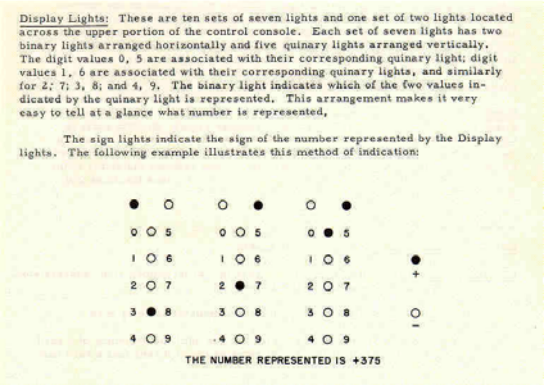
Embedded system programming: HCS12

- Cross developments environments and tools
- XCC12 cross C compiler in particular
- Low level programming in ’C’

Low level programming in C

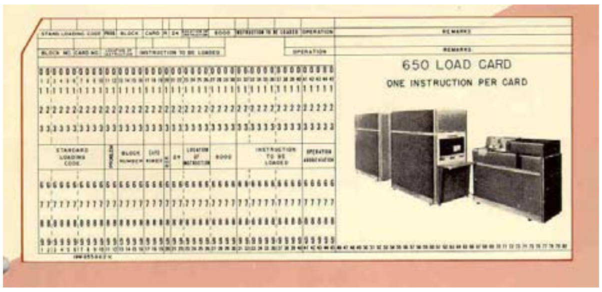
- Short history
- Cross development
- Software libraries
- Embedded assembly code
- Interrupts

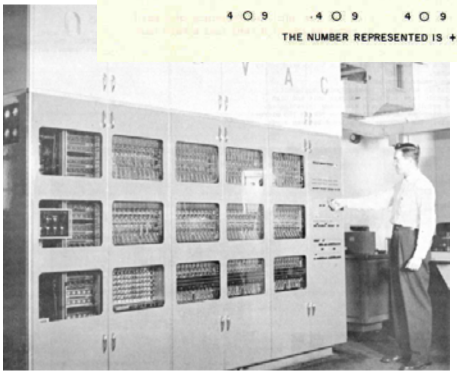


Display Lights: These are ten sets of seven lights and one set of two lights located across the upper portion of the control console. Each set of seven lights has two binary lights arranged horizontally and five quinary lights arranged vertically. The digit values 0, 5 are associated with their corresponding quinary light; digit values 1, 6 are associated with their corresponding quinary lights, and similarly for 2; 7; 3, 8; and 4, 9. The binary light indicates which of the two values indicated by the quinary light is represented. This arrangement makes it very easy to tell at a glance what number is represented,

The sign lights indicate the sign of the number represented by the Display lights. The following example illustrates this method of indication:

THE NUMBER REPRESENTED IS +375





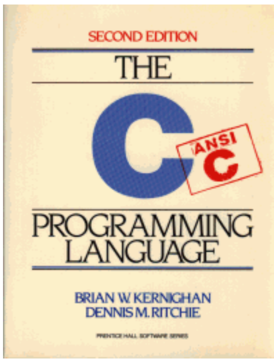
The program language ’C’

”The philosophy of BCPL is not one of the tyrant who thinks he knows best and lays down the law on what is and what is not allowed; rather, BCPL acts more as a servant offering his services to the best of his ability without complaint, ***even when confronted with apparent nonsense****.* ***The programmer is always assumed to know what he is doing and is not hemmed in by petty restrictions.”***

Martin Richards

Brian Kernighan

Dennis Ritchie



’ BCPL’ Basic Combined Programming Language – (Martin Richards) 1966

’B’ - (Johnson/Kernighan) 1973

’C’ – (Kernighan/Ritchie) 1978

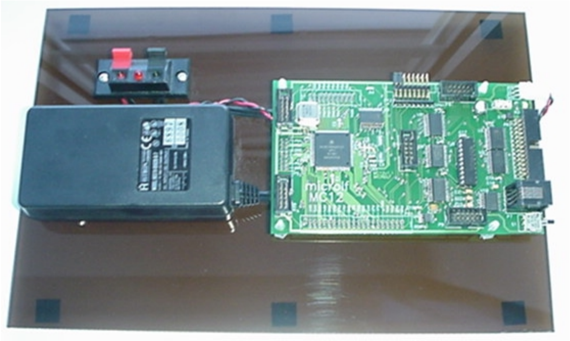
’ANSI C’ – 1983, first standardisation

’C++’ – (Stroustrup) 1986

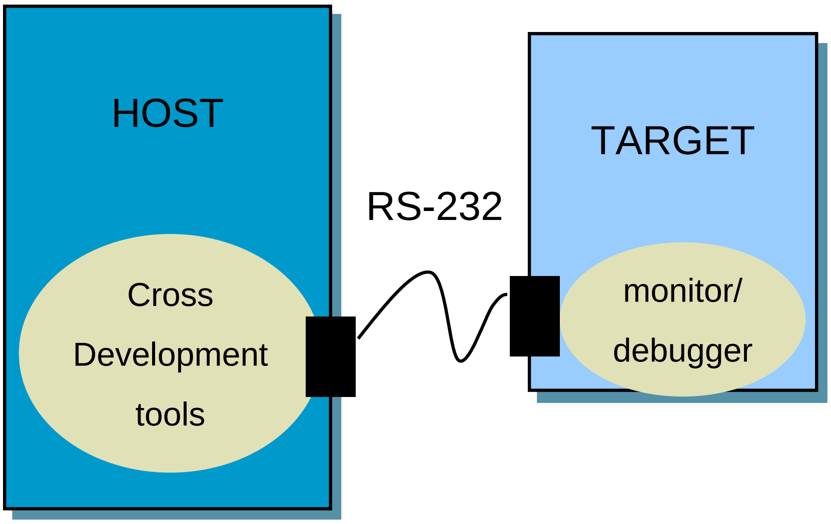
’ISO’ – 1995, 1999

Cross development

- Development for one type of computer (target computer) with another type of computer (host computer)
- Tools:
  - Cross assembler
  - Cross compiler
  - …

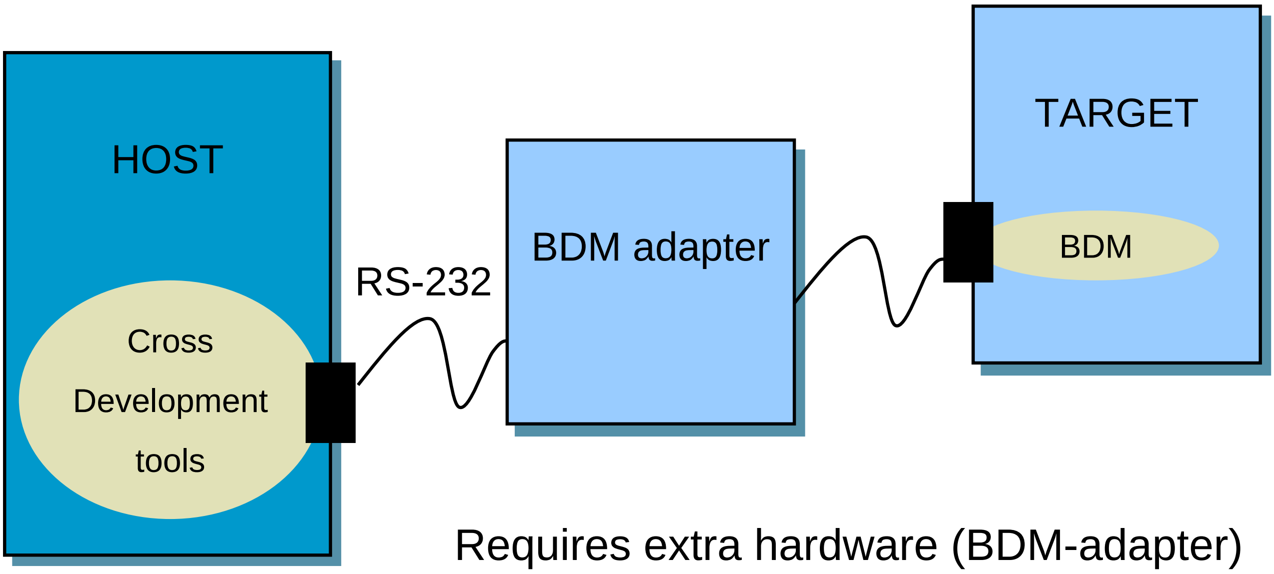


Cross development environment



Requires a resident software debugger in the target

Cross development environment



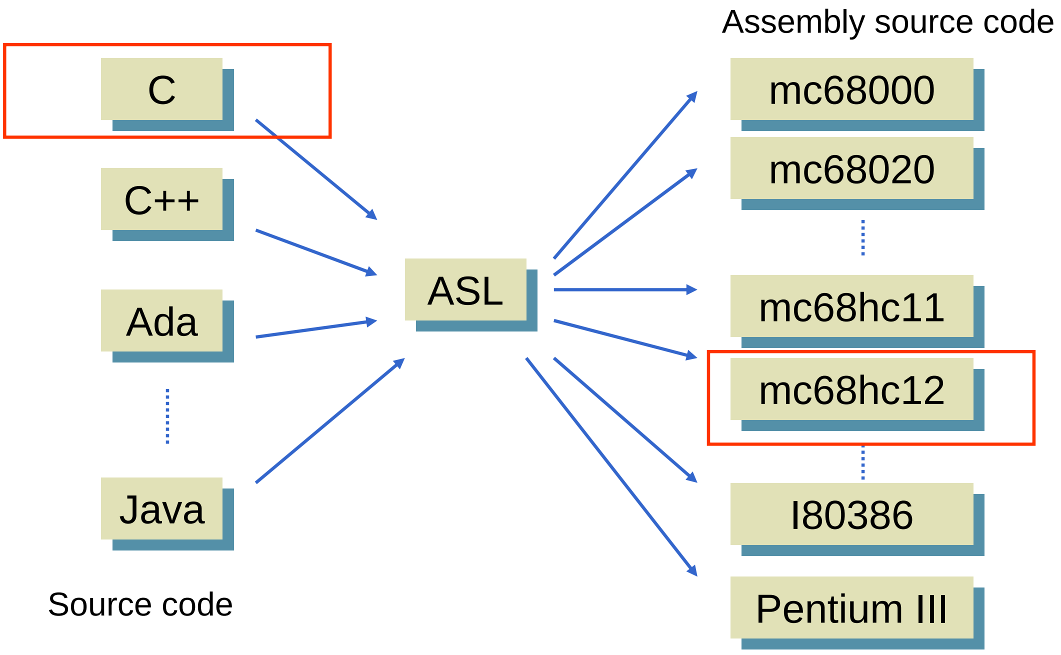
Requires extra hardware (BDM-adapter)

Cross development tools

Host system

- Cross compilers (C/C++/Java/Ada/Fortran...)
- Cross assemblers
- Linker
- Terminal emulation, download
- Simulators
- Debug adapters and software

The compiler



Assembly source code

C

C++

Ada

Java

Source code

ASL

mc68000

mc68020

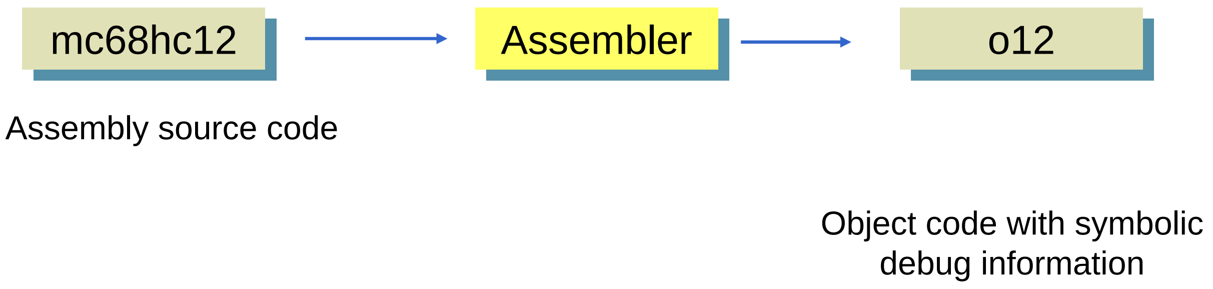
mc68hc11

mc68hc12

I80386

Pentium III

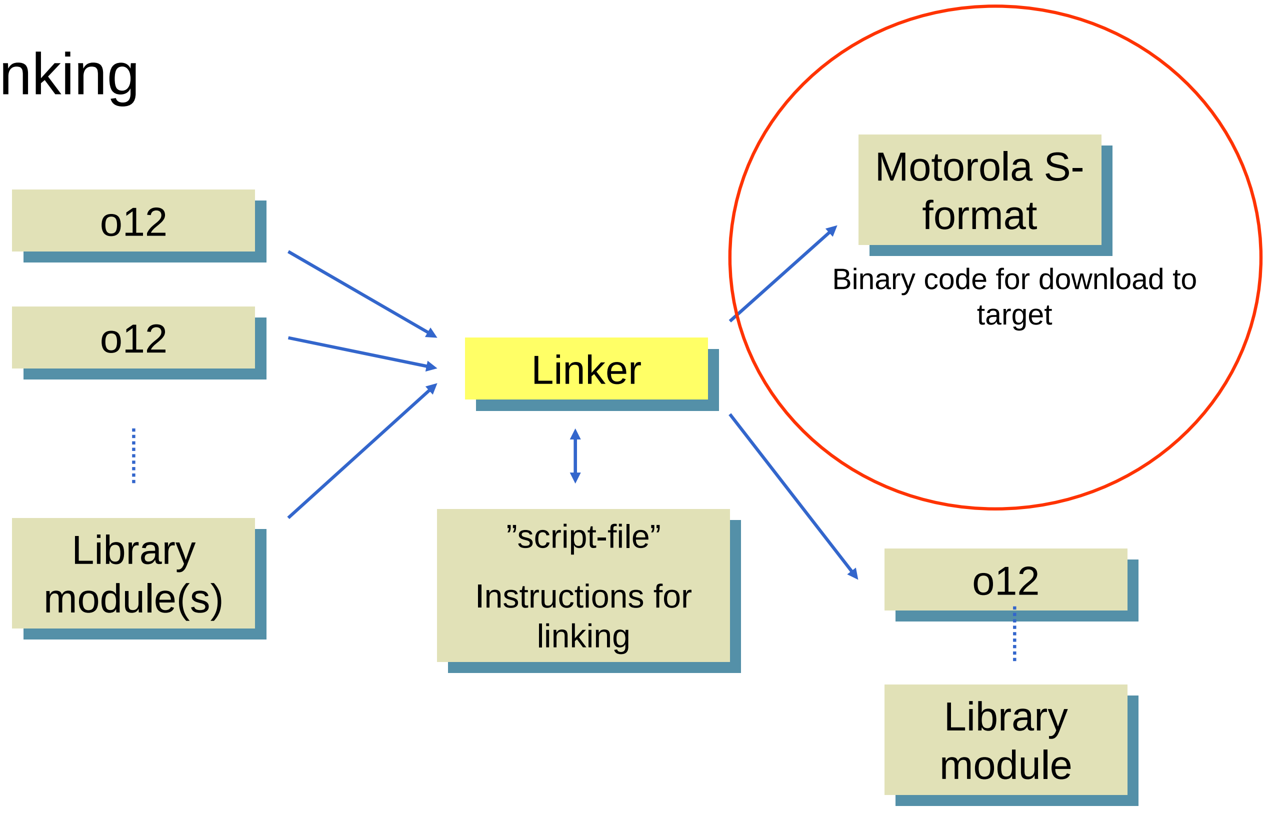
The assembler



Assembly source code

Object code with symbolic debug information

Linking

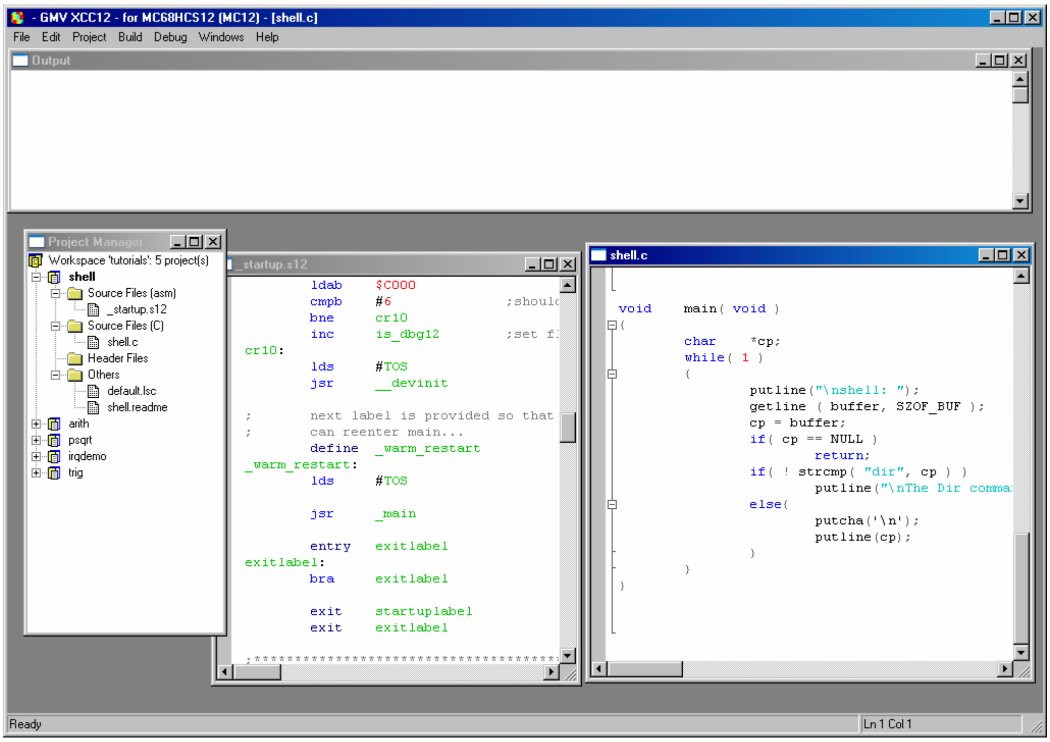


Binary code for download to target

”script-file”

Instructions for linking

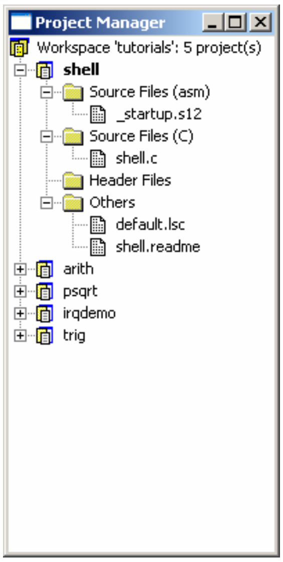
Software development with XCC12



```
        ldab    $C000
        cmpb    #6              ;shoulc
        bne     cr10
        inc     is_dbg12        ;set f:
cr10:
        lds     #TOS
        jsr     __devinit

;       next label is provided so that
;       can reenter main...
        define  _warm_restart
_warm_restart:
        lds     #TOS

        jsr     _main

        entry   exitlabel
exitlabel:
        bra     exitlabel

        exit    startuplabel
        exit    exitlabel

;**************************************
```

```
void    main( void )
{
        char    *cp;
        while( 1 )
        {
                putline("\nshell: ");
                getline ( buffer, SZOF_BUF );
                cp = buffer;
                if( cp == NULL )
                        return;
                if( ! strcmp( "dir", cp ) )
                        putline("\nThe Dir comma:
                else{
                        putcha('\n');
                        putline(cp);
                }
        }
}
```

XCC12 ’Project Manager’



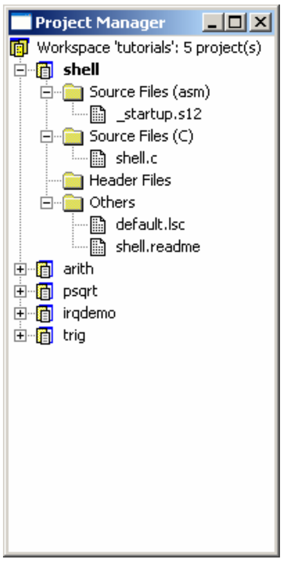
’Project’

All source code files belonging to an application. Result is an executable program

’Workspace’

A practical way of grouping related projects.

XCC12 ’Application’



Project Manager

```
Workspace 'tutorials': 5 project(s)
shell
    Source Files (asm)
        _startup.s12
    Source Files (C)
        shell.c
    Header Files
    Others
        default.lsc
        shell.readme
arith
psqrt
irqdemo
trig
```

Any application requires a startup procedure. The procedure can often be standardised, i.e. several applications use the same startup.

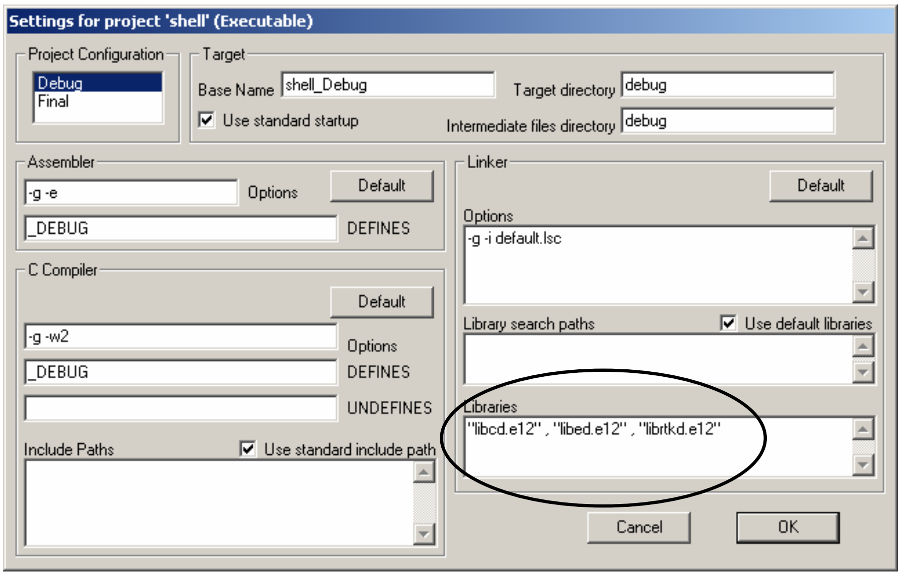
XCC12 Compiler libraries

Performs standard operations wich cannot be handled by a single instruction in CPU12. For example addition of 32-bit numbers.

```
* long int la,lb,lc;
*   lc = la + lb;
  ldd 2+_lb
  ldx _lb
  pshd
  pshx
  ldd 2+_la
  ldx _la
  pshd
  pshx
  jsr add32
  leas 8,sp
  std 2+_lc
  stx _lc
```

’add32’ a function in the precompiled library

XCC12 Standard libraries



Settings for project 'shell' (Executable)

Project Configuration: Debug (selected), Final

Target
Base Name: shell_Debug
Target directory: debug
☑ Use standard startup
Intermediate files directory: debug

Assembler
Options: -g -e
DEFINES: _DEBUG

C Compiler
Options: -g -w2
DEFINES: _DEBUG
UNDEFINES:
Include Paths
☑ Use standard include path

Linker
Options: -g -i default.lsc
Library search paths
☑ Use default libraries
Libraries: "libcd.e12" , "libed.e12" , "librtkd.e12"

Three different libraries with common functions

CC12 ’segment’

The compiler distinguishes the generated code by placing it in one of four possible *segments*.

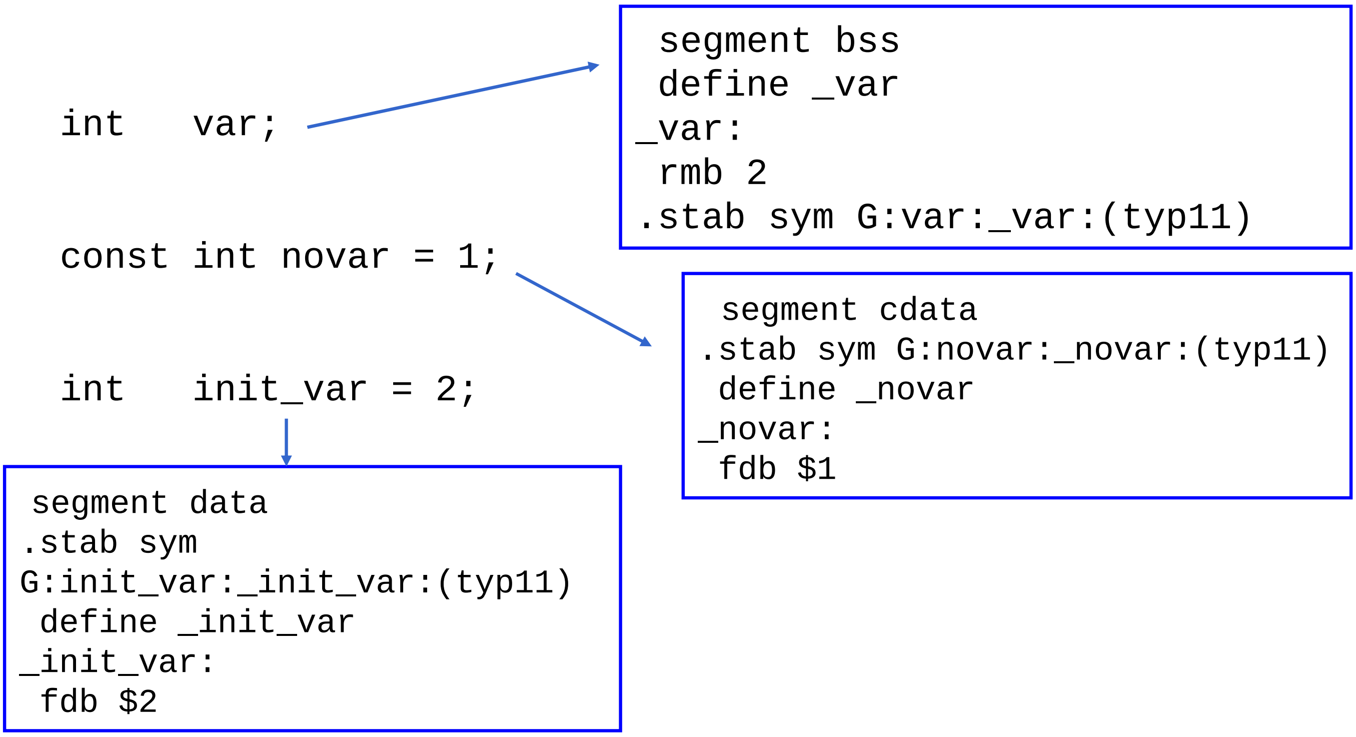
text – this is the segment for machine instructions (executable code)

data – this is the segment for initialised variables. These variables have user defined values when the program starts but they may be changed by the program.

cdata - this segment is also for initialised variables. These variables have user defined values when the program starts and they can *not* be changed by the program.

bss – this segment provides space for variables which don’t have initialised values

CC12 ’segment’ - EXAMPLE



```
int   var;
```

```
 segment bss
 define _var
_var:
 rmb 2
.stab sym G:var:_var:(typ11)
```

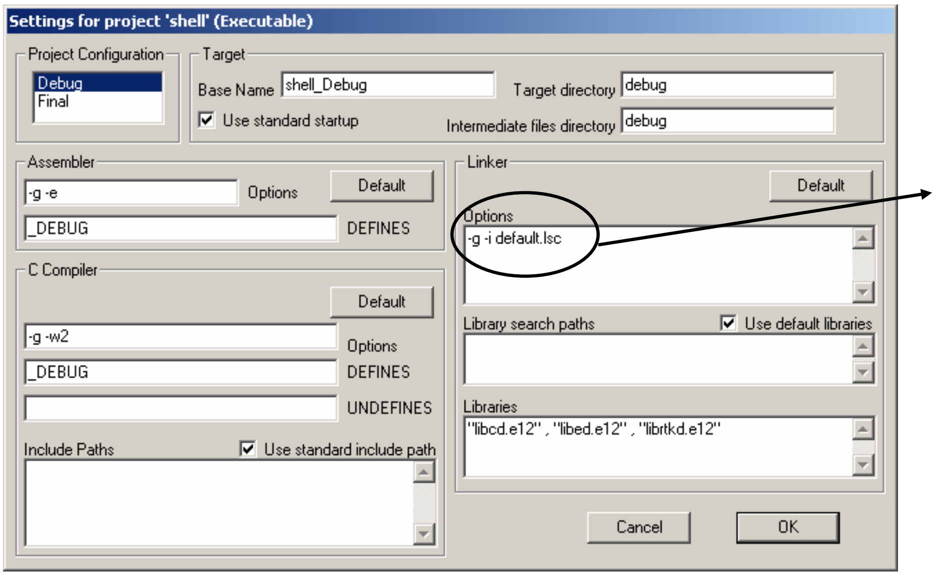
```
const int novar = 1;
```

```
 segment cdata
.stab sym G:novar:_novar:(typ11)
 define _novar
_novar:
 fdb $1
```

```
int   init_var = 2;
```

```
 segment data
.stab sym
G:init_var:_init_var:(typ11)
 define _init_var
_init_var:
 fdb $2
```

XCC12 linker instructions



Settings for project 'shell' (Executable)

Project Configuration: Debug, Final

Target
Base Name: shell_Debug
Target directory: debug
☑ Use standard startup
Intermediate files directory: debug

Assembler
Options: -g -e
DEFINES: _DEBUG

C Compiler
Options: -g -w2
DEFINES: _DEBUG
UNDEFINES:
Include Paths
☑ Use standard include path

Linker
Options: -g -i default.lsc
Library search paths
☑ Use default libraries
Libraries: "libcd.e12" , "libed.e12" , "librtkd.e12"

```
//
//   default.lsc
//   script for QLD
//   for XCC12 applications in RWM
//
...
```

XCC12 ’default.lsc’

```
//        OPTIONS SECTION
          -M        // generate listfile <basename>.map

//        define program entry for debugger
          entry( __start )

          group  ( c , const_group )
          {
                    abs
          }
          group( r , test_group)
          {
                    startupseg,
                    text,
                    cdata,
                    data,
                    bss
          }
          group( r, interrupt_vectors )
          {
                    vectors
          }
          layout
          {
                    0x1000,0x3C80 <= test_group,
                    0x3F80,0x3FFF <= interrupt_vectors
          }
```

XCC12 Embedded assembly code

Operations, which cannot be accomplished with ’C’-syntax, requires ”embedded assembly code”:

```
/* EXAMPLE */
void    main( void )
{
        __asm( " andcc #$F0"); /* clear condition flags */
}
```

NOTE: This is not a part of the ’C’-language and may be compiler dependent.

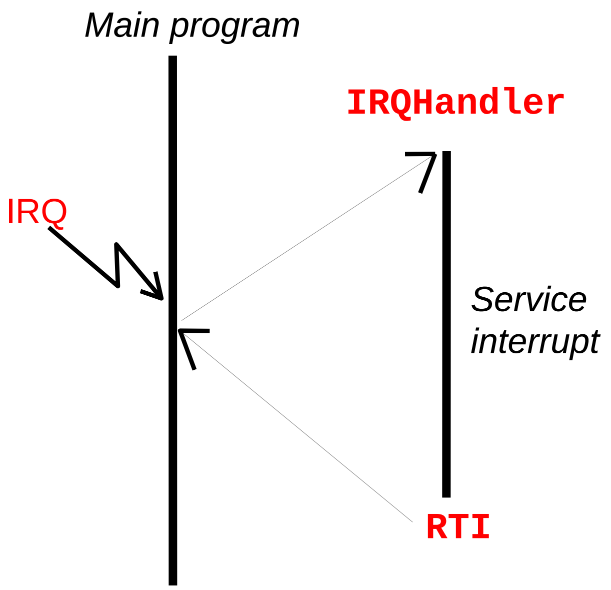
XCC12 Embedded assembly code

Variables and parameters can be referenced by name in embedded assembly code.

```
void callfunc( int aa , int ab )
{
   aa = 1;
   ab = 2;
}
```

```
void callfunc( int aa , int ab )
{
        __asm(" movw #1,%a", aa);
        __asm(" movw #2,%a", ab);
}
```

Remapping interrupt handlers



ROM

Address (hex)	Function
FFFE	RESET, Startvector
FFFC	Clock Monitor Fail, JMP [3FFE]
FFFA	COP Watchdog Timeout, JMP [3FFC]
FFF8	Illegal Op Code, JMP [3FFA]
FFF6	SWI, JMP [3FF8]
FFF4	XIRQ, JMP [3FF4]
FFF2	IRQ, JMP [3FF2]
FF8C FFF0	... JMP [3Fxx]

RWM

Address (hex)	Funktion
3FFE	Not used
3FFC	ClockFailHandler
3FFA	COPFailHandler
3FF8	IllOpHandler
3FF6	SWIHandler
3FF4	XIRQHandler
3FF2	IRQHandler
3F8C 3FF0	...

Specification of interrupt handler

```
__interrupt void name( void );
```

Keyword ”__interrupt” is the first word in the specification.

An interrupt handler can neither have parameters nor return values.

Interrupt vector table in XCC12

```
#pragma  DATA     vectors
/* 3F80 - 3FFF */
__interrupt void(*irqvecs[])() =
{
   .....
   PWMEShutdownHandler,
   PortPIntHandler,
   TimerCh1Handler,
   TimerCh0Handler,
   RTIHandler,
   IRQHandler,
   XIRQHandler,
   SWIHandler,
   IllopHandler,
   COPFailHandler,
   ClockFailHandler,
   ResetHandler
};
```

script file for the application

```
.....

group( r, interrupt_vectors )
        {
                vectors
        }
.....
layout
        {
        ....,
        0x3F80,0x3FFF <= interrupt_vectors
        }
```

Interrupt vector table, static initialisation

declarations

```
__interrupt void IRQHandler( void )
{
        /* Interrupt service routine */
}


#pragma DATA vectors
__interrupt void *irqvecs = IRQHandler;
```

script file

```
group( r, interrupt_vectors )
        {
                vectors
        }
layout
        {
        ....,
        0x3FF2,0x3FF4 <= interrupt_vectors
        }
```

Interrupt vector table, run time initialisation

```
__interrupt void IRQHandler( void )
{
        /* Interrupt service routine  */
}

#pragma DATA vectors
__interrupt void *irqvecs;

#pragma DATA data

void main(void)
{
        irqvecs = IRQHandler;
        ...
}
```

Run time initialisation no vector table

```
__interrupt void IRQHandler( void )
{
        /* Interrupt service routine */
}

void main(void)
{
        *(unsigned short *) 0x3FF2 = (unsigned short) IRQHandler;
        ...
}
```

alternatively...

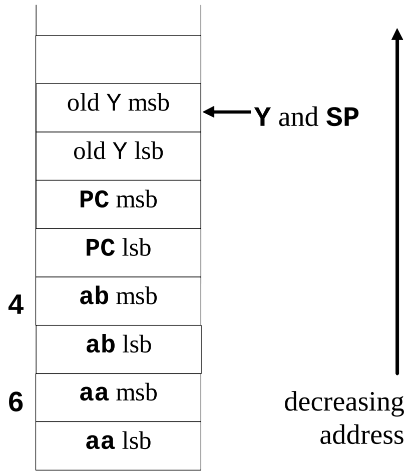
```
#define  SET_IRQ_VECTOR(x,y) *(unsigned short *) y = (unsigned short) x

void main(void)
{
        SET_IRQ_VECTOR( IRQHandler , 0x3FF2 ) ;
        ...
}
```

Function call conventions in XCC12

```
EXAMPLE: Function call
int    a,b;
main()
{
   callfunc( a,b );
}
-----------------------------------------------------------
The following assembly code is generated by the compiler:
   ...
* 0009 |       callfunc( a,b );
   ldd  _b
   pshd
   ldd _a
   pshd
   jsr _callfunc
```

Function conventions in XCC12

```
callfunc( aa , ab )
{
   aa = 1;
   ab = 2;
}
```



old Y msb

old Y lsb

PC msb

PC lsb

4 ab msb

ab lsb

6 aa msb

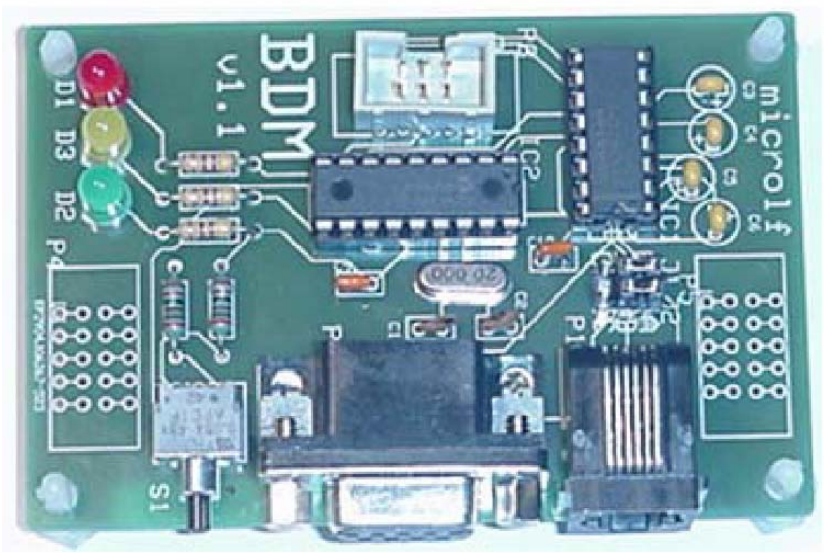
aa lsb

Y and SP

decreasing address

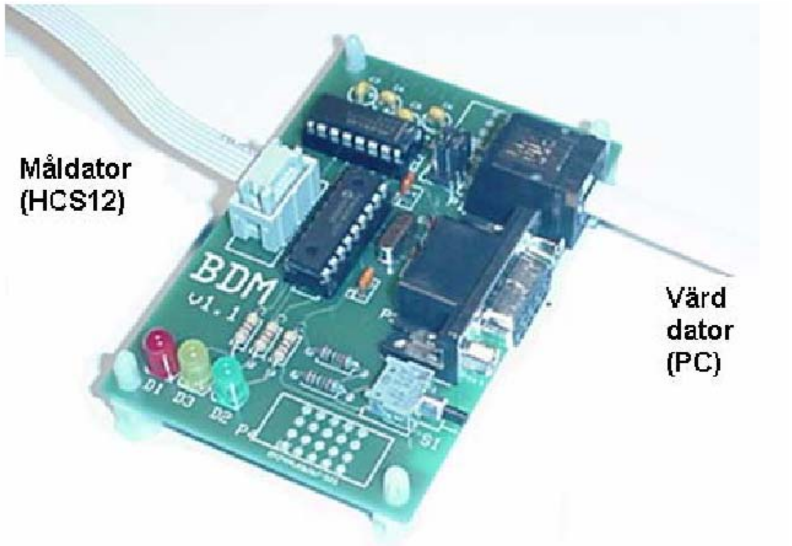
```
The following assembly code is
   generated by the compiler:
   ...
* 0007 |callfunc( aa , ab )
 segment text
 define _callfunc
_callfunc:
   pshy
   tfr  sp,y
* 0008 |{
* 0009 |        aa = 1;
   movw #1,4,y
* 0010 |        ab = 2;
   movw #2,6,y
* 0011 |}
   puly
   rts
```

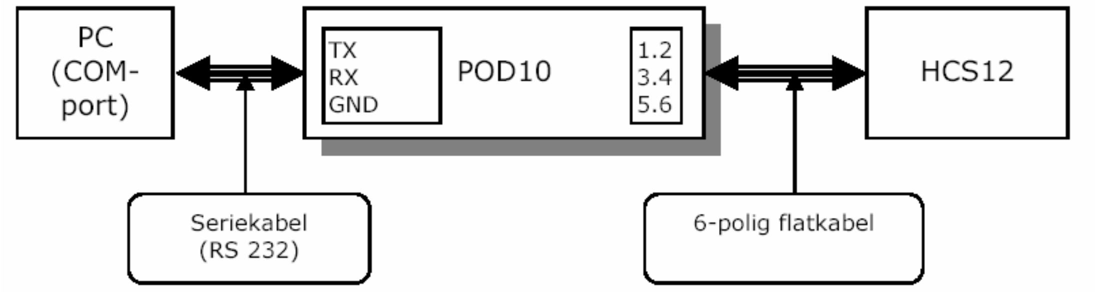
Background Debug Mode –BDM adapter



- Single wire electrical interface
- Used for debugging and programming the FLASH memory in HCS12

BDM adapter (”pod”)




PC (COM-port)

TX
RX
GND

POD10

1.2
3.4
5.6

HCS12

Seriekabel (RS 232)

6-polig flatkabel

Summary

we have got a brief introduction to

- Low level programming in ’C’
- Cross developments environments and tools in general
- XCC12 cross C compiler in particular

which finishes today’s lecture …