C für Mikrocontroller (Embedded C)

Vorlesung Mikroprozessortechnik

HAW Hamburg

22. März 2017

1. Programmentwicklung in C
2. Datentypen
3. Pointer
4. Programmentwicklung und Toolchain
5. Operatoren, Präzedenzen, Bindungen
6. Bitoperationen
7. Speicherklassen

1 Programmentwicklung in C

2 Datentypen

3 Pointer

4 Programmentwicklung und Toolchain

5 Operatoren, Präzedenzen, Bindungen

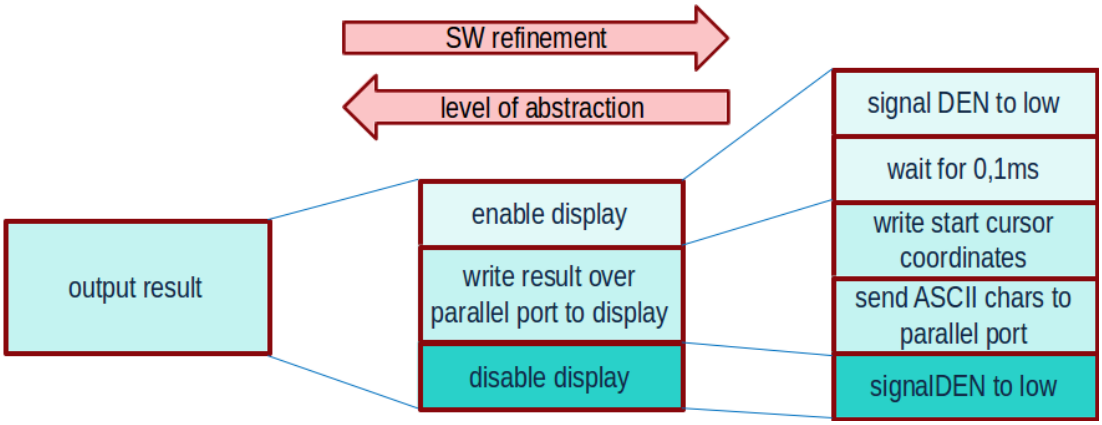
6 Bitoperationen

7 Speicherklassen

Software Design Methodik

- Top-Down-Approach
  - Von der allgemeinen Funktion zur detaillierten Darstellung
  - Decomposition (Stepwise Refinement)
  - wird als gut strukturiert bevorzugt
  - Ein Hilfsmittel dafür: Struktogramme (engl. Structure Charts oder Nassi-Sheidermann-Diagrams)
- Bottom-Up-Approach
  - Von der (gekapselten) Teilfunktion zum (abstrakten) Gesamtsystem
  - Synthese / Composition
  - wird bei schrittweise Aufwachsen der Funktionalität (Seed-Model) und Änderungen/Erweiterungen/Modifikationen angewandt

Top-Down-Approach (Schrittweise Verfeinerung)



Quelle: Lutz Leutelt, Vorlesungsunterlagen

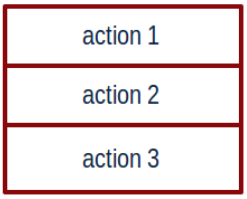
Sequenzen und Blöcke

- **Sequence (of instructions / statements)**

C syntax

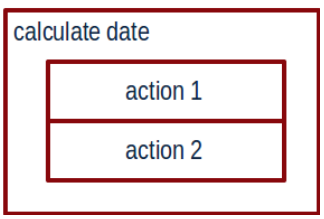
```
statement1;
statement2;
statement3;
```

Nassi-Shneiderman diagram



- **Block of statements**

```
/* comment: calculate date */
{
   statement1;
   statement2;
}
```

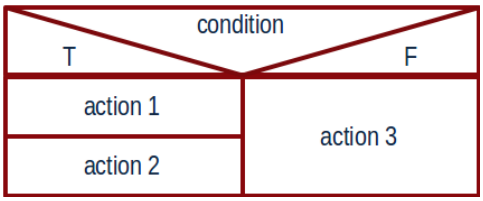


Quelle: Lutz Leutelt, Vorlesungsunterlagen

Bedingte Sequenzen und Blöcke

- **conditional statements**

```
if (condition)
{
   statement1;
   statement2;
}
else
{
   statement3;
}
```

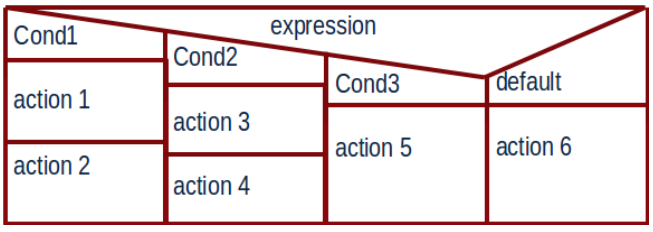


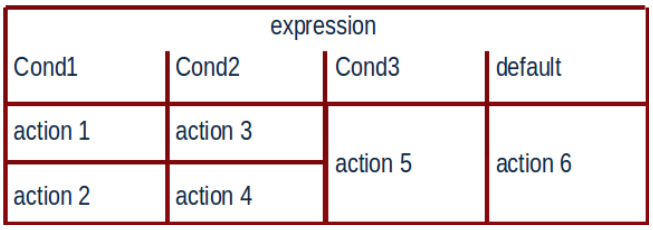
Quelle: Lutz Leutelt, Vorlesungsunterlagen

Fallunterscheidung - Auswahlanweisung

- **case statements**

```
switch (expression)
{
   case Cond1:
      statement1;
      statement2;
      break;
   case Cond2:
      statement3;
      statement4;
      break;
   case Cond3:
      statement5;
      break;
   default:
      statement6;
}
```



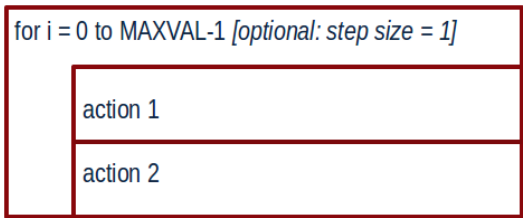


Quelle: Lutz Leutelt, Vorlesungsunterlagen

Zählscheife und abweisende Schleife

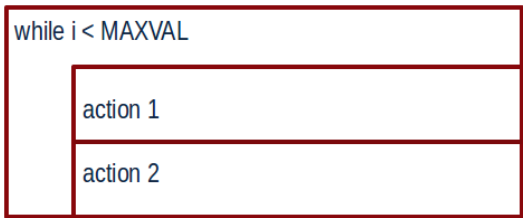
- **for structure**

```
for (i=0;i<MAXVAL;i++)
{
   statement1;
   statement2;
}
```



- **while structure**

```
while (i<MAXVAL)
{
   statement1;
   statement2;
}
```

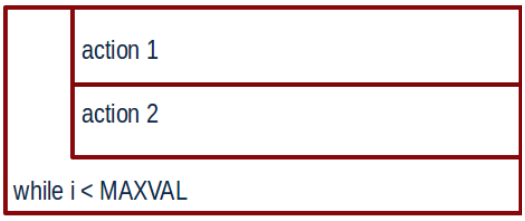


Quelle: Lutz Leutelt, Vorlesungsunterlagen

Nicht-abweisende Schleife und Endlos-Schleife

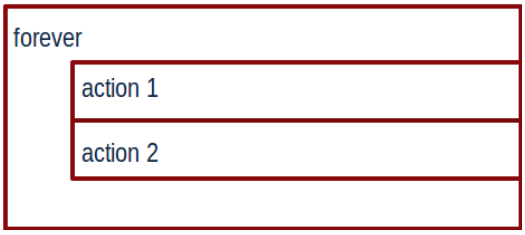
- **do/while structure**

```
do
{
   statement1;
   statement2;
} while (i<MAXVAL)
```



- **endless loop**

```
while (1)
{
   statement1;
   statement2;
}
```

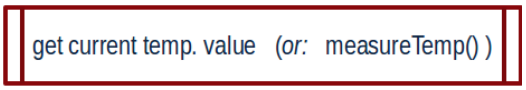


Quelle: Lutz Leutelt, Vorlesungsunterlagen

Funktionsaufruf (Call) und Funktionsdefinition

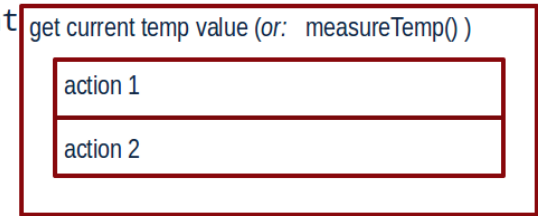
- **function call**

```
/*function call*/
value = measureTemp(var1,var2);
```



- **function definition**

```
char measureTemp(int var1, int
{
   statement1;
   statement2;
   return (value);
}
```



Quelle: Lutz Leutelt, Vorlesungsunterlagen

1 Programmentwicklung in C

2 Datentypen

3 Pointer

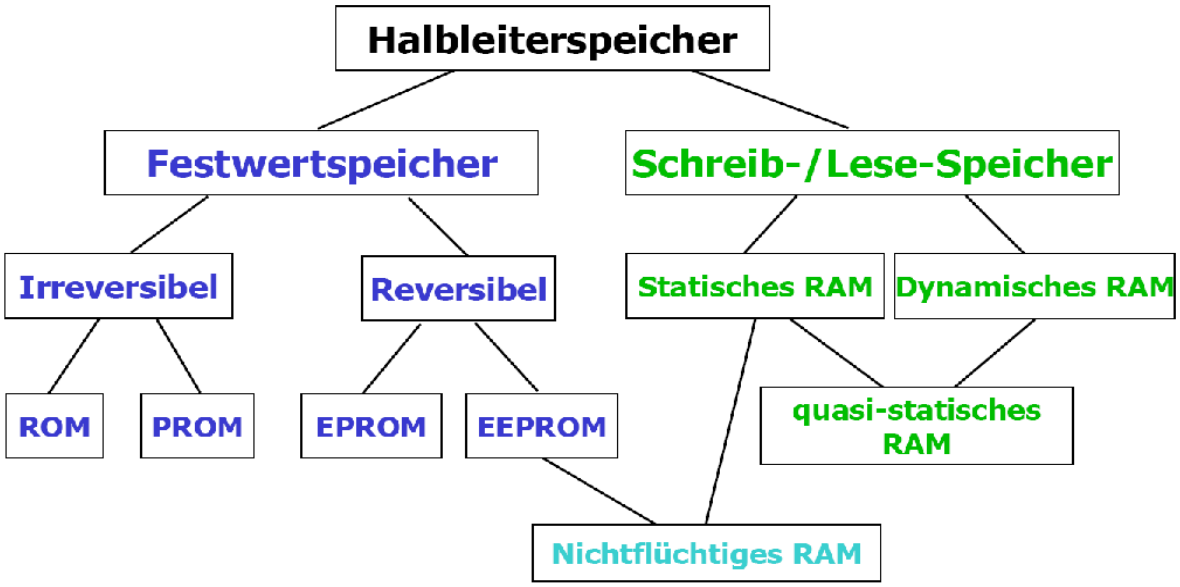
4 Programmentwicklung und Toolchain

5 Operatoren, Präzedenzen, Bindungen

6 Bitoperationen

7 Speicherklassen

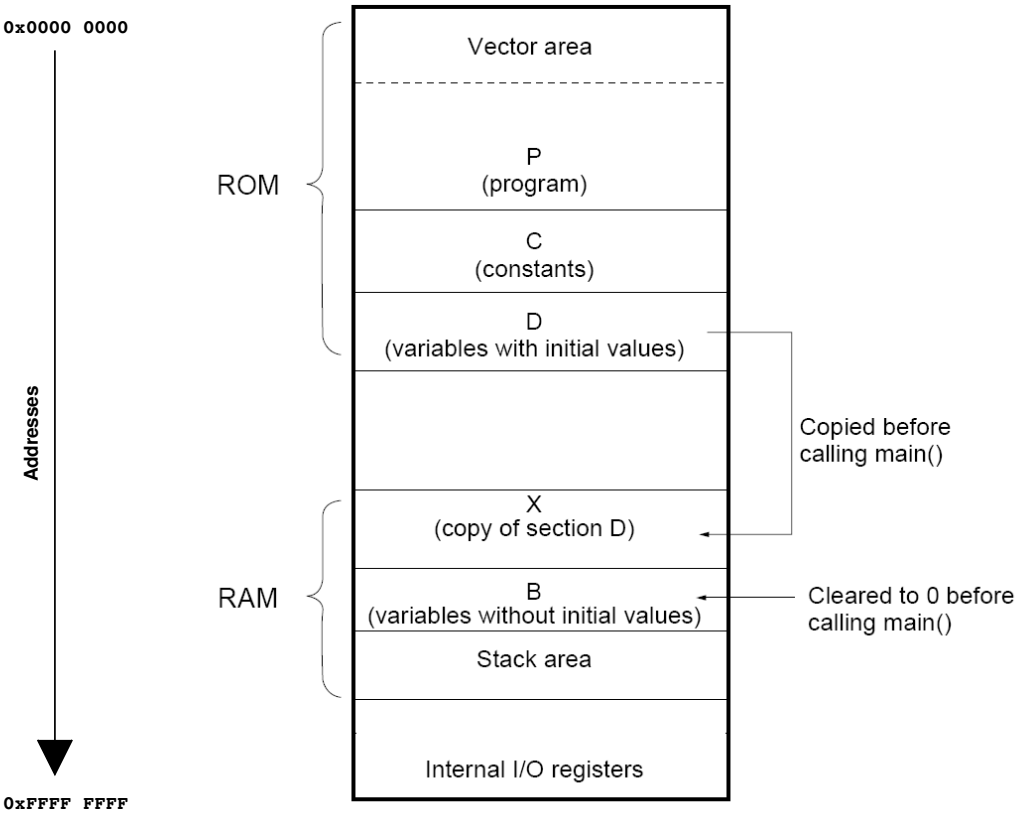
Speicher aus Hardware-Sicht



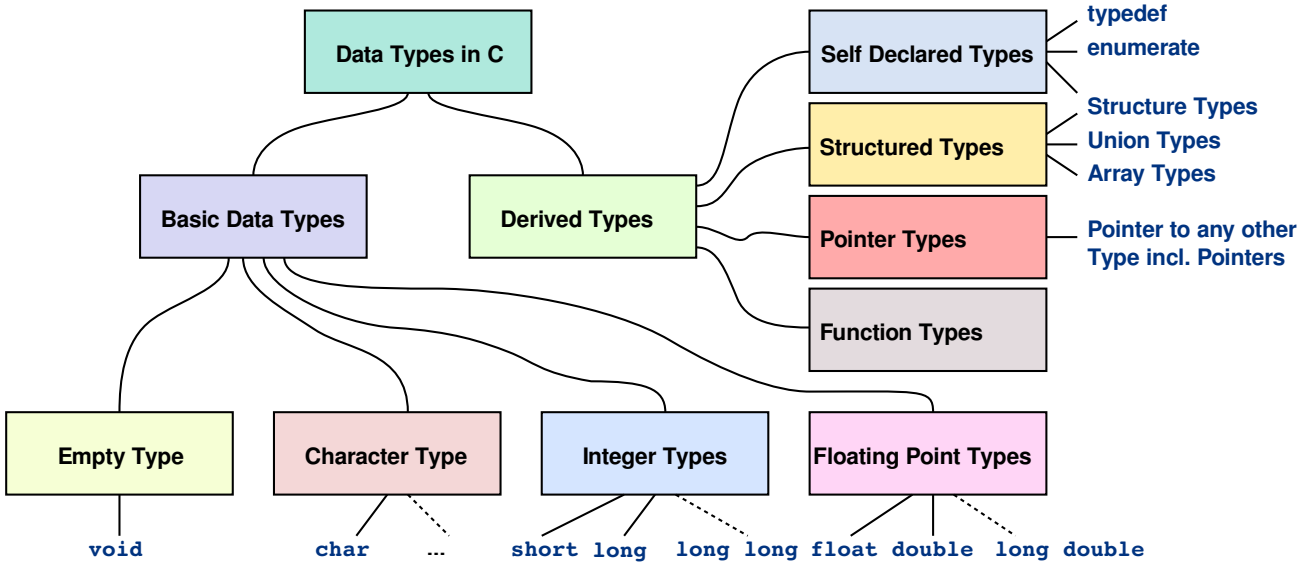
Speicher-Sektionen eines übersetzten C-Programms

	Section Name
Program	P
Variables without initial values	B
Variables with initial values	D
Constants	C
Local variables	Stack area (no section name)

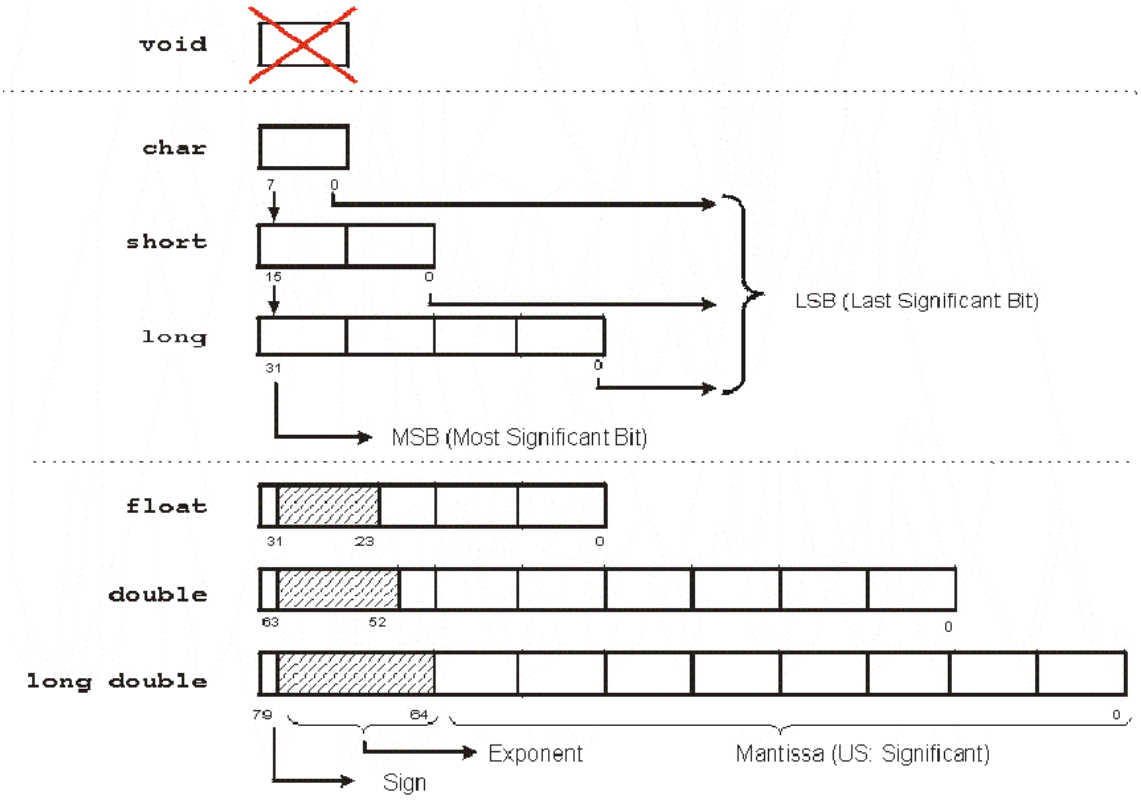
Memory-Map der Sektionen eines übersetzten C-Programms in einem Controller



Datentypen in C



Größe der Basistypen



Wertebereiche der Basistypen (ARM-Default)

Data Type	Bits	Range	Remarks
`unsigned char`	8	0 ... 255	0 ... $2^{8}-1$
`(signed) char`	8	-128 ... 127	-2^{7} ... $2^{7}-1$
`unsigned short`	16	0 ... 65 535	0 ... $2^{16}-1$
`(signed) short`	16	-32 768 ... 32 767	-2^{15} ... $2^{15}-1$
`unsigned long`	32	0 ... 4 294 967 295	0 ... $2^{32}-1$
`(signed) long`	32	-2 147 483 647 ... 2 147 483 647	-2^{31} ... $2^{31}-1$
`float`[1)]	32	$\pm$ 3.4*10^{-38} .. $\pm$3.4*10^{38}	Genauigk. 7 Dez.stellen
`double`[1)]	64	$\pm$ 1.7*10^{-308} .. $\pm$1.7*10^{308}	Genauigk.t 15 Dez.stellen
`long double`[1,2)]	80	$\pm$ 3.4*10^{-4932} .. $\pm$3.4*10^{4932}	Genauigk. 19 Dez.stellen

1) IEEE Standard for Binary Floating-Point Arithmetic for microprocessor systems (ANSI/IEEE Std 754-1985), bzw. IEC-60559:1989 aktuelle Rev. 2008- International version kurz ⇒ IEEE 754.

2) `long double` wird im ARM-Systemen regelmäßig auf `double` abgebildet

Im Headerfile `limits.h` sind die Wertebereiche vordefiniert

Macro	Meaning	Value	Hex value
CHAR_MAX	Maximum value of char	255	0xFF
CHAR_MIN	Minimum value of char	0	0x00
SCHAR_MAX	Maximum value of signed char	127	0x7F
SCHAR_MIN	Minimum value of signed char	-128	0x80
UCHAR_MAX	Maximum val. unsigned char	255	0xFF
SHRT_MAX	Maximum value of short	32 767	0x7FFF
SHRT_MIN	Minimum value of short	-32 768	0x8000
USHRT_MAX	Maximum val. unsigned short	65 535	0xFFFF
INT_MAX	Maximum value of int	2 147 483 647	0x7FFFFFFF
INT_MIN	Minimum value of int	-2 147 483 648	0x80000000
LONG_MAX	Maximum value of long	2 147 483 647	0x7FFFFFFF
LONG_MIN	Minimum value of long	-2 147 483 648	0x80000000
ULONG_MAX	Maximum val. unsigned long	4 294 967 295	0xFFFFFFFF
LLONG_MAX	Maximum value of long long	≈ . 9.2E+18	0x7FFFFFFF FFFFFFFF
LLONG_MIN	Minimum value of long long	≈ .-9.2E+18	0x80000000 00000000
ULLONG_MAX	Max. val. unsigned long long	≈ .1.8E+19	0xFFFFFFFF FFFFFFFF

Variablenvereinbarung: Declaration, Definition, Initialization

Declaration of X = describes, what X is (assign a data type)

→ but do not reserve any memory for X

→ but do not set any value for X

Definition of X = declares X (assign a data type) and creates X (assigns a memory location)

→ but do not set any value for X

Initialization of X = declares X and
creates X (assigns a memory location) and
initialize first contents of X (set values in the memory location)

Variablenvereinbarung - Beispiele

Declaration
```
typedef UC unsigned char;
extern int i;
void sum(long x, long y);
```

Definition
```
int j;
double x[5];
```

Initialization
```
int i = 32;
float z = 2.781;
char c = 'x';
char string[] = "abcdef";
```


Funktionen: Declaration und Definition

Definition = 'Programming' the function

```
int max_of_both (int x, int y)
{
 if (x > y)
      return x;
 else
     return y;
}
```

→ unique in project

Declaration = Function Prototype

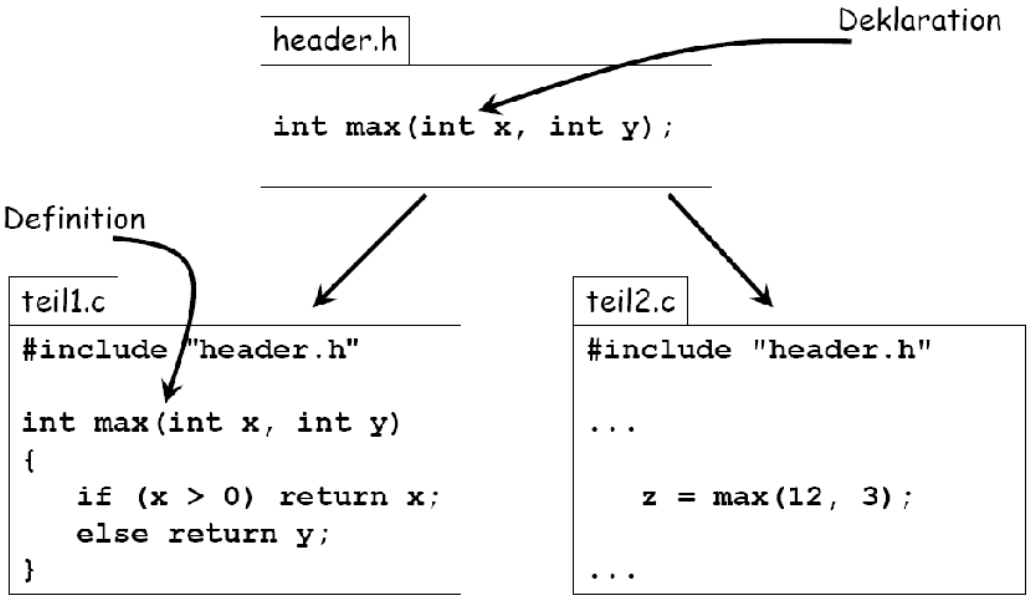
```
int max_of_both (int x, int y);
```

→ inserted in other files of the project without definition

→ used if called before definition

→ used in header files for pre-compiled library functions

Funktionsprototypen in Headerfiles



1 Programmentwicklung in C

2 Datentypen

3 Pointer

4 Programmentwicklung und Toolchain

5 Operatoren, Präzedenzen, Bindungen

6 Bitoperationen

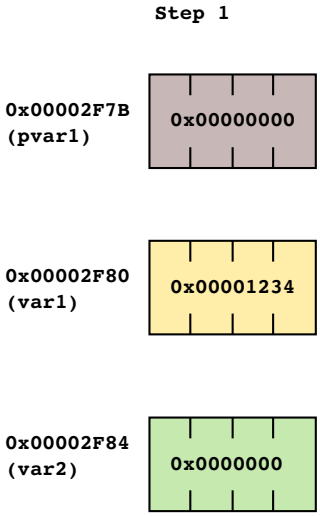
7 Speicherklassen

Pointer als abgeleiteter Datentyp

- Pointer=Variablen für „typisierte“ Adressen
  - Pointer sind selbst Datentypen
  - Pointer sind immer an andere Datentypen gebunden, wie
  - Basistypen oder zusammengesetzer Typen oder Pointer oder Funktionen
- Wert des Pointers
  - Enthält eine Adresse
  - Wenn kein gültiger Wert, üblicherweise als NULL-Pointer gesetzt
  - Wenn ein gültiger Wert, dann die Adresse einer Variable oder eines Elementes oder einer Komponente einer Variable
- Der Pointer durch Pointeroperationen benutzt (gelesen, geschrieben, manipuliert)
  - `&` Adressoperator
  - `*` De-Referenzierung
  - `+, -, ++, --` Pointerarithmetik

Beispiel Pointer 1

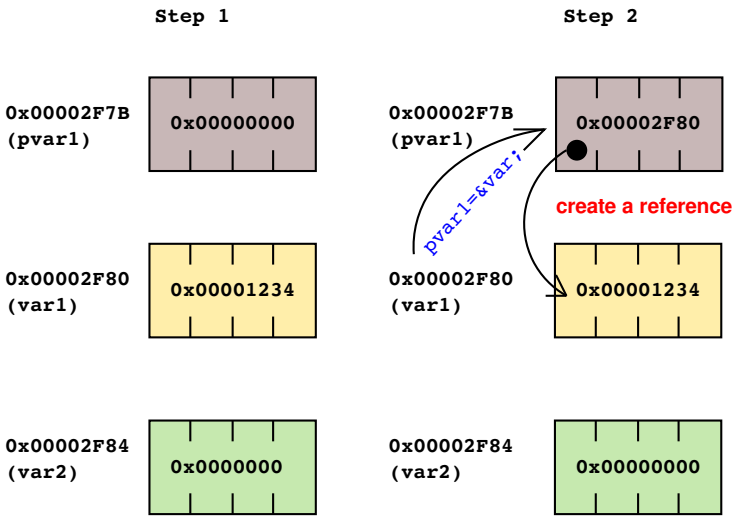
```
// Step 1
long * pvar1;            // variable var2 w. type pointer to long
long  var1 = 0x00001234; // variable var1 w. type long
long  var2 = 0x00000000; // variable var2 w. type long
```



Beispiel Pointer 2

```
// Step 1
long * pvar1;            // variable var2 w. type pointer to long
long  var1 = 0x00001234; // variable var1 w. type long
long  var2 = 0x00000000; // variable var2 w. type long

//Step 2
pvar1 =  &var1 ;     //  copy the address of var1 to pointer pvar1
                     //  set the pointer reference to var1
```

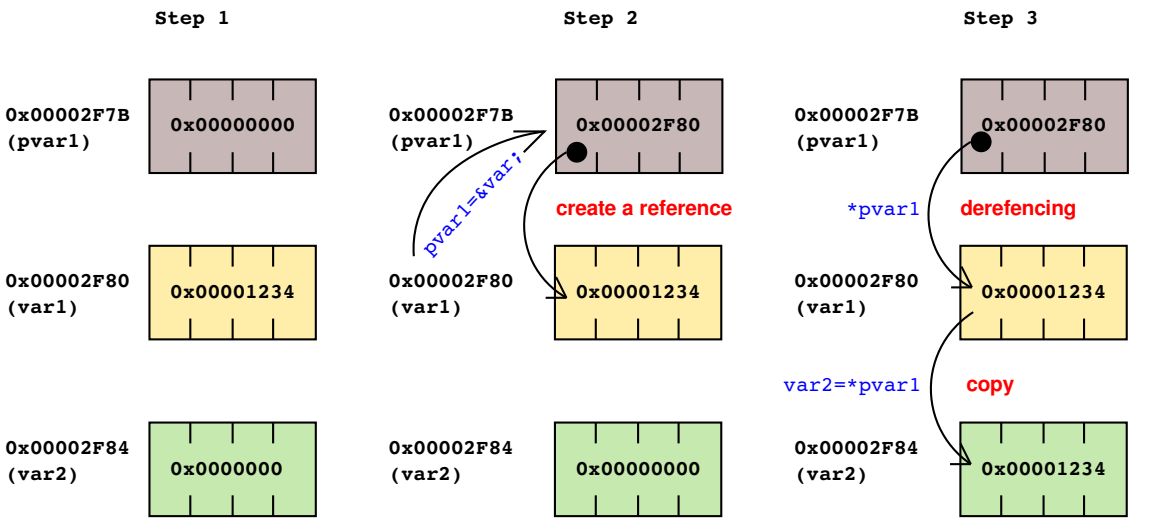


Beispiel Pointer 3

```
// Step 1
long * pvar1;            // variable var2 w. type pointer to long
long  var1 = 0x00001234; // variable var1 w. type long
long  var2 = 0x00000000; // variable var2 w. type long

//Step 2
pvar1 =  &var1 ;     //  copy the address of var1 to pointer pvar1
                     //  set the pointer reference to var1

// Step 3
var2=  * pvar1       // copy a deferenced object of the pointer pvar to var2;
```



Pointer für die zusätzliche Organisation von Call by Reference in C

```
...

/* function with two results (?) */
void take_two(char * X, char * Y)
{
*X = 'a';
*Y = 'b';
}

void main (void)
{
 char A, char B;

/* parameters are the addresses of the variables */
 take_two(&A, &B);

/* parameters A,B are values of the variables */
 printf("A = %c und B = %c", A, B);

}
```


Quiz - Size of ....

Wieviel Speicher 'verbraucht' die folgende Definition auf einem ARM-Controller ?

```
...
 char * bigarray [100];
...
```

1. Jede Addresse hat 32 Bit = 4 Byte
2. Das Array hat 100 Elemente
100 x 4 Byte = 400 Byte

1 Programmentwicklung in C

2 Datentypen

3 Pointer

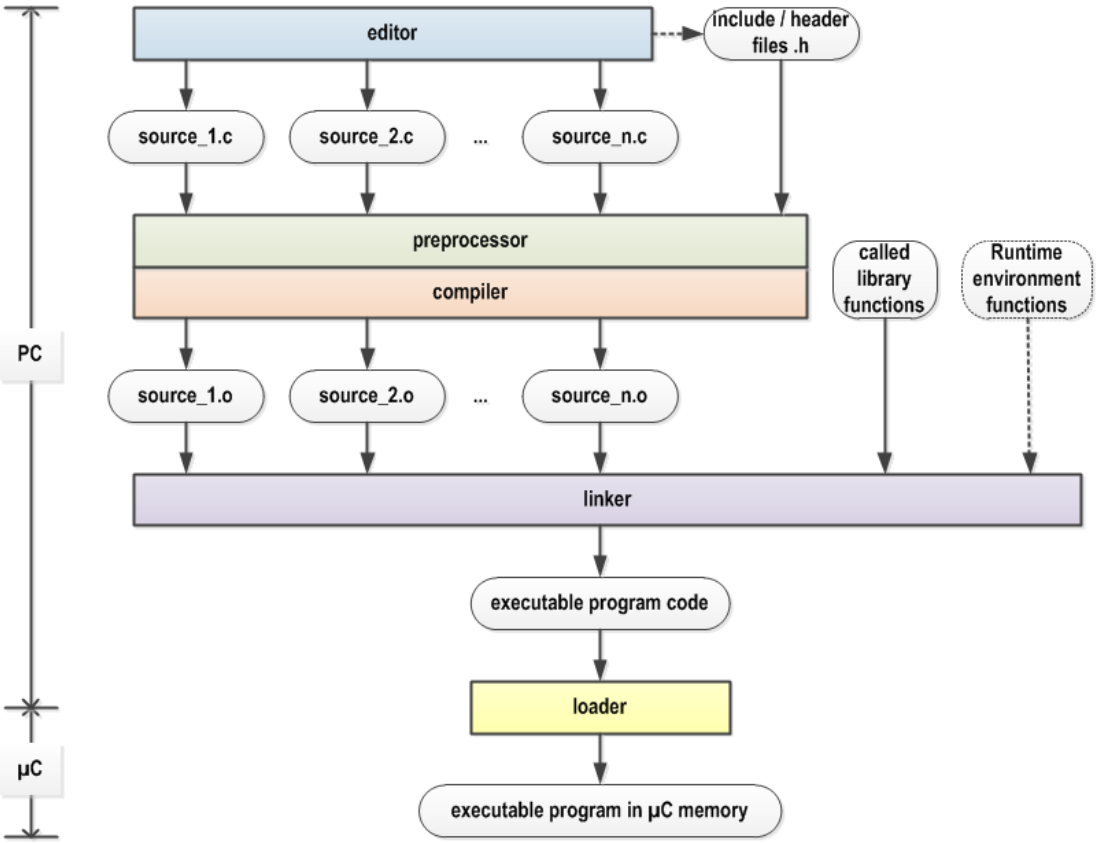
4 Programmentwicklung und Toolchain

5 Operatoren, Präzedenzen, Bindungen

6 Bitoperationen

7 Speicherklassen

Übersetzungsschritte



Quelle: Lutz Leutelt, Vorlesungsunterlagen

Macros im C-Sourcecode

- Der Preprocessor bearbeitet den Sourcecode vor der Übersetzung durch den Compiler
- Die Steuerung Preprocessors erfolgt durch Preprocessor-Direktiven
- Eine Preprocessor-Direktiven beginnt mit #
- Der Preprocessor arbeitet zeilen-orientiert, eine Direktive endet mit der Zeile
- Der Preprocessor erzeugt aus C-Quellcode wieder C-Quellcode (wenn kein Fehler auftritt)
- Mit der `#define` Direktive wird ein Macro erzeugt, z.B.:

  ```
  #define MAXVAL 0x0fF13
  ```

- Das Macro wird vom Preprocessors expandiert, d.h. der Macro-Bezeichner (Name) wird durch die nachfolgende Zeichenkette ersetzt
- Es ist üblich, Macros in großen Buchstaben zu benennen, im Gegensatz zu regelmäßig klein geschriebenen Variablen.

Einfaches Macro - Beispiel

Die Makro-Expansion kann feste, wiederholte benutzte Konstante-Werte für das ganze Programm eintragen

```
...

// Create a Macro
// 1. Directive #define
// 2. MAXSIZE= Macro-Identifier (Macro-Name)
// 3. 0x07F  = Macro-Expansion (Macro-Substitution)

#define MAXSIZE  0x07F

// Use the Macro

char c_array[MAXSIZE+1];
...
for (i = 0;  i<(MAXSIZE-1); i++)
      c_array='A';
...
c_array[MAXSIZE]='\0';
```


Macro - Beispiel 2

Die Makro-Expansion kann ganze C-Anweisungen erzeugen, hier wird eine eigene 'FOREVER-Schleife' erzeugt

```
// Create the Macro
// 1. Directive #define
// 2. FOREVER = Macro-Identifier (Macro-Name)
// 3. for(;;) = Macro-Expansion (Macro-Substitution)
// for (nothing to do; condition = TRUE; nothing);

#define FOREVER for(;;)

// Use the Macro in 3 Examples
// Example 1: a loop without any activity
FOREVER ;

// Example 2:  a loop running a single statement
// FOREVER <single statement>;
FOREVER i++;

// Example 3:   a loop running a block of statements
// FOREVER {  <block of statements>; };
FOREVER
 {
  a++;
  printf("%d\n", a);
 };
```


Macros mit Parametern - Beispiel 3a

Die Macro-Expansion kann mit Parametern in runden Klammern erfolgen

```
/* simple - but not safe - macro directive of a square function */
/* the one parameter X will be multiplied by itself */

#define SQUARE_WITH_A_PROBLEM(X) (X * X)
...
/* now we are using this macro with different parameters*/
result1 = SQUARE_WITH_A_PROBLEM (a);
result2 = SQUARE_WITH_A_PROBLEM (a+1);
...

The source will be expanded after the preprocessor run :
...
result1 = ( a * a );
result2 = ( a + 1 * a + 1);
```

Achtung hier wird in Problem gezeigt !

Macros mit Parametern - Beispiel 3a

Die Makro-Expansion kann mit Parametern in runden Klammern erfolgen

```
/* 'Save macro directive' with brackets for explicit operation
    priority rules */
#define SQUARE(X) ((X)*(X))
...
/* use of this macro */
result1 = SQUARE(a);
result2 = SQUARE(a+1);
...

The Preprocessor expansion takes the additional brackets along.
Now both result1 and result2 are correct calculated.
...
result1 = ( (a)* (a) );
result2 = ( (a + 1) * (a + 1));
```

Mit expliziter, zusätzlicher Klammersetzung wird das Problem gelöst.
Der Präprozessor kennt keine Vorrangregeln/Bindungsregeln der C-Operationen !

$\Longrightarrow$ Klammern in C sind i.d. Regel nützlich, nicht daran sparen!

Preprocessor-Direktiven und Header-Files

- Ein Headerfile wird mit der Direktive `#include` in das Quellfile hereinkopiert

  `#include <stdio.h>` mit Suchpfad im vordefinierten Include-Verzeichnis

  `#include "projectheader.h"` mit Suchpfad im aktuellen Verzeichnis oder als absolute Suchpfadangabe

- Ein Headerfile enthält u.a.

  - Macros, z.B. für Register

    ```
    #define GPIO_PortA_Data_R \
                (*((volatile UC_REG *)(0xFEB0 34F5)))
    ```

  - Function prototypes (z.B. für Libraries)

    ```
    int puts (const char *)
    ```

  - type declarations

    ```
    typedef UC unsigned char;
    ```


Preprocessor-Direktiven vs. C-Statement

- Preprocessor:
  - Alle Direktiven beginnen mit '#'
  - Sie enden am Zeilenende:
    - d.h. mit dem Zeichen 'linefeed' (LF: ASCII-Code 0x0A, Escapesequenz \n)
    - oder den beiden Zeichen 'carriage return' und 'linefeed' (CR LF: ASCII-Code 0x0D 0x0A, Escapesequenz \r\n)
- C-Compiler:
  - Das Zeichen 'linefeed' ist whitespace und wird vollständig ignoriert, Ausnahme in Bezeichnern, das Zeichenpaar 'carriage return' und 'linefeed' ist ebenso whitespace
  - Alle C-Statements enden mit ';',',' oder'}'
  - Alle C-Expessions liefern Werte, sie sind Teil von C-Statements
- Preprocessor und Compiler:
  - Das Zeilenende wird ignoriert wenn die Zeile mit '\' beendet wird
  - Beispiel: `#define PUTLINE puts(\**********\ **********\);`

Bedingte Preprocessor-Direktiven

- Ein Headerfile wird häufig mit der Direktive-Sequenz
  `#ifndef _<name-header_file>`
  `#define _<name-header_file>` begonnen
- Dann wird das Headerfile mit `#endif` beendet
- Beispiel:

```
#ifndef _LIMITS
#define _LIMITS
...
// Here is the headerfile content
...
#endif
```

- Ziel:
  - verhindern von mehrfach gleichen Macros und Deklarationen durch mehrfach eingeschlossene Headerfiles
  - Unabhängigkeit bei der Compilierung der Sourcefiles

1 Programmentwicklung in C

2 Datentypen

3 Pointer

4 Programmentwicklung und Toolchain

5 Operatoren, Präzedenzen, Bindungen

6 Bitoperationen

7 Speicherklassen

Single Term Operators (Einseitige Operatoren)

	Operator	Function	Notes
Single-term operators	-	Negative sign	
	+	Positive sign	
	~	Bit inversion	
	--	Decrement	
	+ +	Increment	
	&	Variable address	&a is the address at which the value of variable a is stored
	*	Content referenced by pointer variable *p is the value referenced by p	

Two Terms Operators (Zweiseitige Operatoren)

	Operator	Function	Notes
Two-term operators	-	Subtraction	
	+	Addition	
	*	Multiplication	
	/	Division	
	%	Remainder of integer division	
	&	Bitwise AND	
	|	Bitwise OR	
	^	Bitwise exclusive OR	
	&&	Logical AND (true or false)	
	||	Logical OR (true or false)	
	>>	Right-shift	(variable name) >> (number of bits to shift)
	<<	Left-shift	(variable name) << (number of bits to shift)

Comparison Operators (Vergleichsoperatoren)

	Operator	Function	Notes
Comparison operators	= =	Equality	
	!=	Inequality	
	>	Greater than	
	<	Lesser than	
	>=	Greater than or equal to	
	<=	Lesser than or equal to	

Assignment Operators (Zuweisungsoperatoren)

	Operator	Function	Notes
Assignment operators	=	Assignment	
	+=	Assignment after addition	
	-=	Assignment after subtraction	
	/=	Assignment after division	
	%=	Assignment after remainder	
	<<=	Assignment after left-shift operation	
	>>=	Assignment after right-shift operation	
	&=	Assignment after logical AND	
	|=	Assignment after logical OR	
	^=	Assignment after logical exclusive OR	

Precedence and Connectivity Rules: Vorrang und Bindungsregeln der Operatoren

	Operator		Connectivity rule*4
First	( ) [ ] -> . ++ --	*1	Left-hand
	! ~ ++ -- + - * & sizeof	*2	Right-hand
	(type)		Right-hand
	* / %	*3	Left-hand
	+ -	*3	Left-hand
	<< >>		Left-hand
	< <= > >=		Left-hand
	= = !=		Left-hand
	&	*3	Left-hand
	^		Left-hand
	|		Left-hand
	&&		Left-hand
	||		Left-hand
	?:		Right-hand
	= += -= *= /= %= &= ^= |= <<= >>=		Right-hand
Last	,		Left-hand

Precedence

Notes

1 Programmentwicklung in C

2 Datentypen

3 Pointer

4 Programmentwicklung und Toolchain

5 Operatoren, Präzedenzen, Bindungen

6 Bitoperationen

7 Speicherklassen

Bitweise Operatoren

`&, &=` Bitweiser AND-Operator

`|, |=` Bitweiser OR-Operator

`^, ^=` Bitweiser XOR-Operator

`~` Bitweiser Komplement-Operator: Invertieren 0 → 1, 1→ 0

`>>, >>=` Bitweise Rechtschieben, von links Nullen auffüllen

`<<, <<=` Bitweise Linksschieben, von rechts Nullen auffüllen

Gegensatz: `&&` Logischer Vergleichs-Operator AND

Für alle Logischen Vergleiche gilt: Resultat nur 0 =FALSE oder 1 = TRUE, Operanden = 0 gelten als FALSE, alle anderen Operandenwerte gelten als TRUE

`||` Logischer Vergleichs-Operator OR

`!, !=, <, >, <=, >=, ==` weitere logische Vergleichs-Operatoren

Beispiel Bitweises AND

```
char x = 0x37;
char y = 0x07;
char z = 0x00;

z = x & y;
```

		Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0		Hex
	Operand x	0	0	1	1	0	1	1	1		0x37
Operation &	Operand y	0	0	0	0	0	1	1	1	&	0x07
	Resultat z	0	0	0	0	0	1	1	1		0x07

Beispiel Bitweises OR

```
char x = 0xB2;
char y = 0x07;
char z = 0x00;

z = x | y;
```

		Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0		Hex
	Operand x	1	0	1	1	0	0	1	0		0xB2
Operation |	Operand y	0	0	0	0	0	1	1	1	|	0x07
	Resultat z	1	0	1	1	0	1	1	1		0xB7

Beispiel Bitweises XOR

```
char x = 0x92;
char y = 0x3C;
char z = 0x00;

z = x ^ y;
```

		Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0		Hex
	Operand x	1	0	0	1	0	0	1	0		0x92
Operation ^	Operand y	0	0	1	1	1	1	0	0	^	0x3C
	Resultat z	1	0	1	0	1	1	1	0		0xAE

Beispiel Bitweises Komplement

```
char x = 0xCD;

char z = 0x00;

z = ~ x;
```

		Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0		Hex
	Operand x	1	1	0	0	1	1	0	1		0xCD
Operation ~	entf.	-	-	-	-	-	-	-	-	~	entf.
	Resultat z	0	0	1	1	0	0	1	0		0x32

Training Bit-Operationen 1

```
char x = 0xAE;
char y = 0xB9;
char z = 0x00;
z = x & y ;
```

		Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Hex
	x									
	y									
	z									

```
char x = 213;
char y = 55;
char z = 00;
z = x | y ;
```

Bitte z auch dezimal angeben !

z: .....

		Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Hex
	x									
	y									
	z									

```
char x = 0x1C;
char y = 0x29;
char z = 0x00;
z = x ^ y ;
```

		Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Hex
	x									
	y									
	z									

```
char x = 64;
char y = 0x0F;
char z = 00;
z = x | y ;
```

Bitte z auch dezimal angeben !

z: ....

		Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Hex
	x									
	y									
	z									

```
char x = 0xF1;
char y = 0x02;
char z = 0x00;
z = x << y ;
```

		Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Hex
	x									
	y									
	z									

```
char x = 0x33;
char y = 0x07;
char z = 00;
z =(~x) & y;
```

		Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Hex
	x									
	y									
	z									

Training Bit-Operationen 2

```
char x = 0xAE;
char y = 0xB9;
char z = 0x00;
z = x & y ;
```

		Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Hex
	x	1	0	1	0	1	1	1	0	0xAE
&	y	1	0	1	1	1	0	0	1	& 0xB9
	z	1	0	1	0	1	0	0	0	0xA8

```
char x = 213;
char y = 55;
char z = 00;
z = x | y ;
```

Bitte z auch dezimal angeben !

z: 247

		Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Hex
	x	1	1	0	1	0	1	0	1	0xD5
|	y	0	0	1	1	0	1	1	1	| 0x37
	z	1	1	1	1	0	1	1	1	0xF7

```
char x = 0x1C;
char y = 0x29;
char z = 0x00;
z = x ^ y ;
```

		Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Hex
	x	0	0	0	1	1	1	0	0	0x1C
^	y	0	0	1	0	1	0	0	1	^ 0x29
	z	0	0	1	1	0	1	0	1	0x35

```
char x = 64;
char y = 0x0F;
char z = 00;
z = x | y ;
```

Bitte z auch dezimal angeben !

z: 79

		Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Hex
	x	0	1	0	0	0	0	0	0	0x40
|	y	0	0	0	0	1	1	1	1	| 0x0F
	z	0	0	1	0	1	1	1	1	0x4F

```
char x = 0xF1;
char y = 0x02;
char z = 0x00;
z = x << y ;
```

		Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Hex
	x	1	1	1	1	0	0	0	1	0xF1
<<	y	0	0	0	0	0	0	1	0	<< 0x02
	z	1	1	0	0	0	1	0	0	0xC4

```
char x = 0x33;
char y = 0x07;
char z = 00;
z =(~x) & y;
```

		Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Hex
~	x	0	0	1	1	0	0	1	1	0x33
	(~x)	1	1	0	0	1	1	0	0	0xCC
&	y	0	0	0	0	0	1	1	1	& 0x07
	z	0	0	0	0	0	1	0	0	0x04

1 Programmentwicklung in C

2 Datentypen

3 Pointer

4 Programmentwicklung und Toolchain

5 Operatoren, Präzedenzen, Bindungen

6 Bitoperationen

7 Speicherklassen

Speicherklassen von Variablen

Storage Class	Place of Declaration	
	Within a Function	**Outside Functions**
None	Local variable of the auto storage class (variable value not preserved)	Global variables; variable names are external names, and only one instance of each variable name is allowed among multiple files. If declared as an extern storage class, the variable can be referenced from a program created in a different file.
auto	As above	Cannot be declared (syntax error)
static	Local variable of the static storage class (variable value is preserved)	Global variable of the static storage class. Cannot be referenced from a program created in a different file.
extern	Global variable reference. Declaration indicating that the declaration to actually secure storage space is performed outside the function.	Global variable reference declared by a program created in a different file. Referencing of global variables in the static storage class is not possible.
register	Local variable having the same properties as the auto storage class. CPU registers allocated as the storage location; program execution efficiency and memory efficiency are improved as a result. However, the number of variables which can be so allocated differs depending on the CPU.	Cannot be declared (syntax error)

Speicherklassen und Lebensdauer

static Heapvariable (Statischer Teil des Heaps)

- die gesamte Laufzeit des Programms werden die Werte erhalten (lebendig, aber nicht unbedingt sichtbar)
- global static variables, local static variables, extern variables

auto Stackvariable

- nur im Block der Vereinbarung werden die Werte erhalten (gültig/lebendig im Block zwischen {})
- lokale Variablen
- der Speicherort auf dem Stack wird nach der Freigabe (Gültigkeits-/Lebensende) erneut genutzt

dynamic Heapvariable (dynamischer Teil des Heaps)

- Anlegen mit `malloc()`
- Freigeben mit `free()`
- Seltener in Controllersystemen

Speicherklassen von Funktionen

Storage Class	Properties of Function Name	Properties of Function
None	External name	Can be called from a function in a different source file
static	Internal name	Cannot be called from a function in a different source file

Using C program functions and variables from an assembler program, an underscore (_) is the first character of names.

Quiz

Bestimmen Sie die Werte der Variablen nach der Ausführung der folgenden Programmzeilen:

```
void main (void){
   char b,c,d;
   char a[] = {0xAA,0xBB,0xCD,0xFE};
   char *p = &a[1];

   b = *(p--) & 0x78;
   c = *(p+2) & ~((1<<3)|(1<<7));
   d = *((p++)+1) | 23;
   ...
```

Antwort: (Werte bitte aufschreiben)

a =

b =

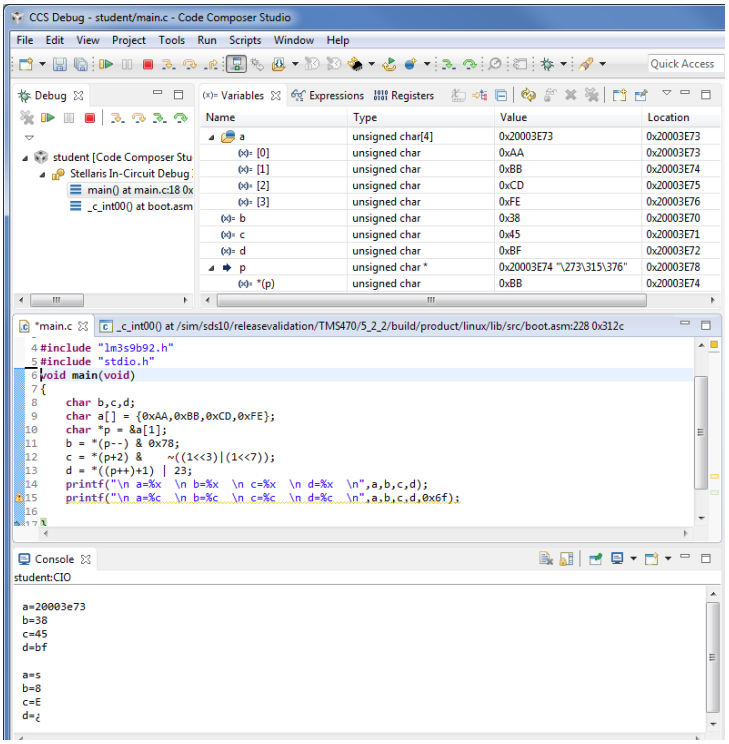
c =

d =

Quiz ... Hinweise

- Bitte investieren bis etwa 15-30 min, um eine Lösung zu finden !
- Die Lösung kommt auf der nächsten Seite.
- Wir haben das Code Composer Studio benutzt, um Fehler auszuschließen. CCS können Sie gern auch nutzen, im Debugger können Sie Zwischenwerte beobachten.
- Die Lösung steht auf der nächsten Seite beabsichtigt auf dem Kopf, damit Sie nicht versehentlich lesen und es besser nochmal selbst versuchen.

Quiz Lösung mit dem Debugger CCS



Zunächst gibt die Funktion printf als hexadezimale Zahlen (Typ int) aus:

Der Ausgabe a zeigt hier die vom System vergebene Adresse*) 0x20003e73 des Elements a[0] mit Wert 0xAA.

Die anderen Werte sind b=0x38, c=0x45, d=0xBF. Führende Nullen werden nicht ausgegeben.

Der Zeiger p wird decrementiert (p–) dann incrementiert (p++), er zeigt also wieder auf die initialisierte Adresse des Elements a[1].

Beim zweiten Aufruf der Funktion printf werden alle Werte als Typ char interpretiert, also nur das niedrigwertigen Bytes als ASCII-Zeichen angezeigt.

*) Dieser Wert kann abweichen.