

Ablauf des Semesters

14.03.	Einführung (Micro_1)	Pr1, Gr1	(Grundaufbau)
21.03.	Programmer, make	Pr1, Gr2	(Grundaufbau)
28.03.	Bin I/O	Pr2, Gr1	(Input Polling)
04.04.	Memory, Fuses	Pr2, Gr2	(Input Polling)
11.04.	LCD - Display, Progmem	Pr3, Gr1	(LCD Display)
18.04.	AVR-Assembler	Pr3, Gr2	(LCD Display)
25.04.	Interrupts, Timer, PWM	Pr4, Gr1	(Interrupts)
02.05.	eagle Schaltungsentwicklung	Pr4, Gr2	(Interrupts)
09.05.	Analog-Digitalwandlung	Pr5, Gr1	(Analog Digital)
16.05	eagle Layout	Pr5, Gr2	(Analog Digital)
23.05	RS232	Pr6, Gr1	(Uart)
30.05.	MP430	Pr6, Gr2	(Uart)
06.06.	TWI	Pr7, Gr1	(TWI)
13.06.	Assembler Intel86	Pr7, Gr2	(TWI)
20.06.	PWM	Pr8, Gr1	(PWM)
27.06.	Konsultation	Pr8, Gr2	(PWM)

mündliche Prüfung (einzeln)

Make

Buildtool der Unixwelt

Basiert auf Abhängigkeiten und Zeitstempel

Aktualität der Quellen/Ziele

Macrodefinitionen

Vordefinierte Makros

Abhängigkeiten/Kommandozeilen

Suffixrules

Aufruf:

`make [-f makefile] [options] [targets] [macro defs]`

Automatismen

Make ohne makefile
Automatische Wahl
des Buildwerkzeugs
nach Extension


```
> ls  
> sinus.c  
> make sinus  
cc    sinus.c  -o sinus  
> ls  
sinus  sinus.c  
>
```

```
> make Date  
g++   Date.cpp -o Date  
>
```

Mehrere Quellfiles

Angabe des Compilers bei .C statt .cpp unnötig, als Linker wird aber cc aufgerufen, was nicht geht!!

Gibt es mehrere Quellfiles, ist ein minimales description file (Makefile) von Nöten:



```
OBJS = mainStr.o CString1.o
CC    = g++

mainStr : $(OBJS)
```

```
> ls
CString1.cpp CString1.h mainStr.cpp makefile makefile~
> make
g++ -c -o mainStr.o mainStr.cpp
g++ -c -o CString1.o CString1.cpp
g++ mainStr.o CString1.o -o mainStr
> ls
CString1.cpp CString1.o mainStr.cpp makefile
CString1.h mainStr mainStr.o makefile~
>
```

Macrodefinitionen

OBJ	= main.o lcd.o
F_CPU	= 1000000
UISP	= uisp -dprog=stk200
MCU_TARGET	= atmega8
OPTIMIZE	= -O2
DEFS	= -DF_CPU=\$(F_CPU)
LIBS	=
CC	= avr-gcc
CFLAGS	= -Wall \$(OPTIMIZE) -mmcu=\$(MCU_TARGET) \$(DEFS)
LDFLAGS	= -Wl,-Map,\$(PRG).map
OBJCOPY	= avr-objcopy
OBJDUMP	= avr-objdump

Prioritäten:

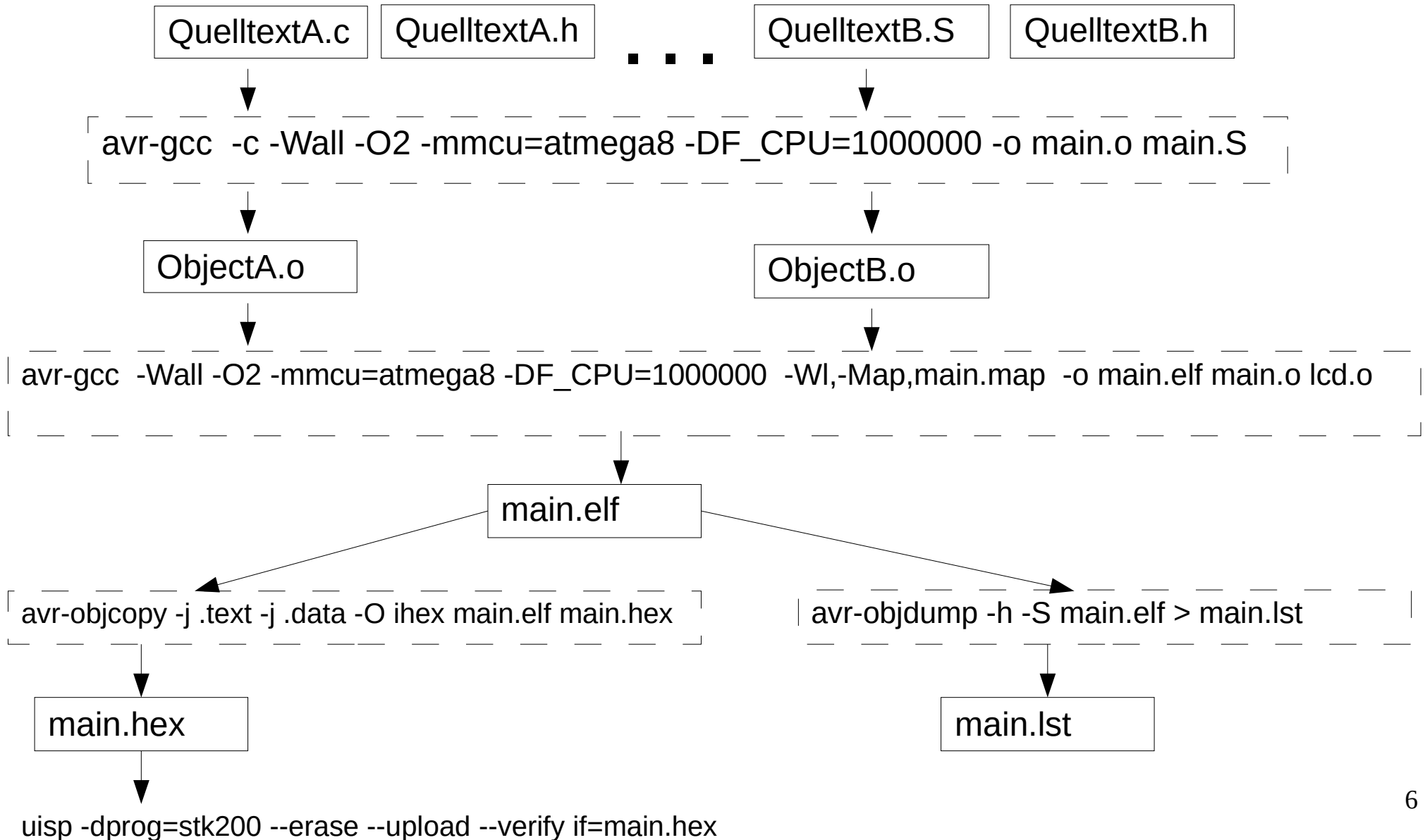
1 internal Macros (defaults)

2 current shell Variables

3 Marcos des Makefiles

4 Macros von der Kommandozeile des make-Aufrufes

Abhängigkeiten



Target
(Ziel)

Abhängigkeiten/builds

all: \$(PRG).elf lst text eeprom

\$(PRG).elf: \$(OBJ)
\$(CC) \$(CFLAGS) \$(LDFLAGS) -o \$@ \$^ \$(LIBS)

Linker

main.o: main.S lcd.h
\$(CC) -c \$(CFLAGS)-o \$@ \$<

Quellen für Target, verwendete Header nicht vergessen

lcd.o: lcd.c lcd.h
\$(CC) -c \$(CFLAGS)-o \$@ \$<

Compiler/Assembler

lst: \$(PRG).lst

%.lst: %.elf
\$(OBJDUMP) -h -S \$< > \$@

clean:

rm -rf *.o \$(PRG).elf *.eps *.png *.pdf *.bak
rm -rf *.lst *.map \$(EXTRA_CLEAN_FILES)

Alle Build-Zeilen beginnen
mit einem Tabulator!!!

Commandline options

http://www.claus-ebert.de/doc/make/quick_reference.html

Option	Beschreibung
-b	Akzeptiere das Makefileformat der vorherigen Implementationen von make.
-d	Debug-Mode - zeigt detaillierte Informationen über interne Flags und die Datei-Timestamps (modified) an.
-e	Umgebungsvariablen überschreiben Makro-Definitionen im Makefile.
-f	Das dieser Option folgende Argument ist das Makefile. Wenn das Argument ein Bindestrich ist wird der Standard-Input verwendet. Es können mehrere Makefiles mit der -f Option angegeben werden. (immer nur ein File per -f) Diese Makefiles werden dann wie in einziges behandelt.
-i	Ignoriere Fehler. (Das gleiche wie .IGNORE: im Makefile)
-k	Ein Fehler beendet die Arbeit an dem aktuellen Zweig der Hierarchie, aber nicht für die anderen Zweige
-n	Schreibt alle Kommandos auf den Bildschirm, aber führt sie NICHT aus
-p	Schreibt Makro-Definitionen, Suffixes, suffix rules und Makefile-Variable auf den Bildschirm
-q	Liefert 0 oder nicht null zurück, je nach dem ob alle Ziel-Dateien aktuell sind oder nicht
-r	Default-Regeln werden abgeschaltet
-s	Schreibt die ausgeführten Kommandos NICHT auf den Bildschirm (Das gleiche bewirkt auch .SILENT: im Makefile)
-t	Toucht alle Zielformateien (setzt den Zeitstempel auf jetzt. Somit erscheinen diese Ziele beim nächsten make-Lauf als up to date)

Einige Interne Macros für Quellen und Ziele

`$@`: Name des aktuellen Zieles, wird oft beim
Compilieren in Verbindung mit `-o` verwendet

`$?`: Liste der Quellen, die neuer sind als das Ziel

`$<`: Name der ersten Quelle

`^`: Liste aller Quellen, durch Leerzeichen getrennt

Beispiel:

Suffix Rules

Regeln, die mit Hilfe der Extensions in den Dateinamen gesteuert werden.

Sie gestatten, die Steuerdatei von make sehr viel kompakter zu gestalten. Die Abhängigkeiten werden allerdings nicht so sauber abgebildet

Mit Suffixregeln werden die Ziele(Targets) aus den gleichnamigen Quellen (Dependencies, prerequisites) nach den geltenden Standards gebaut (.c: gcc, .S: Assembler, .cpp oder .C: g++)

Abhängigkeiten von verwendeten Headerfiles werden nicht beachtet.

Beispiel

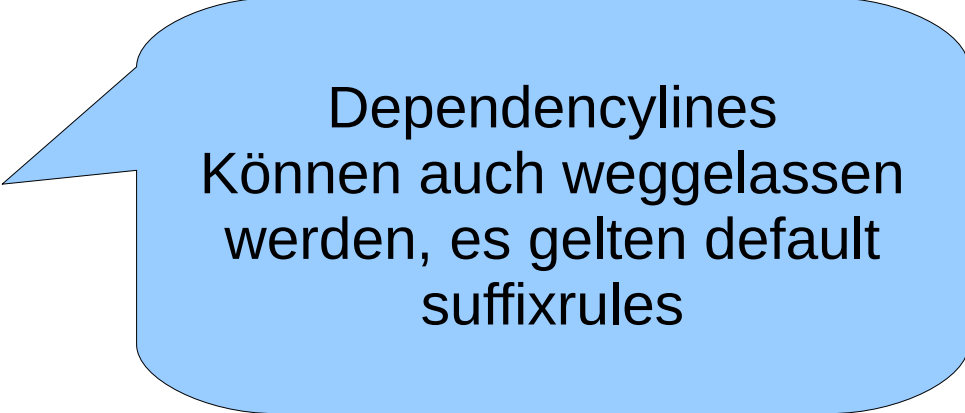
PRG = main
OBJ = main.o lcd.o

...

\$(PRG).elf: \$(OBJ)
\$(CC) \$(CFLAGS) \$(LDFLAGS) -o \$\$ \$^ \$(LIBS)

#main.o: main.S lcd.h
\$(CC) -c \$(CFLAGS)-o \$\$ \$<

#lcd.o: lcd.c lcd.h
\$(CC) -c \$(CFLAGS)-o \$\$ \$<



Dependencylines
Können auch weggelassen
werden, es gelten default
suffixrules

Mit suffixrules geht es auch

```
PRG          = main
OBJ          = main.o lcd.o

...

$(PRG).elf: $(OBJ)
    $(CC) $(CFLAGS) $(LDFLAGS) -o $@ $^ $(LIBS)

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

Da es im Verzeichnis ein main.c und ein main.S gibt, wird main.c der Vorzug eingeräumt, es wird immer zunächst main.c kompiliert oder nach einem .y bzw. .lex -file gesucht. Gibt es keines davon, wird main.S assembliert.

Weiterführende Regeln

Weiterführende Regeln gibt es für

- den Bau von Libraries

- die Arbeit mit verschachtelten Verzeichnisstrukturen, bei denen dann jedes Unterverzeichnis ein eigenes Makefile enthält

Das Kommando `make -p 2>/dev/null | less` liefert alle Variablen und Voreinstellungen, die make u.Ust. verwendet. (sehr viel > 1500 Zeilen)