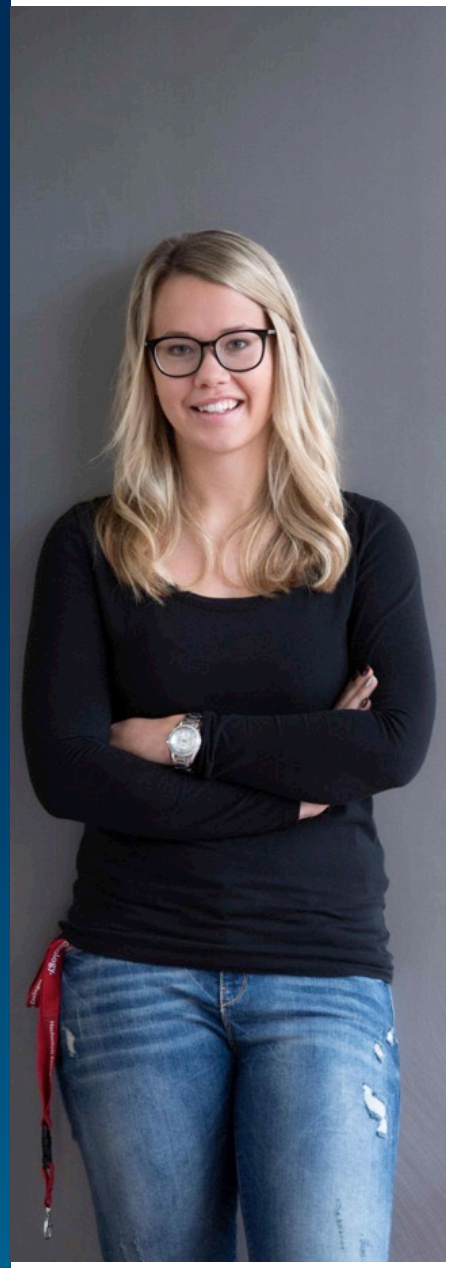
HOCHSCHULE
ESSLINGEN



LECTURE COMPUTER ARCHITECTURE

MODULAR PROGRAMMING IN C AND ASSEMBLER

RAINER KELLER

HOCHSCHULE
ESSLINGEN

CONTENT

GOALS FOR TODAY

- Learn Inline Assembly usage on HCS12
- Learn possibilities of C++

INTRODUCTION

HOCHSCHULE ESSLINGEN

Most professional embedded system programs are written in C, sometimes C++, because coding in C is faster and programs are more maintainable than assembler code. Nevertheless, a small part of these programs often will be programmed in other languages, especially in assembler, because

- **Algorithms in assembler** can use special features of the CPU's instruction set, which a C- compiler may not use. Thus assembler code may run faster and/or require less memory. Special commands like BIT SET/CLEAR may be much faster than AND/OR operations, or instructions, which add and multiply in a single operation like HCS12's EMACS instruction, may not be used, because they are not available in all types of CPUs.

- **Accessing special registers or functions** of the microcontroller, for which no equivalent C-operation is available, i.e. there are no C-statements to access status register CCR directly.

There are two basic solutions for the interaction of C and assembler code:

- **Inline-Assembler**: Embed assembler instructions into C code.
- **Assembler-Module**: Assembler subroutines, which are called from C code.

INLINE ASSEMBLER (HCS12 C COMPILER)

Single assembler instructions can be integrated in C-code: `asm instr;`

For (small) blocks of asm instructions within a C function use `asm {…}`
(although the capabilities are very limited, in comparison to GNU `gcc`)

Because the C-compiler uses the same CPU registers as the assembler, some rules ensure a integration between C and assembler code:

- **Variables and constants** shall be declared **in C**. Assembler code can access these variables by their name. Global C-variables can be used in all assembler instructions which work with direct addressing. Local C-variables require assembler instructions which allow register-indirect addressing via **SP**. The linker will handle the translation between C variable name and the CPU's address mode.
- At the end of each block of assembler instructions, the Stack Pointer must have the same value as at the start of the block.
- The address of a C-variable is available as `@variableName` in the assembler block.
- Elements of C-structures and unions can be used, e.g. `LDD myStruct.myElement`
- Elements of C-arrays can be used, e.g.: `int a[20]; asm LDD a:24` accesses element `a[12]`.

  Note: The C-array index must be converted into a byte offset in the assembler instruction, i.e. `variableName : C-Index · sizeof(variableType)`

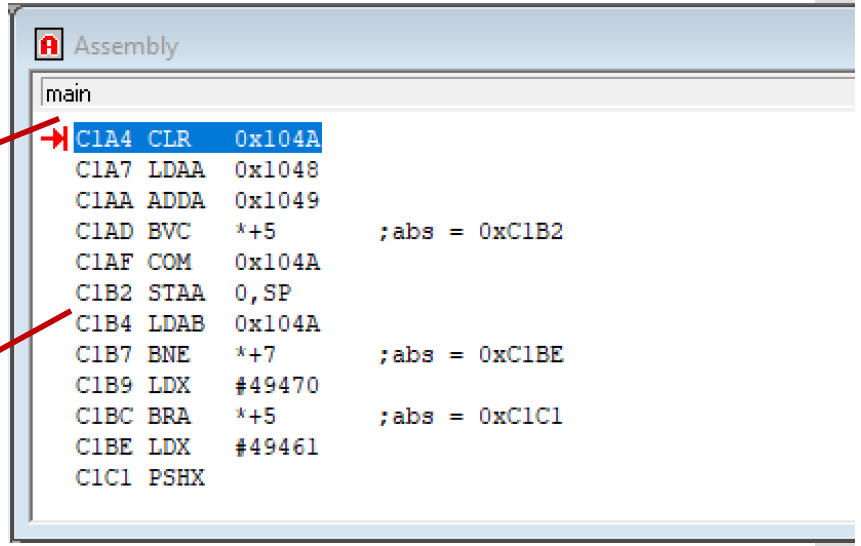
Some compilers allow to freely modify all CPU registers in assembler blocks (e.g. HCS12 CodeWarrior). Check the compiler manual!

INLINE ASSEMBLER

Example Program `InlineAssembler.mcp`

- Add two 8-bit variables to indicate overflow:

```
char s1, s2;          // Global variables
char flag;
void main(void) {
  char r;             // Local variable
  for(;;) { ...
    s1 = . . .        // Input summands
    s2 = . . .        // from terminal
    …
    flag = 0;
    asm {
          LDAA s1  // Compute r = s1 + s2, if overflow occurs
          ADDA s2  // set flag to 1
          BVC noov // Check for overflow, branch if it did not
          COM flag // … and set flag if it did…
    noov: STAA r   // Store result (r is a local var on the Stack)
    }
    sprintf(temp, "\nc =%d + %d = %d %s\n", s1, s2, r, flag ?
                  "Overflow" : "No overflow");
    PutString(temp);   // Output result to terminal
    asm SWI;           // Exit program (stop simulator)
}
```



C WITH ASSEMBLER SUBROUTINES 1/4

HOCHSCHULE ESSLINGEN

Assembler-Subroutines for C-Programs

(see [3.13 Chapter HC12 Backend – Call Protocol and Calling Conventions and Chapter Stack Frames])

To pass parameters to a subroutine and return results, the caller and the callee must use a common strategy and data types for parameter passing:

```
returnType function(C_Type1 param1, . . ., C_TypeN paramN)
```

Several methods exist:

- **Use global variables**: Advantage: No size limit (other than RAM size). Disadvantage: Recursive subroutine calls (Reentrancy) impossible.
- **Use registers**: Advantage: Fast. Disadvantage: Limited size/number of registers. This method is mainly used in stand-alone assembler programs.
- **Use the stack**: Advantage: Flexible. Disadvantage: Slow, error-prone addressing of parameters.

On some systems Calling Conventions are not standardized, so each programming language and each compiler comes up with its own solution. The CodeWarrior HCS12-C-Compiler uses the following methods:

- Result of a function is returned in a register (D, see table on the next page).
- If a function has a fixed number N of parameters: Parameters 1 to N-1 passed on the stack „from left to right“ (PASCAL calling convention). However, the last parameter paramN is passed in a register (D, see table).

C WITH ASSEMBLER SUBROUTINES 2/4

HOCHSCHULE ESSLINGEN

Data type of last parameters Data type of return value	Size in Bytes	CPU-Register
`char`	1	`B`
`int` `(Near) Pointer *any_C_type`	2	`D`
`long`	4	`X` (upper 2 byte), `D` (lower 2 Byte)

Big data structures (arrays, structures, unions) should be passed by reference, rather than by value, i.e. only a pointer to the data structure is passed (see [3.13])

- The calling program has to remove the parameters from stack again, after the subroutine returned. This is done by modifying SP by the number of bytes, which have been passed as parameters on the stack. I.e.

`param1` → Stack ("Push")	Parameter passing via stack
…	"left to right" (decrements SP)
`paramN` → Register	Last parameter in register
Call the subroutine	
Store return value	
`SP`+number of parameter bytes → `SP`	"Clear the stack" (increment SP)

- **Note**: If a function has a variable number of parameters, e.g. `printf()`, `sprintf()`, …, all parameters are passed on stack „from right to left", i.e. `paramN` is pushed first, parameter `param1` is pushed last to the stack before calling the subroutine (C calling convention).

C WITH ASSEMBLER SUBROUTINES 3/4

HOCHSCHULE ESSLINGEN

The **interface of a function** must be defined exactly for compiler, asm & linker.
Interface in calling –program:

- Function prototype:
  ```
  C_type function (C_type1 param1, …, C_typeN paramN);
  ```
  (note: The C keyword "`extern`" is optional, but usage is not recommended)
- Reference to a global asm variable (possible, but considered bad style):
  ```
  extern C_type assemblerVariable;
  ```

Interface in called assembler-program:

- Export the assembler subroutine to other program modules:
  ```
  XDEF nameAsmRoutine       // at beginning of file
  nameAsmRoutine: …         // Label at begin of code
  ```
- Export a global assembler variable to other program modules:
  ```
  XDEF assemblerVariable   // at beginning of file
  assemblerVariable DS.x … // normal declaration
  ```
- Import of global C-variable:
  ```
  XREF C_variable              // at beginning of file
  ```

C WITH ASSEMBLER SUBROUTINES 4/4

HOCHSCHULE ESSLINGEN

Example Program: CwithASM.mcp

- Add two 8-bit variables and indicate overflow

Main program in C:

```
char asmComp(char s1, char s2);
char s1, s2, r, flag=0;

void main(void) {
  r = asmCompute(s1, s2);
  …
}
```

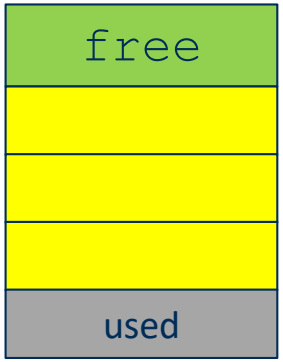
Compiled into:

```
LDAB s1
PSHB                    ;(1)
LDAB s2
JSR asmCompute          ;(2)
LEAS 1,+SP              ;(5)
STAB r
```

Subroutine in assembler:

```
    XDEF asmCompute    ; Export of ASM function
    XREF flag          ; Import of global C variable
.init: SECTION         ; Assembler program code in ROM
asmCompute:
    LDAA 2, SP         ; Get s1 from stack (3)
    ABA                ; s1 + s2 (in reg. B) → reg. A
    BVC noov           ; Check for overflow
    COM flag           ; and set global C variable
noov:                  ; (bad programming style)
    TFR A, B           ; Move result to B
    RTS                ; Return to caller (4)
```

State of Stack:



HOCHSCHULE
ESSLINGEN

Stack & Stack Frames

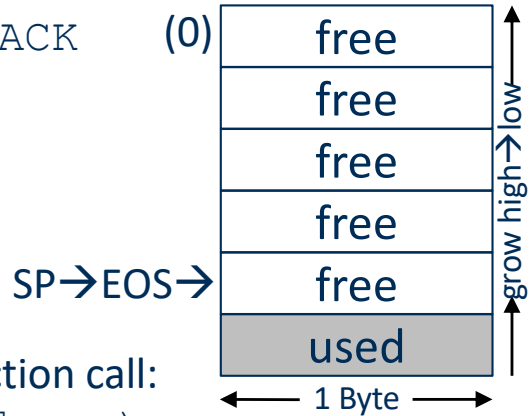
STACK-FRAME 1/2

Local variables are considered good style in higher level languages. Unfortunately, assembler does not allow to directly declare local variables, but we can use the same concept which C- compilers use to assign local variables (so-called auto variable) in a so-called **Stack-Frame**:

Example program `LocalVar.mcp`: Search for the maximum in an array → See code on **next page**

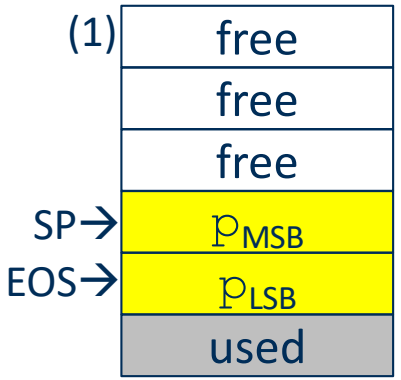
- State of the stack when the program is running:

(0) `LDS __SEG_END_STACK`

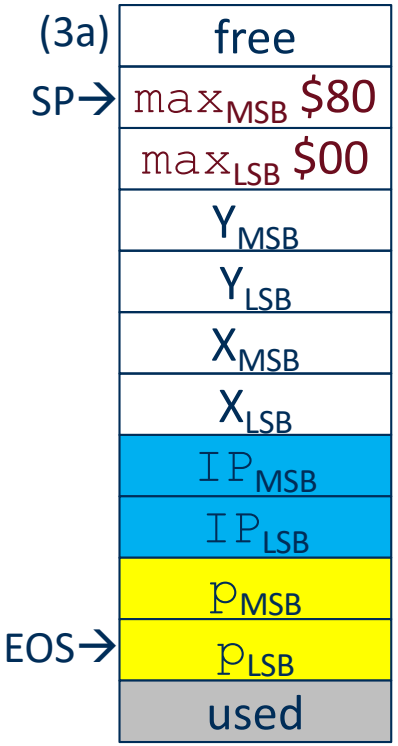


(1) Prior to the `JSR` of function call:

```
int asmMax(int * p, char n);
ret = asmMax(p, n);
```



(3a) In `asmMax` after storing to local var:
Stack Frame is allocated using `LEAS 2, -SP`



STACK-FRAME 2/2

HOCHSCHULE ESSLINGEN

Some C-compilers expect the subroutine to save & restore registers (but never restore the return register!). Not required with CodeWarrior's HCS12 compiler, but good style.

```
int asmMax (int *pArray, char n);      // Prototype of assembler function in C
int val, array[] = { 47, 1600, -4500, 2000, }; // Array
…
val = asmMax(array, 6);                // Calling the function          (1)
```

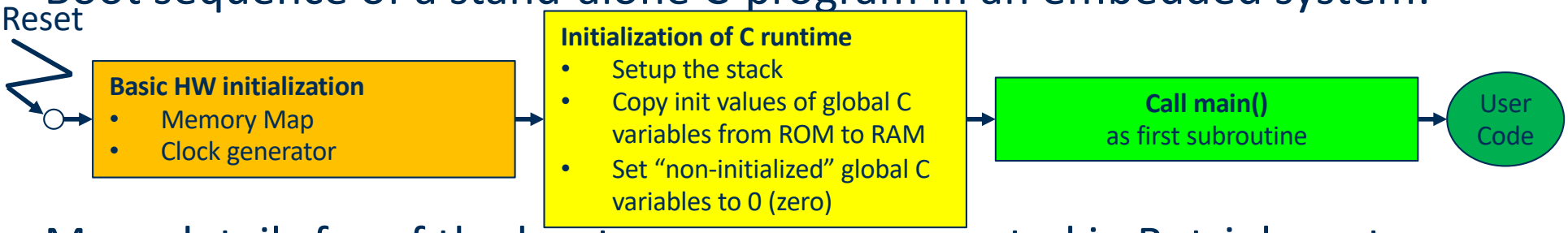
C-Function

```
int asmMax
  (int *pArray, char n) {
  int max= -32768;




  for ( ; n > 0; n--) {
    if (*pArray > max)
      max = *pArray;
    pArray++;
  }


  return max;
}
```

Assembler Function

```
asmMax: PSHX          ; Save registers          (2)
        PSHY
        LEAS 2, -SP   ; Allocate stack space    (3)
        MOVW -32768, 0, SP; ... Init local var  (3a)
        LDX 8, SP     ; pArray (from Stack)→X   (4)
for:    LDY 0, X      ; current array elem *pArray
        CPY 0, SP     ; compare it with max
        BLE next      ; is smaller? then jump
        STY 0, SP     ; if greater: copy → max
next:   LEAX 2, +X    ; pArray++
        DECB          ; n-- for loop counter
endFor: BNE for       ; if n>0, repeat
        LDD 0, SP     ; return max
        LEAS 2, +SP   ; deallocate stack space  (5)
        PULY          ; Restore stack
        PULX
        RTS           ; Return to caller
```

Boot sequence of a stand-alone C-program in an embedded system:



More details for of the boot process are presented in Betriebssysteme.

Workflow when optimizing a program for size and/or speed:

- Define architecture for the program, which is easy to understand & maintain
- Select the most efficient algorithms and data types for (e.g. sort algos)
- Write and debug the program for correctness first (not speed or size!).
- Turn on the strongest optimization levels of the compiler/linker toolchain.
- Profile program, to find parts responsible for program size and speed.
- Optimize the critical parts using better algos, intrinsics, inline assembler, …

Typical compiler/linker optimizations include

- Variable placement, register usage
- Code modifications: Loop unrolling or loop-invariant code removal, dead-code removal, ...
- "Smart linkers" link only those functions of a library, which are actually used.

EMBEDDED C/C++: COMPILER & LINKER OPTIMIZATION

All compiler/linker toolchains can – to a limited extent – optimize programs, but user must specify the optimization level,

Visual C/C++	GNU C/C++
`// Optimize for code size:` `cl /O1 myprogram.cpp`	`// Optimize for code size:` `gcc -Os myprogram.cpp`
`// Optimize for code speed:` `cl /O2 myprogram.cpp`	`// Optimize for code speed:` `gcc -O2 myprogram.cpp`

Optimizations, where compiler may need the software engineer's help:

- Memory in embedded systems is limited, smaller systems are short of RAM. Constants should be declared `const`, … for the linker will put it in ROM.
- Stack should be kept small:
  - Limit the size of local variables per function, no large local arrays
  - Don't use recursive algos, each recursion adds to the stack
  - Limit the nesting of subroutines, each subroutine & ISR adds to the stack
  - Estimate stack usage in the development phase and monitor stack usage during run-time. E.g. periodically check SP, or initialize the stack memory area with a defined byte pattern and periodically check, which part was overwritten, i.e. used.

EMBEDDED C/C++: INLINE FUNCTIONS

HOCHSCHULE ESSLINGEN

Calling a subroutine and returning from it is slow, i.e. is ineffective for short subroutines (call overhead, stack operations, etc.): **Inlining** helps:

Let the compiler decide / or provide hint, whether the function is implemented as subroutine or the function body copied into the calling program. → Faster code, but code may be duplicated multiple times…

HCS12 CodeWarrior C	Visual C/C++

`#pragma inline`
`int strcpy(char *dest, char *src)`
`{…}`
	`__inline`
	`int strcpy (char *dest, char*src)`
	`{…}`

Note: In GCC and C++ better use keyword inline, in GCC also Attributes!

Register Variables:

To speed up programs, variables should be held in registers. Good compilers optimize register usage automatically. The programmer may declare frequently used local variables with keyword `register`, e.g. `register int a;`

The compiler will try to put this variable into a register, if possible. However, many CPUs (HCS12, 80x86, ...) have only a small number of registers, so the speed-up may be small.

EMBEDDED C/C++: VOLATILE VARIABLES

HOCHSCHULE ESSLINGEN

From SW's point of view, HW registers of peripherals are normal global variables. For all variables, the compiler assumes that **only its own** code statements will change the value of a variable. Because reading memory is slow, a compiler may cache a variable in a register. Thus the compiler may not notice, when the value in memory changes. E.g. in the following program (left column), the compiler reads timer counter register `TCNT` only once. To ensure, that always the most current value of a hardware register is used, declare hardware registers with keyword `volatile`:

	`short *pTCNT=0x0044`	`short volatile *pTCNT=0x0044`

`if (*pTCNT > 100)`
`  …`
 	
`if (*pTCNT > 200)`
`…`
	`LDD TCNT`
	`CPD #100`
	`…`
	`CPD #200`
	`…`
		`LDD TCNT`
		`CPD #100`
		`…`
		`LDD TCNT`
		`CPD #200`

Compiler intrinsics

Compilers provide compiler-/HW-specific ways to indirectly access low-level CPU-features. These intrinsics are implemented as C macros with inline ASM.

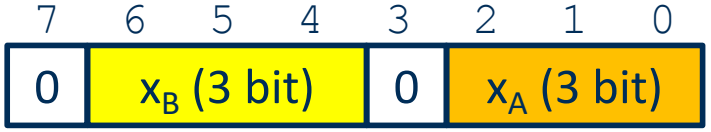
HCS12 CodeWarrior C	Visual C/C++
`EnableInterrupts` → shortcut `asm CLI`	`_enable()`
`DisableInterrupts` → shortcut `asm SEI`	`_disable()`

EMBEDDED C/C++: BITWISE OPERATIONS

HOCHSCHULE ESSLINGEN

Peripherals or low-level communications frequently need to modify single bits or groups of bits, without changing other parts of a hardware register. Microprocessors provide special instructions, e.g. `BSET`, `BCLR`, `BRSET`, ...
As there are no equivalent C statements, C programs must use awkward sequences of AND, OR and shift operations.
Example: In PWMPRCLK $x_B=2_D=010_B$ shall be set, other bits must not be changed.

Register `PWMPRCLK`



Solution in C:
```
#define PWMPRCLK (*(char*) 0x00A3)// Ptr to register
PWMPRCLK = (PWMPRCLK & 0x8F) | 0x20;
```

The & clears bits 6..4. Then the | sets bit 5. A non-optimal C compiler does the same, i.e. uses the AND / OR instructions for byte operands, rather than using the more efficient bit operations BSET / BCLR. The C-programmer must make sure, that constants 0x8F and 0x20 actually address the correct bits. A better solution is to use a C bit-field:

```
typedef struct {
  char xA   : 3;
  char res3: 1;     // reserved bit 3
  char xB   : 3;
  char res7: 1;     // reserved bit 7
} PRCLK;
#define PWMPRCLK (*(PRCLK*) 0x00A3)   // Ptr to register
PWMPRCLK.xB = 2;    // Set xB = 2D
```

EMBEDDED C/C++: COMPATIBILITY & PORTABILITY

HOCHSCHULE ESSLINGEN

Assembler is not portable between CPU families which have a different programming model (instruction set, register set, and address modes). They must be rewritten.

But **low-level C-programs** may not be portable, either:
Each CPU-family has a different set of peripherals. Even if they do the same, e.g. digital I/O, register details are different. This applies even within a CPU family, if it is manufactured by different suppliers, e.g. ARM CPUs of Samsung and Apple have the same CPU instruction set, but have proprietary peripherals.
→ Hardware-dependent parts of programs should be isolated, so that code which needs to be modified, when the CPU family is changed, is easily identified → **HW drivers**.

C has – despite ANSI C89/90 or ISO C99 standards – **gaps** in it's specification, leading to undefined behaviour → Tools like Clangs & GCC's UBSanitizer helps.
E.g. the C standard does not define the signedness of `char`, leaving it to the implementation whether char is actually a `unsigned char` or a `signed char`.
The C standard suggests the size of an `int` to be the same size of the major data registers and ALU of the CPU, e.g. 16 bit for HCS12, 64 bit for x86-64…

→ Programmers should include `stdint.h` and use their definitions like:

```
typedef unsigned char uint8_t;
```

EMBEDDED C/C++: EXTENSIONS

Most compilers provide **proprietary language extensions**; e.g. in CodeWarrior:

C-standard provides compiler-specific extensions using keyword `#pragma`:

```
#pragma TRAP_PROC      // Defines the following subroutine as ISR
void myInterruptServiceRoutine(void) {…}
```

CodeWarrior's proprietary solution requires only a single line of code. However, its non-standard keyword may not be understood by other C compilers:

```
void interrupt 3 myInterruptServiceRoutine(void) {…}
```

Intrinsics (see above) are typically compiler-specific, i.e. not portable.

Memory addresses may be accessed in a C-std conformant way via pointers, e.g.

```
char *p = 0x0001;  // Assign PortB's address to ptr
*p = *p | 0x02;    // Set Bit 1 in Port B
```

Or via CPP macro:

```
#define PORTB  (*((char*) 0x001))
PORTB |= 0x02;
```

Or non-std via:

```
char PORTB @0x0001;// Assign address to variable
PORTB |= 0x02;     // Set Bit 1 in PortB
```

Note: All of the above should be detected by the Configuration of the Software, e.g. using AutoConf/AutoMake, or CMake or other build tools and then selectively compiled using Pre-Processor #`ifdef`s and hiding by CPP macros.

EMBEDDED C/C++: EXECUTION SPEED

HOCHSCHULE ESSLINGEN

Speed in CPU clock cycles, 1 clock = 42ns @ f_{BUSCLK}=24MHz, measured with the HCS12 True Time Simulator, C-code with compiler default optimization, all operands and results are global variables, execution clock cycles include the arithmetic instruction plus fetching the operands and storing the results from/to memory.

Operation		Data type of operands a, b, c			
		int (16 bit)	long (32 bit)	float (IEEE 32)	double (64-bit)
Addition/Subtraction	`c=a+b`	9	21	203	500
Multiplication	`c=a*b`	12	83	285	4174
Division	`a=c/b`	21	143	1049	5324
Type conversion from `int a` to:		b=(int)a 6	b=(long)a 22	b=(float)a 127	b=(double)a 969

Analysis of results:

- Add and subtract is fast, multiply is slower (if a CPU does not have a full operand-width hardware multiplication unit), divide is a disaster (right-shifting??)…
- Data types, which are longer than the size of the ALU (16bit @ HCS12) lead to slow execution.
- Integer arithmetic is much faster than floating-point arithmetic (if the CPU does not have a floating-point ALU)
  - → Use integer arithmetic **and smallest-size data types**, but check overflow and resolution
  - → Use efficient algorithms, i.e. algorithms with the fewest number of multiply operations and no divisions, if possible

EMBEDDED C/C++: C++ 1/3

HOCHSCHULE ESSLINGEN

C++ has a bad reputation due to code size & speed esp. in embedded sys:

- C++ compilers are much more complex than C compilers
- Some vendors don't spend much development effort on C++ features in their compilers, esp. if the compiler is for free or has no large community

→ Use a compiler with a large sales volume or a costly professional compiler: (e.g. GNU C → Microsoft Visual C++ → Intel C++)

- Engineers need to know, which language features do cause overhead (see e.g. Scot Meyer: Effective C++, Addision-Wesley Publishers)
  No runtime overhead or code bloat (because the C++ compiler may resolve them during compile-time) have:
  - C++ class attributes are implemented in the same way as C structure variables, but with two additional function calls (constructor & destructor), when the class is instantiated. However the same code is required when a C structure is initialized.
  - C++ class methods are implemented like C functions with one additional parameter (a `this` pointer to the associated object)
  - Simple inheritance of non-virtual classes, default arguments for functions, namespaces, overloaded functions and operators, private/protected/public declarations only specify the scope of variables and methods during compilation.
  - C++ classes and structures as function parameters should be passed by (a const) reference (= pointer), not by value (=copy of the class/structure).

EMBEDDED C/C++: C++ 2/3

- **Dynamic creation** of C++ objects via `new`/and `delete` may lead to the same problems like dynamic C data structures created via malloc()/free()
  → heap fragmentation, memory leaks, out of heap memory problems, etc.
- C++ uses much **stricter rules for data type compatibility** and requires many explicit type casts rather than the more relaxed automatic casts in C. To allow type checks when linking code modules, compilers "decorate" C++ function and variable names in the object code with data type information ("name mangling"):

  Programmer defines: `int fcn(char x);`
  C++ compiler mangles into: `?fcn@@YAHD@Z`

where the @...'s indicate the type of the parameter and the return value. Unfortunately, name mangling is compiler specific and only done in C++. If this function is called from C or from an assembler module, which do not use mangling, the linker will stop with an error, because the function names between the C++ and the non-C++ module do not match. A cross-language compatible C++ function must be declared as

```
extern "C" int fcn(char x)
```

to turn off name mangling for this function. Mangling is done in the compile-link phase, it has no influence on code size or runtime execution speed.
Turning off for larger code-sections, use: `extern "C" { … }`

EMBEDDED C/C++: FEATURES WITH NEGATIVE EFFECTS

HOCHSCHULE ESSLINGEN

- Inheritance from a **virtual base class** causes an array of pointers attached to the class (Virtual Function Table VTABLE) → memory size increases. The table is used to indirectly call the function implementations via the pointers → runtime increases.
- **Exception handling** with `try {} catch {}` stores the current state of the program (CPU registers etc.) on the stack, before the `try {}` block is executed → stack size increases. If an exception occurs inside the try block, this state is restored to recover from the exception → runtime increases. As in normal program flow no exception should occur, compilers try to optimize the `try {}` path at the expense of the `catch {}` path.
- **Expensive library** functions: Some standard library functions like `iostream()` have to take care about any possible parameter combination and thus require a lot of code. For embedded applications there are simplified versions of the standard libraries, e.g. uClibc-NG, NewLib or **DietLibC** instead of glibc for GNU compilers.
- **Templates** may or may not **increase code size**, depending on how complex and nested the generated classes are. They may lead to very performance code!
- **Runtime-Type Identification RTTI** (`dynamic_cast`) allows to call a function with different object types, not known during compile time. This is one of the most esoteric C++ features and thus, compiler implementations may be very inefficient. To be avoided.

LECTURE: LITERATURE

HOCHSCHULE ESSLINGEN

According to list of literature:

- Patterson, D., Hennessy, J.: *Computer Organization and Design*, Kaufmann, 2011
  (Deutsche Übersetzung: Rechnerorganisation und –entwurf, Spektrum)
- Hennessy, J., Patterson, D.: *Computer Architecture: A quantitative Approach, **5th edition***, Morgan Kaufmann, 2017
- Tanenbaum, A., Austin, T.: *Structured Computer Organization*, Pearson, 6th edition, 2013
- Tanenbaum, A., Austin, T.: *Rechnerarchitektur: von der digitalen Logik zum Parallelrechner*, Pearson, 6th ed., 2014
- Huang, H.W.: *The HCS12/9S12. An Introduction to the HW and SW interface*, Thomson Learning, 2009
- Beierlein, T.: *Taschenbuch Mikroprozessortechnik*, Carl-Hanser Verl., 2011