HOCHSCHULE ESSLINGEN

LECTURE COMPUTER ARCHITECTURE

EMBEDDED PROGRAMMING

RAINER KELLER

CONTENT

1 Basic features of the HCS12

2 Hello World in an embedded world

3 Register model, Data Types, Addressing

4 Instruction Set

5 Stack

6 Code size and execution speed

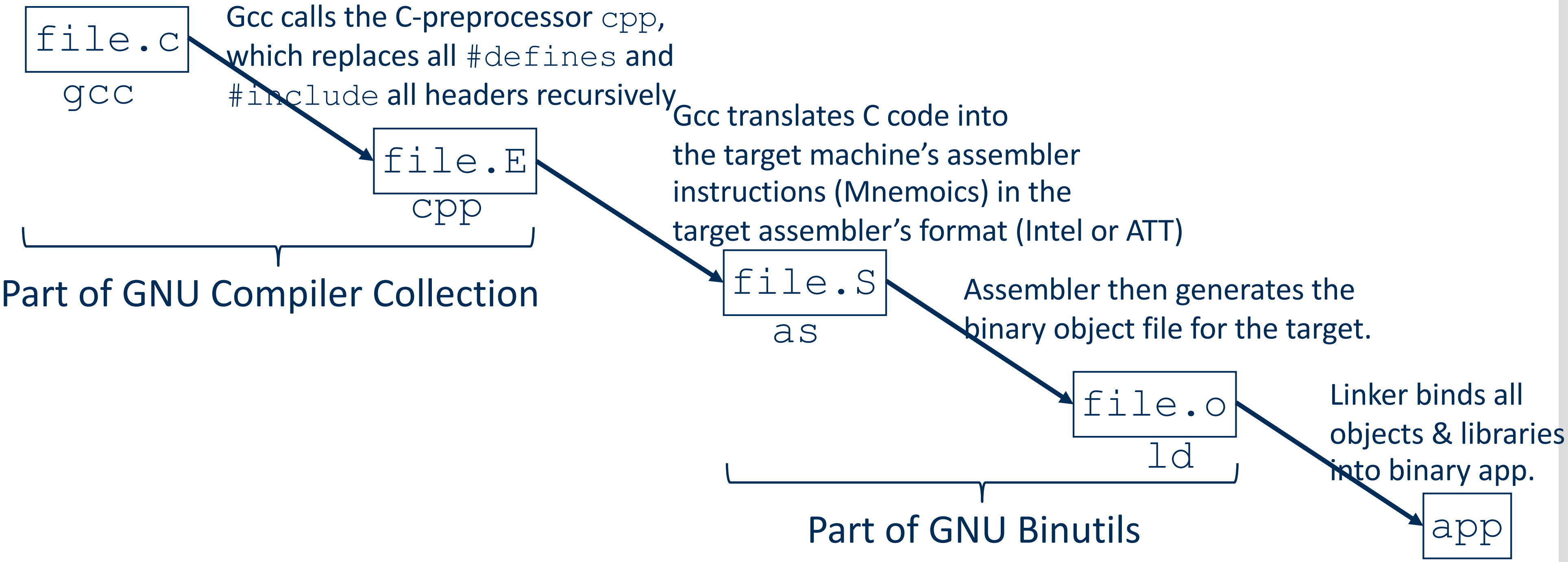
GOALS FOR TODAY

- How the Compiler translates into Machine Language and
- How that executes on the CPU
- Difference between ABI vs. API
- Know the Dragon12 components

HOW A COMPILER WORKS

- A Compiler translates one byte representation to another; or better:
- From (human readable) language to binary representation of CPU:

```
gcc -Wall -O2 -o app file.c # All Warnings, Optimization
```



`file.c` `gcc`

Gcc calls the C-preprocessor `cpp`, which replaces all `#defines` and `#include` all headers recursively

`file.E` `cpp`

Part of GNU Compiler Collection

Gcc translates C code into the target machine's assembler instructions (Mnemoics) in the target assembler's format (Intel or ATT)

`file.S` `as`

Assembler then generates the binary object file for the target.

`file.o` `ld`

Part of GNU Binutils

Linker binds all objects & libraries into binary app.

`app`

- The compiler has to map the C and Assembler code and data to an executable binary for target machine → ABI for Cross Development

ABI & API

APPLICATION BINARY INTERFACE

The application binary interface (ABI) defines:

- Processor instruction set and availability to user (CPU operating mode)
- How data is accessed (size of types, alignment of data, MSB/LSB?)
- Calling Convention:
  - How parameters are passed (Call-by-ref./…-value, reg-/stack-usage)
  - Which registers are caller-, which are callee-saved (on stack…)
  - Symbol naming/visibility for constants/functions/methods (think classes/namespaces)

And is defined by:

- The given hardware (CPU and the memory architecture), e.g.:
  - x86 with 20 Bit address space, RAM+ROM within 640kB to 1 MB…
  - x86-64 with 64 bit address space, RAM+ROM within 640kB, many GPR, legacy FP registers, modern AVX, AVX2 and AVX512 registers

- The chosen Compiler & Tools provided for this Operating system.
  e.g. GNU C Compiler on Linux Operating system (vs. Fortran, vs. Win)

32-BIT AND 64-BIT REGISTER USAGE

Architectures have been extended from 8- and 16-bit time-and-again…

Upon resizing registers, the ABIs had to be redefined as well.

Concepts (I=`integer`, L=`long`, LL=`long long`, P=Pointer) for 64bit:

Type	ILP32	LLP64	LP64	ILP64
`short`	16	16	16	16
`int`	32	32	32	64
`long`	32	32	64	64
`long long`	64	64	64	64
`pointer, size_t`	32	64	64	64

- ILP32 used on x86-64 as `x32` and on ARM `arm64ilp32` for Linux
- LLP64 Microsoft Windows on x86-64 and IA64 (including MinGW)
- LP64 Most Unixes like Linux, BSD (MacOS), but also CygWin
- ILP64 Port of Solaris on SPARC64

Why is that important?

Casting Pointer to `long` (or some other “small” integer) is **wrong**.

Use `uintptr_t` or `intptr_t` when casting..

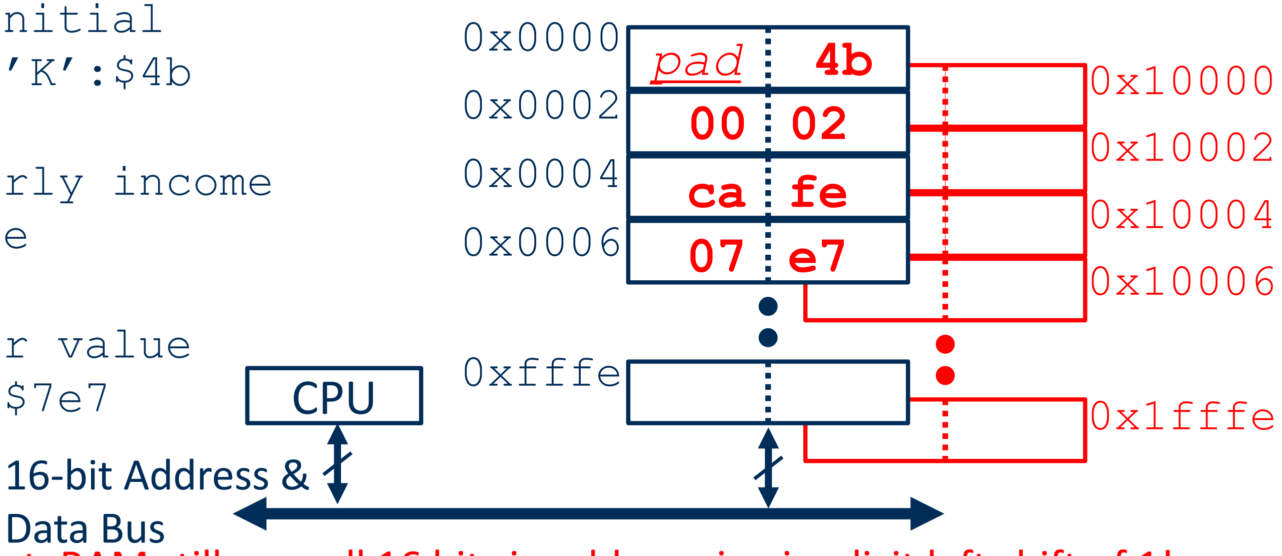
ABI OF A REDUCED INSTRUCTION SET COMPUTER

Consider a 16-bit RISC CPU with von-Neumann architecture:

- Registers are 16-bit wide, the natural data word size n_{DAT} = 16 bit
- Address and Data bus are the same, so n_{ADR} = 16 bit
- Smallest addressable unit n_{min} = 2 Bytes

  Address space: $N = 2^{nADR} * n_{min} = 2^{16}$ Byte $* 2 = 128$ kB
- Multibyte values stored in most-significant byte MSB-order: Big Endian
- Compiler has to acknowledge alignment and either pad data in structures < 4 Bytes, or re-order data in structures:

```
struct values {
    // Last name initial
    // here ASCII 'K':$4b
    char initial;
    // 4 Bytes yearly income
    // here: $2cafe
    int income;
    // 2 Bytes year value
    // here: year $7e7
    short year;
}
```



1. Due to n_{min}=2 HW alignment, RAM still uses all 16 bits in address, i.e. implicit left shift of 1!
2. As an Exercise: Please visualize the storage in case of a little Endian System

HOCHSCHULE ESSLINGEN

Hardware
Freescale 68HC12 / HCS12
Dragon-12 Board

INFORMATION ON FREESCALE HCS12

- **Von Neumann** architecture
- Complex Instruction Set (**CISC**)
- Data word size n_{DAT} = 16 bit
- Address word size n_{ADR} = 16 bit
- Smallest addressable unit n_{min} = 1 Byte
  Address space: $N = 2^{nADR} * n_{min} = 2^{16}$ Byte = 64 kB
  extensible via memory banking (pages)
- No memory alignment, i.e. instructions & data can start at any address
- Multi-byte values stored in **Big Endian** (MSB first) sequence
  Memory Access requires address of the first byte and length of data

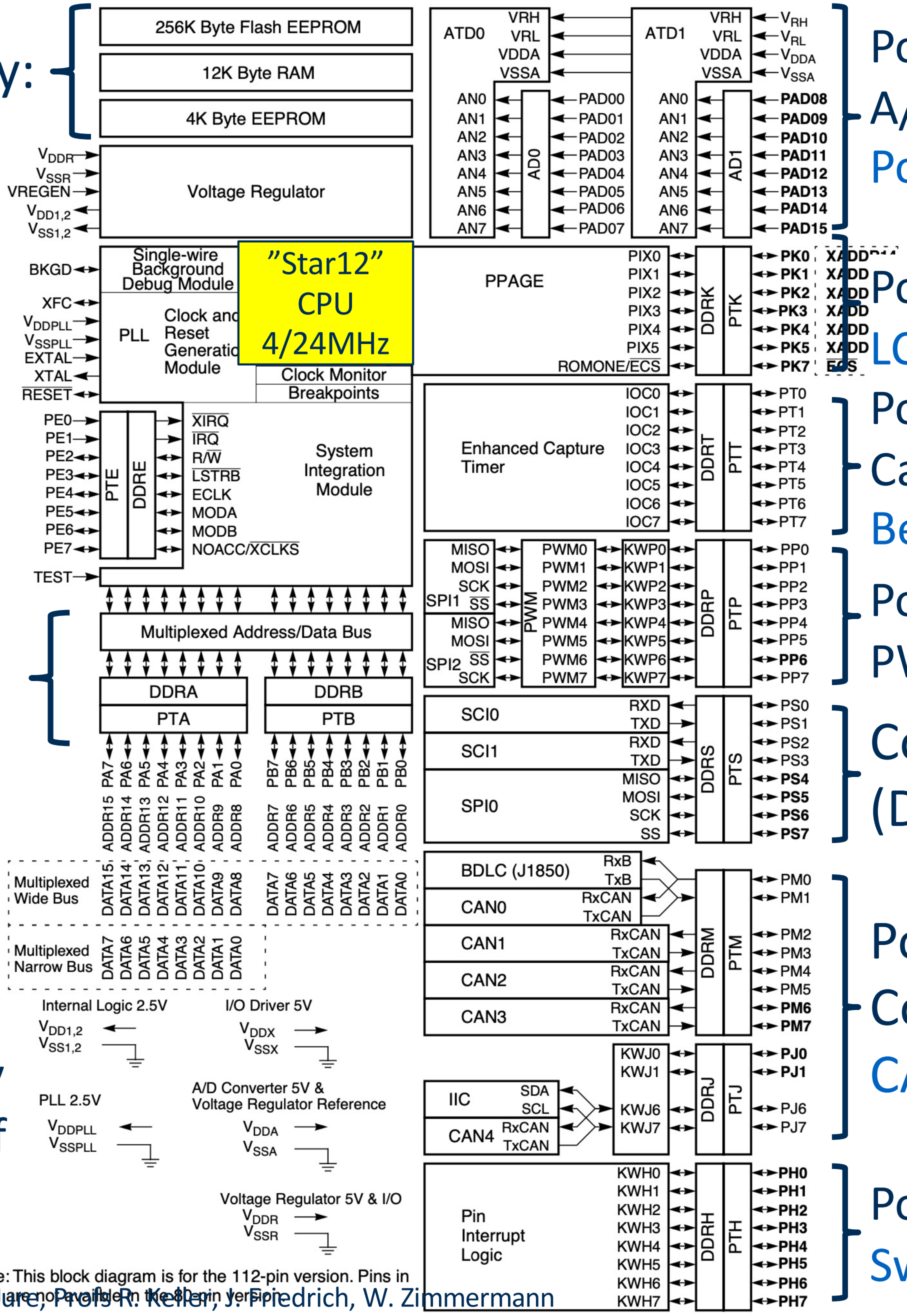
Literature:

- https://www.nxp.com/docs/en/data-sheet/MC9S12DP256.pdf
- Barret, S.: *Embedded systems design and applications with the 68HC12 and HCS12*, Pearson, 2005, Available in Bücherei Esslingen
- Lipovski, G.: *Introduction to microcontrollers: architecture, programm., and interfacing for the Freescale 68HC12*, Elsevier, Online (via VPN)

BLOCK DIAGRAM: MICROCONTROLLER MC9S12DP256

HOCHSCHULE ESSLINGEN

MC9S12DP256 112-Pin Block Diagram


256K Byte Flash EEPROM
12K Byte RAM
4K Byte EEPROM
Voltage Regulator
VDDR
VSSR
VREGEN
VDD1,2
VSS1,2
ATD0
ATD1
VRH
VRL
VDDA
VSSA
AN0 AN1 AN2 AN3 AN4 AN5 AN6 AN7
AD0
AD1
PAD00 PAD01 PAD02 PAD03 PAD04 PAD05 PAD06 PAD07
PAD08 PAD09 PAD10 PAD11 PAD12 PAD13 PAD14 PAD15
BKGD
Single-wire Background Debug Module
"Star12" CPU 4/24MHz
XFC
VDDPLL
VSSPLL
EXTAL
XTAL
RESET
PLL
Clock and Reset Generation Module
Clock Monitor
Breakpoints
PPAGE
PIX0 PIX1 PIX2 PIX3 PIX4 PIX5
ROMONE/ECS
DDRK
PTK
PK0 PK1 PK2 PK3 PK4 PK5 PK7
XADDR14 XADDR15 XADDR16 XADDR17 XADDR18 XADDR19 ECS
PE0 PE1 PE2 PE3 PE4 PE5 PE6 PE7
PTE
DDRE
XIRQ
IRQ
R/W
LSTRB
ECLK
MODA
MODB
NOACC/XCLKS
System Integration Module
TEST
Enhanced Capture Timer
IOC0 IOC1 IOC2 IOC3 IOC4 IOC5 IOC6 IOC7
DDRT
PTT
PT0 PT1 PT2 PT3 PT4 PT5 PT6 PT7
Multiplexed Address/Data Bus
DDRA
PTA
DDRB
PTB
PA7 PA6 PA5 PA4 PA3 PA2 PA1 PA0
PB7 PB6 PB5 PB4 PB3 PB2 PB1 PB0
ADDR15 ADDR14 ADDR13 ADDR12 ADDR11 ADDR10 ADDR9 ADDR8
ADDR7 ADDR6 ADDR5 ADDR4 ADDR3 ADDR2 ADDR1 ADDR0
Multiplexed Wide Bus
DATA15 DATA14 DATA13 DATA12 DATA11 DATA10 DATA9 DATA8
DATA7 DATA6 DATA5 DATA4 DATA3 DATA2 DATA1 DATA0
Multiplexed Narrow Bus
DATA7 DATA6 DATA5 DATA4 DATA3 DATA2 DATA1 DATA0
SPI1
SPI2
MISO MOSI SCK SS
PWM
PWM0 PWM1 PWM2 PWM3 PWM4 PWM5 PWM6 PWM7
KWP0 KWP1 KWP2 KWP3 KWP4 KWP5 KWP6 KWP7
DDRP
PTP
PP0 PP1 PP2 PP3 PP4 PP5 PP6 PP7
SCI0
SCI1
RXD TXD
SPI0
DDRS
PTS
PS0 PS1 PS2 PS3 PS4 PS5 PS6 PS7
BDLC (J1850)
RxB TxB
CAN0
CAN1
CAN2
CAN3
RxCAN TxCAN
DDRM
PTM
PM0 PM1 PM2 PM3 PM4 PM5 PM6 PM7
IIC
SDA SCL
CAN4
KWJ0 KWJ1 KWJ6 KWJ7
DDRJ
PTJ
PJ0 PJ1 PJ6 PJ7
Pin Interrupt Logic
KWH0 KWH1 KWH2 KWH3 KWH4 KWH5 KWH6 KWH7
DDRH
PTH
PH0 PH1 PH2 PH3 PH4 PH5 PH6 PH7
Internal Logic 2.5V
VDD1,2
VSS1,2
I/O Driver 5V
VDDX
VSSX
PLL 2.5V
VDDPLL
VSSPLL
A/D Converter 5V & Voltage Regulator Reference
VDDA
VSSA
Voltage Regulator 5V & I/O
VDDR
VSSR
Note: This block diagram is for the 112-pin version. Pins in bold are not available in the 80-pin version.

Internal memory:

Circuits on the Dragon-12 Board marked in blue!

Port A: 8-Bit inout
Port B: 8-Bit out
LEDs and LCD Display

The HCS12 micro-controller family has ~120 different memory configurations & mix of peripherals

Port AD: 2x 8-Bit in A/D Convertor
Potentiometer

Port K: 8 bit inout
LCD Display

Port T: 8 bit inout
Capture/Compare In/out
Beeper

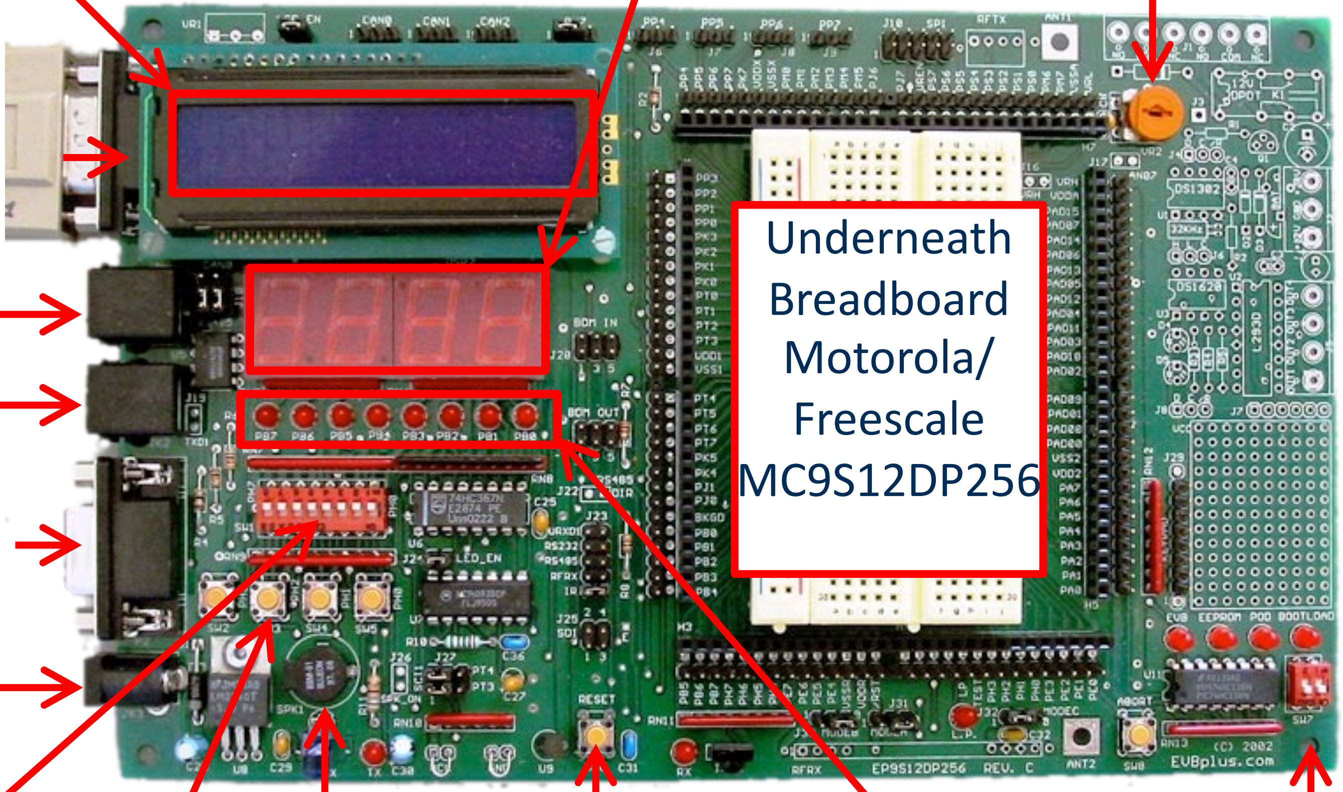
Port P: 8 bit out
PWM Output (no Dragon12)

CodeWarrior Dev Suite
(Debugging via SCI0)

Port M/J: 8 bit inout
Controller Area Network
CAN bus

Port H: 8 bit in
Switches & Buttons

DRAGON-12 EVALUATION BOARD



LCD Display
2 rows with 16 characters
Port K7...0

LED 7-segment display
4 digits
Port B7...0 & P3...0

Potentiometer
Port AD7...0

Serial Interface SCI0
(Debugger)

CAN
Bus

Serial Interface SCI1
(User Program)

Power
Supply

Underneath
Breadboard
Motorola/
Freescale
MC9S12DP256

8 DIP Switches
Port H7...0

4 Buttons
Port H3...0

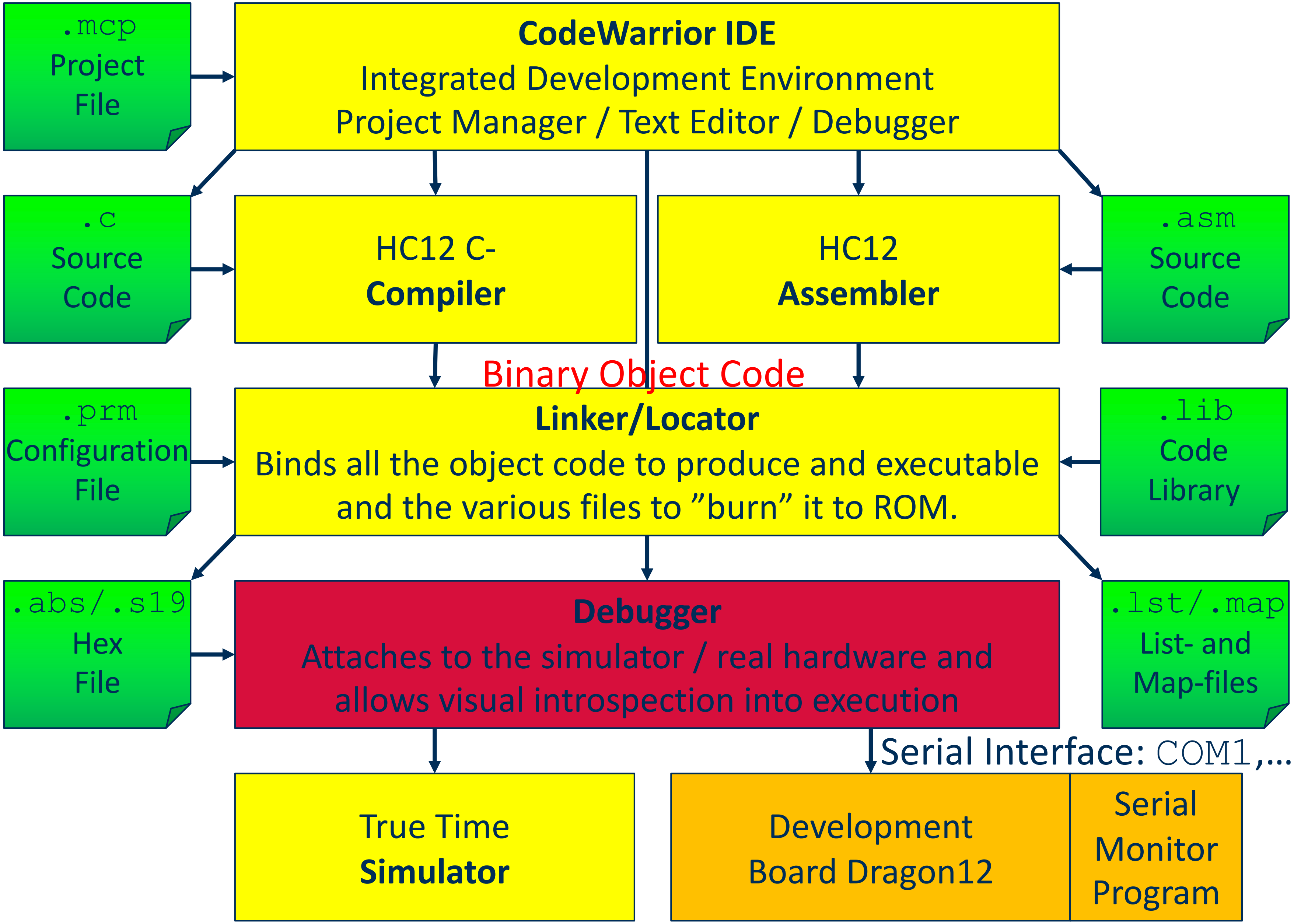
Beeper
Port T5

Reset
Button

8 Single LEDs
Port B7...0 & J1

Debug
Mode

IDE FREESCALE CODEWARRIOR



.mcp
Project
File

CodeWarrior IDE
Integrated Development Environment
Project Manager / Text Editor / Debugger

.c
Source
Code

HC12 C-
Compiler

HC12
Assembler

.asm
Source
Code

Binary Object Code

.prm
Configuration
File

Linker/Locator
Binds all the object code to produce and executable and the various files to ”burn” it to ROM.

.lib
Code
Library

.abs/.s19
Hex
File

Debugger
Attaches to the simulator / real hardware and allows visual introspection into execution

.lst/.map
List- and
Map-files

Serial Interface: COM1,…

True Time
Simulator

Development
Board Dragon12

Serial
Monitor
Program

MC9S12DP256 MEMORY MAP

The HCS12 has several operating modes. In the lab / Dragon12 we use the “Normal Single Chip Mode” without external memory.

$0000	HW/SW Interface: Registers to control the on-chip-peripherals 1kB
$0400	EEPROM 3kB
$1000	RAM 12kB
$4000	Flash-ROM 16kB
$8000	Paged Flash-ROM 16kB
$C000	Flash-ROM 16kB
$FFFF	

All peripherals are **memory-mapped**, i.e. for SW, their registers look like variables

The EEPROM has 4kB, but 1kB is shadowed by the peripheral registers.

Stack for debug monitor program at the end of the RAM area (36 Bytes, don’t overwrite)

This address range can be used to map additional 16kB Flash-ROM pages (Page Window selected by PPAGE-register) → Memory extension to > 64kB

`$F780` … `$FE00`: Debugger monitor program
`$FF00` … `$FFFF`: Interrupt Vector Table 256B

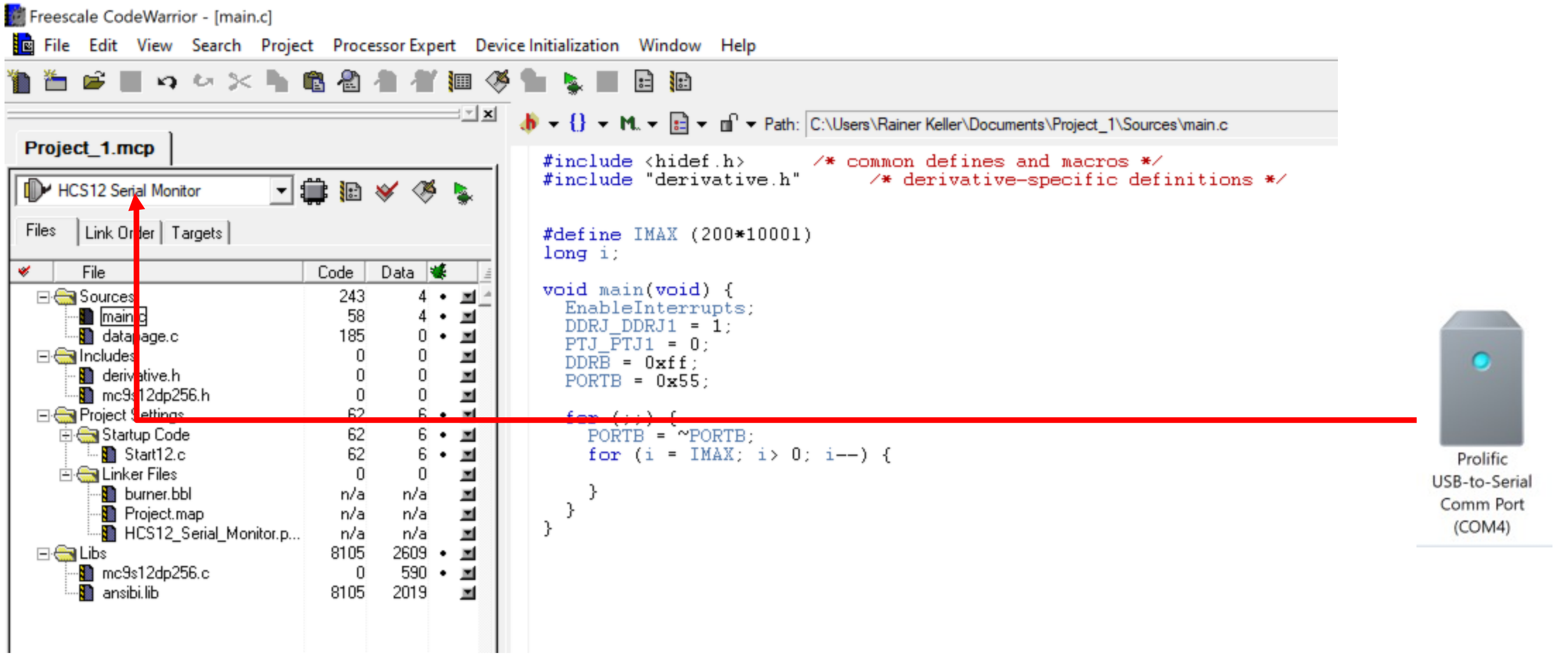
See `MC9S12DP256.pdf`, **p121**

HOCHSCHULE ESSLINGEN

Programming the Freescale 68HC12 / HCS12

INSTALLATION OF CODEWARRIOR V5.1

1. Follow the installation script `CA3_CW_Installation.pdf`
2. Start the IDE:


Freescale CodeWarrior - [main.c]
File Edit View Search Project Processor Expert Device Initialization Window Help
Path: C:\Users\Rainer Keller\Documents\Project_1\Sources\main.c
Project_1.mcp
HCS12 Serial Monitor
Files
Link Order
Targets
Prolific
USB-to-Serial
Comm Port
(COM4)

File	Code	Data	
Sources	243	4	•
main.c	58	4	•
datapage.c	185	0	•
Includes	0	0	
derivative.h	0	0	
mc9s12dp256.h	0	0	
Project Settings	62	6	•
Startup Code	62	6	•
Start12.c	62	6	•
Linker Files	0	0	
burner.bbl	n/a	n/a	
Project.map	n/a	n/a	
HCS12_Serial_Monitor.p...	n/a	n/a	
Libs	8105	2609	•
mc9s12dp256.c	0	590	•
ansibi.lib	8105	2019	

```
#include <hidef.h>      /* common defines and macros */
#include "derivative.h"      /* derivative-specific definitions */


#define IMAX (200*10001)
long i;

void main(void) {
  EnableInterrupts;
  DDRJ_DDRJ1 = 1;
  PTJ_PTJ1 = 0;
  DDRB = 0xff;
  PORTB = 0x55;

  for (;;) {
    PORTB = ~PORTB;
    for (i = IMAX; i> 0; i--) {

    }
  }
}
```

3. Creating a new project, the IDE queries the target CPU
   You don’t need the Rapid Development Options
4. The default settings will create an pre-generated `main.c` which includes `derivative.h`, which includes `mc9s12dp256.h`
5. In case You have HW: choose HCS12 Serial Monitor (select COM1…)

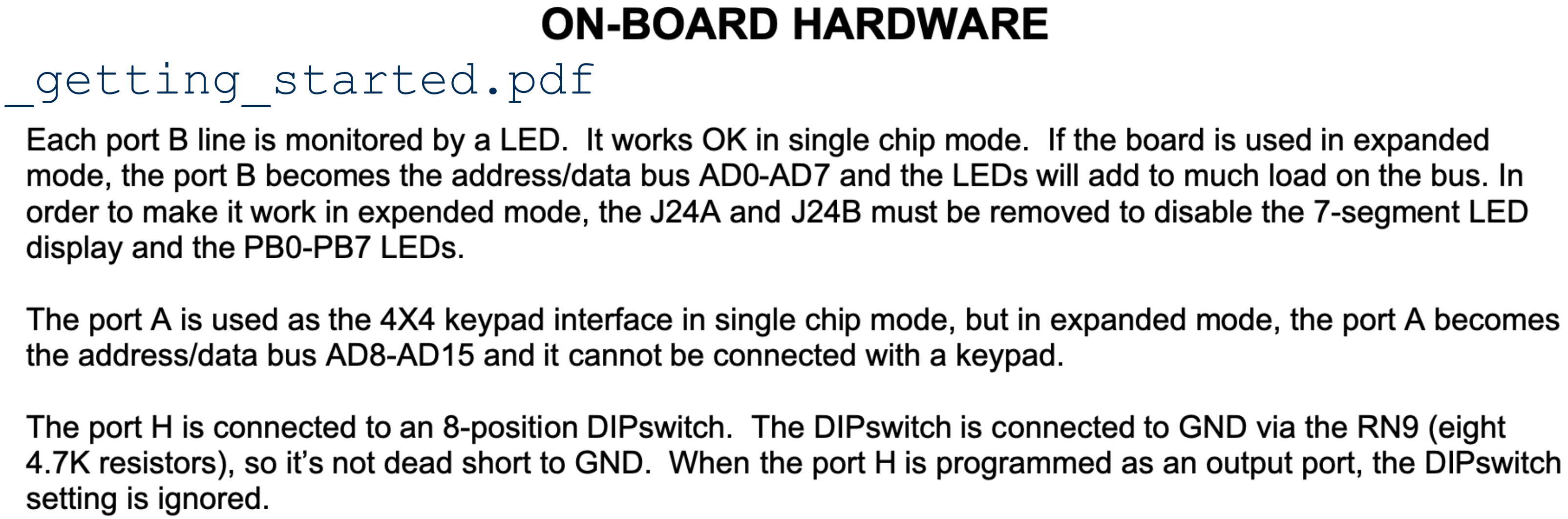
HELLO EMBEDDED WORLD 1/7

The first program is always “Hello World”…
However, as we’re on a typical embedded system, we don’t have a Keyboard, or a display… But we have LEDs:
Through the Toggle Ports we may switch Blinking LEDs.

This requires some setup:

Step 1: What is the hardware setup of the evaluation board?
Where are the LEDs connected and how can they be controlled?

See
`Dragon12_getting_started.pdf`



ON-BOARD HARDWARE

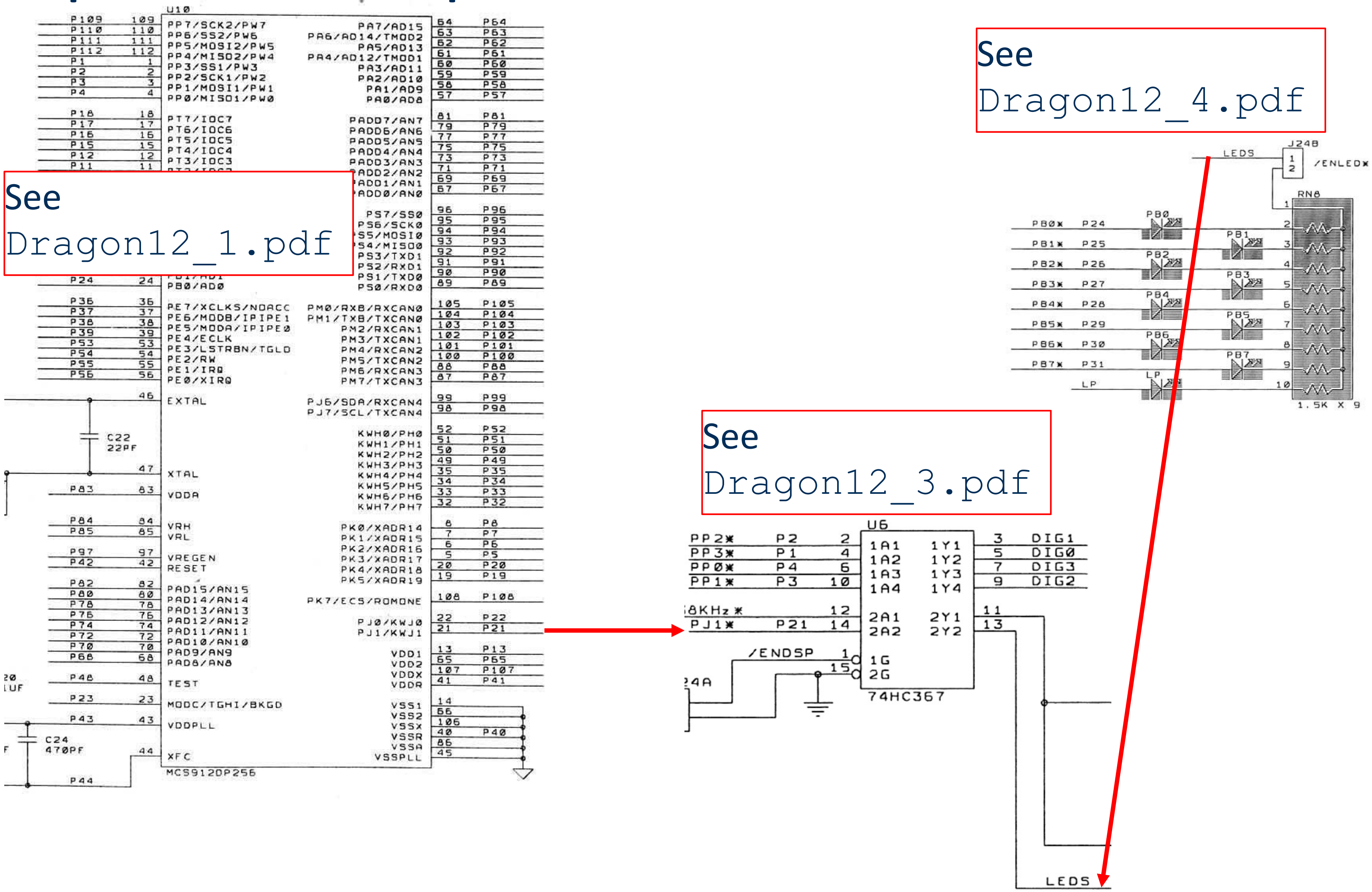
Each port B line is monitored by a LED. It works OK in single chip mode. If the board is used in expanded mode, the port B becomes the address/data bus AD0-AD7 and the LEDs will add to much load on the bus. In order to make it work in expended mode, the J24A and J24B must be removed to disable the 7-segment LED display and the PB0-PB7 LEDs.

The port A is used as the 4X4 keypad interface in single chip mode, but in expanded mode, the port A becomes the address/data bus AD8-AD15 and it cannot be connected with a keypad.

The port H is connected to an 8-position DIPswitch. The DIPswitch is connected to GND via the RN9 (eight 4.7K resistors), so it’s not dead short to GND. When the port H is programmed as an output port, the DIPswitch setting is ignored.

HELLO EMBEDDED WORLD 2/7

Step 2: Hardware setup of the microcontroller? Please note Revision E!



See
Dragon12_1.pdf

See
Dragon12_4.pdf

See
Dragon12_3.pdf

Port J1 as Output set to 0, and Port B0…7 as output and set to 0 or 1…

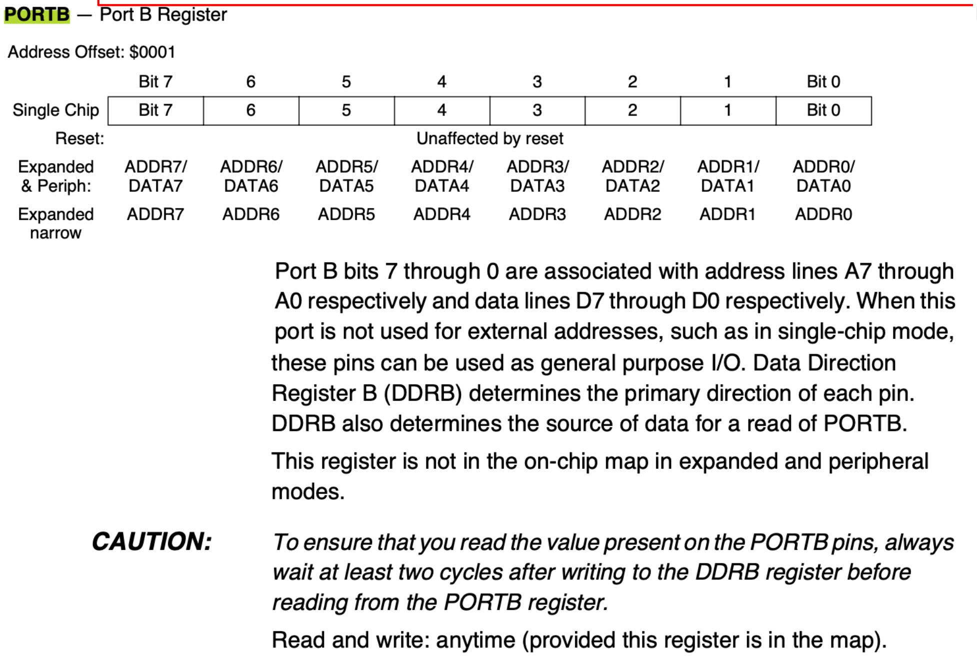
HELLO EMBEDDED WORLD 3/7

Step 2: Hardware setup of the microcontroller?

Where are I/O ports in the memory address range and how to program?

DDR: Data Direction Register, Bit 0: high-impedance input, Bit 1: output

See `MC9S12DP256.pdf` P66ff & p129ff



PORTB — Port B Register

Address Offset: $0001

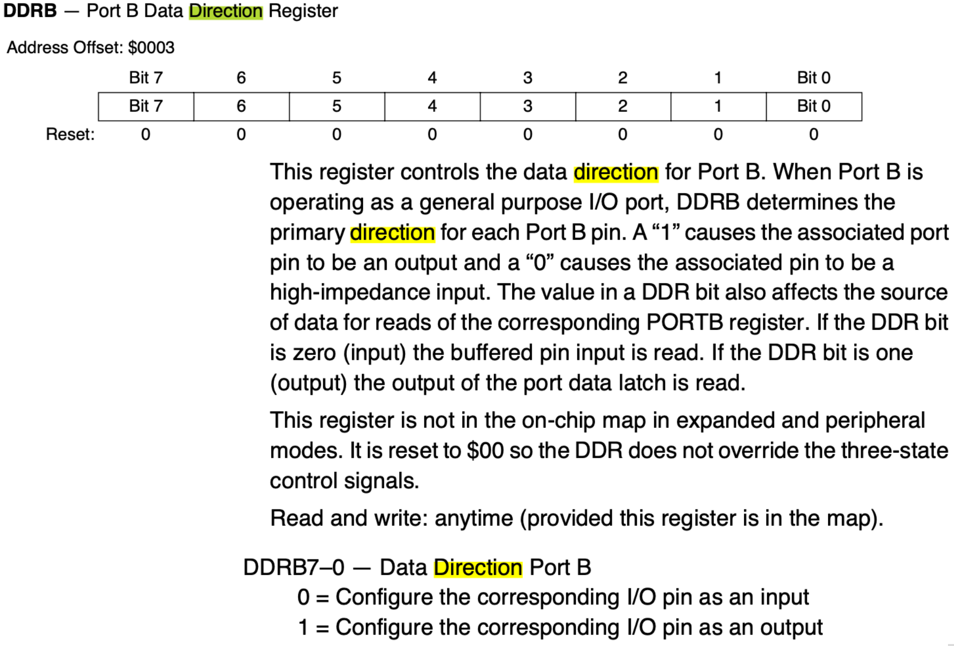
	Bit 7	6	5	4	3	2	1	Bit 0
Single Chip	Bit 7	6	5	4	3	2	1	Bit 0
Reset:	Unaffected by reset							
Expanded & Periph:	ADDR7/ DATA7	ADDR6/ DATA6	ADDR5/ DATA5	ADDR4/ DATA4	ADDR3/ DATA3	ADDR2/ DATA2	ADDR1/ DATA1	ADDR0/ DATA0
Expanded narrow	ADDR7	ADDR6	ADDR5	ADDR4	ADDR3	ADDR2	ADDR1	ADDR0

Port B bits 7 through 0 are associated with address lines A7 through A0 respectively and data lines D7 through D0 respectively. When this port is not used for external addresses, such as in single-chip mode, these pins can be used as general purpose I/O. Data Direction Register B (DDRB) determines the primary direction of each pin. DDRB also determines the source of data for a read of PORTB.

This register is not in the on-chip map in expanded and peripheral modes.

CAUTION: *To ensure that you read the value present on the PORTB pins, always wait at least two cycles after writing to the DDRB register before reading from the PORTB register.*

Read and write: anytime (provided this register is in the map).



DDRB — Port B Data Direction Register

Address Offset: $0003

	Bit 7	6	5	4	3	2	1	Bit 0
	Bit 7	6	5	4	3	2	1	Bit 0
Reset:	0	0	0	0	0	0	0	0

This register controls the data direction for Port B. When Port B is operating as a general purpose I/O port, DDRB determines the primary direction for each Port B pin. A "1" causes the associated port pin to be an output and a "0" causes the associated pin to be a high-impedance input. The value in a DDR bit also affects the source of data for reads of the corresponding PORTB register. If the DDR bit is zero (input) the buffered pin input is read. If the DDR bit is one (output) the output of the port data latch is read.

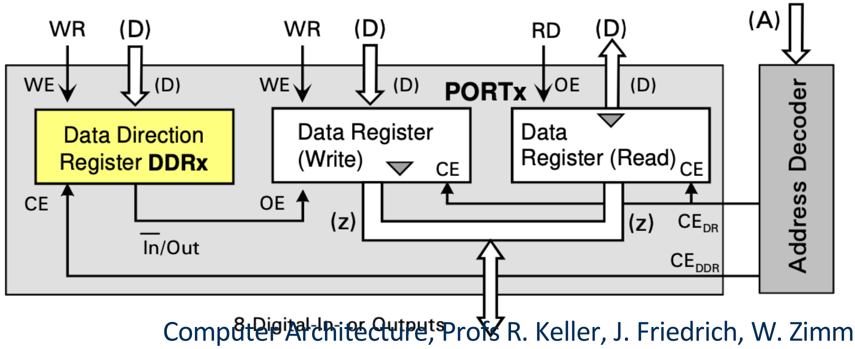
This register is not in the on-chip map in expanded and peripheral modes. It is reset to $00 so the DDR does not override the three-state control signals.

Read and write: anytime (provided this register is in the map).

DDRB7–0 — Data Direction Port B

0 = Configure the corresponding I/O pin as an input

1 = Configure the corresponding I/O pin as an output



Same for Port J:

Data register PTJ at address $0268

Data direction register DDRJ at address $026A

HELLO EMBEDDED WORLD 4/7

Step 3: Development environment, program design and coding

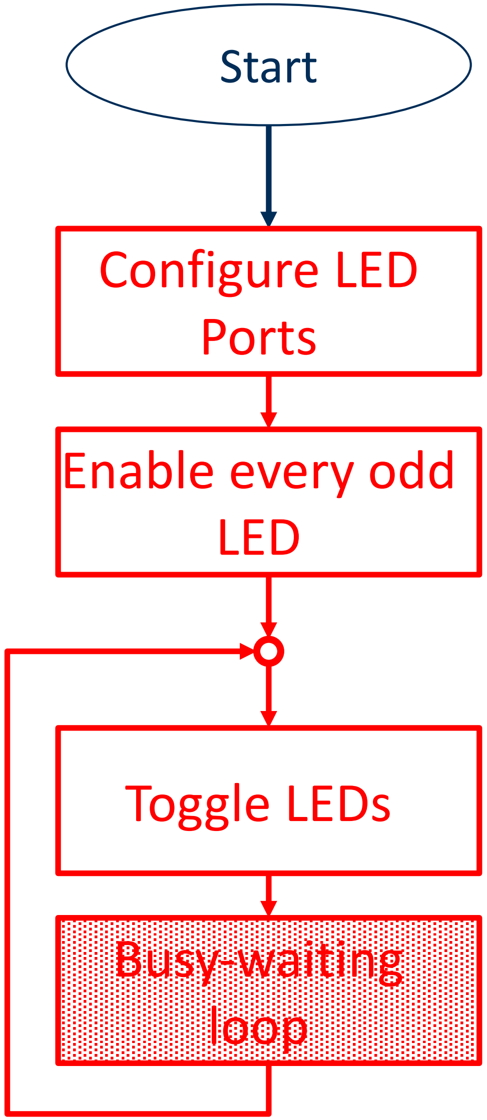
How do I write and compile a program?

- Installation and use the IDE see documentation package
- Instead of coding hexadecimal addresses for Port B, the IDE provides include files defining symbols for registers and their respective bit, e.g.

Predefined Symbols in Include-Files for		For C Programs mc9s12dp256.h	For ASM Programs mc9s12dp256.inc
Port B	Port	#define PORTB (*(char*) 0x0001)	PORTB: equ $0001
	PortB Bit0	#define PORTB_BIT0 PORTB.Bits.BIT0	
	…		
	DDRB	#define DDRB (*(char*) 0x0003)	DDRB: equ $0003
Port J	Port	#define PTJ (*(char*) 0x0268)	PTJ: equ $0268
	PTJ Bit0	#define PTJ_PTJ0 PTJ.Bits.PTJ0	
	…		

HELLO EMBEDDED WORLD 5/7

Code Design



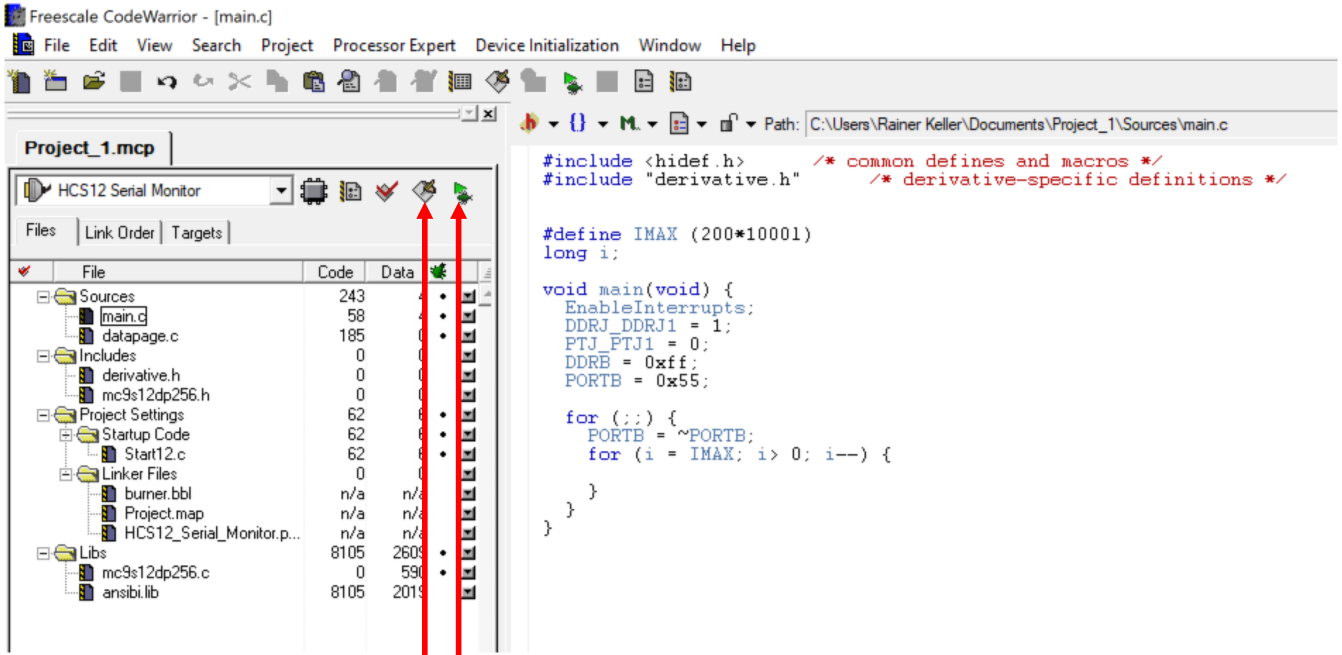
C-Code (see project `BlinkingLeds.mcp`)

```
#include <hidef.h>          // Defines for Debugger
#include <mc9s12dp256.h>    // CPU specific defines

// #pragma LINK_INFO DERIVATIVE "mc9s12dp256b"
#define IMAX 200*1000L      // Delay loop counter
long i;

void main(void) {
    EnableInterrupts;// Allow for debugger
    DDRJ_DDRJ1 = 1;  // Port J.1 as output
    PTJ_PTJ1 = 0;    // J.1=0 --> Activate LEDs
    DDRB = 0xff;     // Port B all Pins as outputs
    PORTB = 0x55;    // Turn on every other LED
    for(;;) {        // Main/Endless Loop
        PORTB = ~PORTB;//Toggle LEDs (Bitwise Not)
        for (i=IMAX; i>0; i--) {
            // Delay loop
        }
    }
}
```

HELLO EMBEDDED WORLD 6/7

Step 4: Development Environment

How do I compile a program?



Freescale CodeWarrior - [main.c]
File Edit View Search Project Processor Expert Device Initialization Window Help
Project_1.mcp
HCS12 Serial Monitor
Files | Link Order | Targets

Path: C:\Users\Rainer Keller\Documents\Project_1\Sources\main.c

✓ File	Code	Data		
Sources	243	4	•	
main.c	58	4	•	
datapage.c	185	0	•	
Includes	0	0		
derivative.h	0	0		
mc9s12dp256.h	0	0		
Project Settings	62	6	•	
Startup Code	62	6	•	
Start12.c	62	6	•	
Linker Files	0	0		
burner.bbl	n/a	n/		
Project.map	n/a	n/		
HCS12_Serial_Monitor.p...	n/a	n/		
Libs	8105	260	•	
mc9s12dp256.c	0	59	•	
ansibi.lib	8105	201		

```
#include <hidef.h>      /* common defines and macros */
#include "derivative.h"      /* derivative-specific definitions */


#define IMAX (200*10001)
long i;

void main(void) {
  EnableInterrupts;
  DDRJ_DDRJ1 = 1;
  PTJ_PTJ1 = 0;
  DDRB = 0xff;
  PORTB = 0x55;

  for (;;) {
    PORTB = ~PORTB;
    for (i = IMAX; i> 0; i--) {

    }
  }
}
```

- Once You’re done writing code, You may “make” it….
  This takes all .c/.asm files in Link Order and compiles and links them
- In order to execute it in the Simulator / on the HCS12, click the debug

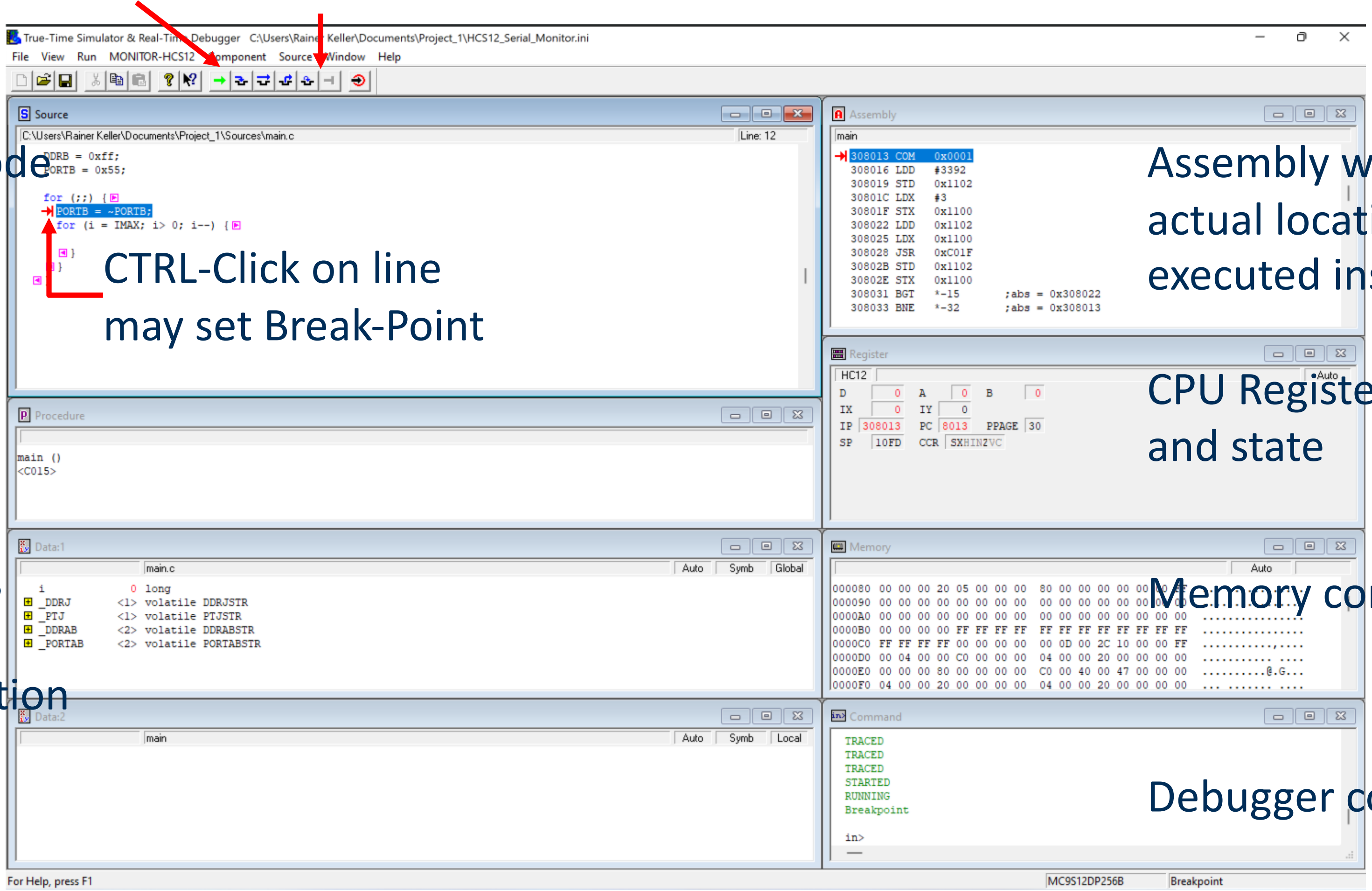
HELLO EMBEDDED WORLD 7/7

Step 5: Debug Environment

How do I debug my running program?

Continue execution

Single step / Instr. step


True-Time Simulator & Real-Time Debugger C:\Users\Rainer Keller\Documents\Project_1\HCS12_Serial_Monitor.ini
File View Run MONITOR-HCS12 Component Source Window Help
Source
C:\Users\Rainer Keller\Documents\Project_1\Sources\main.c
Line: 12
Assembly
main
Register
HC12
Auto
Procedure
Data:1
main.c
Auto
Symb
Global
Data:2
main
Auto
Symb
Local
Memory
Auto
Command
For Help, press F1
MC9S12DP256B
Breakpoint

```
DDRB = 0xff;
PORTB = 0x55;

for (;;) {
  PORTB = ~PORTB;
  for (i = IMAX; i> 0; i--) {

  }
}
```

```
308013 COM    0x0001
308016 LDD    #3392
308019 STD    0x1102
30801C LDX    #3
30801F STX    0x1100
308022 LDD    0x1102
308025 LDX    0x1100
308028 JSR    0xC01F
30802B STD    0x1102
30802E STX    0x1100
308031 BGT    *-15        ;abs = 0x308022
308033 BNE    *-32        ;abs = 0x308013
```

```
D      0    A      0    B      0
IX     0    IY     0
IP 308013   PC  8013    PPAGE 30
SP   10FD   CCR SXHINZVC
```

```
main ()
<C015>
```

```
   i          0  long
⊞ _DDRJ      <1>  volatile DDRJSTR
⊞ _PTJ       <1>  volatile PTJSTR
⊞ _DDRAB     <2>  volatile DDRABSTR
⊞ _PORTAB    <2>  volatile PORTABSTR
```

```
000080  00 00 00 20 05 00 00 00  80 00 00 00 00 00 00 FF
000090  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
0000A0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ................
0000B0  00 00 00 00 FF FF FF FF  FF FF FF FF FF FF FF FF   ................
0000C0  FF FF FF FF 00 00 00 00  00 0D 00 2C 10 00 00 FF   ...........,....
0000D0  00 04 00 00 C0 00 00 00  04 00 00 20 00 00 00 00   ........... ....
0000E0  00 00 00 80 00 00 00 00  C0 00 40 00 47 00 00 00   ..........@.G...
0000F0  04 00 00 20 00 00 00 00  04 00 00 20 00 00 00 00   ... ....... ....
```

```
TRACED
TRACED
TRACED
STARTED
RUNNING
Breakpoint

in>
```

Source Code

CTRL-Click on line may set Break-Point

Assembly with actual location of executed instruction

CPU Registers and state

Variables and Visualization

Memory content

Debugger commands

HOCHSCHULE ESSLINGEN

Evaluating
Freescale 68HC12 / HCS12

MEMORY REQUIREMENTS HELLO EMBEDDED WORLD

The memory requirements are available in file `proj_name.map`:

```
Summary of section sizes per section type:
READ_ONLY (R):    8E (dec:142) ← ROM: Program code + constant data
READ_WRITE (R/W):104 (dec:260) ← RAM: Variable data 4 Byte (+Stack 256 B)
NO_INIT (N/I):   23D (dec:573) ← Peripheral registers (fixed for all progs…)
```

The C compiler doesn't (by default) optimize… By manually programming in machine language (assembler), faster and smaller code is possible.

```
Summary of section sizes per section type:
READ_ONLY (R):    2A (dec:42)  ← ROM: Program code + constant data
READ_WRITE (R/W):100 (dec:256) ← RAM: Variable data 0 Byte (+Stack 256 B)
```

Execution (in CPU clock Ticks in Simulator, for loop IMAX length of 1):

Run time	C	ASM
CPU Reset until Toggle	101 clocks	21 clocks
1 loop cycle Toggle LEDs	53 clocks	19 clocks

Here the stack could have been reduced in the C program – in the ASM without debugging support, we could've eliminated the Stack altogether.

EMBEDDED HELLO WORLD IN ASM

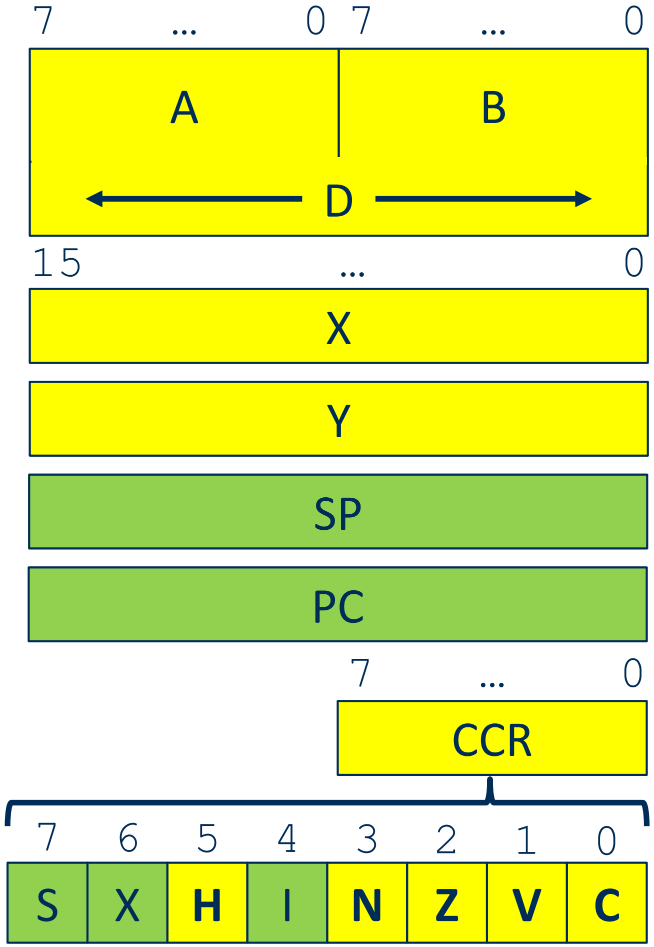
Blinking LEDs, optimized HCS12 Assembler (BlinkingLedsAsm.mcp)

```
    INCLUDE 'derivative.inc'  ; Generated file, includes m9s12dp12.inc
    XDEF Entry, _Startup, main; Export symbols to reference in C/C++
    XREF __SEG_END_SSTACK     ; Symbol defined by linker: end of stack
IMAX: EQU 2048                ; Symbolic constant: Delay count
main:    SECTION
_Startup:
Entry:
    LDS  #__SEG_END_SSTACK  ; initialize the stack pointer
    CLI                     ; enable interrupts
    BSET DDRJ, #2           ; Bit Set: Port J.1 as output
    BCLR PTJ, #2            ; Bit Clear: J.1=0 --> Activate LEDs
    MOVB #$FF, DDRB         ; $FF -> DDRB: Port B.7...0 as outputs (LEDs)
    MOVB #$55, PORTB        ; $55 -> PORTB: Turn on every other LED
loop:
    COM PORTB               ; Complement PortB: Toggle every other LED
    LDX #IMAX               ; X contains counter
waitO:
    LDY #IMAX               ; Two nested counter loops with registers X and Y
waitI:
    DBNE Y, waitI           ; Decrement Y, branch to waitI if not equal to 0
    DBNE X, waitO           ; Decrement X, branch to waitO if not equal to 0
    BRA   loop              ; Branch to loop creating an endless loop
```

Register Model
Freescale 68HC12 / HCS12

REGISTER MODEL ACCESSIBLE IN ASSEMBLER

See `001-S12CPUV2- ReferenceManual.pdf`, p25



Accumulator:
16 bit register D, can be used in two halves as two 8 bit registers A and B for arithmetic-logic operations

Index Register X: For data and/or pointers

Index Register Y: For data and/or pointers

Stack Pointer SP: Pointer to Stack

Program Counter PC: Pointer to next instruction

Condition Code Register (status register)
Status bits for arithmetic/logic operations and control

Carry/Borrow (last op produced carry; for signed)

e**X**ternal interrupt mask

o**V**erflow (unsigned)

Half carry (for BCD operations)

Interrupt mask

Zero

Negative

Stop disable, ignore stop command (state after CPU reset: S=1)

DATA TYPES, OPERAND ADDRESSING

		HCS12		80x86
		Assembler	HCS12 C *1	Visual C++
Natural numbers (unsigned) *2				
Whole numbers (signed, 2s-complement)				

8 bit	`-128 ... +127`
	`0 … 255`
		`DC.B, DS.B` *3	`char`
			`unsigned char`	

16 bit	`-32768 … +32767`
	`0 … 65535`
		`DC.W, DS.W` *3	`short int`
			`unsigned short`
				`short`
				`unsigned short`

32 bit	`-2147483648…+2147483647`
	`0 … 4294967296`
		`DC.L, DS.L` *3	`long`
			`unsigned long`
				`int / long`
				`unsigned int/long`

Floating point numbers:				
	IEEE 32-Bit		`float, double` *1	`float`
	IEEE 64-bit		`(double)` *1	`double`
Addresses / Pointers (to all data types)		16 bit (near pointer)	16 bit (near pointer)	32 bit or 64 bit
Bit field		1 bit	8, 16 or 32 bit	32 bit
Enumeration		--	16 bit	32 bit
Array		*3	`datatype name[count]`	
Structure, union		--	`struct, union`	

CODING OF NUMBERS AND STRING CONSTANTS

	HCS12 Asm	C
Decimal (Base 10)	`-34, 128`	
Hexadecimal (Base 16)	`$3f8a, -$3f`	`0x3f8a`
Octal (Base 8)	`@7345`	`07345 – don’t use ;)`
Dual	`%10101001`	`0b10101001`
Floating-point	--	`3.141` or `1.6e-19`
ASCII character	`'Z'`	
ASCIIZ string *4	`"This is a string",0`	`"This is a string"`

*1 Bit size of most data types are configurable via HCS12 compiler options

*2 Assembler does not differentiate between `signed` and `unsigned` data

*3 Variable in RAM memory: `name: DS.B  count`
defines 8 bit variables in RAM, which can be used via their `name`. The variables are **not** initialized. Use `count` > 1 to define an array. Use `DS.W` and `DS.L` to define 16 bit or 32 bit variables and arrays.
Constant in ROM memory: `name: DC.B val` or `name: DC.B count,val`
Similarly defines constant in ROM initialized to value `val`.

*4 In C, Strings are implicitly zero-terminated, this has to be explicitly specified in ASM

OPERAND ADDRESSING MODES 1/4

See `001-S12CPUV2- ReferenceManual.pdf`, p25ff

HCS12 is a two-address CPU, i.e. a CPU instruction can have up to two operands. One of the operands (destination operand) will be overwritten by the instruction result:

Register operands Instr. SRC, DEST

(Explicit) **Register Address**	`INST reg[, optional_reg2]` Registers are explicitly specified as operands. Rarely used, HCS12 prefers implicit registers	
Example:	`TFR D, X`	Copy value of register D to register X

Implicit **(Register) Address**	`INST` The operand (one of registers A, B, D, X, Y, SP) is implicitly used in the instruction mnemonic.	
Example:	`INX`	Increment the value of register X

Memory variable operand

Direct Address DIR 8bit, EXT 16bit Address	`INST address` The operand's memory address is part of the instruction. Programmers typically use variable names rather than addresses. The address is assigned by the linker from the compiler's / assembler's output.	

Example:	`LDD var1`
	`LDD $2000`
		Load D with the value of the variable `var1`.
		Load D with the value at memory address $2000.

OPERAND ADDRESSING MODES 2/4

Constant operands

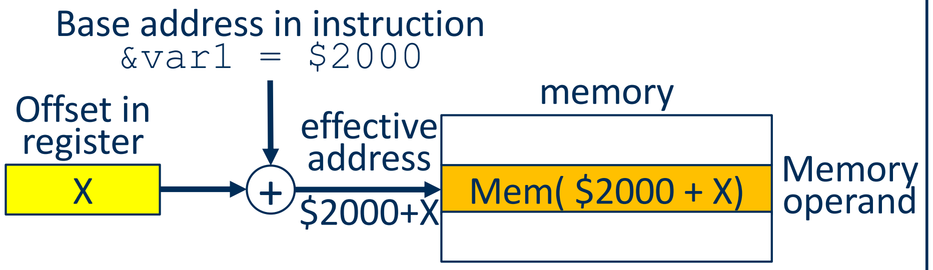
Immediate Operand	`INST #const` The operand is part of the instruction. Constants must be marked by `#...`, e.g. `#20`, `#-20`, `#$0A` or `#%10010110`	

Example:	`LDD #$b010`
	`LDD #var1`
		Load constant `0xb010` into register D
		Load D with the address of variable `var1`.

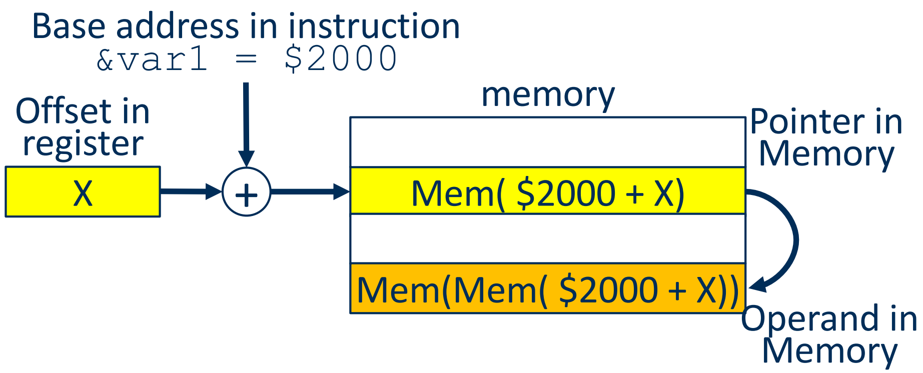
Indirect Address in various variants (Motorola / Freescale term: "Indexed")		
Register-indirect... Indexed IMM	`INST 0, reg`$_{X,Y,SP}$ Memory address in register X, Y, SP, i.e. register used as pointer.	
Example:	`LDD 0, X`	Load register D with the value at memory address stored in X (**indirect address**)
... with Pre- or Post-Increment or Decrement Auto Increment IDX	`INST const`$_{1,\ldots,+8}$`, {+|-}reg`$_{X,Y,SP}$ `INST const`$_{1,\ldots,+8}$`, reg`$_{X,Y,SP}$`{+|-}` The pointer in register X, Y or SP will be incremented or decremented by constant 1, ... or 8 before (pre) or after (post) using the pointer to address the operand	

Example:	`LDD 2, -X`
	`LDD 4, X+`
		Load memory value to which X points into register D, decrement X by 2 before
		..., increment X by 4 afterwards

OPERAND ADDRESSING MODES 3/4

… with index/offset Indexed IDX 5 bit constant IDX1 9 bit constant IDX2 16 bit constant	INST const, $\texttt{reg}_{X,Y,SP,PC}$ address=const + $\text{reg}_{X,Y,SP,PC}$ INST $\texttt{reg}_{A,B,D}$, $\texttt{reg}_{X,Y,SP,PC}$ address=$\text{reg}_{A,B,D}$ + $\text{reg}_{X,Y,SP,PC}$ The operand’s address is the sum of a constant plus register X, Y or SP or the sum of the two registers A, B or D plus X, Y or SP.	
Example:	LDY var1, X LDY D, X	Load Y with var1[X], i.e. the value at memory address var1+X (indexing array var1 by index X)  Base address in instruction &var1 = $2000 Offset in register X + effective address $2000+X memory Mem($2000 + X) Memory operand Load Y with contents of memory address D + X
Memory-indirect **… with index** Indexed-Indirect [IDX2]	INST [const, $\texttt{reg}_{X,Y,SP,PC}$] INST [D, $\texttt{reg}_{X,Y,SP,PC}$] The operand’s memory address is in a pointer in memory. This memory address will be addressed via another pointer, which is calculated as the value of register X, Y, SP or PC plus a constant or register D (Note: A, B not allowed here).	
Example:	LDY [D, X] new ASM syntax using []	Load Y with the value of the memory cell, to which the memory pointer points, to which D+X points.

OPERAND ADDRESSING MODES 4/4

Example:	LDY [var1, X]	Load Y with the memory cell to which a pointer in var1[X] points. (access via an array of pointer)  Base address in instruction &var1 = $2000 Offset in register X + memory Mem($2000 + X) Mem(Mem($2000 + X)) Pointer in Memory Operand in Memory

Branch instructions use so called **relative addressing** (Motorola/ Freescale-name REL). Relative addresses use the current value of the IP and add a constant offset, which is included in the instruction. The programmer need not care about details, but simply uses a label as the target of the branch:

```
start:    …
          BRA start
```

Unfortunately, normal branches limit the offset to 8 bit. However, there is a Long Branch version LBRA using a 16bit offset (see chapter 2.6).

For more information, please go to `CA3_AddressingModes.pdf`

Instruction Set 1
Freescale 68HC12 / HCS12

INSTRUCTION SET 1: DATA TRANSPORT

Different instructions:

- Data transport instr. move data from RAM (incl. stack)/ROM to register
- Arithmetic logic instruction
- Compare and branch instructions (including software interrupts)
- Miscellaneous instructions

Abbreviations:

$\texttt{reg}_{\texttt{A,B,D}}$…	One of the registers `A`, `B`, `D`, …
`mem`	Memory operand with arbitrary memory addressing (direct, indexed, indirect-indexed)
`imm`	Immediate operand
`mem_i`	Either `mem` or `imm`
`adr`	Code address relative to `PC`
`LD{AA|AB|..}`	Abbreviation for `LDAA`, `LDAB` or `LDS`
`8bit` or `16bit`	Used as index: Size of an operand

If not stated otherwise, all instructions do modify CCR status bits N, Z, V, C depending on the instruction’s result such that conditional branches may directly use the result without a preceding compare instructions.

DATA TRANSPORT INSTRUCTIONS 1/5

HOCHSCHULE
ESSLINGEN

Transport Instructions (Status bits N, Z, V, C are modified by LD… & ST… instructions only)

LD{AA|AB|D|X|Y|S} mem_i *2	mem_i → reg$_{A,B,D,X,Y,SP}$	**LoaD** register from memory A, B are loaded with an 8 bit, D, X, Y, SP are loaded with an 16 bit value
ST{AA|AB|D|X|Y|S} mem *2	reg$_{A,B,D,X,Y,SP}$ → mem	**ST**ore register to memory
TFR reg$_{A,B,D,X,Y,SP,CCR}$, reg_dest$_{A,B,D,X,Y,SP,CCR}$ *1	reg → reg	**TransFeR** register to register If the source reg is 8 bit and the dest. reg. is 16 bit, the MSB tales the sign of the 8 bit value (Sign Ex).
EXG reg$_{A,B,D,X,Y,SP,CCR}$, reg$_{A,B,D,X,Y,SP,CCR}$ *1	reg ↔ reg	**EX**chan**G**e register Swap register contents
TAB, TBA TSX, TSY, TXS, TYS TAP, TPA XGDX, XGDY	A → B and B→A SP→X,SP→Y,X→SP,Y→S A→CCR, CCR→A D ↔ X, X ↔ D	Variants of TFR and EXG (shorter opcodes)
MOVB mem_i, mem MOVW mem_i, mem *1	mem_i → mem 8 bit mem_i → mem 16 bit	**MOV**e Byte **MOV**e Word
SEX reg$_{A,B,CCR}$, reg$_{D,X,Y,SP}$ *1	reg$_{A,B,C}$→reg$_{D,X,Y,SP}$	**Sign EX**tension Copy from 8 to 16 bit for 2s-complement (same as TFR)

*1 These do not modify CCR bits N, Z, V, C.

*2 These do modify CCR bits N, Z, V, but not C!

DATA TRANSPORT INSTRUCTIONS 2/5

Calculate a pointer (indexed or indirect address = effective address)

`LEA{X|Y|S} mem` *1	Address of mem → $\text{reg}_{X,Y,SP}$	**L**oad **E**ffective memory **A**ddress into register Note: Calculation is done during runtime, not compile time

Stack (see chapter 2.6)

`PSH{A|B|C}` *1 `PSH{D|X|Y}` *1	SP-1 → SP, $\text{reg}_{A,B,CCR}$→Stack SP-2 → SP, $\text{reg}_{D,X,Y}$→Stack	**PuSH** register to stack Copy register on stack
`PUL{A|B|C}` *1 `PUL{D|X|Y}` *1	Stack→$\text{reg}_{A,B,CCR}$ SP+1→SP Stack→$\text{reg}_{D,X,Y}$ SP+2→SP	**PUL**l register from stack Copy from stack to register

*1 All instructions (except `PULC`) do not modify CCR bits `N`, `Z`, `V`, `C`.

Note: `PSH`… and `PUL`… may be substituted by `ST`… and `LD`…:

E.g.:

```
STAA 1, -SP  = PSHA
STD 2, -SP   = PSHD
LDAA 1, SP+  = PULA
LDD 2, SP+   = PULD
```

Modification of `SP` without actually copying data (required in ch.4):

```
LEAS n, -SP  Allocate stack space for n byte
LEAS n, +SP  Free n byte from Stack
```

DATA TRANSPORT INSTRUCTIONS 3/5

Example program 1 (CodeWarrior project AsmIntro.mcp)

```
.data:  SECTION    ; Globale Variable (nicht initialisiert) im RAM
var1:   ds.w 1     ;     short var1
var2:   ds.b 1     ;     char var2
var3:   ds.b 2     ;     char var3[2]

.const: SECTION    ; Globale Konstanten im ROM
const1: dc.b       $00, $11, $22, $33 ; const char const1[4]={0x00,…}

.init:  SECTION    ; Beginn der Code Section im ROM, main als externer
main:   …          ; Fkt.-name… Stack, Debugger init. Ints
    LDD    #$1234  ; D=0x1234 # = Zahl wird als Konst. Genutzt
                   ; Implizite Register Addressierung, immediate Adr.
    TFR    D, X    ; X=D; Addressierungsarten: 2x explizite Reg.-Addr.

    STD    var1    ; var1=0x1234; D implizite Reg.-Addr.; Direkte Adr.
    STAA   var2    ; var2=A=0x12;
    STD    var3    ; "var3=D" (Achtung Array); writes 2 byte big-end.
    LDD    const1  ; "D = const1"
    LDD    #const1 ; "D = &const1" ; loads the address
```

DATA TRANSPORT INSTRUCTIONS 4/5

Continued

```
                           ; D=&const1 (from previous instruction)

    LDD     #$0001         ; Hausaufgabe: Addressierungsarten
                             verstehen und was die Befehle machen
    LDX     D, Y           ;
    LDX     const1, Y      ;
    LDY     #const1        ;
    LDAA    1, Y+          ;
    LDAA    2, +Y          ;
    LDAA    1, -Y          ;
    LDAA    1, Y-          ;
    LDD     #const1        ;
    STD     var1           ;
    LDX     #0000          ;
    LDD     var1, X        ;
    LDD     [var1, X]      ;
```

DATA TRANSPORT INSTRUCTIONS 5/5

Continued

```
LDD     #$AAAA     ; D = 0xAAAA
LDX     #$5555     ; X = 0x5555
LDAA    #$7F       ; A = 0x7f
TFR     A, X       ;
LDAA    #$80       ; A = 0x80
TFR     A, X       ;
TFR     X, B       ;
MOVW    #$5678, var1;
MOVW    var1, var2 ;
LDX     #var3      ; X = &var3
MOVB    var1, 0,X  ;
MOVB    0,X,  1,X  ;
LDD     var1       ;
LDD     var1+1     ;s
```

INSTRUCTION SET 2: ARITHMETIC AND LOGIC OPS 1/3

Add, subtract, increment, decrement, invert sign

AB{A|X|Y} SBA	B+A→A, B+X→X, B+Y→Y A-B → A	**Add B** to …/**SuB**tract from **A, X or Y** (A, B are loaded with a 8 bit, D, X, Y, are loaded with a 16 bit value)
ADD{A|B|D} mem_i SUB{A|B|D} mem_i	reg$_{A,B,D}$+mem_i→reg$_{A,B,D}$ reg$_{A,B,D}$-mem_i→reg$_{A,B,D}$	**ADD** 8 bit ± 8 bit or 16 bit ± 16 bit **SUB**tract
ADC{A|B} mem_i$_{8bit}$ SBC{A|B} mem_i$_{8bit}$ (ADC, SBC not with D)	reg$_{A,B}$+mem_i+C→reg$_{A,B}$ reg$_{A,B}$–mem_i–C→reg$_{A,B}$	**AD**d with **C**arry-Bit (8 bit only) **S**u**B**tract with **C**arry-Bit (8 bit only)
INC mem$_{8bit}$ IN{CA|CB|X|Y|S} *1 DEC mem$_{8bit}$ DE{CA|CB|X|Y|S} *1 (INC, DEC not with D)	mem+1 → mem reg$_{A,B,X,Y,S}$+1→reg$_{A,B,X,Y,S}$ mem–1 → mem reg$_{A,B,X,Y,S}$–1→reg$_{A,B,X,Y,S}$	**INC**rement memory (8 bit only) **IN**crement register **DEC**rement memory (8 bit only) **DE**crement register
CLR mem$_{8bit}$ CLR{A|B} (CLR not with D)	0 → mem 0 → reg$_{A,B}$	**CL**ea**R** byte (Load with 0)
NEG mem8bit NEG{A|B} (NEG not with D)	–mem → mem –reg$_{A,B}$ → reg$_{A,B}$	**NEG**ate byte Multiply with -1 (invert sign), sets C=1, if A≠0 or B≠0, sets V=1, if A=$80 or B=$80!

*1 INS and DES do not modify the CCR bits N, Z, V and C.

INSTRUCTION SET 2: ARITHMETIC AND LOGIC OPS 2/3

Bitwise logical Operations

COM mem$_{8bit}$ COM{A|B}	/mem → mem /reg$_{A,B}$ → reg$_{A,B}$	**COM**plement (1’s complement, bitwise NOT)
AND{A|B} mem_i$_{8bit}$ ANDCC imm$_{8bit}$ ORA{A|B} mem_i$_{8bit}$ ORCC imm$_{8bit}$ EOR{A|B} mem_i$_{8bit}$	reg$_{A,B}$ AND mem_i→reg$_{A,B}$ CCR AND imm→CCR reg$_{A,B}$ OR mem_i →reg$_{A,B}$ CCR OR imm→CCR reg$_{A,B}$ XOR mem_i →reg$_{A,B}$	Bitwise **AND** Bitwise **OR** Bitwise **E**xclusive **OR**

Bit Operations

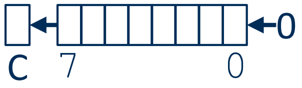
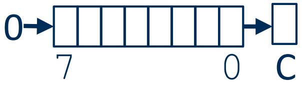
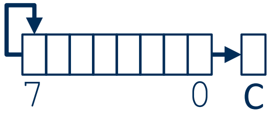
CLC, SEC CLV, SEV	0 → C, 1→ C 0 → V, 1 → V	**CL**ear / **SE**t **C**arry bit in CCR **CL**ear / **SE**t o**V**erflow bit in CCR
BCLR mem_i$_{8bit}$, imm BSET mem_i$_{8bit}$, imm	mem AND /imm→mem mem OR /imm→mem	**Bit CL**e**R** (8bit only) **Bit SET** (8bit only)

Multiply, Divide

MUL EMUL, EMULS	A x B → D unsigned D x Y → (Y, D) unsigned/signed	**MUL**tiply 8bit x 8bit → 16 bit 16bit x 16bit → 32 bit
IDIV, IDIVS EDIV, EDIVS FDIV	D / X → X Remainder in D (Y, D) / X → Y Re. in D Unsigned/signed D*2^{16} / X → X Re. in D	**DIV**ide 16bit x 16bit → 16 bit 32bit x 16bit → 16 bit “Pseudo 32 bit” / 16 bit → 16 bit

INSTRUCTION SET 2: ARITHMETIC AND LOGIC OPS 3/3

Shift and Rotate

`LSL mem`$_{8bit}$ `LSL{A|B|D}` (same as ASL)	mem << 1 → mem reg$_{A,B,D}$ << 1 → reg$_{A,B,D}$	**Logical Shift Left** Shift left by 1 bit for signed and unsigned values. MSB shifted to CCR C bit. LSB cleared to 0.
`ASL mem`$_{8bit}$ `ASL{A|B|D}` 	mem << 1 → mem reg$_{A,B,D}$ << 1 → reg$_{A,B,D}$	**Arithmetic Shift Left** Shift left by 1 bit for signed and unsigned values. MSB shifted to CCR C bit. LSB cleared to 0.
`LSR mem`$_{8bit}$ `LSR{A|B|D}` 	mem >> 1 → mem reg$_{A,B,D}$ >> 1 → reg$_{A,B,D}$	**Logical Shift Right** Shift right by 1 bit for unsigned values. LSB shifted to CCR C bit. MSB cleared to 0.
`ASR mem`$_{8bit}$ `ASR{A|B}`  (`ASR` not with D)	mem >> 1 → mem reg$_{A,B}$ >> 1 → reg$_{A,B}$ (`MSB`, i.e. sign is kept intact)	**Arithmetic Shift Right** Shift right by 1 bit for signed values. LSB shifted to CCR C bit. MSB (=sign value) is not changed.
`ROL mem`$_{8bit}$ `ROL{A|B}` (`ROL` not with D)	mem << 1 → mem+C reg$_{A,B}$ << 1 → reg$_{A,B}$+C	**ROtate Left** Rotate left by 1 bit. Carry Bit shifted to LSB, MSB shifted to Carry Bit.
`ROR mem`$_{8bit}$ `ROR{A|B}` (`ROR` not with D)	mem >> 1 → mem+C*8 reg$_{A,B}$ >> 1 → reg$_{A,B}$+C*8	**ROtate Right** Rotate right by 1 bit. Carry Bit shifted to MSB, LSB shifted to Carry Bit.

Multiplication by 2

Division by 2!

What could these shift operations be used for?

Combinations of e.g. `TFR A, B LSLA; LSLA; ADB;` may be faster than `MUL` by 5.
(… but is not on HCS12, e.g. `MUL` takes only 1 clock cycles, `EMUL` 3 cycles…)

ARITHMETIC AND LOGIC INSTRUCTIONS EXAMPLE 1/3

Example Program 1 (CodeWarrior project `AsmIntro2.mcp`)

C-Program	Equivalent Assembler-Program

`char a08 = 1, c08 = 3;`
`int  a16 = 1, b16 = 2, c16 = 3;`
`long a32 = 1, b32 = 2, c32 = 3;`
`unsigned char  cu08 = 3;`
`unsigned int   cu16 = 3;`
`void main(void) {`	

`  c16 = a16 + b16;  // Add 16 bit`	

`  c32 = a32 + b32;  // Add 32 bit`	`LDD     a32+2`
	`ADDD    b32+2`
	`STD     c32+2`
	`LDD     a32`
	`ADCB    b32+1`
	`ADCA    b32`
	`STD     c32`

`  c08 = (char) c16; // signed 16 → 8bit`	

`  cu08 = (unsigned char) cu16;`
`                 // unsigned 16 → 8bit`
	`LDAB    cu16+1`
	`STAB    cu08`

The 32bit operation is split into 2x 16 bit with carry using B and A registers.

ARITHMETIC AND LOGIC INSTRUCTIONS EXAMPLE 2/3

C-Program	Equivalent Assembler-Program
`c16 = c08;          // signed 8 → 16 bit`	
`cu16 = cu08;        // unsigned 8 → 16 bit`	
`cu16 = cu16 >> 2; // Shift right unsigned`	
`c16 = c16 >> 2;   // Shift right unsigned`	**LDD c16** **ASRA** **RORB** **ASRA** **RORB** **STD c16**
`c08 = c08 | 0x81; // Set bits 7 and 0 to 1`	
`a08 = a08 & ~0x81;// Set bits 7 and 0 to 0`	

ARITHMETIC AND LOGIC INSTRUCTIONS EXAMPLE 3/3

C-Program	Equivalent Assembler-Program
`c16 = a16 ^ b16;  // Bitwise Exclusive OR`	Please note: the EOR instruction is not available for 16 (or 32 bit…)
`c16 = a16 & b16;  // Bitwise AND`	Same: AND needs to be split into two instructions of 8 bit each…
`c16 = a16 && b16; // Logical AND`	see listing below

```
     LDD   a16; Load 16 bit a16 into D
     CPD   #0 ; Compare D against immediate 0
     BEQ   L1 ; if a16 zero branch to L1
     LDD   b16; Load 16 bit b16 into D
     CPD   #0 ; Compare D against immediate 0
     BNE   L2 ; if b16 not zero branch to L2
L1:  LDY   #0 ; FALSE case: Load Y with 0
     BRA   L3 ; and jump out
L2:  LDY   #1 ; TRUE case: Load Y with 1
L3:  STY   c16; Store the result of Y in c16
```

```
}
```

INSTRUCTION SET 3: COMPARE AND BRANCH 1/3

Compare and Test

`CBA` `CMP{A|B} mem_i`$_{8bit}$ `CP{D|X|Y|SP} mem_i`$_{16bit}$	Compute A – B Compute reg$_{A,B}$ – mem_i Com. reg$_{D,X,Y,SP}$ – mem_i	**Compare** Compare register with register, variables or constant. Sets bits in CCR.
`TST mem`$_{8bit}$ `TST{A|B}`	Compute mem – 0 Compute reg$_{A,B}$ – 0	**TeST** if operand is 0 or negative If Operand is 0 or negative, set bits in CCR.
`BIT{A|B} mem_i`$_{8bit}$	Com. reg$_{A,B}$ AND mem_i	**BIt Test** Like AND, but only sets the CCR bits.

Unconditional and conditional branches

(check, but do not change CCR bits N, Z, V, C)

`JMP mem`	mem → PC	**JuMP** like {L}BRA but can use indirect/indexed addressing
`{L}BRA adr`	adr → PC	**BR**anch **A**lways
`{L}BRN adr`	No Operation, NOP	**BR**anch **N**ever
`{L}BCC adr`	adr → PC, if C=0	**BR**anch if **C**arry **C**lear
`{L}BCS adr`	…, if C=1	**BR**anch if **C**arry **S**et
`{L}BNE adr`	…, if Z=0	**BR**anch if **N**ot **E**qual
`{L}BEQ adr`	…, if Z=1	**BR**anch if **EQ**ual
`{L}BPL adr`	…, if N=0	**BR**anch if **PL**us (positive)
`{L}BMI adr`	…, if N=1	**BR**anch if **MI**nus (negative)
`{L}BVC adr`	…, if V=0	**BR**anch if o**V**erflow **C**lear
`{L}BVS adr`	…, if V=1	**BR**anch if o**V**erflow **S**et

INSTRUCTION SET 3: COMPARE AND BRANCH 2/3

{L}BGT adr	adr → PC, if >	**BR**anch if **G**rea**T**er
{L}BGE adr	…, if >=	**BR**anch if **G**reater or **E**qual
{L}BEQ adr	…, if ==	**BR**anch if **EQ**ual
{L}BLE adr	…, if <=	**BR**anch if **L**ess or **E**qual
{L}BLT adr	…, if <	**BR**anch if **L**ess
		Use after compare/arith. ops of **signed** values.
{L}BHI adr	adr → PC, if >	**BR**anch if **HI**gher
{L}BHS adr	…, if >=	**BR**anch if **H**igh or **S**ame
{L}BEQ adr	…, if ==	**BR**anch if **EQ**ual
{L}BLS adr	…, if <=	**BR**anch if **L**ower or **S**ame
{L}BLO adr	…, if <	**BR**anch if **L**ower
		Use after a compare/arith. ops of **unsigned** values.
BRCLR mem$_{8bit}$, imm, adr	adr→PC, if mem & imm=0	**BR**anch if bits are **CL**ea**R**ed
BRSET mem$_{8bit}$, imm, adr	adr→PC, if /mem & imm=0	**BR**anch if bits are SET

All conditional branches check the status bits in the CCR register, which have been set by a previous operation, typically a compare.

The `adr` here is a code memory address, which the programmer most often specifies via a label. The instruction uses relative addressing. Normal branches with 8 bit offset can only jump -128..127 Bytes from the current instruction pointer location. If You jump over a longer distance, use the long branch instruction {L} variant, which use 16 bit offsets and thus may reach any HCS12 address.

INSTRUCTION SET 3: COMPARE AND BRANCH 3/3

Loop Instructions

(These instructions do not modify CCR bits N, Z, V, C)

IBEQ $reg_{A,B,D,X,Y,SP}$, adr DBEQ $reg_{A,B,D,X,Y,SP}$, adr IBNE $reg_{A,B,D,X,Y,SP}$, adr DBNE $reg_{A,B,D,X,Y,SP}$, adr	$reg_{A,B,D,X,Y,SP}$ ±1 → $reg_{A,...}$ adr → PC, if $reg_{A,...}$ = 0 adr → PC, if $reg_{A,...}$!= 0	**I**ncrement/**D**ecrement register and ... **B**ranch if **EQ**ual to 0 ... **B**ranch if **N**ot **E**qual to 0
TBEQ $reg_{A,B,D,X,Y,SP}$, adr TBNE $reg_{A,B,D,X,Y,SP}$, adr	adr → PC, if $reg_{A,...}$ = 0 if $reg_{A,...}$!= 0	**T**est register and **B**ranch if ...

COMPARE AND BRANCH INSTRUCTIONS EXAMPLE 1/3

Example Program 3 (CodeWarrior project `AsmIntro2.mcp`)

C-Program	Equivalent Assembler-Program

```
if (c16 <= 32)         // if - else


{ a08 = 4;
   …

} else {
  a08 = 8;
  …
}
…
if (cu16 <= 32)       // if - else



{…
}
…

for (;;);             // endless loop
```

```
  LDD    c16
  CPD    #32        ; Compare with 32
  BLE    L1         ; Branch if Lower/Eq
  MOVB   #4, a08
  …
  BRA    L2
L1:
  MOVB   #8, a08
  …

L2: …
  LDD    cu16
  CPD    #32       ; Compare with 32
  BGT    L3
  …

L3: …

  BRA *+0
```

COMPARE AND BRANCH INSTRUCTIONS EXAMPLE 2/3

C-Program

```
for (c08 = 0; c08 < 3; c08++) {// for

    c16 = c16 + a16;

}
```

```
while (c08 <= 32) {        // while-do
    a16++;
}
```

```
do { … } while (c08 <= 32);// do-while
```

Equivalent Assembler-Program

```
     CLR  c08
     BRA  L4
L0:  LDD  c16
     ADDD a16
     STD  c16

L1:  INC  c08
L4:  LDAB c08
     CMPB #3
     BLT  L0

     BRA  L3
L2:  LDX  a16
     INX
     STX  a16
L3:  LDAB c08
     CMPB #32
     BLE  L2
```

COMPARE AND BRANCH INSTRUCTIONS EXAMPLE 3/3

C-Program

```
enum { NONE, ONE, TWO } eVal;
…
switch (eVal)         // Switch-Case






{  case NONE: …
              break;
   case ONE:  …
              break;
   case TWO:  …
              break;
}
```

Equivalent Assembler-Program

```
NONE:    EQU  0    ; Values of the
ONE:     EQU  1    ; enumeration
TWO:     EQU  2
eVal:    DS.W 1    ; Enumeration
         …
switch:  LDD eVal ; Compute
         LSLD     ; index into
         TFR D, X ; branch table
         JMP [swK,X]
swK:     DC.W     caseNONE ; Branch
         DC.W     caseONE  ; table
         DC.W     caseTWO

caseNONE:    …
         BRA  endCase
caseONE:     …
         BRA  endCase
caseTWO:     …
         BRA endCase
endCase:
```

INSTRUCTION SET 4: SUBROUTINE CALLS & STATE 1/2

Subroutine Calls

(Do not change CCR bits N, Z, V, C)

`JSR mem`	mem → PC Saves return address on stack	**Jump to SubRoutine** Like BSR, but can use indirect/indexed destination address
`{L}BRS adr`	adr → PC Saves return address on stack	**Branch to SubRoutine** Like JSR, but only relative addresses (shorter opcode than JSR).
`RTS`	Restores the return address from stack	**ReTurn from Subroutine**
`CALL, RTC`	Subroutine call and return for memory sizes > 64kB.	

Interrupts (see Chapter 3)

(Do not change CCR bits N, Z, V, C)

Interrupt = subroutine, which will be called by a hardware event. An interrupt will store registers on the stack.

`RTI`	Restores regs from stack.	**ReTurn from Interrupt** (don't use in terrupts)
`CLI`	0 → I Debugger shows `ANDCC #$EF`	**CLear Interrupt mask** Global interrupt enable.
`SEI`	1 → I Debugger shows `ORCC #$10`	**SEt Interrupt mask** Global interrupt disable.
`SWI`	Store return address and register set X, Y, D, CCR on stack, not maskable, does disable interrupts I=1.	**SoftWare Interrupt** Call the SWI interrupt Service Routine (used by debug monitor)

INSTRUCTION SET 4: SUBROUTINE CALLS & STATE 2/2

`TRAP`	Like SWI	**TRAP** for unimplemented opcodes Call the TRAP Interrupt Service Routine

Miscellaneous Operations

`NOP`	--	**No OP**eration
`WAI, STOP`	**WAI**t and **STOP** Energy saving mode: Turn CPU off without/with all on-chip peripherals. Operation will be resumed via an interrupt. Should not be used when testing a program with the HCS12 debugger.	

`MEM, REV, EMIN…, EMAX…,`
`MIN…, MAX…, ETBL, TBL, …`
	Instructions to implement Fuzzy Logic minimum and maximum operations and data table access. See CPU.	

SUBROUTINE CALLS & CPU STATE EXAMPLE 1/3

Example Program 4 (CodeWarrior project `AsmIntro2.mcp`)

C-Program	Equivalent Assembler-Program
`int betrag(int x) {`	`betrag:`
`  return x > 0 ? x : -x;`	`        CPD  #0`
	`        BGT  L0`
	`        COMA`
	`        COMB`
	`        ADDD #1`
`}`	`L0:     RTS`
`void main(void) {`	`main:   …`
`  …`	`        LDD  a16`
`  c16 = betrag(a16);`	`        JSR  betrag`
`  …`	`        STD  c16`
`}`	

Simplest way to pass parameters: Parameters in register(s)
Return value(s) in register(s)

Functions with many parameters: Pass parameters via Stack, see Ch. 4

Note: Depending on configuration, the C-compiler may optimize the code and thus the assembler code generated will look different from the code shown here. For example programs, optimization was turned off.

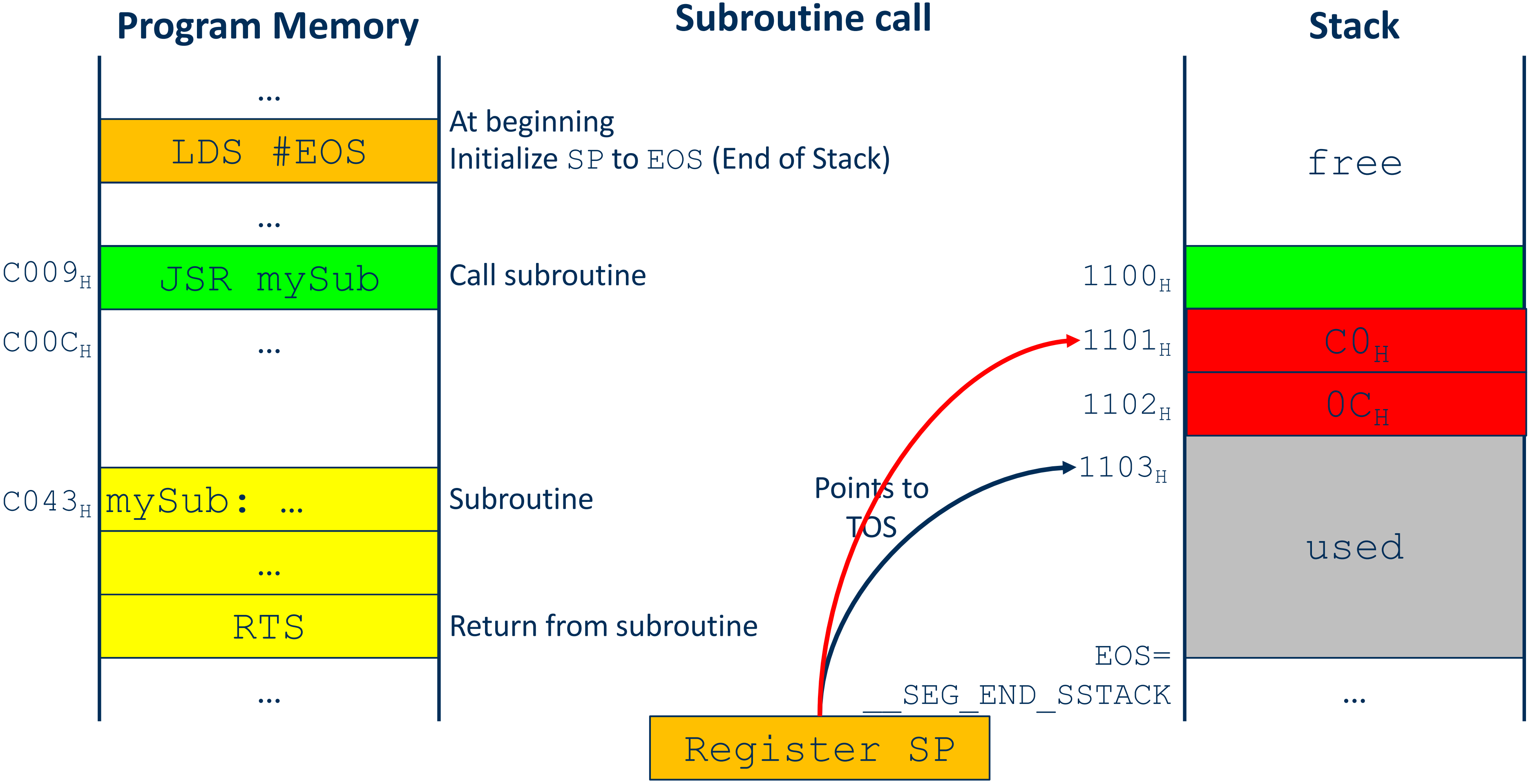
Stack

STACK 1/3

Purpose: RAM memory area to temporarily save registers, return addresses & local vars.

Idea: Last-In-First-Out (LIFO) memory, filled from the end (End of Stack, EOS).
Read/write access with register-indirect addressing via the stack pointer SP.
SP points to the last byte pushed onto the stack (Top of Stack, TOS).

Example:
Subroutine call



STACK 2/3

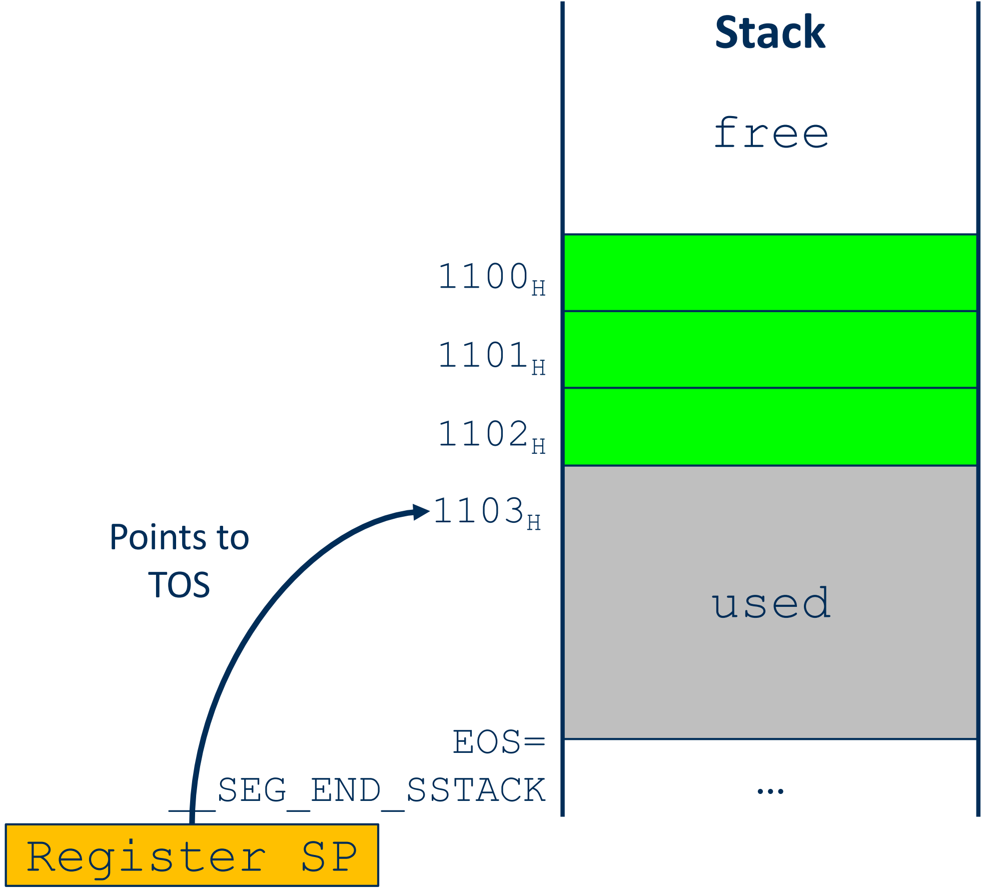
Example: Save registers to Stack (CodeWarrior project AsmIntro.mcp)

Program Memory

```
    LDS #EOS
    …
1:  LDD #$1122
2:  LDX #$3344
3:  LDY #$5566
4:  PSHX
5:  PSHA
6:  PSHB
    …
7:  PULD
    …
8:  PULX
9:  LDY -2, SP
```

Register Content

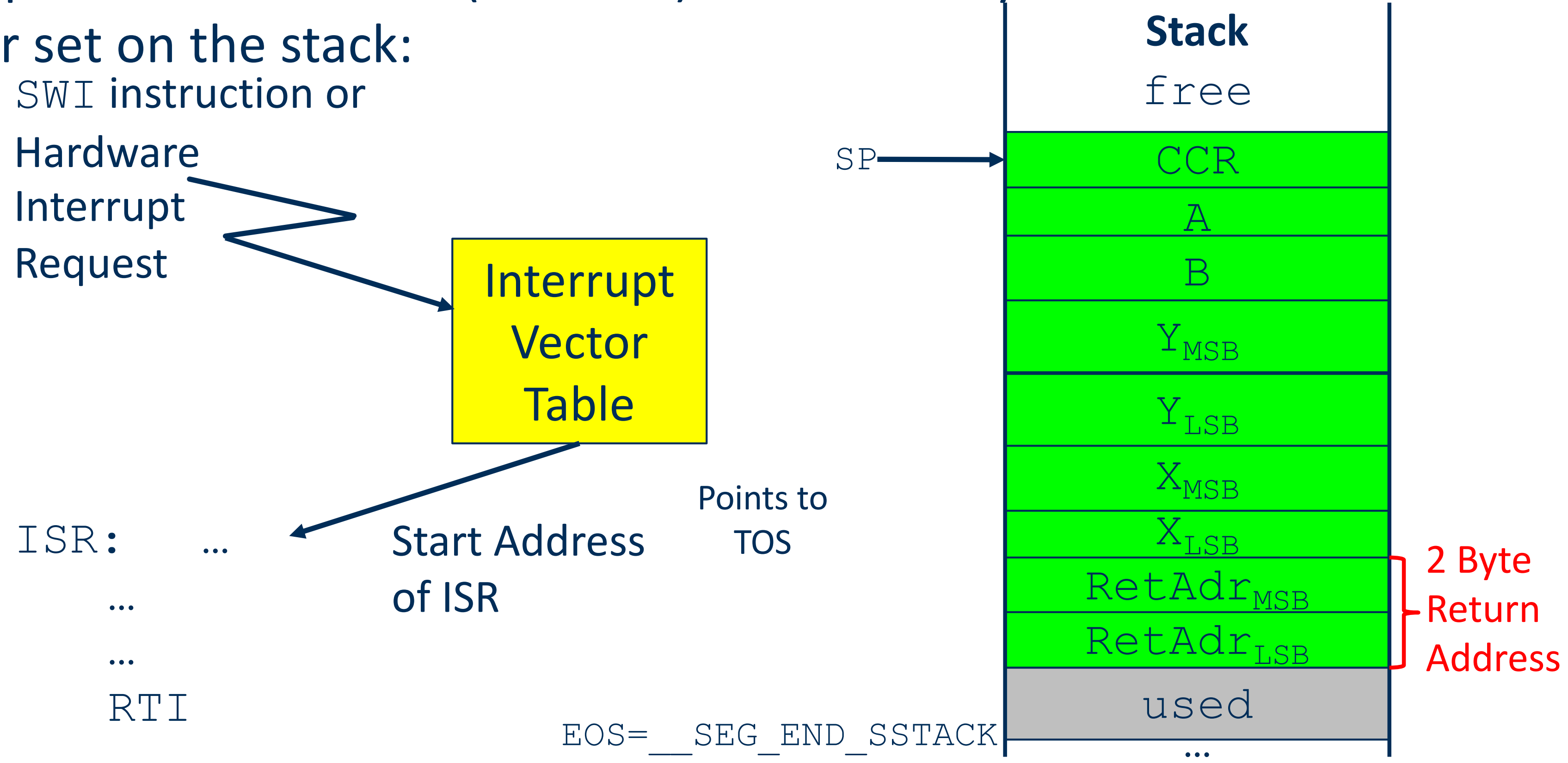
D		X	Y
A	B		



- Allocate stack in RAM, automatically done by Linker [*1]
- Initialize SP at beginning of program: `LDS #__SEG_END_SSTACK` [*1]
- Stack pointer managed (inc/dec) automatically by hardware (push/pull/rts)
- Number of bytes stored on stack and retrieved from stack **must be** balanced

STACK 3/3

Interrupt Service Routines (See ch.3) automatically save & restore register set on the stack:



SWI instruction or Hardware Interrupt Request

Interrupt Vector Table

ISR: …
…
…
RTI

Start Address of ISR

Points to TOS

Stack

free

SP

CCR

A

B

Y_{MSB}

Y_{LSB}

X_{MSB}

X_{LSB}

$RetAdr_{MSB}$

$RetAdr_{LSB}$

2 Byte Return Address

used

EOS=__SEG_END_SSTACK

…

*1 The Codewarrier HCS12 development tools do define the stack size in linker control files `Simulater_Linker.prm` and `Monitor_Linker.prm`. The default size is `STACKSIZE` 0x100 (256 Bytes). The linker provides a symbol `__SEG_END_SSTACK`, which points to the end of the stack. Assembler programs use this to initialize the stack pointer `SP`:

```
        ; Import symbols
        XREF __SEG_END_SSTACK     ; End of Stack
        ; Beginning of program code
main:   LDS #__SEG_END_SSTACK     ; Initialize stack pointer
```

INSTRUCTION SIZE AND EXECUTION SPEED 1/3

- Instruction Size (Opcode length)
  HCS12 opcodes are 1 or 2 Byte long plus a variable number of bytes for a direct operand address or an immediate operand or an operand index. Constants are stored as 5, 9 or 11 bit values when possible, to save memory space. Total instruction length is 1 to 6 byte.
- Execution Time (Instruction clock cycles)
  The number of clock cycles required to execute an instruction depends on the length of the instruction (cycles to read the instruction from memory), the location of the operands and result (read/write registers or memory) plus the actual execution of the operation. Reading/writing 2 bytes from a register or internal ROM/RAM memory typically takes 1 CPU clock cycle. See next page for examples.
- Detailed info can be found in literature reference [3.1, chapter 6.7 and appendix A], but is hard to read, because there are many dependencies. An easy way to find out is as follows:
  - The size of an instruction (including operands) can be seen in the Disassembly listing of the IDE's source code editor (right click to open the listing) or in the Disassembly-Window of the debugger. The total size of a program can be found in the Linker/Locator's Map file.
  - The execution time of an instruction or program can be "measured" in the HCS12 simulator (debugger in simulation mode), see CPU Cycle display in the debugger's register window).

INSTRUCTION SIZE AND EXECUTION SPEED 2/3

Rules of Thumb:

… for Instruction Size: Opcode + Operand Address Info

• Opcode length for most instructions:	1 byte	(`MOVB`, `MOVW`, `TFR`: 2 byte)
• Direct address operand:	2 byte	(if address ≥ 2 byte)
• Immediate operand or index/offset constant:	1 / 2 byte	(if constant ≥ 2 byte)
• Implicit register address:	0	(included in opcode)

… for Execution Time: Fetch Instructions+Fetch Operand(s) +Execute Operations+Store Result

• Read/Write memory access: (instruction or operand)	1 cycle per 2 byte	
• Register operand access:	0	(included in execute operation)
• Calculate pointers register-indirect:	1 cycle	
memory-indirect:	2 cycles	(includes read pointer from memory)
• Execute arithmetic logic instruction:	1 cycle	

INSTRUCTION SIZE AND EXECUTION SPEED 3/3

- Instruction size & speed depend on type and operand addressing mode. E.g.

Address mode *1 Operands		Instruction	Length in byte	Speed in CPU clock cycles *2
Source Operand	Destination			
Immediate (IMM)	Register	LDD #1234	3	2
Register indirect (IDX)	Register	LDD 0, X	2	3
Register indirect with increment	Register	LDD 2, X+	2	3
Memory direct (EXT)	Register	LDD var1	3	3
Register indirect with index (IDX2)	Register	LDD var1, X	4	4
Memory indirect with index ([IDX2])	Register	LDD [var1, X]	4	6
Register	Register	TFR D, X	2	1
Register indirect	Register indirect	MOVW 0, X, 0, Y	4	5
Memory direct	Memory direct	MOVW var1, var2	6	6
Direct		JMP address	3	3
Direct		JSR address	3	4
		JSR [address]	3	7
Implicit		RTS	1	5
Register implicit		INX	1	1
Memory direct	Register implicit	ADDD var1 (16+16bit)	3	3
Register implicit	Register implicit	EMUL (16x16bit)	1	3
Register implicit	Register implicit	EDIV (32/16 bit)	1	12

*1 var1, var2 … 16 bit variables in internal ROM

*2 CPU clock period 42 ns @ f_{BUSCLK} = 24 MHz

LECTURE: LITERATURE

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