

U1 3. Übung

- Besprechung Aufgabe 0
- Make und Makefiles
- Aufgabe 2

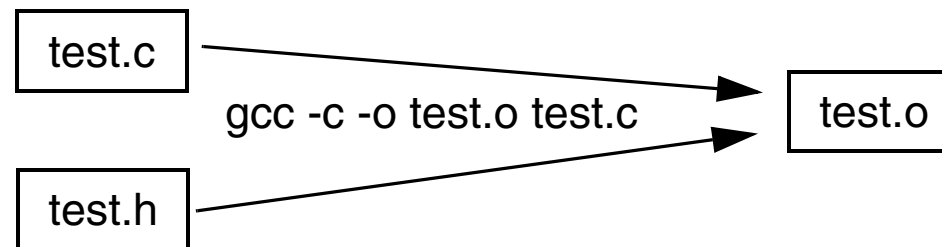
U1-1 Hinweise

- Die Kommandos auf den folgenden Folien dienen der Veranschaulichung
- Sind nicht *direkt* auf die Übungsaufgabe übertragbar
- Insbesondere fehlen die bekannten, notwendigen Compileroptionen!

U1-2 Make

■ Erzeugung von Dateien aus anderen Dateien.

◆ z.B. Erzeugung einer .o-Datei aus einer C-Datei durch C-Compiler

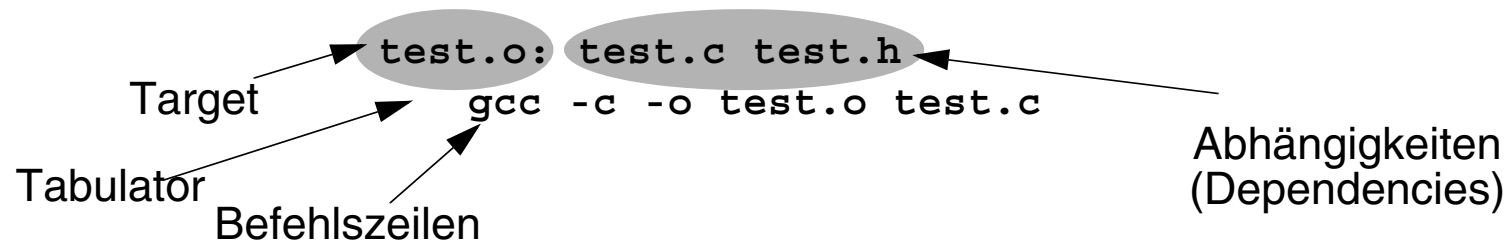


■ Ausführung von *Update*-Operationen

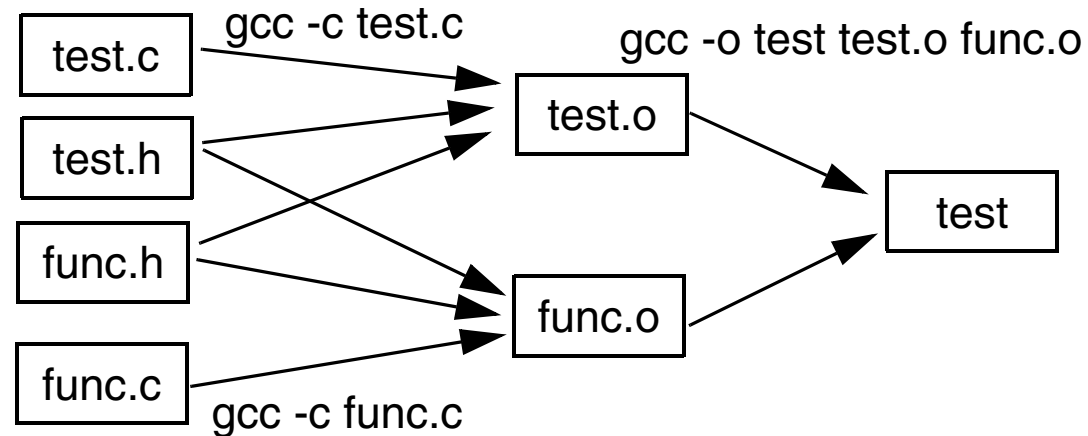
■ Regeldatei mit dem Namen **Makefile**:

◆ Targets (was?) und Abhängigkeiten (woraus?)

◆ Befehlszeilen (wie?)



1 Beispiel mit mehreren Modulen



```
test: test.o func.o
      gcc -o test test.o func.o
```

```
test.o: test.c test.h func.h
      gcc -c test.c
```

```
func.o: func.c func.h test.h
      gcc -c func.c
```

2 Allgemeines

- Kommentare beginnen mit # (bis Zeilenende)
- Befehlszeilen müssen mit TAB beginnen
- Nach der letzten Befehlszeile einer Regel muss eine Zeile folgen, die weder mit TAB noch mit # beginnt
- das zu erstellende Target kann beim `make`-Aufruf angegeben werden (z.B. `make test`)
 - ◆ ohne explizite Target-Angabe bearbeitet make das erste Target im Makefile
- beginnt eine Befehlszeile mit '@', wird sie nicht ausgegeben
- beginnt eine Befehlszeile mit '-', führt ein Fehler nicht zum Abbruch
- jede Zeile wird in einer neuen Shell ausgeführt
 - ◆ `cd` in einer Zeile hat keine Auswirkung auf die nächste Zeile

3 Makros

- in einem Makefile können Makros definiert werden

```
SOURCE = test.c func.c
```

- Verwendung der Makros mit `$(NAME)` oder `${NAME}`

```
test: $(SOURCE)
    gcc -o test $(SOURCE)
```

- Erzeugen neuer Makros durch Ersetzung in existierenden Makros

```
OBJS = $(SOURCE:.c=.o)
```

- ◆ In allen Wörtern, die auf den Suchstring `.c` enden, wird dieser durch `.o` ersetzt

- Erzeugung neuer Makros durch Konkatination

```
ALLOBJS = $(OBJS) hallo.o
```

4 Dynamische Makros

- `$@` Name des Targets (hier: *test*)

```
test: $(SOURCE)
      gcc -o $@ $(SOURCE)
```

- `$*` Basisname des Targets (ohne Dateiendung, hier *test*)

```
test.o: test.c test.h
      gcc -c $*.c
```

- `$<` Name einer Abhängigkeit (in impliziten Regeln)

5 Eingebaute Regeln und Makros

- make enthält eingebaute Regeln und Makros (`make -p` zeigt diese an)

- Wichtige Makros:

- ◆ `CC` C-Compiler Befehl
- ◆ `CFLAGS` Optionen für den C-Compiler
- ◆ `LD` Linker Befehl
(in der Praxis wird aber meist gcc verwendet, weil direkter Aufruf von ld die Standard-Bibliotheken nicht mit einbindet - gcc ruft intern bei Bedarf automatisch ld auf)
- ◆ `LDFLAGS` Optionen für den Linker

- Wichtige Regeln:

- ◆ `.c.o` C-Datei in Objektdatei übersetzen
- ◆ `.c` C-Datei übersetzen und linken

6 Suffix Regeln

- Allgemeine Regel zur Erzeugung einer Datei mit einer bestimmten Endung aus einer gleichnamigen Datei mit einer anderen Endung.

- Beispiel: Erzeugung von **.o**-Dateien aus **.c**-Dateien

```
.c.o:
    $(CC) $(CFLAGS) -c $<
```

- Dateiendungen müssen deklariert werden, als Abhängigkeiten des Spezialtargets **.SUFFIXES**

```
.SUFFIXES: .c .o
```

- Explizite Regeln überschreiben die Suffix-Regeln

```
test.o: test.c
    $(CC) $(CFLAGS) -DXYZ -c $<
```

- Regeln ohne Kommandos können Abhängigkeiten überschreiben

```
test.o: test.c test.h func.h
```

- ◆ die Suffix-Regel wird weiterhin zur Erzeugung herangezogen

7 Beispiel verbessert

```
SOURCE = test.c func.c
OBJS = $(SOURCE:.c=.o)
HEADER = $(SOURCE:.c=.h)

test: $(OBJS)
    $(CC) $(LDFLAGS) -o $@ $(OBJS)

# Suffix-Regeln
.SUFFIXES: .c .o

.c.o:
    @echo Folgende C-Datei wird neu uebersetzt: $<
    $(CC) $(CFLAGS) -c $<

# korrekte Abhaengigkeiten
test.o: test.c $(HEADER)

func.o: func.c $(HEADER)
```

8 GNU Make Erweiterungen

- Funktionsumfang von POSIX.2 make sehr eingeschränkt
- viele Make Implementierungen mit z.T. inkompatiblen Erweiterungen
 - ◆ BSD Make (verschiedene Variationen)
 - ◆ Sun Make (Solaris)
 - ◆ Microsoft nmake
 - ◆ smake
 - ◆ GNU Make (gmake), installiert als *make* im CIP-Pool

9 Pseudo-Targets

- Dienen nicht der Erzeugung einer gleichnamigen Datei
- Deklaration als Abhängigkeiten des Spezial-Targets **.PHONY**

```
.PHONY: all clean install
```

- ◆ so deklarierte Targets werden immer gebaut, auch wenn eine gleichnamige Datei bereits existiert, die aktueller als die Abhängigkeiten ist

- Aufräumen mit `make clean`

```
clean:  
    rm -f $(OBJS) test
```

- Projekt bauen mit `make all` (Konvention: `all` ist immer erstes Target)

```
all: test
```

- Installieren mit `make install`

```
install: all  
    cp test /usr/local/bin
```

10 Eingebaute Funktionen

- Ausgabe eines Shell-Kommandos einem Verzeichnis zuweisen

```
CURRENTDIR = $(shell pwd)
```

- Dateinamen nach einem Shell-Wildcard-Muster suchen

```
SOURCE = $(wildcard *.c)
```

11 Dynamische Makros

- $\$^$ Mit Leerzeichen getrennte Liste aller Abhängigkeiten

12 Einbinden anderer Makefiles

- `include`-Anweisung (am Zeilenanfang, ohne Tabulator)
`include /proj/i4spic/xxx.mk`
- die Datei wird an Stelle der `include`-Anweisung eingebunden

U1-3 Aufgabe 2: Makefile

- Baut `timing.elf`
- Bindet `debug.mk` ein
- Pseudotarget `all`: Baut `timing.elf`
- Pseudotarget `clean`: löscht alle *von Make erzeugten* Dateien
- Alle notwendigen Compiler-Optionen sollen verwendet werden