Bedienungshinweise für *XEmacs*, *make* und GNU Debugger

Labor "Rechnerorganisation"
Berufsakademie Stuttgart
Fachrichtung Nachrichtentechnik

Stand 1. Februar 2007

`http://home.ntz.de/phleps/programming/werkzeuge.pdf`

Inhaltsverzeichnis

1 Zweck der Unterlage

Diese Unterlage gibt Hinweise zum Entwickeln von C-Programmen unter UNIX mit

- dem *XEmacs* als Entwicklungsumgebung,
- GNU *make* zum Steuern des Übersetzens der Programme und
- dem GNU Debugger *gdb*.

Die Beschreibung bezieht sich auf *XEmacs* Version 20.4, GNU *make* Version 3.76 und *gdb* Version 4.16. Ausführliche Informationen können den im *XEmacs* integrierten Info-Seiten bzw. der Online-Hilfe zu *make* bzw. *gdb* (Aufruf: **`man make`** bzw. **`man gdb`**) entnommen werden.

2 Konfiguration des *XEmacs*

Der XEmacs ist ein sehr mächtiges Werkzeug. Er bietet viele Möglichkeiten, um die Konfiguration an die persönlichen Vorlieben anzupassen.

2.1 Konfiguration der UNIX-Umgebung

Um mit dem XEmacs arbeiten zu können, müssen folgende Angaben in Ihrer UNIX-Arbeitsumgebung korrekt angegeben sein:

- Die Variable *PATH* muss das Verzeichnis enthalten, in dem das Programm **`xemacs`** steht.
  Beim *XEmacs* von Linux-Distributionen sollte *PATH* automatisch auch das Verzeichnis enthalten, in dem der *XEmacs* installiert wurde.
- Die Variable *DISPLAY* muss korrekt gesetzt sein, damit X Window die Ausgaben zum richtigen Bildschirm schickt.
  Dies ist automatisch der Fall, wenn Sie mit X Window
  - direkt am lokalen Rechner arbeiten, oder
  - über SSH (= Secure Shell) bzw. **`slogin`** auf einem anderen Rechner arbeiten und *ForwardX11* aktiviert ist.

Falls diese Variablen nicht schon von vornherein richtig gesetzt sind, kann jeder Anwender sie in den Konfigurationsdateien in seinem HOME-Verzeichnis nach seinem persönlichen Bedarf setzen. Diese Konfigurationsdateien sind, je nach verwendeter Shell, bei der

bash die Datei **`.bashrc`** oder **`.bash_profile`**

ksh z. B. die Datei **`.kshrc`** oder **`.profile`**

Da die Namen der Konfigurationsdateien mit einem Punkt beginnen, werden Sie von den Befehlen **`ls`** und **`ll`** nur mit der Option **`-a`** angezeigt, also z. B. mit

```
ls -al
```

Wenn Konfigurationsdateien von Kollegen übernommen werden, müssen Sie prüfen, ob diese Dateien absolute Pfadangaben enthalten. Solche Pfadangaben müssen auf die eigenen Verzeichnisse angepasst werden!

2.2 Konfigurationsdateien für *XEmacs*

Die Datei `http://home.ntz.de/phleps/programming/xemacs-cfg-1.1.tgz` enthält folgende Konfigurationsdateien, die Sie als Basis für Ihre eigene *XEmacs*-Konfiguration verwenden können:

- **`.emacs`**
- **`.emacs-places`**
- **`.xemacs-options`**

Da auch diese Dateinamen mit einem Punkt beginnen, gelten die Hinweise des Abschnitts 2.1 entsprechend.

3 Arbeiten mit dem *XEmacs*

Der *XEmacs* wird mit dem Befehl

`xemacs` *Dateiname*

gestartet.
Die wesentlichen Funktionen des *XEmacs* sind über die Menü-Leiste und über die Schaltflächen erreichbar. Wenn der Maus-Cursor über einer Schaltfläche steht, wird die Bedeutung der Schaltfläche jeweils am unteren Rand des XEmacs-Fensters angezeigt.
Wer Tasten-Kürzel mag und sie kennt, kann natürlich auch damit arbeiten.
Hilfe ist über die Schaltfläche Info erreichbar.

4 Programme übersetzen mit *make*

Syntax:	**`make`**	erzeugt das im Makefile definierte *Default-Target*
	`make` *dateiname*	baut *dateiname*

Das Programm *make* erleichtert das Erzeugen von Programmen oder Dateien, die aus mehreren Quelldateien bestehen. *make* wurde ursprünglich meistens zusammen mit C-Compilern ausgeliefert, lässt sich aber allgemein einsetzen.
Von *make* gibt es viele verschiedene Implementierungen mit unterschiedlichen Leistungsmerkmalen und teilweise individuellen syntaktischen Erweiterungen. Die leistungsfähigste Variante, die ich kenne, ist *GNU make*, die für alle gängigen Systeme verfügbar ist.
Die folgende Beschreibung gilt für *GNU make*, verwendet jedoch nur solche Merkmale, die auch mit den meisten anderen *make*-Varianten arbeiten sollten.
Das Programm *make* verfolgt die Idee, dass eine Datei (Ziel-Datei = *target*) aus einer oder mehreren Quelldateien = *dependencies* erzeugt wird, indem ein Befehl = *command* – oder eine Liste von Befehlen – ausgeführt wird.
Dabei wird vorausgesetzt, dass eine Ziel-Datei nur dann neu erstellt werden muss,

- wenn sie noch nicht existiert, oder
- wenn mindestens eine ihrer Quelldateien neueren Datums ist als die Ziel-Datei.

make verfolgt diese Abhängigkeiten über mehrere Stufen und kann auf beliebige Dateien, z. B. auch auf Text-Dateien, angewendet werden.
Die Abhängigkeiten der Dateien und die Regeln zum Erzeugen der Ziel-Dateien werden als Text in Beschreibungs-Dateien = *Makefiles* abgelegt, die *make* liest und abarbeitet. Diese Dateien haben normalerweise den Namen *Makefile* oder *makefile*.

4.1 *Makefile*-Syntax

Regeln im *Makefile* haben folgende Syntax:

```
target:    dependencies
        command
        ...
```

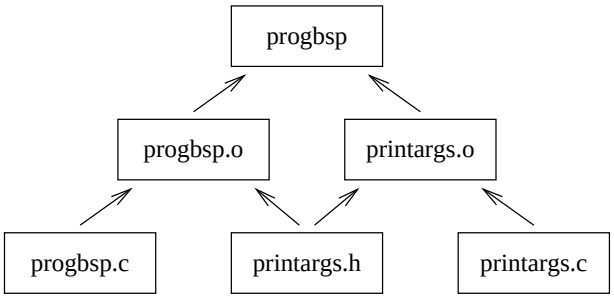
oder

```
target:    dependencies \
           dependencies
        command
        ...
```

oder

```
target:    dependencies ; command
        command
        ...
```

Jedes Ziel (=*target*) muss am Zeilenanfang stehen. Zwischen dem Doppelpunkt, der hinter dem Ziel steht, und der ersten Quelldatei muss mindestens ein Tabulator-Zeichen sein.[1]
Befehlszeilen müssen mit einem Tabulator-Zeichen beginnen.
Zwischen den Zeilen mit der Abhängigkeit und den zugehörigen Befehlszeilen dürfen keine Leerzeilen sein! Leerzeilen markieren das Ende einer Regel.
Beispiel:



Für das Zusammenbauen von *progbsp* könnte folgendes *Makefile* verwendet werden:

```
#
# Makefile für das Programm "progbsp":
# Sehr ausführliche Version 1

#       Makro-Definitionen:
#       zu verwendender Compiler:
CC      = gcc

#       Compiler-Flags, hier für GNU C-Compiler gcc:
CFLAGS  = -Wall -ansi -pedantic -g


#       Default-Target = progbsp: Regel mit zugehörigem Befehl
progbsp:        progbsp.o printargs.o
```

[1] Manche Implementierungen von **make** begnügen sich auch mit Leerzeichen statt Tabulatoren.

```
        $(CC) $(CFLAGS) progbsp.o printargs.o -o progbsp

#       Erstellen der Objekt-Dateien: Regeln mit jeweils zugehörigem Befehl
progbsp.o:      progbsp.c printargs.h
        $(CC) -c $(CFLAGS) progbsp.c

printargs.o:    printargs.c printargs.h
        $(CC) -c $(CFLAGS) printargs.c
```

Wenn das aktuelle Verzeichnis nur die Dateien *Makefile*, *printargs.c*, *printargs.h* und *progbsp.c* enthält, würde die *XEmacs*-Schaltfläche Compile oder der Befehl **make** bzw. **make progbsp** folgenden Ablauf bewirken:

1. **make** sucht nach einer Beschreibungs-Datei und findet *Makefile*. Diese Datei wird zeilenweise gelesen und bearbeitet:

2. Alle Zeichen, die zwischen # und dem Ende der betreffenden Zeile stehen, werden als Kommentar behandelt und nicht weiter untersucht.

3. Die Zeilen

   CC = gcc

   CFLAGS = -Wall -ansi -pedantic -g

   definieren und besetzen die Makros *CC* und *CFLAGS*. Diese Makro-Namen sind allgemein üblich für den Compiler und die Compiler-Flags. Default-Einstellung ist

   CC = cc

   Unter UNIX kann der C-Compiler sowohl zum Übersetzen, als auch für den Aufruf des Linkers verwendet werden.

4. Die Zeilen

   progbsp:<TAB>progbsp.o printargs.o

   <TAB>$(CC) $(CFLAGS) progbsp.o printargs.o -o progbsp

   sind die erste in *Makefile* gefundene Regel.

   Wenn **make** ohne Argumente aufgerufen wird, betrachtet **make** das Ziel der ersten Abhängigkeit als zu erstellendes Endprodukt[2]. Im Beispiel hätten die Aufrufe **make progbsp** und **make** die selbe Wirkung.

   Das Endprodukt ist in diesem Fall das Programm *progbsp*, das entsprechend der Zeile

   progbsp:<TAB>progbsp.o printargs.o

   von den Dateien *progbsp.o* und *printargs.o* abhängt.

   Die direkt folgende Zeile (ohne Leerzeilen dazwischen und mit einem Tabulator beginnend!)

   <TAB>$(CC) $(CFLAGS) progbsp.o printargs.o -o progbsp

   gibt an, mit welchem Befehl die Ziel-Datei aus den Quelldateien erzeugt wird. Für den Compiler und die Compiler-Flags werden die oben definierten Makros verwendet.

[2] Bei *Makefiles*, die mehrere Programme erstellen sollen, wird dazu üblicherweise das Default-Target *all* verwendet, bei dem alle zu erstellenden Programme als Abhängigkeiten angegeben werden. Die Regel enthält keinen Befehl. Dadurch wird keine Datei mit dem Namen *all* erstellt.

5. Da die angegebenen Quelldateien *progbsp.o* und *printargs.o* nicht existieren, sucht **make** weiter nach Regeln, mit denen diese Dateien erstellt werden können.

   Passende Regeln werden in den folgenden Zeilen gefunden. Da die Dateien *progbsp.c*, *printargs.h* und *printargs.c* existieren und dafür keine Abhängigkeiten mehr angegeben sind, beginnt **make** nun mit dem Ausführen der Befehle in umgekehrter Reihenfolge der gefundenen Abhängigkeiten:

6. Erstellen von *progbsp.o*:

   ```
   gcc -c -Wall -ansi -pedantic -g progbsp.c
   ```

   Die Option **-c** bewirkt, dass nur übersetzt, aber nicht gelinkt wird.

7. Erstellen von *printargs.o*

   ```
   gcc -c -Wall -ansi -pedantic -g printargs.c
   ```

8. Erstellen von *progbsp*:

   ```
   gcc -Wall -ansi -pedantic -g progbsp.o printargs.o -o progbsp
   ```

4.2 Allgemeine Regeln, automatische Variablen

In der oben gezeigten ersten Version des *Makefile* wurde bei jeder Regel der zugehörige Befehl explizit mit angegeben. Dies kann jedoch durch Einführen von allgemeinen Regeln vereinfacht werden. Eine allgemeine Regel enthält nur die Datei-Endungen, bei denen sie anzuwenden ist, und den zugehörigen Befehl, z. B.

```
#       Allgemeine Regel, mit der aus einer C-Quelldatei (Endung ".c") eine
#       Objektdatei (Endung ".o") erzeugt wird:
.c.o:
        $(CC) -c $(CFLAGS) $*.c
```

Eine allgemeine Regel wird immer dann angewendet, wenn die Datei-Endungen einer Abhängigkeit passen, aber kein Befehl angegeben ist.
Zum allgemeingültigen Formulieren von Regeln bietet *make* einige Variablen, die bei jeder angewendeten Regel entsprechend der Abhängigkeiten automatisch besetzt werden. Die wichtigsten sind:

$@ Name der Ziel-Datei, die mit der Regel erzeugt wird

$< Name der ersten Quelldatei der Regel

$? Liste aller Quelldateien, die neuer sind als die Ziel-Datei

$+ Liste aller Quelldateien der Regel

$* Name der Ziel-Datei ohne Datei-Endung (z. B. *test* bei Ziel-Datei *test.o*)

Damit lässt sich das *Makefile* vereinfachen:

```
#
# Makefile für das Programm "progbsp"
#
# Version 2 mit allgemeinen Regeln.

#       zu verwendender Compiler:
CC      = gcc

#       Compiler-Flags, hier für GNU C-Compiler gcc:
CFLAGS  = -Wall -ansi -pedantic -g


#       Allgemeine Regel, mit der Objektdateien (Endung ".o") zu einem
#       Programm (Endung "") gelinkt werden:
.o:
        $(CC) $(CFLAGS) $+ -o $@

#       Allgemeine Regel, mit der aus einer C-Quelldatei (Endung ".c") eine
#       Objektdatei (Endung ".o") erzeugt wird:
.c.o:
        $(CC) -c $(CFLAGS) $*.c


#       Default-Target = progbsp: Regel enthält nur noch die Abhängigkeit
progbsp:        progbsp.o printargs.o

#       Abhängigkeiten der Objekt-Dateien:
progbsp.o:      progbsp.c printargs.h

printargs.o:    printargs.c printargs.h
```

Im Programm **`make`** sind viele allgemeine Regeln schon eingebaut. Sie werden immer dann angewendet, wenn keine passenden Befehle im *Makefile* gefunden werden. Die allgemeinen Regeln in Version 2 können deshalb auch weggelassen werden:

```
#
# Makefile für das Programm "progbsp"
#
# Version 3: Da hier keine Regeln angegeben sind, verwendet make seine
# eingebauten Regeln.

#       zu verwendender Compiler:
CC      = gcc

#       Compiler-Flags, hier für GNU C-Compiler gcc:
CFLAGS  = -Wall -ansi -pedantic -g


#       Default-Target = progbsp: Regel enthält nur noch die Abhängigkeit
progbsp:        progbsp.o printargs.o

#       Abhängigkeiten der Objekt-Dateien:
progbsp.o:      progbsp.c printargs.h

printargs.o:    printargs.c printargs.h
```

Mit den eingebauten Regeln lässt sich **`make`** auch zum Zusammenbauen von einfachen Programmen verwenden, die nicht im *Makefile* angegeben sind. Wenn z. B. das Programm *childproc* aus der Quelldatei *childproc.c* erzeugt werden soll, reicht dazu der Befehl

```
make childproc
```

auch wenn kein *Makefile* existiert.

5 Bedienen des GNU-Debuggers *gdb*

5.1 Voraussetzungen

Die Quelldateien, deren Code mit dem Debugger untersucht werden soll, müssen mit der Debug-Option übersetzt und gelinkt sein. Sowohl bei *gcc*, als auch beim HP Compiler, ist dies die Option **`-g`**.
Wenn die Option **`-g`** beim Übersetzen oder Linken vergessen wurde, lässt sich der Debugger trotzdem starten, zeigt aber den zum Programm gehörenden Quellcode nicht an.

5.2 Starten des Debuggers

Der GNU-Debugger kann wahlweise über die UNIX-Kommandozeile oder über den *XEmacs* gestartet werden:

- Start über die UNIX-Kommandozeile:
  `gdb` *Programm-Name*
- Start aus *XEmacs*:
  - Schaltfläche Debug (die mit der Wanze)
  - Auswahl der Datei des ausführbaren Programms mit der **mittleren** Maus-Taste

Der Debugger meldet sich danach mit einem Text und der Eingabeaufforderung *(gdb)*.
Wenn *gdb* über den *XEmacs* gestartet wird, bietet der *XEmacs* eine etwas komfortablere Oberfläche zur Bedienung des *gdb*.

5.3 Starten des Programms im Debugger

Bevor das Programm im Debugger gestartet wird, sollte (mindestens) ein Breakpoint gesetzt werden, z. B. zum Anhalten bei Funktion *main()*:

```
b main
```

Danach kann das Programm mit dem Befehl

`r` *Kommandozeilen-Argumente*

gestartet werden. Das Programm läuft bis zum ersten Breakpoint, zeigt die entsprechende Quellcode-Zeile an und wartet auf weitere Befehle.

5.4 Übersicht der wichtigsten Befehle des gdb

Befehl	Erklärung
help	Ausgabe der gdb-Befehle
<Enter>	wiederholt den letzten Befehl
set args *ProgArg*	besetzt die Kommandozeilen-Argumente für den Programmaufruf mit *ProgArg*
r	startet das Programm. Wenn vorher Argumente mit **set args** gesetzt wurden, werden diese Argumente dem Programm übergeben.
r *ProgArg*	startet das Programm mit den Kommandozeilen-Argumenten *ProgArg*
target core *file*	Core-Datei *file* mit dem Debugger untersuchen
attach *Prozess-ID*	Laufenden Prozess mit *Prozess-ID* mit dem Debugger untersuchen
b *Funktionsname*	setzt einen Breakpoint bei Funktion *Funktionsname*
print *expr*	gibt den Ausdruck *expr* aus
display *expr*	zeigt den Ausdruck *expr* bei jeder Unterbrechung des Programms an
watch *expr*	"*Watchpoint*": Programm hält an, wenn sich der Wert des Ausdrucks *expr* ändert
s *var* **=** *value*	setzt den Inhalt der Variablen *var* auf *value*
info break	zeigt alle gesetzten "*Breakpoints*" und "*Watchpoints*" an
info args	zeigt alle Argumente der aktuellen Funktion an
info display	zeigt alle gesetzten "*display*" Ausdrücke
c	"*continue*": Programm fortsetzen, z. B. bis zum nächsten "*Breakpoint*"
next	führt die nächste Programm-Zeile (der aktuellen Funktion) aus. Funktionsaufrufe werden wie eine Programm-Zeile behandelt.
step	führt den nächsten Programmschritt aus. Bei Funktionsaufrufen hält das Programm beim ersten Befehl der aufgerufenen Funktion.
finish	setzt das Programm bis zum Ende der aktuellen Funktion fort
bt	"*backtrace*": zeigt die Aufruf-Hierarchie der Funktionen und ihrer Argumente
up	Aufruf-Hierarchie: Wechsel in die aufrufende Funktion
up	Aufruf-Hierarchie: Wechsel in die nächsttiefere (d. h. aufgerufene) Funktion
l	zeigt einige Quellcode-Zeilen vor und hinter der aktuellen Position an. Diese Funktion erübrigt sich, wenn *gdb* vom *XEmacs* gestartet wurde.
q	Programm beenden