX - System V
 ABI Version:
                                     EXEC (Executable file)
 Type:
                                     Intel 80386
 Machine:
 Version:
                                     0x1
                                     0x80482d0
 Entry point address:
                                     52 (bytes into file)
 Start of program headers:
 Start of section headers:
                                     10148 (bytes into file)
                                     0x0
 Flags:
 Size of this header:
                                     52 (bytes)
 Size of program headers:
                                     32 (bytes)
 Number of program headers:
                                     6
 Size of section headers:
                                     40 (bytes)
 Number of section headers:
                                     29
 Section header string table index: 26
```

Gleich zu Beginn der Datei finden sich vier Identifikations-Bytes: Ein Zeichen mit ASCII-Code 0x7f wird von den ASCII-Werten der Zeichen E (0x45), L (0x4c) und F (0x46) gefolgt, woran alle ELF-verarbeitenden Tools erkennen, dass es sich um eine Datei im gewünschten Format handelt. Außerdem finden sich einige Angaben zur verwendeten Architektur – in diesem Fall einem Pentium III-Rechner, einer IA32-kompatiblen Maschine. Die Klasseninformation (ELF32) gibt völlig zu Recht an, dass es sich dabei um eine 32-Bit-Maschine handelt (auf Alphas, IA-64, Sparc64 und anderen 64-Bit-Plattformen wäre in dem Feld der Wert ELF64 zu finden).

Der Typ der Datei ist EXEC, was bedeutet, dass es sich um eine ausführbare Datei handelt. Mit Hilfe des Versionsfeldes kann zwischen verschiedenen Revisionen des ELF-Standards unterschieden werden; da die initiale Version 1 allerdings immer noch aktuell ist, wird dieses Feature momentan nicht benötigt. Ebenfalls finden sich einige Angaben über Größe und Indexpositionen verschiedener Bestandteile einer ELF-Datei, auf deren genaue Bedeutung wir später eingehen werden. Da die Größe der Abschnitte nicht statisch vorgeschrieben ist, sondern sich je nach Programm dynamisch ändert, müssen die entsprechenden Daten im Header zugänglich gemacht werden.

Welche Felder ändern sich, wenn man die Objektdatei anstelle der ausführbaren Datei betrachtet? Der Einfachheit halber zeigen wir nur die interessanten Felder, die sich in der Ausgabe von readelf ändern:

```
wolfgang@meitner> readelf -h test.o
...
Type: REL (Relocatable file)
...
Start of program headers: 0 (bytes into file)
...
Size of program headers: 0 (bytes)
Number of program headers: 0
```

<sup>1</sup> Neben den hier gezeigten Argumenten besitzt das Programm noch einige weitere Kommandozeilenoptionen, die in der Manual-Page readelf (1) dokumentiert sind bzw. mit readelf --help angezeigt werden können.

Der Dateityp wird als REL ausgewiesen; es handelt es sich also um eine relozierbare Datei, deren Code an eine beliebige Stelle verschoben werden kann.<sup>2</sup> Ebensowenig besitzt die Datei eine Programmheader-Tabelle, die für Linkobjekte nicht benötigt wird; die betreffenden Größen sind daher alle auf 0 gesetzt.

# **E.1.2** Programmheader-Tabelle

Da die Objektdatei keine Programmheader-Tabelle enthält, können wir sie nur in der ausführbaren Datei untersuchen:

```
wolfgang@meitner> readelf -l test
Elf file type is EXEC (Executable file)
Entry point 0x80482d0
There are 6 program headers, starting at offset 52
Program Headers:
                 Offset
                         VirtAddr
                                    PhysAddr
                                               FileSiz MemSiz Flg Align
  Type
                 0x000034 0x08048034 0x08048034 0x000c0 0x000c0 R E 0x4
  INTERP
                 0x0000f4 0x080480f4 0x080480f4 0x00013 0x00013 R
      [Requesting program interpreter: /lib/ld-linux.so.2]
  LOAD
                 0x000000 0x08048000 0x08048000 0x0046d 0x0046d R E 0x1000
  LOAD
                 0x000470 0x08049470 0x08049470 0x00108 0x0010c RW 0x1000
  DYNAMIC
                 0x000480 0x08049480 0x08049480 0x000c8 0x000c8 RW 0x4
  NOTE
                 0x000108 0x08048108 0x08048108 0x00020 0x00020 R
 Section to Segment mapping:
  Segment Sections...
   00
   01
          .interp
          .interp .note.ABI-tag .hash .dynsym .dynstr .gnu.version
   02
          .gnu.version_r .rel.dyn .rel.plt .init .plt .text .fini .rodata
   03
          .data .eh_frame .dynamic .ctors .dtors .jcr .got .bss
   04
          .dvnamic
   05
          .note.ABI-tag
```

Unter der Überschrift Program headers werden sechs Abschnitte aufgelistet, aus denen sich das fertige Programm im Speicher zusammensetzt. Für jeden Abschnitt werden Größe und Position im virtuellen und physikalischen Adressraum<sup>3</sup>, Flags, Zugriffsberechtigungen und Ausrichtungsvorschrift (Alignment) angegeben; zusätzlich wird ein Typ spezifiziert, der den Abschnitt genauer charakterisiert. Das Beispielprogramm verwendet fünf verschiedene Typen, die folgende Bedeutung haben:

- PHDR nimmt die Programmheader-Tabelle im Speicher auf.
- INTERP gibt an, welcher "Interpreter" aufgerufen werden muss, nachdem das Programm von der ausführbaren Datei in den Speicher transferiert wurde. Mit Interpreter ist hier *nicht* gemeint, dass der Inhalt der Binärdatei von einem zusätzlichen Programm interpretiert werden muss, wie es beispielsweise bei Java-Bytecode und der Java Virtual Machine der Fall ist.

<sup>2</sup> Dies bedeutet vor allem, dass nur relative Sprungziele anstelle absoluter Ziele im Assemblercode verwendet werden können.

<sup>3</sup> Die Angaben zur physikalischen Adresse werden ignoriert, da diese bekanntlich dynamisch durch den Kern zugewiesen wird, je nachdem, welche physikalischen Speicherseiten in die entsprechenden Positionen des virtuellen Adressraums eingeblendet werden. Die Angaben sind nur auf Maschinen von Bedeutung, die keine MMU und daher keinen virtuellen Speicher besitzen, beispielsweise kleine Embedded-Prozessoren.

Vielmehr bezieht sich die Angabe auf ein Programm, das zum Auflösen der unaufgelösten Referenzen dient, indem zusätzlich benötigte Bibliotheken eingebunden werden.

Normalerweise wird hierfür /lib/ld-linux.so.2, /lib/ld-linux-ia-64.so.2 etc. verwendet, die die benötigten dynamischen Bibliotheken in den virtuellen Adressraum einfügen. Für beinahe jedes Programm muss die C-Standardbibliothek libc.so eingeblendet werden; hinzu kommen verschiedene Bibliotheken wie GTK, die mathematische Bibliothek, libjpeg

- LOAD bezeichnet einen Abschnitt, der aus der Binärdatei in den virtuellen Adressraum eingeblendet wird. Darin finden sich konstante Daten (beispielsweise Zeichenketten), der Objektcode des Programms etc.
- Abschnitte des Typs DYNAMIC enthalten Informationen, die vom dynamischen Linker (also dem in INTERP angegebenen Interpreter) verwendet werden.
- NOTE speichert herstellerspezifische Informationen, auf die wir allerdings nicht genauer eingehen wollen.

Die verschiedenen Abschnitte im virtuellen Adressraum werden mit Daten bestimmter Sektionen gefüllt, die sich in der ELF-Datei befinden. Der zweite Teil der readelf-Ausgabe spezifiziert daher, welche Sektionen in welchen Abschnitt geladen werden (Section to Segment Mapping).<sup>4</sup>

Andere Plattformen verwenden prinzipiell die gleiche Vorgehensweise, wobei aber je nach Architektur andere Sektionen in die einzelnen Bereiche eingeblendet werden, wie man am Beispiel von IA-64 sieht:

```
wolfgang@meitner> readelf -l test_ia64
Elf file type is EXEC (Executable file)
Entry point 0x400000000000004e0
There are 7 program headers, starting at offset 64
Program Headers:
              Offset
                              VirtAddr
                                              PhysAddr
 Туре
                              MemSiz
                                               Flags Align
              FileSiz
 PHDR
              0x00000000000188 0x00000000000188 R E
 INTERP
              0x0000000000001c8 0x400000000001c8 0x400000000001c8
              0x000000000000018 0x000000000000018 R
     [Requesting program interpreter: /lib/ld-linux-ia64.so.2]
 LOAD
              0x0000000000009f0 0x000000000009f0
 LOAD
              0x0000000000009f0 0x600000000009f0 0x600000000009f0
              0x000000000000270 0x000000000000280
 DYNAMIC
              0x0000000000009f8 0x600000000009f8 0x600000000009f8
              0x0000000000001a0 0x0000000000001a0 RW
 NOTE
              0x0000000000001e0 0x400000000001e0 0x400000000001e0
              IA_64_UNWIND
              \tt 0x00000000000009a8 \ 0x4000000000009a8 \ 0x4000000000009a8
              0x000000000000048 0x000000000000048
                                                     8
 Section to Segment mapping:
 Segment Sections...
```

<sup>4</sup> Achtung: Dabei handelt es sich nicht um Segmente in dem Sinn, wie sie bei IA32-Prozessoren zur Implementierung verschiedener isolierter Bereiche des virtuellen Adressraums verwendet werden, sondern um einfache Bereiche im Adressraum.

```
01 .interp
02 .interp .note.ABI-tag .hash .dynsym .dynstr .gnu.version .gnu.version_r
.rela.IA_64.pltoff .init .plt .text .fini .rodata .opd
.IA_64.unwind_info .IA_64.unwind
03 .data .dynamic .ctors .dtors .jcr .got .IA_64.pltoff .sdata .sbss .bss
04 .dynamic
05 .note.ABI-tag
06 .IA_64.unwind
```

Neben der Verwendung von 64-Bit-Adressen fällt vor allem auf, dass ein weiterer Abschnitt gegenüber der IA-32-Variante hinzukommt, der den Typ IA-64\_UNWIND besitzt: Darin werden *Unwind-Informationen* gespeichert, die zur Analyse von Stack Frames verwendet werden (beispielsweise dann, wenn ein Backtrace generiert werden soll), da dies auf IA-64 aus Architekturspezifischen Gründen nicht mehr durch eine simple Analyse des Stackinhalts möglich ist.<sup>5</sup> Auf die genaue Bedeutung der verschiedenen Sektionen werden wir im nächsten Abschnitt eingehen.

Achtung: Die Abschnitte können sich auch überschneiden, wie aus der Readelf-Aufgabe für IA-32 sichtbar wird: Abschnitt 02 vom Typ LOAD reicht von 0x08048000 bis 0x8048000 + 0x0046d = 0x0804846d. Er enthält das Segment .note. ABI-tag. Der selbe Bereich im virtuellen Adressraum wird allerdings auch zur Realisierung des Abschnitts 06 (vom Typ NOTE) verwendet, der den Bereich von 0x08048108 bis 0x08048108 + 0x00020 = 0x08048128 belegt und damit *innerhalb* von Abschnitt 02 liegt. Dieses Verhalten wird vom Standard aber explizit erlaubt.

### E.1.3 Sektionen

Der Inhalt von Abschnitten wird beschrieben, indem die Sektionen angegeben werden, deren Daten hineinkopiert werden sollen. Zur Verwaltung der Sektionen einer Datei dient eine weitere Tabelle, die als *Sektionsheader-Tabelle* bezeichnet wird, wie Abbildung E.1 gezeigt hat. Wiederum kann readelf verwendet werden, um die vorhandenen Sektionen einer Datei anzuzeigen:

```
wolfgang@meitner> readelf -S test.o
There are 10 section headers, starting at offset 0x114:
```

```
Section Headers:
                                                               ES Flg Lk Inf Al
  [Nr] Name
                        Type
                                        Addr
                                                 Nff
                                                        Size
                                        00000000 000000 000000 00
                                                                           0 0
  [0]
                        NUL.L.
                                                                        0
                                                                    AX
                        PROGBITS
  [ 1] .text
                                        00000000 000034 000065 00
                                                                        0
                                                                            0
  [ 2] .rel.text
                        REL.
                                        00000000 000374 000030 08
                                                                        8
                                                                           1
                        PROGBITS
                                                                    WA
  [ 3] .data
                                        00000000 00009c 000000 00
                                                                            0
  [ 4] .bss
                        NORTES
                                        00000000 00009c 000000 00
                                                                    WΑ
                                                                        0
                                                                            0
                        PROGBITS
                                        00000000 00009c 000025 00
                                                                        0
  [5] .rodata
                                                                            0
  [ 6] .comment
                        PROGBITS
                                        00000000 0000c1 000012 00
                                                                        0
                                                                            0
                                                                              1
  [7] .shstrtab
                        STRTAB
                                        00000000 0000d3 000041 00
                                                                        0
                                                                            0
                                                                              1
  [8] .symtab
                        SYMTAR
                                        00000000 0002a4 0000b0 10
                                                                        9
                                                                               4
                                        00000000 000354 00001d 00
  [ 9] .strtab
                        STRTAR
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings)
  I (info), L (link order), G (group), x (unknown)
  O (extra OS processing required) o (OS specific), p (processor specific)
```

<sup>5</sup> IA-64 verwendet *Register Stacks*, um die lokalen Variablen einer Prozedur zu speichern. Der Prozessor reserviert dazu automatisch ein "Fenster" aus dem umfangreichen Gesamtregistersatz des Prozessors. Je nach Bedarf können Teile dieser Register in den Speicher ausgelagert werden, was für das Programm aber transparent ist. Da die Größe der Register Stacks für jede Prozedur unterschiedlich ist und je nach Aufrufkette verschiedene Register ausgelagert sein können, kann ein Backtrace nicht mehr durch einfaches Zurückverfolgen der Stack Frames über die Framepointer erfolgen, wie es bei den meisten anderen Architekturen möglich ist. Erst die zusätzlich gespeicherten Unwind-Informationen, auf die wir allerdings nicht genauer eingehen wollen, ermöglichen dies.

Das angegebene Offset (in diesem Fall 0x114) bezieht sich die Binärdatei. Die Sektionsinformationen müssen für ausführbare Dateien nicht in das fertige Prozessimage kopiert werden, das im virtuellen Adressraum angelegt wird. Die Informationen finden sich allerdings immer in der Binärdatei.

Jede Sektion ist mit einem Typ versehen, der die Semantik der enthaltenen Daten festlegt. Die wichtigsten Werte in unserem Beispiel sind PROGBITS (Informationen, deren Interpretation dem Programm obliegt, beispielsweise der Binärcode <sup>6</sup>) und SYMTAB (Symboltabelle), REL (Relokationsinformationen). STRTAB wird verwendet, um Strings zu speichern, die für das ELF-Format selbst von Bedeutung, aber nicht direkt mit dem Programm verknüpft sind, beispielsweise die symbolischen Bezeichnungen von Sektionen wie .text oder .comment.

Ebenfalls wird für jede Sektion das Offset innerhalb der Binärdatei und ihr Umfang festgehalten. Das Adressfeld könnte verwendet werden, um anzugeben, an welche Position im virtuellen Adressraum eine Sektion geladen werden soll; da es sich bei vorherigem Beispiel allerdings um ein Linkobjekt handelt, ist die Zieladresse nicht festgelegt, weshalb 0 als Wert eingesetzt wird. Flags legen fest, wie auf die einzelnen Sektionen zugegriffen werden darf oder wie sie behandelt werden müssen. Vor allem das A-Flag ist interessant; es regelt, ob die Daten einer Sektion beim Laden der Datei in den virtuellen Adressraum kopiert werden sollen oder nicht.

Obwohl Sektionen beliebige Namen tragen dürfen, finden sich unter Linux (und auch allen anderen Unix-Derivaten, die das ELF-Binärformat verwenden) einige Standardsektionen, die teilweise zwingend notwendig sind. Überall enthalten ist eine Sektion der Bezeichnung .text, die den Binärcode und damit die Programm-Informationen, die mit der Datei verknüpft sind, enthält. In .rel.text finden sich Relokationsinformationen für den Textabschnitt, auf die wir weiter unten genauer eingehen werden.

Die ausführbare Datei enthält einige zusätzliche Informationen:

```
wolfgang@meitner> readelf -S test There are 29 section headers, starting at offset 0x27a4:
```

Section Headers:										
[Nr]	Name	Туре	Addr	Off	Size	ES	Flg	Lk	${\tt Inf}$	Al
[ 0]		NULL	00000000	000000	000000	00		0	0	0
[ 1]	.interp	PROGBITS	080480f4	0000f4	000013	00	Α	0	0	1
[ 2]	.note.ABI-tag	NOTE	08048108	000108	000020	00	Α	0	0	4
[ 3]	.hash	HASH	08048128	000128	000030	04	Α	4	0	4
[ 4]	.dynsym	DYNSYM	08048158	000158	000070	10	Α	5	1	4
[ 5]	.dynstr	STRTAB	080481c8	0001c8	00005e	00	Α	0	0	1
[ 6]	.gnu.version	VERSYM	08048226	000226	00000e	02	Α	4	0	2
[7]	.gnu.version_r	VERNEED	08048234	000234	000020	00	Α	5	1	4
[ 8]	.rel.dyn	REL	08048254	000254	800000	80	Α	4	0	4
[ 9]	.rel.plt	REL	0804825c	00025c	000018	80	Α	4	b	4
[10]	.init	PROGBITS	08048274	000274	000018	00	AX	0	0	4
[11]	.plt	PROGBITS	0804828c	00028c	000040	04	AX	0	0	4
[12]	.text	PROGBITS	080482d0	0002d0	000150	00	AX	0	0	16
[13]	.fini	PROGBITS	08048420	000420	00001e	00	AX	0	0	4
[14]	.rodata	PROGBITS	08048440	000440	00002d	00	Α	0	0	4
[15]	.data	PROGBITS	08049470	000470	00000c	00	WA	0	0	4
[16]	.eh_frame	PROGBITS	0804947c	00047c	000004	00	WA	0	0	4
[17]	.dynamic	DYNAMIC	08049480	000480	0000c8	80	WA	5	0	4
[18]	.ctors	PROGBITS	08049548	000548	800000	00	WA	0	0	4
[19]	.dtors	PROGBITS	08049550	000550	800000	00	WA	0	0	4

<sup>6</sup> Der Binärcode eines Programms wird häufig auch als *Text* bezeichnet. Nichtsdestotrotz handelt es sich dabei natürlich um binäre Informationen, wie es für Maschinencode üblich ist.

<sup>7</sup> Sektionen, deren Name mit einem Punkt beginnt, werden vom System selbst verwendet. Wenn eine Applikation eigene Sektionen definieren will, sollten diese keinen Punkt als Präfix besitzen, um Konflikte mit den Systemsektionen zu vermeiden.

```
[20] .jcr
                        PROGBITS
                                        08049558 000558 000004 00
                                                                            0
                                                                               4
  [21] .got
                        PROGBITS
                                        0804955c 00055c 00001c 04
                                                                            0
  [22] .bss
                        NOBITS
                                        08049578 000578 000004 00
                                                                            0
  [23] .stab
                        PROGBITS
                                        00000000 000578 0007ь0 Ос
  [24] .stabstr
                        STRTAB
                                        00000000 000d28 001933 00
                                                                        0
                        PROGBITS
  [25] .comment
                                        00000000 00265b 00006c 00
                                                                            0
  [26] .shstrtab
                                        00000000 0026c7 0000dd 00
                        STRTAB
                                                                        0
                                                                            0
  [27] .symtab
                                        00000000 002c2c 000450 10
                        SYMTAB
                                                                       28
                                                                          31
 [28] .strtab
                        STRTAB
                                        00000000 00307c 0001dd 00
Kev to Flags:
 W (write), A (alloc), X (execute), M (merge), S (strings)
 I (info), L (link order), G (group), x (unknown)
 O (extra OS processing required) o (OS specific), p (processor specific)
```

Anstelle der zehn Sektionen der Objektdatei besitzt die ausführbare Datei 29 Sektionen, von denen für unsere Zwecke aber nicht alle interessant sind. Wichtig sind vor allem:

- .interp enthält den Dateinamen des Interpreters als ASCII-Zeichenkette.
- .data enthält initialisierte Daten, die zu den normalen Daten des Programms beitragen und während seiner Laufzeit modifiziert werden können (beispielsweise vorinitialisierte Strukturen).
- .rodata enthält read-only-Daten, die zwar gelesen, aber nicht modifiziert werden dürfen. Der Compiler verpackt beispielsweise alle statischen Zeichenketten, die in printf-Anweisungen auftreten, in diese Sektion.
- .init und .fini enthalten Code für die Prozessinitialisierung und -termination, die üblicherweise automatisch vom Compiler hinzugefügt und nicht vom Applikationsprogrammierer kontrolliert werden.
- .hash ist eine Hashtabelle, mit deren Hilfe schnell auf alle Einträge der Symboltabelle zugegriffen werden kann, ohne dass eine lineare Suche über alle darin enthaltenen Elemente durchgeführt werden muss.

Die Adressfelder der Sektionen enthalten im Fall der ausführbaren Datei gültige Werte, da der Code an bestimmte festgelegte Positionen im virtuellen Adressraum transferiert werden soll (unter Linux wird für Anwendungen der Speicherbereich ab 0x08000000 verwendet).

## **E.1.4** Symboltabelle

Die Symboltabelle ist ein wichtiger Bestandteil jeder ELF-Datei, da sich darin alle (globalen) Variablen und Funktionen befinden, die von einem Programm implementiert oder verwendet werden. Man spricht von *undefinierten* Symbolen, wenn sich ein Programm auf ein Symbol bezieht, das nicht in seinem eigenen Code definiert wird (in unserem Beispiel die Funktion printf, die sich in der C-Standardbibliothek findet); Referenzen dieser Art müssen entweder beim statischen Linken mit anderen Objektmodulen bzw. Bibliotheken oder beim dynamischen Linken während des Ladens (mit Hilfe von ld-linux.so) aufgelöst werden. Das nm-Tool kann verwendet werden, um eine Liste aller Symbole zu erzeugen, die ein Programm definiert und verwendet:

```
wolfgang@meitner> nm test.c
00000000 T add
U exit
000001a T main
U printf
```

Die linke Spalte zeigt den Symbolwert an; hier handelt es sich um die Position in der Objektdatei, an der sich die Definition des Symbols findet. In unserem Beispiel treten zwei verschiedene Symboltypen auf: Funktionen, die im Textsegment definiert werden (wie das Kürzel T angibt) und undefinierte Referenzen, die durch ein U gekennzeichnet werden. Die undefinierten Referenzen besitzen logischerweise keinen Symbolwert.

Wesentlich mehr Symbole finden sich in der ausführbaren Datei. Da die meisten davon aber vom Compiler automatisch generiert und für interne Zwecke des Laufzeitsystems verwendet werden, zeigen wir nur die bereits aus der Objektdatei bekannten Elemente:

```
wolfgang@meitner> nm test
08048388 T add
U exit@@GLIBC_2.0
080483a2 T main
U printf@@GLIBC_2.0
```

exit und printf sind weiterhin undefiniert, mittlerweile wurde aber ein Hinweis hinzugefügt, aus welcher Version der GNU-Standardbibliothek die Funktionen mindestens entnommen werden müssen (im vorliegenden Fall darf beispielsweise keine Version kleiner als 2.0 verwendet werden, was bedeutet, dass das Programm mit Libc5 und Libc4 nicht funktioniert<sup>8</sup>). Die vom Programm definierten Symbole add und main wurden mittlerweile aber an feste Positionen im virtuellen Adressraum verschoben, an die ihr Code beim Laden der Datei positioniert wird.

Wie wird der Symboltabellen-Mechanismus im ELF-Format realisiert? Drei Sektionen werden zur Aufnahme der relevanten Daten verwendet:

- . symtab stellt die Verbindung zwischen dem Namen eines Symbols und seinem Wert her. Der Symbolname wird allerdings nicht direkt als Zeichenkette, sondern indirekt als Kennzahl kodiert, die als Index in ein Stringarray verwendet wird.
- strtab nimmt das Stringarray auf.
- . hash speichert eine Hashtabelle, mit deren Hilfe ein Symbol schnell gefunden werden kann.

Vereinfacht ausgedrückt besteht jeder Eintrag der .symtab-Sektion aus zwei Elementen: Der Position des Namens in der Stringtabelle und dem zugeordneten Wert (wie wir weiter unten sehen werden, ist die Situation in der Realität etwas komplizierter, da mehr Informationen für jeden Eintrag berücksichtigt werden müssen).

# E.1.5 Stringtabellen

Abbildung E.2 auf der nächsten Seite zeigt, wie Stringtabellen zur Verwaltung von Zeichenketten für ELF-Dateien realisiert werden.

Das erste Byte der Tabelle ist immer ein Nullbyte, anschließend folgen die durch Nullbytes voneinander getrennten Strings.

Um einen String zu selektieren, muss eine Position angegeben werden, die als Index ins Array dient: Dadurch werden alle Zeichen selektiert, die sich vor dem nächsten Nullbyte befinden (wenn die Position eines Nullbytes als Index verwendet wird, entspricht dies einem leeren String). Dies ermöglicht (wenn auch sehr eingeschränkt) die Verwendung von Teilstrings, indem nicht die Startposition, sondern irgendeine Position in der Mitte eines Strings als Index gewählt wird.

<sup>8</sup> Die Versionsnummerierung erscheint etwas seltsam, ist aber korrekt: Libc4 und Libc5 waren spezielle C-Standardbibliotheken für Linux; Glibc 2.0 war die erste systemübergreifende Variante der Bibliothek, die die alten Versionen abgelöst hat.

	0	1	2	3	4	5	6	7	8	9
0	0/	a	d	d	\0	Н	у	p	е	r
10	k	a	s	t	е	n	\0	m	a	i
20	n	\0								

Abbildung E.2: Stringtabelle für ELF-Dateien

. strtab ist nicht die einzige Stringtabelle, die sich standardmäßig in einer ELF-Datei findet: .shstrtab wird verwendet, um die textuellen Bezeichnungen der einzelnen Sektionen (beispielsweise .text) in der Datei unterzubringen.

# **E.2** Datenstrukturen im Kern

Das ELF-Dateiformat wird im Kern an zwei Stellen verwendet: Zum einen zur Behandlung ausführbarer Dateien und Bibliotheken, zum anderen zur Implementierung von Modulen. Beide Bereiche verwenden zwar unterschiedlichen Code, um die Daten nach ihren Vorstellungen zu lesen und zu manipulieren, greifen aber auf die gleichen Datenstrukturen zurück, die wir in diesem Abschnitt vorstellen wollen. Basis ist die Headerdatei <elf.h>, in der die Vorgaben des Standards praktisch unverändert umgesetzt werden.

## **E.2.1** Datentypen

Da ELF ein prozessor- und architekturunabhängiges Format ist, kann es sich nicht auf eine bestimmte Wortlänge oder Datenausrichtung (Little oder Big Endian) verlassen – zumindest für die Elemente der Datei, die auf allen Systemen gelesen und verstanden werden können sollen (Maschinencode, wie er sich beispielsweise im .text-Segment findet, ist natürlich in der Darstellungsweise des Hostsystems gespeichert, um keine umständlichen Konvertierungsoperationen vornehmen zu müssen). Der Kern definiert deshalb einige Datentypen, die auf allen Architekturen die gleiche Bitzahl aufweisen:

```
<elf.h>
          /* 32-bit ELF base types. */
          typedef __u32
                          Elf32_Addr;
                          Elf32_Half;
          typedef __u16
          typedef __u32
                          Elf32 Off:
          typedef __s32
                          Elf32_Sword;
          typedef __u32
                          Elf32_Word;
          /* 64-bit ELF base types. */
          typedef __u64
                           Elf64_Addr;
          typedef __u16
                          Elf64_Half;
          typedef __s16
                          Elf64_SHalf;
          typedef __u64
                          Elf64_Off;
          typedef __s32
                          Elf64_Sword;
          typedef __u32
                          Elf64_Word;
          typedef __u64
                          Elf64_Xword;
          typedef __s64
                          Elf64_Sxword;
```

Da der Architektur-spezifische Code ohnehin Integer-Datentypen mit klar definierter Signedness und Bitanzahl definieren muss, können die vom ELF-Standard geforderten Datentypen ohne größeren Aufwand als direkte Typedefs implementiert werden.

## E.2.2 Header

Für die verschiedenen Header, die im ELF-Format auftreten, steht je eine Datenstruktur für 32und 64-Bit-Rechner zur Verfügung.

#### **ELF-Header**

Der Identifikationsheader wird auf 32-Bit-Architekturen durch folgende Datenstruktur repräsentiert:

```
typedef struct elf32_hdr{
                                                                                           <elf.h>
 unsigned char e_ident[EI_NIDENT];
 Elf32_Half
                e_type;
 Elf32_Half
                e machine:
 Elf32 Word
                e_version;
 Elf32 Addr
                e_entry; /* Entry point */
 Elf32 Off
                e_phoff;
 Elf32_Off
                e_shoff;
 Elf32_Word
                e_flags;
 Elf32 Half
                e ehsize:
 Elf32_Half
                e_phentsize;
 Elf32 Half
                e_phnum;
 Elf32 Half
                e shentsize:
 Elf32_Half
                e_shnum;
                e_shstrndx;
 Elf32 Half
} Elf32_Ehdr;
```

- e\_ident nimmt 16 (EI\_NIDENT) Bytes auf, die auf allen Architekturen durch den Datentyp char repräsentiert werden. Die ersten vier halten ein Nullbyte und die Buchstaben 'E', 'L' und 'F' fest, wie wir bereits weiter oben besprochen haben. Zusätzlich werden einige andere Bitpositionen mit bestimmten Bedeutungen belegt:
  - EI\_CLASS (4) gibt die "Klasse" der Datei an, die zur Unterscheidung zwischen 32und 64-Bit-Files verwendet wird. Momentan sind daher die Werte ELFCLASS32 und ELFCLASS64 definiert.<sup>9</sup>
  - EI\_DATA (5) legt fest, welche Endianess das Format verwendet: ELFDATA2LSB steht für least significant byte (und damit Little Endian), während ELFDATA2MSB für most significant byte und damit Big Endian steht.
  - EI\_VERSION (6) gibt die Revision des ELF-Headers (dessen Version potentiell unabhängig von der Version des Datenteils ist) an, nach dem sich die Datei richtet. Momentan ist nur EV\_CURRENT erlaubt, was der ersten Version entspricht.
  - Ab Position EI\_PAD (7) wird der Identifikationsteil des Headers mit Nullbytes aufgefüllt, da die verbleibenden Positionen (noch) nicht benötigt werden.
- e\_type unterscheidet zwischen verschiedenen Typen von ELF-Dateien, die in Tabelle E.1 vorgestellt werden.
- e\_machine legt fest, für welche Architektur der Dateiinhalt bestimmt ist. Tabelle E.2 zeigt die verschiedenen verfügbaren Möglichkeiten, wobei nur die von Linux unterstützten Varianten berücksichtigt werden.

<sup>9</sup> Der Elf-Standard definiert in vielen Fällen – auch hier – Konstanten, die für "undefiniert" oder "ungültig" stehen, die wir in der folgenden Beschreibung der Einfachheit halber aber außer Acht lassen werden.

Tabelle E.1: ELF-Dateitypen

Wert	Bedeutung
ET_REL	Relozierbare Datei (Objektdatei)
ET_EXEC	Ausführbare Datei
ET_DYN	Dynamische Bibliothek
$ET_CORE$	Coredump (Speicherabzug)

Tabelle E.2: Von ELF unterstützte Architekturen

Wert	Bezeichnung
EM_SPARC	32-Bit-Sparc
EM_SPARC32PLUS	32-Bit-Sparc in Ausführung "v8 Plus"
EM_SPARCV9	64-Bit-Sparc
EM_386 und ELF_486	IA-32
EM_IA_64	IA-64
EM_X86_64	AMD64
EM_68K	Motorola 68k
EM_MIPS	MIPS
EM_PARISC	Hewlet-Packard PA-Risc
EM_PPC	PowerPC
EM_PPC64	PowerPC 64
EM_SH	Hitachi SuperH
EM_S390	IBM S390
EM_S390_OLD	Früherer Interimswert für S390
EM_CRIS	Axis Communications Cris
EM_V850	NEC v850
EM_H8_300H	Hitachi H8/300H
EM_ALPHA	Alpha AXP

- e\_version nimmt Versionsinformationen auf, wodurch verschiedene ELF-Varianten unterschieden werden k\u00f6nnen. Momentan ist allerdings nur Version 1 der Spezifikation definiert, die durch EV\_CURRENT repr\u00e4sentiert wird.
- e\_entry gibt den Einsprungpunkt im virtuellen Speicher an, an dem die Ausführung beginnt, nachdem das Programm geladen und im Speicher plaziert worden ist.
- e\_phoff gibt das Offset an, an dem sich die Programmheader-Tabelle in der Binärdatei befindet.
- e\_shoff gibt das Offset an, an dem sich die Sektionsheader-Tabelle befindet.
- e\_flags kann Prozessor-spezifische Flags aufnehmen; momentan werden diese aber vom Kern nicht verwendet.
- e\_ehsize legt die Größe des Headers in Bytes fest.
- phentsize gibt an, wie viele Bytes ein Eintrag in der Programmheader-Tabelle umfasst (alle Einträge sind gleich groß).
- e\_phnum gibt die Anzahl der Einträge in der Programmheader-Tabelle an.
- e\_shentsize gibt an, wie viele Bytes ein Eintrag in der Sektionsheader-Tabelle umfasst (alle Einträge sind gleich groß).
- e\_shnum legt legt die Anzahl der Einträge in der Sektionsheader-Tabelle fest.

 e\_shstrndx hält die Indexposition fest, an der sich die Stringtabelle mit den Sektionsnamen in der Headertabelle befindet.

Analog wird eine 64-Bit-Datenstruktur definiert, die sich nur dadurch von der 32-Bit-Variante unterscheidet, dass die entsprechenden 64-Bit-Datentypen anstelle ihrer 32-bittigen Äquivalente verwendet werden, wodurch der Header etwas größer wird. Völlig identisch bei beiden Varianten bleiben allerdings die ersten 16 Bytes, da beide Architekturtypen ELF-Dateien für Maschinen mit unterschiedlicher Wortbreite daran erkennen:

```
typedef struct elf64_hdr {
                                                                                                <elf.h>
                                            /* ELF "magic number" */
 unsigned char e_ident[16];
 Elf64_Half e_type;
 Elf64_Half e_machine;
 Elf64_Word e_version;
                                   /* Entry point virtual address */
 Elf64_Addr e_entry;
 Elf64_Off e_phoff;
                                   /* Program header table file offset */
 Elf64_Off e_shoff;
                                   /* Section header table file offset */
 Elf64_Word e_flags;
 Elf64_Half e_ehsize;
 Elf64_Half e_phentsize;
 Elf64_Half e_phnum;
 Elf64_Half e_shentsize;
 Elf64_Half e_shnum;
 Elf64_Half e_shstrndx;
} Elf64_Ehdr;
```

#### Programmheader

Die Programmheader-Tabelle setzt sich aus mehreren Einträgen zusammen, die wie Einträge eines Arrays behandelt werden (die Anzahl der Einträge wird durch e\_phnum im ELF-Header festgelegt). Als Datentyp für die Einträge wird eine eigene Struktur definiert, die auf 32-Bit-Rechnern folgenden Inhalt hat:

```
typedef struct elf32_phdr{
                                                                                            <elf.h>
 Elf32_Word
                p_type;
 Elf32_Off
                p_offset;
 Elf32\_Addr
                p_vaddr;
 Elf32 Addr
                p_paddr;
 Elf32_Word
                p_filesz;
 Elf32 Word
                p_memsz;
 Elf32 Word
                p_flags;
 Elf32_Word
                p_align;
} Elf32_Phdr;
```

- p\_type legt fest, um welche Art von Segment es sich beim aktuellen Eintrag handelt. Dazu ist eine Reihe von Konstanten definiert:
  - PT\_NULL kennzeichnet ein unbenutztes Segment.
  - PT\_LOAD wird für ladbare (loadable) Segmente verwendet, die von der Binärdatei in den Speicher transferiert werden, bevor das Programm ausgeführt werden kann.
  - PT\_DYNAMIC zeigt an, dass die Sektion Informationen für den dynamischen Linker enthält, worauf wir in Abschnitt E.2.6 eingehen werden.
  - PT\_INTERP legt fest, dass in der aktuellen Sektion der Programminterpreter spezifiziert wird, der für das dynamische Linking verwendet wird. Üblicherweise handelt es sich dabei um ld-linux.so, wie wir bereits weiter oben festgestellt haben.

 PT\_NOTE zeigt ein Segment an, in dem vom Compiler zusätzliche herstellerspezifische Informationen untergebracht werden können, auf die wir aber nicht eingehen werden.

Zusätzlich werden die Varianten PT\_LOPROC und PT\_HIGHPROC definiert, die für Maschinen-spezifische Zwecke verwendet werden können, wovon der Kernel aber keinen Gebrauch macht.

- p\_offset gibt das Offset (gerechnet in Bytes ab Anfang der Binärdatei) an, ab dem sich Daten des beschriebenen Segments befinden.
- p\_vaddr legt fest, an welche Position des virtuellen Adressraums die Daten des Segments transferiert werden (für Segmente des Typs PT\_LOAD). Systeme, die keine virtuelle, sondern nur physikalische Adressierung unterstützen, verwenden stattdessen die in p\_paddr gespeicherte Angabe.
- p\_filesz gibt an, wie groß (in Bytes) das Segment in der Binärdatei ist.
- p\_memsz legt fest, wie groß (in Bytes) das Segment im virtuellen Adressraum ist. Größenunterschiede mit dem physikalischen Segment werden durch Abschneiden von Daten oder Auffüllen mit Nullbytes realisiert.
- p\_flags nimmt Flags auf, die die Zugriffsberechtigungen auf das Segment festlegen: PF\_R erlaubt den Lese-, PF\_W den Schreib- und PF\_X den Ausführungszugriff.
- p\_align gibt an, wie das Segment im Speicher und in der Binärdatei ausgerichtet werden soll (die Adressen p\_vaddr und p\_offset müssen modulo p\_align teilbar sein). Ein p\_align-Wert von 0x1000 = 4096 bedeutet beispielsweise, dass das Segment an 4KiB-Seiten ausgerichtet werden muss.

Eine analoge Datenstruktur ist für 64-Bit-Architekturen definiert. Der einzige Unterschied zur 32-Bit-Variante besteht darin, dass andere Datentypen verwendet werden; die Bedeutung der Einträge ändert sich aber nicht:

```
<elf h>
           typedef struct elf64_phdr {
            Elf64_Word p_type;
             Elf64_Word p_flags;
             Elf64_Off p_offset;
                                               /* Segment file offset */
             Elf64_Addr p_vaddr;
                                               /* Segment virtual address */
             Elf64_Addr p_paddr;
                                               /* Segment physical address */
             Elf64_Xword p_filesz;
                                               /* Segment size in file */
            Elf64_Xword p_memsz;
                                               /* Segment size in memory */
             Elf64_Xword p_align;
                                               /* Segment alignment, file & memory */
           } Elf64_Phdr;
```

#### Sektionsheader

Die Sektionsheader-Tabelle wird durch ein Array implementiert, dessen Einträge je eine Sektion enthalten. Die einzelnen Sektionen bilden den Inhalt der Segmente, die in der Programmheader-Tabelle definiert werden. Folgende Datenstruktur repräsentiert eine Sektion:

```
Elf32_Off sh_offset;
Elf32_Word sh_size;
Elf32_Word sh_link;
Elf32_Word sh_info;
Elf32_Word sh_addralign;
Elf32_Word sh_entsize;
} Elf32_Shdr;
```

- sh\_name legt den Namen der Sektion fest; allerdings wird nicht die Zeichenkette selbst, sondern ein Index in die Sektionsheader-Stringtabelle gespeichert.
- sh\_type legt den Typ der Sektion fest, wofür folgende Alternativen zur Auswahl stehen:
  - SH\_NULL kennzeichnet eine nicht verwendete Sektion, deren Daten ignoriert werden.
  - SH\_PROGBITS enthält Programm-spezifische Informationen, deren Format nicht genauer festgelegt ist und die uns nicht weiter interessieren.
  - SH\_SYMTAB enthält eine Symboltabelle, auf deren Aufbau wir in Abschnitt E.2.4 zurückkommen werden. Auch SH\_DYNSYM enthält eine Symboltabelle; auf die Unterschiede zwischen beiden Varianten werden wir weiter unten genauer eingehen.
  - SH\_STRTAB kennzeichnet eine Sektion, in der sich eine Stringtabelle befindet.
  - SH\_RELA und SHT\_RELA werden für Relokationssektionen verwendet, deren Aufbau wir in Abschnitt E.2.5 behandeln werden.
  - SH\_HASH legt eine Sektion fest, in der eine Hashtabelle gespeichert wird, mit deren Hilfe Einträge in Symboltabellen schneller gefunden werden können, wie bereits weiter oben angesprochen.
  - SH\_DYNAMIC enthält Informationen, die beim dynamischen Linken verwendet werden, worauf Abschnitt E.2.6 n\u00e4her eingeht.

Zusätzlich existieren die Werte SHT\_HIPROC, SHT\_LOPROC, SHT\_HIUSER und SHT\_LOUSER, die für Prozessor- bzw. Anwendungs-spezifische Zwecke reserviert sind und uns nicht weiter interessieren.

- sh\_flags legt fest, ob schreibend auf die Sektion zugegriffen werden darf (SHF\_WRITE), ob Speicher im virtuellen Adressraum reserviert werden soll (SHF\_ALLOC) und ob die Sektion ausführbaren Maschinencode enthält (SHF\_EXECINSTR).
- sh\_addr gibt die Position im virtuellen Adressraum an, an der die Sektion eingeblendet werden soll.
- sh\_offset legt die Position in der Datei fest, an der die Sektion beginnt.
- sh\_size gibt an, wie viele Bytes die Daten der Sektion umfassen.
- sh\_link verweist auf einen anderen Eintrag in der Sektionsheader-Tabelle, dessen Interpretation sich je nach Sektionstyp unterscheidet. Wir werden gleich genauer darauf eingehen.
- sh\_info wird zusammen mit sh\_link verwendet; die genaue Bedeutung werden wir ebenfalls gleich klären.
- sh\_addralign legt fest, wie die Daten der Sektion im Speicher ausgerichtet werden sollen.

■ sh\_entsize gibt an, wie viele Bytes die Einträge der Sektion umfassen, wenn es sich dabei um Elemente konstanter Größe handelt – wie beispielsweise bei einer Symboltabelle.

Die Elemente sh\_link und sh\_info werden je nach Sektionstyp mit unterschiedlichen Bedeutungen verwendet:

- Sektionen des Typs SHT\_DYMAMIC verwenden sh\_link, um auf die Stringtabelle zu verweisen, die von den Daten der Sektion verwendet wird. sh\_info bleibt unbenutzt und wird auf 0 gesetzt.
- Hashtabellen, d.h. Sektionen des Typs SHT\_HASH, verweisen mit Hilfe von sh\_link auf die Symboltabelle, deren Einträge gehasht werden. sh\_info ist unbenutzt.
- Relokationssektionen des Typs SHT\_REL und SHT\_RELA benutzen sh\_link, um auf die zugehörige Symboltabelle zu verweisen. sh\_info gibt den Index der Sektion in der Sektionsheader-Tabelle an, auf die sich die Relokationen beziehen.
- Für Symboltabellen (SHT\_SYMTAB und SHT\_DYNSYM) legt sh\_link fest, welche Stringtabelle verwendet wird, während sh\_info die Indexposition in der Symboltabelle unmittelbar hinter dem letzten lokalen Symbol (vom Typ STB\_LOCAL) angibt.

Wie üblich existiert eine separate Datenstruktur für 64-Bit-Rechner, deren Inhalt sich aber nicht von der 32-Bit-Variante unterscheidet:

```
<elf.h>
           typedef struct elf64_shdr {
             Elf64_Word sh_name;
                                                 /* Section name, index in string tbl */
             Elf64_Word sh_type;
                                                 /* Type of section */
             Elf64_Xword sh_flags;
                                                 /* Miscellaneous section attributes */
             Elf64_Addr sh_addr;
                                                 /* Section virtual addr at execution */
             Elf64_Off sh_offset;
                                                 /* Section file offset */
             Elf64_Xword sh_size;
                                                 /* Size of section in bytes */
             Elf64_Word sh_link;
                                                 /* Index of another section */
             Elf64_Word sh_info;
                                                 /* Additional section information */
             Elf64_Xword sh_addralign;
                                                 /* Section alignment */
             Elf64_Xword sh_entsize;
                                                 /* Entry size if section holds table */
           } Elf64_Shdr;
```

Der ELF-Standard definiert einige Sektionen mit festen Namen, die zur Realisierung von Standardaufgaben verwendet werden, die in den meisten Objektdateien notwendig sind. Alle Namen beginnen mit einem Punkt, um sie von benutzerdefinierten bzw. nicht standardmäßigen Sektionen zu unterscheiden. Die wichtigsten Standardsektionen sind:

- . bss nimmt uninitialisierte Datenabschnitte des Programms auf, die vor dem Start mit Nullbytes gefüllt werden.
- .data enthält initialisierte Daten des Programms beispielsweise vorinitialisierte Strukturen, die bereits zum Übersetzungszeitpunkt mit statischen Daten gefüllt wurden, die aber während des Programmablaufs geändert werden können.
- rodata nimmt nur lesbare Daten auf, die vom Programm verwendet, aber nicht manipuliert werden beispielsweise Zeichenketten,
- .dynamic, .dynstr nehmen die dynamischen Informationen auf, auf die wir am Ende dieses Kapitels eingehen werden.

<elf.h>

- .interp nimmt den Namen des Programminterpreters als Zeichenkette auf.
- shstrtab enthält eine Stringtabelle, in der die Namen der Sektionen festgelegt werden.
- . strtab nimmt eine Stringtabelle auf, die vor allem die Zeichenketten enthält, die für die Symboltabelle benötigt werden.
- . symtab enthält die Symboltabelle der Binärdatei.
- .init und .fini enthalten Maschinencode, der zur Initialisierung bzw. beim Beenden des Programms ausgeführt wird. Der Inhalt dieser Sektionen wird üblicherweise automatisch vom Compiler und dessen Hilfstools generiert, um eine passende Ablaufumgebung zu schaffen
- . text nimmt den Haupt-Maschinencode auf, der das eigentliche Programm ausmacht.

# E.2.3 Stringtabellen

Das Format von Stringtabellen wurde bereits in Abschnitt E.1.5 besprochen. Da ihr Aufbau sehr dynamisch ist, kann der Kern keine feste Datenstruktur zur Verfügung stellen, sondern muss die vorhandenen Daten "manuell" analysieren.

## E.2.4 Symboltabellen

Symboltabellen enthalten alle Informationen, die notwendig sind, um Symbole eines Programms zu ermitteln, mit Werten zu belegen oder zu relozieren. Wie wir bereits festgestellt haben, existiert ein spezieller Sektionstyp, der zur Aufnahme von Symboltabellen dient; die Tabellen selbst setzen sich aus Einträgen zusammen, deren Aufbau durch folgende Datenstruktur festgelegt ist:

```
typedef struct elf32_sym{
  Elf32_Word     st_name;
  Elf32_Addr     st_value;
  Elf32_Word     st_size;
  unsigned char st_info;
  unsigned char st_other;
  Elf32_Half     st_shndx;
} Elf32_Sym;
```

Prinzipielle Aufgabe eines Symbols ist es, eine Zeichenkette mit einem Wert zu verbinden. Der Wert zum Symbol printf ist beispielsweise die Adresse der printf-Funktion im virtuellen Adressraum, an der sich der Maschinencode der Funktion findet. Als Wert für ein Symbol kann aber auch ein absoluter Wert verwendet werden, der beispielsweise als numerische Konstante interpretiert wird.

Der genaue Verwendungszweck eines Symbols wird durch das Element st\_info festgelegt, das in zwei Teile aufgespalten wird (die exakte Bitaufteilung zwischen den Abschnitten interessiert hier nicht). Folgende Informationen werden festgelegt:

- Der Gültigkeitsbereich (Binding) des Symbols. Dies gibt an, wie weit ein Symbol sichtbar ist; drei verschiedene Einstellungen sind möglich:
  - Lokale Symbole (STB\_LOCAL) sind nur innerhalb einer Objektdatei gültig und für andere
     Teile des Programms, die beim Linken zusammengefügt werden, nicht sichtbar. Es ist

- kein Problem, wenn mehrere Objektdateien eines Programms Symbole mit identischen Bezeichnungen definieren, solange es sich bei allen nur um lokale Symbole handelt: Sie beeinflussen sich gegenseitig nicht.
- Globale Symbole (STB\_GLOBAL) sind nicht nur in der Objektdatei sichtbar, in der sie definiert werden, sondern können von allen anderen Objektdateien referenziert werden, die an einem Programm beteiligt sind. Jedes globale Symbol darf in einem Programm nur ein einziges Mal definiert werden, da der Linker ansonsten eine Fehlermeldung liefert. Undefinierte Referenzen, die auf ein globales Symbol verweisen, werden bei der Relokation mit der Position des Symbols versorgt. Wenn undefinierte Referenzen auf globale Symbole nicht aufgelöst werden können, wird die Ausführung des Programms bzw. das statische Zusammenbinden verweigert.
- Weiche Symbole (STB\_WEAK) sind ebenfalls im gesamten Programm sichtbar, können aber mehrfach definiert werden. Wenn ein globales und ein lokales Symbol mit dem gleichen Namen in einem Programm existiert, wird dem globalen Symbol automatisch der Vorrang eingeräumt.
  - Programme werden auch dann statisch oder dynamisch gebunden, wenn ein weiches Symbol undefiniert bleibt in diesem Fall wird ihm der Wert 0 zugewiesen.
- Der *Typ* des Symbols wird aus verschiedenen Alternativen ausgewählt, wobei für unsere Zwecke nur drei davon relevant sind (eine Beschreibung der anderen Möglichkeiten findet sich im ELF-Standard):
  - STT\_0BJECT gibt an, dass es sich um ein Datenobjekt wie Variablen, Arrays oder Zeiger handelt.
  - STT\_FUNC wird verwendet, wenn ein Symbol eine Funktion oder eine Prozedur repräsentiert
  - STT\_NOTYPE repräsentiert ein Symbol unbekannten Typs, was für undefinierte Referenzen verwendet wird.

Neben st\_name, st\_value und st\_info finden sich noch andere Elemente in der Elf32\_Sym-Struktur, die folgende Bedeutung haben:

- st\_size gibt die Größe des Objekts an, beispielsweise die Länge eines Pointers oder die Anzahl an Bytes, die in einem struct-Objekt enthalten sind. Der Wert kann auch auf 0 gesetzt sein, wenn die Größe nicht bekannt ist.
- **st\_other** wird in der aktuellen Version des Standards noch nicht verwendet.
- st\_shndx enthält den Index einer Sektion (in der Sektionsheader-Tabelle), mit der das Symbol verbunden ist üblicherweise ist es im Code dieser Sektion definiert. Allerdings gibt es zwei Spezialwerte, die eine besondere Bedeutung haben:
  - SHN\_ABS zeigt an, dass das Symbol einen absoluten Wert besitzt, der sich durch die Relokation nicht ändert und immer konstant bleibt.
  - SHN\_UNDEF kennzeichnet undefinierte Symbole, die durch externe Quellen (wie andere Objektdateien oder Bibliotheken) aufgelöst werden müssen.

Wie üblich gibt es auch eine 64-Bit-Variante der Symboldatenstruktur, die – bis auf die verwendeten Datentypen – den gleichen Inhalt wie das 32-Bit-Pendant besitzt:

readelf kann auch verwendet werden, um alle Symbole zu ermitteln, die sich in der Symboltabelle eines Programms befinden. Für die Objektdatei test. o sind fünf Einträge besonders wichtig (die restlichen Elemente werden automatisch vom Compiler generiert und interessieren uns hier nicht weiter):

```
wolfgang@meitner> readelf -s test.o
                                      Vis
   Num:
          Value Size Type
                               Bind
                                               Ndx Name
     1: 00000000
                               LOCAL DEFAULT ABS test.c
                     0 FILE
     7: 00000000
                    26 FUNC
                               GLOBAL DEFAULT
                                                 1 add
     8: 00000000
                    O NOTYPE
                               GLOBAL DEFAULT
                                               UND printf
     9: 0000001a
                    75 FUNC
                               GLOBAL DEFAULT
                                                 1 main
                    O NOTYPE
    10: 00000000
                               GLOBAL DEFAULT
                                               UND exit
```

Der Name der Quelldatei wird als Absolutwert gespeichert – er ist immer konstant und verändert sich durch Relokationen nicht. Das lokale Symbol verwendet den weiter oben nicht besprochenen Typ STT\_FILE, dessen einziger Zweck die Verknüpfung einer Objektdatei mit dem Namen ihrer Quelldatei ist.

Die beiden in der Datei definierten Funktionen – main und add – werden als globale Symbole des Typs STT\_FUNC gespeichert. Beide Symbole beziehen sich auf das Segment mit Kennzahl 1, wobei es sich um das Textsegment der Datei handelt, in dem sich der Maschinencode der Funktionen befindet.

Die Symbole printf und exit werden als undefinierte Referenzen des Typs STT\_UNDEF definiert, weshalb sie beim Linken des Programms mit Funktionen aus der Standardbibliothek (oder irgendeiner anderen Bibliothek, die Symbole dieses Namens definiert) verbunden werden müssen. Da der Compiler nicht weiß, um welche Art von Symbol es sich handelt, besitzen sie den Typ STT\_NOTYPE.

# E.2.5 Relokationseinträge

## Prinzip

Als Relokation bezeichnet man den Prozess, der undefinierte Symbole in ELF-Dateien mit gültigen Werten verbindet. Für unser Standardbeispiel test. o bedeutet dies, dass die undefinierten Referenzen auf printf und add durch die Adressen ersetzt werden müssen, an denen sich der entsprechende Maschinencode im virtuellen Adressraum des Prozesses befindet. Diese Ersetzung muss an allen Stellen in der Objektdatei durchgeführt werden, an denen eines der Symbole verwendet wird. Im Fall von Userspace-Programmen ist der Kern nicht an der Symbolersetzung beteiligt, da diese vollständig durch externe Hilfstools durchgeführt wird. Anders verhält es sich

bei Kernmodulen, wie wir im Modulkapitel gezeigt haben: Da der Kern die Rohdaten eines Moduls genau so überreicht bekommt, wie sie in der Binärdatei gespeichert sind, muss er sich um die Durchführung der Relokation kümmern.

Um Stellen zu kennzeichnen, die reloziert werden müssen, wird eine spezielle Tabelle mit Relokationseinträgen in jeder Objektdatei gespeichert. Jeder Eintrag enthält folgende Informationen:

- Ein Offset, das die Position des Eintrags angibt, der modifiziert werden soll.
- Eine Referenz auf das Symbol (als Index in eine Symboltabelle), das die Daten liefert, die in die Relokationsposition eingesetzt werden sollen.

Um zu verdeutlichen, wie Relokationsinformationen verwendet werden, greifen wir wieder auf das weiter oben eingeführte Testprogramm test.c zurück. Zunächst können mit readelf alle Relokationseinträge angezeigt werden, die sich in der Datei finden:

```
wolfgang@meitner> readelf -r test.o
Relocation section '.rel.text' at offset 0x374 contains 6 entries:
 Offset
            Info
                    Туре
                                    Sym. Value
                                               Sym. Name
          00000501 R_386_32
00000009
                                                 .rodata
                                     00000000
          00000802 R_386_PC32
                                                printf
0000000e
                                     00000000
00000046
          00000702 R_386_PC32
                                     00000000
                                                add
00000050
          00000501 R_386_32
                                     00000000
                                                 .rodata
                                                printf
00000055
          00000802 R_386_PC32
                                     00000000
00000061
          00000a02 R_386_PC32
                                     00000000
```

Die in der Spalte "Offset" vorhandenen Informationen werden verwendet, wenn der Maschinencode auf Funktionen oder Symbole verweist, von denen noch nicht klar ist, an welcher Stelle sie sich im virtuellen Adressraum befinden werden, wenn das Programm ausgeführt wird bzw. wenn test.o zu einer ausführbaren Datei gelinkt wird. Der Assemblercode von main zeigt einige Funktionsaufrufe, die sich an den Offset-Positionen 0x46 (add), 0xe und 0x55 (printf) und 0x61 (exit) befinden, was mit Hilfe des objdump-Tools sichtbar gemacht werden kann. Die relevanten Zeilen sind kursiv wiedergegeben:

```
wolfgang@meitner> objdump --disassemble test.o
0000001a <main>:
       55
 1a:
                                        %ebp
                                 push
        89 e5
                                        %esp,%ebp
 1b:
                                 mov
        83 ec 18
                                         $0x18,%esp
 1d:
                                 sub
 20:
        83 e4 f0
                                         $0xffffffff0,%esp
                                 and
 23:
        ъ8 00 00 00 00
                                         $0x0, %eax
                                 mov
 28:
        29 c4
                                 sub
                                         %eax.%esp
        c7 45 fc 03 00 00 00
                                         $0x3,0xfffffffc(%ebp)
 2a:
                                 movl
 31:
        c7 45 f8 04 00 00 00
                                        $0x4,0xfffffff8(%ebp)
                                 movl
                                         Oxfffffff8(%ebp),%eax
 38:
        8b 45 f8
                                 mov
        89 44 24 04
                                         %eax,0x4(%esp,1)
 3b:
                                 mov
                                        Oxfffffffc(%ebp),%eax
 3f:
        8b 45 fc
                                 mov
        89 04 24
 42:
                                         %eax,(%esp,1)
                                 mov
 45:
        e8 fc ff ff ff
                                        46 <main+0x2c>
                                 call
                                         weax,0xffffffff4(%ebp)
        89 45 f4
 4a:
                                 mov
        c7 04 24 17 00 00 00
                                        $0x17,(%esp,1)
 4d:
                                 movl
        e8 fc ff ff ff
                                        55 <main+0x3b>
 54:
                                 call
        c7 04 24 00 00 00 00
                                        $0x0.(%esp,1)
 59:
                                 movl
 60:
        e8 fc ff ff ff
                                 call
                                        61 <main+0x47>
```

Wenn die Adressen der Funktionen printf und add bestimmt sind, müssen sie an den genannten Offset-Positionen eingefügt werden, um korrekten ablauffähigen Code zu erzeugen.

#### **Datenstrukturen**

Aus technischen Gründen gibt es zwei unterschiedliche Arten von Relokationsinformationen, die durch leicht verschiedene Datenstrukturen repräsentiert werden. Die erste Variante bezeichnet man als normale Relokation; die Einträge der Relokationstabelle, die in einer Sektion des Typs SHT\_REL untergebracht werden, sind dabei durch folgenden Datentyp definiert:

```
typedef struct elf32_rel {
    Elf32_Addr    r_offset;
    Elf32_Word    r_info;
} Elf32_Rel;
```

Während r\_offset die Position des Eintrags angibt, der reloziert werden soll, liefert r\_info nicht nur eine Position in der Symboltabelle, sondern zusätzliche Informationen darüber, um welchen Relokationstyp es sich handelt (wir gehen gleich darauf ein). Dies wird durch Unterteilung des Werts in zwei Abschnitte erreicht, wobei hier nicht weiter interessant ist, wie die Aufteilung genau erfolgt.

Die alternative Variante – als Relokationseintrag mit konstantem Addendum bezeichnet – darf nur in Sektionen des Typs SHT\_RELA auftreten; die Einträge dieser Sektion sind durch folgende Datenstruktur definiert:

Neben den auch in der ersten Variante vorhandenen Feldern r\_offset und r\_info gibt es ein zusätzliches Element r\_addend, das einen als *Addendum* bezeichneten Wert aufnehmen kann, der bei der Berechnung des Relokationswerts je nach Relokationstyp unterschiedlich berücksichtigt wird.

Achtung: Der Addendums-Wert ist auch bei der Verwendung von elf32\_rel vorhanden: Obwohl er nicht explizit in der Datenstruktur festgehalten wird, verwendet der Linker den Wert als Addendum, der sich an der Speicherstelle befindet, in die die berechnete Relokationsgröße eingefügt werden soll. Wozu dieser Wert gut ist, werden wir weiter unten anhand eines Beispiels klären.

Für beide Relokationstypen existieren funktional äquivalente 64-Bit-Datenstrukturen, für die sich eine genauere Beschreibung erübrigt:

```
typedef struct elf64_rel {
    Elf64_Addr r_offset; /* Location at which to apply the action */
    Elf64_Xword r_info; /* index and type of relocation */
} Elf64_Rel;

typedef struct elf64_rela {
    Elf64_Addr r_offset; /* Location at which to apply the action */
    Elf64_Xword r_info; /* index and type of relocation */
    Elf64_Sxword r_addend; /* Constant addend used to compute value */
} Elf64_Rela;
```

#### Relokationstypen

Der ELF-Standard definiert viele verschiedene Relokationstypen, wobei für jede unterstützte Architektur ein eigener Satz existiert. Die meisten dieser Typen werden verwendet, wenn dynamische Bibliotheken oder positionsunabhängiger Code erzeugt werden; auf manchen Plattformen

 – allen voran IA-32 – müssen zusätzlich viele Designfehler oder historischer Ballast kompensiert werden. Glücklicherweise kommt der Kern, der lediglich an der Relokation von Modulen interessiert ist, mit zwei verschiedenen Relokationstypen aus:

- PC-relative Relokation
- Absolute Relokation

PC-relative Relokation erzeugt Relokationseinträge, die auf Adressen im Speicher zeigen, die relativ zum *Program Counter* (PC, Programmzähler)<sup>10</sup> definiert sind. Dies wird vor allem bei Unterprogrammaufrufen benötigt. Die alternative Relokationsform erzeugt absolute Adressen, wie ihr Name unmissverständlich andeutet. Üblicherweise werden diese verwendet, um sich auf Daten im Speicher zu beziehen, die bereits zur Übersetzungszeit bekannt sind, beispielsweise Stringkonstanten.

Auf IA-32-Rechnern werden die beiden Relokationstypen durch die Konstanten R\_386\_PC32 (PC-relative Relokation) und R\_386\_32 (absolute Relokation) repräsentiert. Das Relokationsergebnis berechnet sich wie folgt:

R\_386\_32 : 
$$Result = S + A$$
  
R\_386\_32 :  $Result = S - P + A$ 

A steht dabei für den Addendums-Wert, der im Fall von IA-32 implizit über den Speicherinhalt der Relokationsposition gegeben ist. S gibt den Wert des Symbols wieder, der in der Symboltabelle gespeichert ist, und P steht für das Offset der Relokationsposition, d.h. die Stelle in der Binärdatei, an die die errechneten Daten geschrieben werden sollen. Gehen wir vorerst davon aus, dass der Addendums-Wert gleich 0 ist: Dies bedeutet, dass absolute Relokationen einfach den Wert des Symbols, der sich in der Symboltabelle findet, in die Relokationsposition einfügen, während für PC-relative Relokationen die Differenz zwischen Symbolposition und Relokationsposition berechnet wird — mit anderen Worten ausgedrückt bedeutet dies, dass berechnet wird, wie viele Bytes von der Relokationsposition entfernt sich das Symbol befindet.

Der Addendums-Wert wird in beiden Fällen hinzuaddiert und erzeugt eine lineare Verschiebung des Ergebnisses, auf deren Bedeutung wir eingehen werden, nachdem wir die Vorgehensweise an einem konkreten Beispiel klargemacht haben.

**Beispiel für relative Verschiebungen** In der Testdatei test.o findet sich folgende call-Anweisung:

```
45: e8 fc ff ff ff call 46 <main+0x2c>
```

e8 ist der Opcode des call-Befehls, während 0xfffffffc (Little-Endian-Notation!) der Wert ist, der dem fertigen Befehl als Parameter übergeben wird. Da IA-32 normale Relokationen anstelle von Add-Relokationen verwendet, handelt es sich um den Addendums-Wert. 0xfffffffc ist also noch nicht die endgültige Adresse, sondern muss erst den Relokationsprozess durchlaufen. Dezimal entspricht 0xfffffffc übrigens dem Wert —4, wobei beachtet werden muss, dass die Zweierkomplementnotation zur Darstellung vorzeichenbehafteter Ganzzahlen verwendet wird. Achtung: Das objdump-Tool zeigt auf der rechten Seite nicht das Argument der call-Anweisung an,

<sup>10</sup> Zur Erinnerung: Beim Programmzähler handelt es sich um ein spezielles Prozessor-Register, das festlegt, an welcher Stelle im Maschinencode sich der Prozessor während der Ausführung befindet.

sondern erkennt automatisch, dass ein Relokationseintrag auf die entsprechende Speicherposition verweist, weshalb diese Information eingefügt wird.

Wie die Relokationstabelle zeigt, handelt es sich bei der Relokationsposition 46 um einen Aufruf der add-Funktion:

```
00000046 00000702 R_386_PC32 00000000 add
```

Da die Sektionen der Binärdatei bereits an ihre endgültige Position im Speicher verschoben worden sind, bevor die Relokation ausgeführt wird, ist die Position von add im Speicher bekannt. Nehmen wir an, dass add an Position 0x08048388 positioniert wurde. Die main-Funktion soll sich an Position 0x080483a2 befinden, was bedeutet, dass die Relokationsposition, in die das Relokationsergebnis hineingeschrieben werden soll, an Position 0x80483ce zu finden ist.

Die Berechnung des Relokationsergebnisses wird durchgeführt, indem die Formel für PCrelative Relokationen angewandt wird:

```
Result = S - P + A
= 0x08048388 - 0x80483ce + (-4)
= 134513544 - 134513614 - 4
= -74
```

Das Resultat entspricht dem Code, der sich in der ausführbaren Datei test findet, wie man mit objdump feststellen kann:

```
80483cd: e8 b6 ff ff ff call 8048388 <add>
```

0xffffffb6 entspricht in der dezimalen Darstellung der Zahl -74, was der Leser unter Beachtung der Rahmenbedingungen "Little Endian" und "Zweierkomplementnotation" leicht nachprüfen kann. objdump zeigt in der symbolischen Darstellung auf der rechten Seite der Ausgabe nicht das relative Sprungziel an, sondern rechnet dies in einen absoluten Wert um, um es dem Programmierer leichter zu machen, die entsprechende Stelle im Maschinencode zu finden.

Auf den ersten Blick scheint das Ergebnis falsch zu sein: Der Maschinencode der add-Anweisung findet sich 70 Bytes (0x46) vor der Relokationsposition, wie wir bereits gesehen haben, nicht 74 Bytes! Die Verschiebung um 4 Bytes nach hinten wird durch den Addendums-Wert bewirkt. Warum setzt der Compiler diesen bei der Generierung der Objektdatei test.o auf -4, anstelle ihn auf 0 zu belassen? Der Grund dafür liegt in der Arbeitsweise von IA-32=Prozessoren: Der Programmzähler zeigt immer auf die Anweisung, die der gerade ausgeführten Anweisung als Nächstes folgt – und ist daher 4 Bytes "zu groß", wenn der Prozessor das absolute Sprungziel anhand der relativen Angabe im Maschinencode berechnet. Der Compiler muss entsprechend 4 Bytes vom relativen Sprungziel abziehen, um letztendlich an die korrekte Position im Programm zu gelangen.

Absolute Relokationen folgen demselben Schema, wobei die Berechnung allerdings etwas einfacher ist, da nur die Zieladresse des gewünschten Symbols mit dem Addendums-Wert kombiniert werden muss, weshalb wir hier nicht näher darauf eingehen.

## **E.2.6** Dynamisches Linken

ELF-Dateien, die dynamisch mit Bibliotheken verbunden werden müssen, um ablaufen zu können, sind aus Sicht des Kerns relativ uninteressant: Die Referenzen in Modulen können allesamt durch Relokationen aufgelöst werden, während das dynamische Linken von Userspace-

Programmen vollständig durch ld. so im Userspace erledigt wird. Wir werden daher nur überblicksweise auf die Bedeutung der dynamischen Abschnitte eingehen.

Zwei Sektionen werden verwendet, um die Daten aufzunehmen, die vom dynamischen Linker benötigt werden:

- dynsym enthält eine Symboltabelle mit allen Symbole, die durch externe Referenzen aufgelöst werden.
- dynamic enthält ein Array mit Elementen des Typs Elf32 Dyn, die verschiedenste Informationen liefern, auf die wir gleich eingehen.

Der Inhalt von .dynsym kann mit readelf abgefragt werden:

```
\verb|wolfgang@meitner>| readelf --syms| test|\\
Symbol table '.dynsym' contains 7 entries:
   Num:
          Value Size Type
                               Bind
                                      Vis
                                                Ndx Name
    0: 00000000
                     O NOTYPE LOCAL DEFAULT
                                               UND
     1: 08049474
                     O OBJECT GLOBAL DEFAULT
                                                15 __dso_handle
    2: 0804829c
                   206 FUNC
                               GLOBAL DEFAULT
                                               UND __libc_start_main@GLIBC_2.0 (2)
    3: 080482ac
                    47 FUNC
                               GLOBAL DEFAULT
                                               UND printf@GLIBC_2.0 (2)
     4: 080482bc
                   257 FUNC
                               GLOBAL DEFAULT
                                               UND exit@GLIBC_2.0 (2)
    5: 08048444
                     4 OBJECT GLOBAL DEFAULT
                                                14 _IO_stdin_used
     6: 00000000
                     O NOTYPE
                               WEAK
                                      DEFAULT
                                               UND __gmon_start__
```

Neben einigen Symbolen, die bei der Produktion der ausführbaren Datei automatisch eingefügt werden und die uns hier nicht weiter interessieren, finden sich auch die Funktionen print und exit, die explizit im Code verwendet wurden. Die zusätzliche Angabe @GLIBC\_2.0 spezifiziert, dass *mindestens* Version 2.0 der GNU-Standardbibliothek verwendet werden muss, um die Referenzen aufzulösen.

Der Datentyp für die Arrayeinträge der .dynamic-Sektion wird zwar im Kern definiert, aber nirgends verwendet, da die Auswertung der Informationen im Userspace erfolgt:

```
<elf.h> typedef struct dynamic{
        Elf32_Sword d_tag;
        union{
            Elf32_Sword d_val;
            Elf32_Addr d_ptr;
        } d_un;
        } Elf32_Dyn;
```

d\_tag dient dazu, verschiedene Tags zu unterscheiden, die angeben, welche Art von Informationen durch den Eintrag beschrieben werden, während d\_un entweder eine virtuelle Adresse oder eine Ganzzahl enthält, die je nach Tag anders interpretiert wird.

Die wichtigsten Tags sind:

 DT\_NEEDED legt fest, welche dynamischen Bibliotheken benötigt werden, um das Programm ausführen zu können. d\_un zeigt auf einen Stringtabelleneintrag, der den Namen der Bibliothek angibt.

 $F\ddot{u}r\ die\ Testapplikation\ \texttt{test.c}\ ist\ nur\ die\ C-Standardbibliothek\ n\"{o}tig,\ wie\ \texttt{readelf}\ zeigt:$ 

```
wolfgang@meitner> readelf --dynamic test

Dynamic segment at offset 0x480 contains 20 entries:
Tag Type Name/Value
0x00000001 (NEEDED) Shared library: [libc.so.6]
```

Reale Programme wie beispielsweise der Editor emacs benötigen signifikant mehr dynamische Bibliotheken, um zu funktionieren:

```
\verb|wolfgang@meitner>| readelf| --dynamic| / usr/bin/emacs|
Dynamic segment at offset 0x1ea6ec contains 36 entries:
                                          Name/Value
 0x0000001 (NEEDED)
                                         Shared library: [libXaw3d.so.7]
 0x0000001 (NEEDED)
                                         Shared library: [libXmu.so.6]
 0x0000001 (NEEDED)
                                         Shared library: [libXt.so.6]
 0x0000001 (NEEDED)
                                         Shared library: [libSM.so.6]
 0x0000001 (NEEDED)
                                         Shared library: [libICE.so.6]
 0x0000001 (NEEDED)
                                         Shared library: [libXext.so.6]
 0x0000001 (NEEDED)
                                         Shared library: [libtiff.so.3]
 0x0000001 (NEEDED)
                                         Shared library: [libjpeg.so.62]
 0x0000001 (NEEDED)
                                         Shared library: [libpng.so.2]
 0x0000001 (NEEDED)
                                         Shared library: [libz.so.1]
 0x00000001 (NEEDED)
                                         Shared library: [libm.so.6]
 0x00000001 (NEEDED)
                                         Shared library: [libungif.so.4]
 0x0000001 (NEEDED)
                                         Shared library: [libXpm.so.4]
 0x0000001 (NEEDED)
                                         Shared library: [libX11.so.6]
 0x00000001 (NEEDED)
                                         Shared library: [libncurses.so.5]
                                         Shared library: [libc.so.6]
 0x0000001 (NEEDED)
 0x000000f (RPATH)
                                         Library rpath: [/usr/X11R6/lib]
```

- DT\_STRTAB legt die Position der Stringtabelle fest, in der sich die Namen aller benötigten dynamischen Bibliotheken und Symbole finden, die für die dynamische Sektion gebraucht werden.
- DT\_SYMTAB spezifiziert die Position der Symboltabelle, in der sich alle für die dynamische Sektion benötigten Informationen befinden.
- DT\_INIT und DT\_FINI legen die Adressen der Funktionen fest, die bei der Initialisierung bzw. bei der Beendigung des Programms aufgerufen werden.