9. Kapitel

Assemblieren und Binden

Kapitel 9: Assemblieren und Binden

0. Einbettung
1. Assemblieren
   - Prinzip
   - Speicherabbildung
   - Sprunglänge
   - Segmentierter Speicher
2. Binden
   - Prinzip
   - Beispiel ELF

Assemblieren und Binden

to assemble: zusammensetzen, montieren

Aufgaben:

- Befehle und Konstante binär codieren (Assemblierer)
- Speicherdarstellung berechnen:
  Befehls-, Konstanten-, Datensegment
- symbolische Adressen auflösen:
  - Assemblierer: interne Adressen
  - Binder: externe Adressen
- Lademodul herstellen (Binder)
  - Lademodul: vom Systemlader in den Speicher zur Ausführung ladbarer Code
- Sonderfall Übersetzen und Ausführen: Code wird nach dem Assemblieren und Binden sofort geladen und ausgeführt
  - Übersetzer Teil der Laufzeitumgebung

Assemblieren

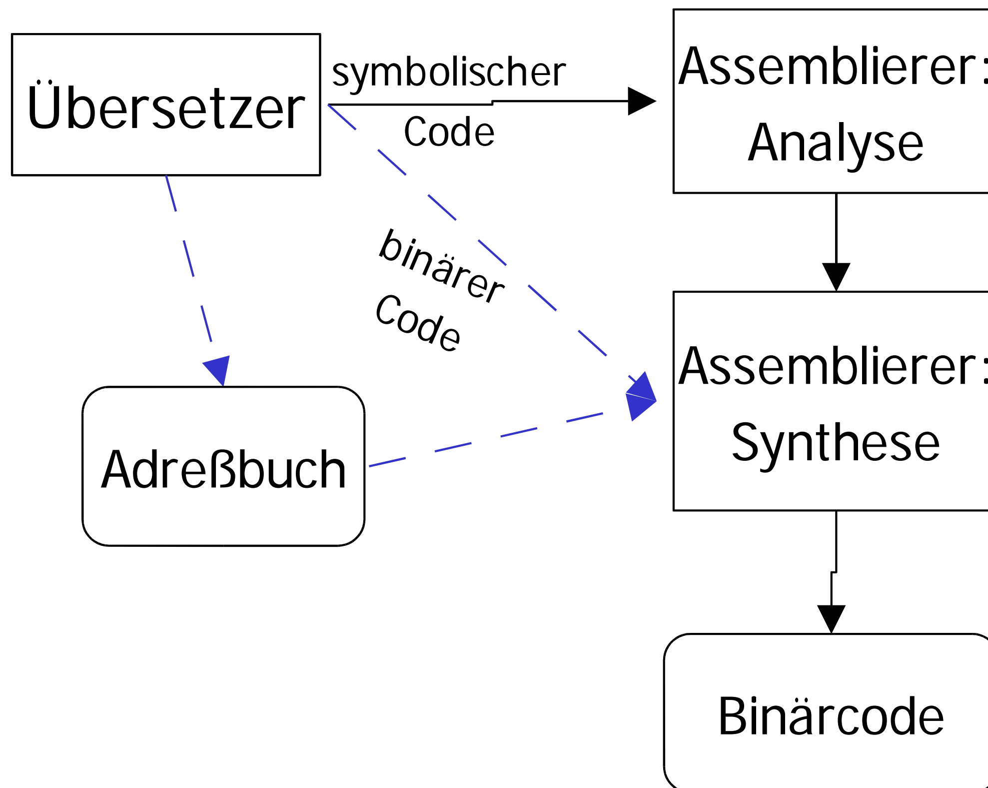
Schritte:

1. Programm in symbolischer Maschinensprache einlesen
2. Syntax prüfen
3. Semantik prüfen, gegebenenfalls Makros auflösen
   z.B. Zulässigkeit Registernummern, Definition aller symbolischen Adressen, ...
4. Speicherabbildung festlegen, gegebenenfalls Prozeduren und Grundblöcke anordnen, Adreßbuch vervollständigen
5. symbolische Adressen auflösen, Kurz-/Langsprünge berechnen,
6. Codeerzeugung: Binärcodierung berechnen, lokale Befehlsanordnung

Einsicht:

- Assemblierer ist voller Übersetzer
- Schritte 1-3 kann auch von Codeerzeugung des vorangehenden Übersetzers übernommen werden

Übersetzen und Assemblieren



Assemblierer: Analyse

Aufgaben:

- Symbolentschlüsselung
- syntaktische Analyse (sehr einfach)
- semantische Analyse:
  - Namensanalyse, unterscheide
    - Bezeichner für Befehle, Register, Schlüsselwörter (section, macro,...)
    - Makronamen, Makroparameter (hier nicht behandelt)
    - Marken von Befehlen und Daten

      ordne allen Verwendungen von Marken in Befehlen die Definition zu

      Gültigkeitsbereiche:
    - global
    - lokal in Sektion
    - makrospezifisch (hier nicht behandelt)
  - Adreßbuch: Definitionstabelle des Assemblierers
  - Konsistenzprüfung: alle Befehle zulässig? Befehle vollständig? Adreßmodi zulässig? Konstanten korrekt geschrieben?

Speicherabbildung I (getrennte Assemblierung)

unterscheide nach Zielen

- absolut adressierter Code (nur in Ausnahmefällen)
- relativ adressierter Code (Standardfall, hier unterstellt)

unterscheide nach System-/Hardwarekonventionen

- getrennte Segmente für Befehle, schreibgeschützte Daten, Variablenspeicher?

Aufgaben:

- initialisiere Segmente
- ordne Befehle/Daten in Eingangsreihenfolge im jeweiligen Segment an
  - auf Ausrichtung an Wort-/Doppelwort/Vierfachwortgrenzen achten!
- zähle während der Anordnung mit und trage den Zähler als (Relativ-) Adresse im Adreßbuch ein (Vervollständigung des Adreßbuchs)

Speicherabbildung II (Assemblierung integriert in Übersetzer)

Unterscheidung nach Zielen und System-/Hardware-Konventionen wie zuvor

Übersetzer liefert an:

- Pseudobefehle *neues Gebiet* (liefert Marke an Übersetzer zurück), *schließe Gebiet* (ergibt Bereich), jeder Grundblock bzw. markiertes Datum ist ein Bereich
- binäre Daten (Länge gezählt in Bits):
  (Teile von) Befehlen, binär codierte Daten
- Marken M und Differenzen M-N
  (als Operanden, samt Längenangabe in Bits)

Assemblierer ordnet Bereiche an (siehe Befehlsanordnung) und trägt sie in die Segmente ein, wie zuvor

Unterschied zu Fall I: Zuordnung Gebrauch-Definition Marke bereits geleistet

Assemblierer: Synthese

Fall I (getrennte Assemblierung):

- Zuordnung Gebrauch von Marken zu Definition berechnen
- Befehle binär codieren
- Kurz-/Langsprünge auflösen
- Gebrauch von Marken durch Adressen gemäß Adreßbuch ersetzen
- Code ausgeben entsprechend Formatvorgabe des Binders:
  - Eingangskennung (magische Zahl)
  - Kommentar (Copyrightvermerk)
  - nicht aufgelöste externe Adreßsymbole
  - Code
  - Daten
  - Info für Testhilfe/Speicherbereiniger

Fall II: wie Fall I ohne die ersten beiden Punkte

Kurz-/Langsprünge: Problem

Problem:

Prozessoren haben oft 2 oder mehr verschiedene Sprungbefehle:

- $jr\ M$, $|M|$ beschränkt z.B. $|M| < 2^{11}$ oder $M < 2^{12}$
- $jr\ M$, $|M|$ unbeschränkt

  M Adresse relativ zum Befehlszählerstand

  Befehle haben unterschiedliche Länge und Ausführungszeit
- auch andere Befehle wertabhängig, z.B. $addi\ R,c$, c konstant

Aufgabe:

- wähle günstigsten wertabhängigen Befehl (Programmlänge, Ausführungszeit)
- Entscheidung erst möglich, wenn Adressen aller Marken bekannt

Kurz-/Langsprünge: Lösung

Datenstrukturen:

- Liste L wertabhängiger Befehle
- Adreßbuch A mit Einträgen $(M, adr_M, erhöhe)$
  - Invariante: $erhöhe$ = *Summe der Verlängerungen wertabhängiger Befehle vor M*

Eingangsannahmen:

- alle Befehle adressiert relativ zur gleichen Anfangsadresse (gleiches Segment)
- alle Befehle kurz ($erhöhe = 0$)
- L enthält alle wertabhängigen Befehle

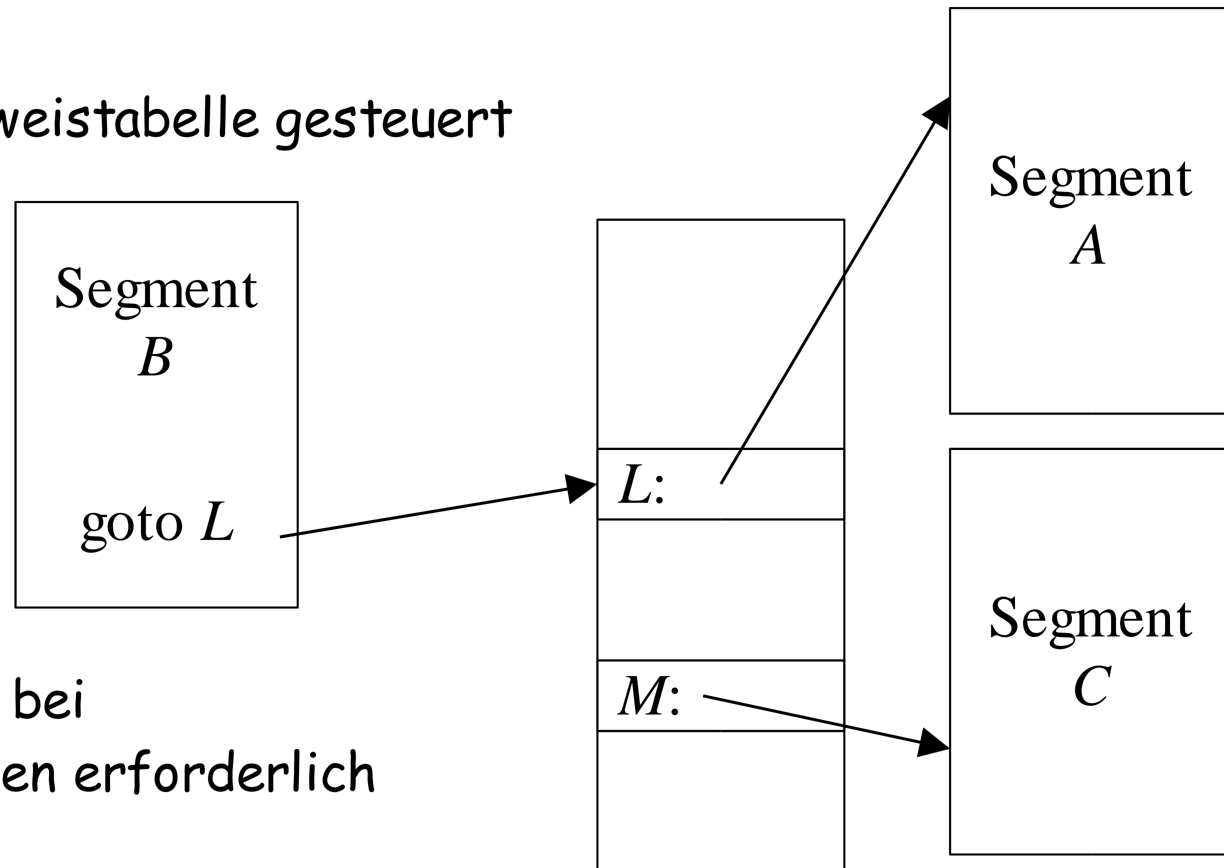
Verfahren (Hanglberger 1977, Szymanski 1978):

loop
 for all $b \in L$ **loop**
  if b zu kurz **then** verlängere(b); aktualisiere(A); streiche b aus L **end**
 end
until kein zu streichender Befehl in L gefunden

- Aufwand $O(|L|_{init}^2)$, praktisch jedoch linearer Aufwand

mehrere Befehlssegmente

- Kurz-/Langsprünge in jedem Segment einzeln behandeln
- Querverweise: Alternativen:
  - sind Langsprünge
  - werden über Verweistabelle gesteuert



- Verweistabelle auch bei polymorphen Aufrufen erforderlich

Kapitel 9: Assemblieren und Binden

0. Einbettung
1. Assemblieren
   - Prinzip
   - Speicherabbildung
   - Sprunglänge
   - Segmentierter Speicher
2. Binden
   - Prinzip
   - Beispiel ELF

Binden

- Vorgegeben:
  - assembliertes, relativ adressiertes Programm
  - Bibliotheken weiterer assemblierter, relativ adressierter Programmstücke
- Aufgaben des Binders (engl. *linker*, keine politische Einordnung!)
  - externe Adressen auflösen
    - Bibliotheken durchsuchen
    - gegebenenfalls Vorbereitung zur dynamischen Verbindung mit gemeinsamen Bibliotheken
  - bei oo-Sprachen: gegebenenfalls zusätzliche polymorphe Alternativen
  - ladefähigen Code herstellen
- Adreßauflösung genau wie im Assemblierer zwischen Segmenten
- hoher Aufwand wegen (zu) vieler zu durchsuchender Bibliotheksdateien
- transitive Hülle aller gefundenen externen Adressen
- intelligente Pufferverwaltung nötig

Eingabeformat des Binders

Binärdateien enthalten:

- Globale Information (Größe aller Daten, Datum der Erstellung, usw.)
- Binärdaten (Programm und Daten)
  - Startadresse
- Verweise auf Stellen, die geändert werden müssen, falls der Binder Anfangsadressen ändert (Relokation)
- Symbole
  - zugänglich für andere Programmteile (Export)
  - benötigt von anderen Programmteilen (Import)
- Informationen für Speicherbereinigung und Testunterstützung

Ausgabeformate des Binders

Binder verknüpft Eingaben zu einer Ausgabe in demselben Format – mit möglichem Datenverlust

Beispiele:

- DOS COM Dateien (keine Informationen, nur Programm und Daten)
- DOS EXE
- UNIX a.out (mehr als 25 Jahre alt)
  - Derivate: NAMGIC, ZMAGIC, QMAGIC
  - ELF (Executable and Linking Format)

UNIX ELF

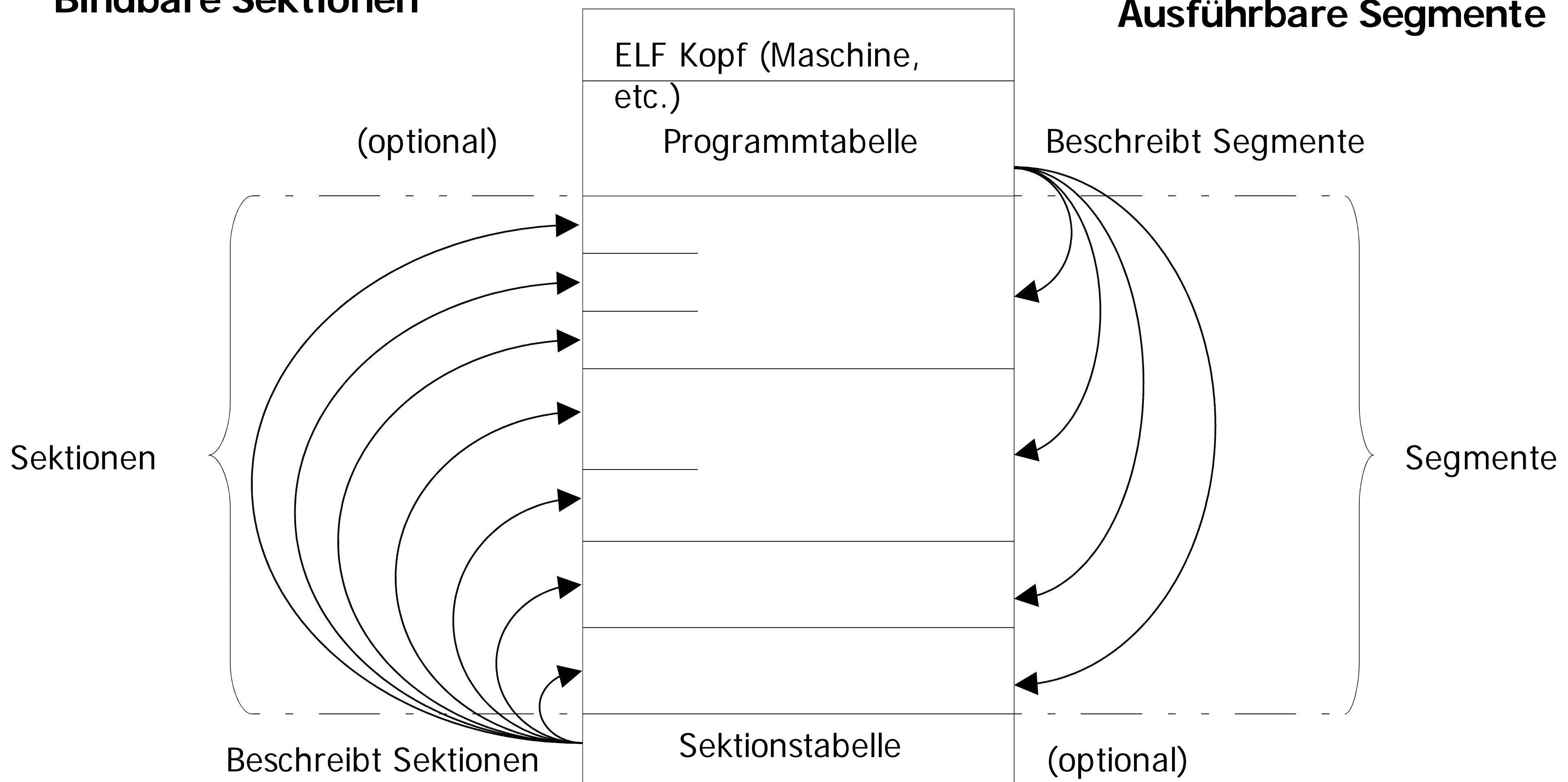
Drei Varianten von ELF:

- Relozierbar
  - werden von Assemblierer oder Übersetzer erzeugt
  - werden vom Binder zum Programm transformiert
- Ausführbar
  - Relokation abgeschlossen
  - Symbole aufgelöst
- Bibliothek (shared libraries)
  - enthält Programmcode, Daten und Symbole

ELF kennt:

- *Sektionen* für ausführbare Programmteile, Daten, usw.
- *Segmente* bestehen aus Sektionen

Sichten auf ELF

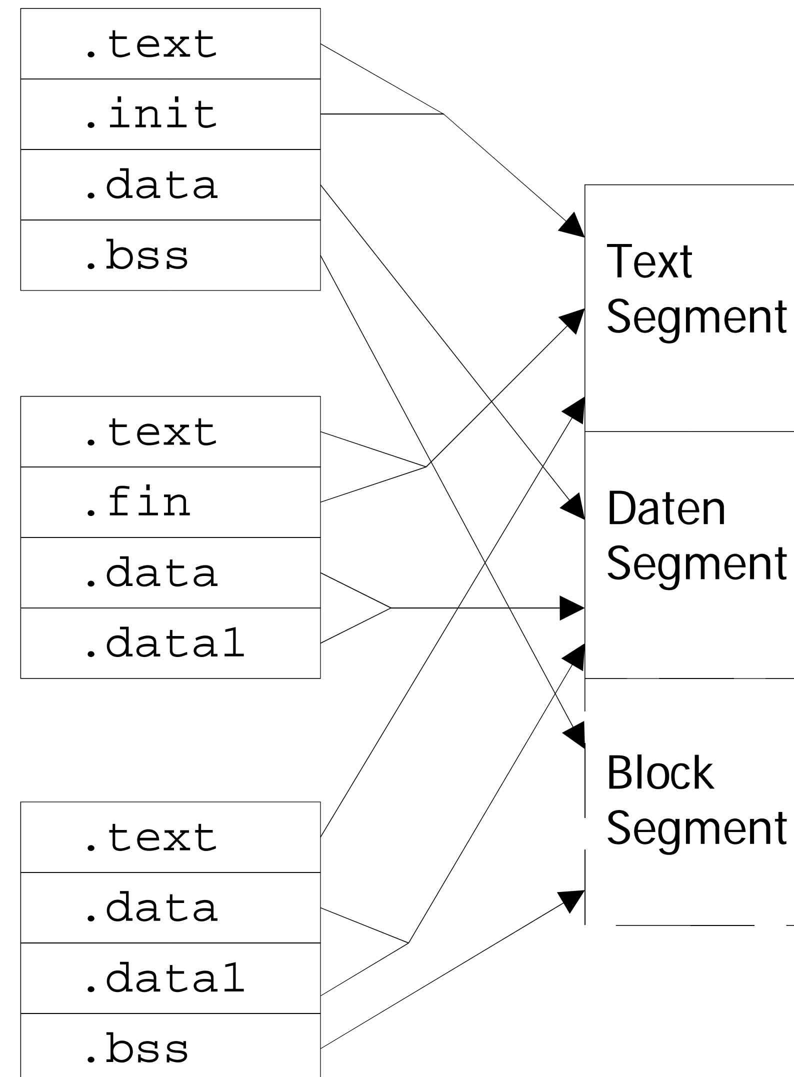


Relozierbar oder Bibliothek

- Besteht aus Menge von Sektionen
- Jede Sektion enthält genau einen Typ
  - `PROGBITS` - Programm, Daten und Testinformation
  - `NOBITS` - wie `PROGBITS` ohne Speicherallokation
  - `SYMTAB` und `DYNSYM` – Symboltabelle
  - `STRTAB` - Stringtabelle
  - `REL` und `RELA` - Relokationsinformation (Substitution, Addition)
  - `DYNAMIC` und `HASH` – Information für dynamisches Binden und Laufzeit Haschtabelle
- Zusätzliche Attribute: `ALLOC`(Sektion benötigt Speicher), `WRITE` (Sektion ist schreibbar) und `EXECINSTR` (Sektion enthält Code)
  - Bsp.: `.text` in Assemblerquellen ist Typ `PROGBITS` mit Attributen `ALLOC` und `EXECINSTR`

Ausführbar

- identisches globales Format wie bisher
- Anordnung so, daß Datei in den Speicher abgebildet und ausgeführt werden kann
- äquivalente Sektionen werden in Segmenten zusammengefaßt
- enthält i.a. nur wenige Segmente (für Code und Daten)



Beispiel

```
extern int i; extern void f(int); int main(){ f(i); return(0);}
```

```
ELF Kopf
```

- Identifikator für ELF-Format “0x7fELF” (magic number)
- Typ: relozierbar oder ausführbar oder Bibliothek
- Prozessor
- Version
- Anzahl und Größe der Einträge in Programmtabelle
- Anzahl und Größe der Einträge in Sektionentabelle
- Index der Stringtabelle
- ...

Beispiel

```
extern int i; extern void f(int); int main(){ f(i); return(0);}
```

	Sektionentabelle
	ELF Kopf
Sek 0	0 .. 0
Sek 1	9, STRTAB
Sek 2	27, SYMTAB
Sek 3	41, REL
Sek 4	0, PROGBITS, ALLOC, EXECINSTR

- 5 Sektionen in Objektdatei
  - reservierte Sektion, enthält 0 Bytes
  - Sektion für Stringtabelle (mit Adresse)
  - Sektion für Symboltabelle
  - Sektion für Relokationsinformation
  - Sektion für Programmcode

Beispiel

```
extern int i; extern void f(int); int main(){ f(i); return(0);}
```

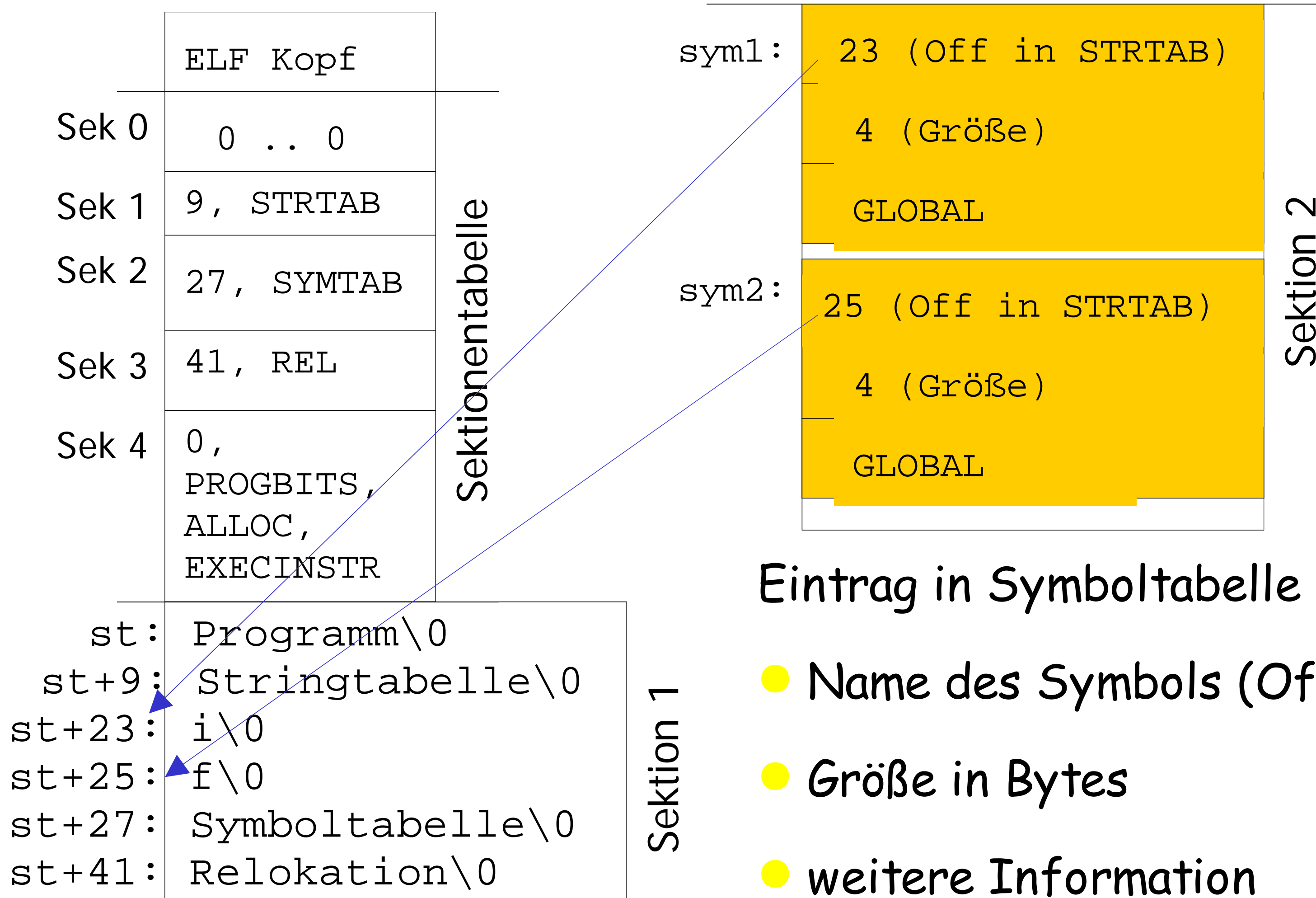
	Sektionentabelle
	ELF Kopf
Sek 0	0 .. 0
Sek 1	9, STRTAB
Sek 2	27, SYMTAB
Sek 3	41, REL
Sek 4	0, PROGBITS, ALLOC, EXECINSTR

	Sektion 1
st:	Programm\0
st+9:	Stringtabelle\0
st+23:	i\0
st+25:	f\0
st+27:	Symboltabelle\0
st+41:	Relokation\0

- Stringtabelle enthält:
  - symbolische Namen der Sektionen
  - Bezeichner des Programms

Beispiel

```
extern int i; extern void f(int); int main(){ f(i); return(0);}
```

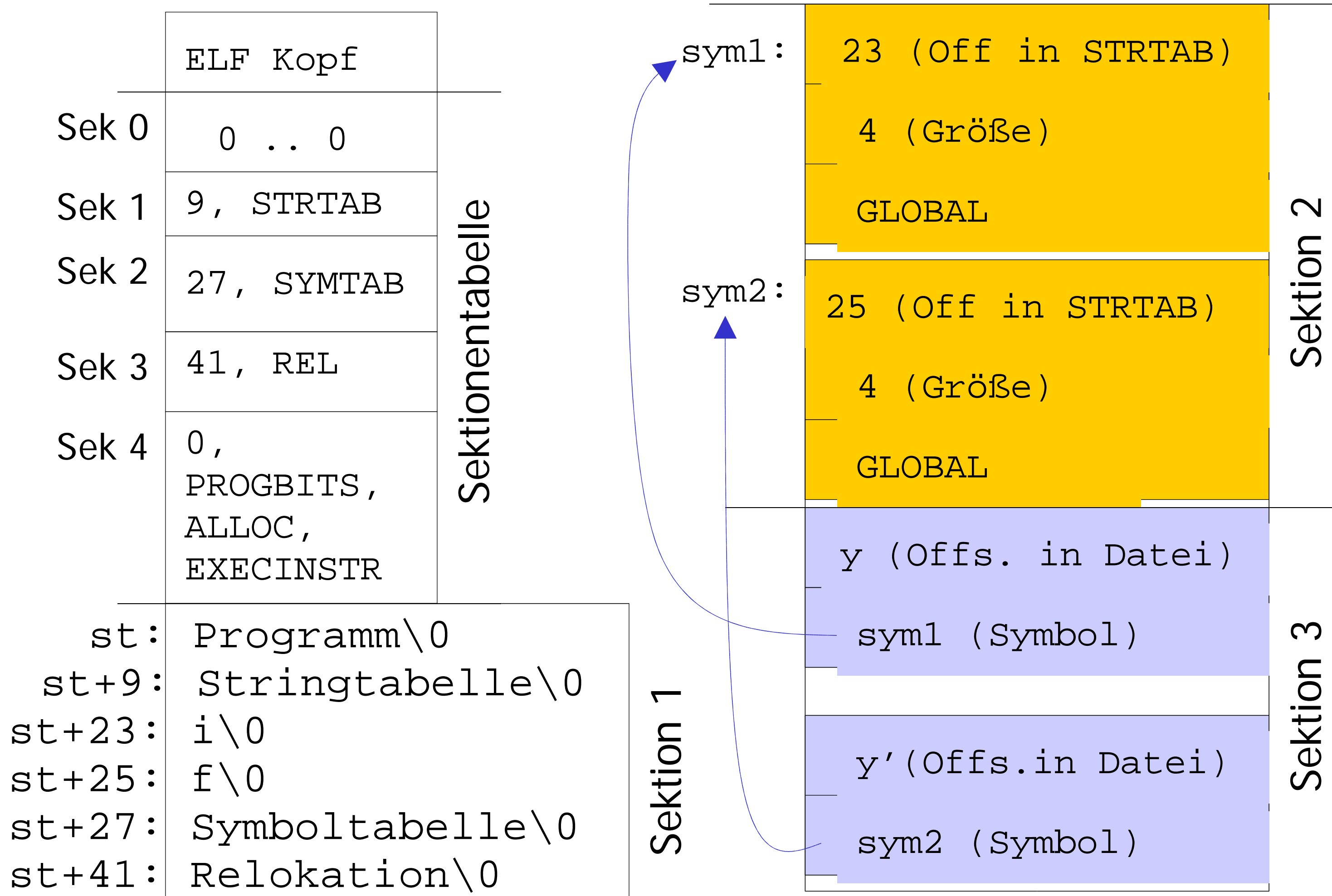


Eintrag in Symboltabelle enthält:

- Name des Symbols (Offset in STRTAB)
- Größe in Bytes
- weitere Information

Beispiel

```
extern int i; extern void f(int); int main(){ f(i); return(0);}
```



Relokationsinfo enthält:

- Offset der zu ändernden Stelle
- zu änderndes Symbol

Beispiel

```
extern int i; extern void f(int); int main(){ f(i); return(0);}
```

	ELF Kopf
Sek 0	0 .. 0
Sek 1	9, STRTAB
Sek 2	27, SYMTAB
Sek 3	41, REL
Sek 4	0, PROGBITS, ALLOC, EXECINSTR

Sektionentabelle

	Sektion 1
st:	Programm\0
st+9:	Stringtabelle\0
st+23:	i\0
st+25:	f\0
st+27:	Symboltabelle\0
st+41:	Relokation\0

Sektion 2

sym1:	23 (Off in STRTAB)
	4 (Größe)
	GLOBAL
sym2:	25 (Off in STRTAB)
	4 (Größe)
	GLOBAL

Sektion 3

y (Offs. in Datei)
sym1 (Symbol)
y'(Offs.in Datei)
sym2 (Symbol)

Sektion 4

	load
y:	i
	call
y':	f