week	lecture	Topics
4	System Startup – Detailed	- The system startup file (example: KEIL) Macro definitions, configuration registers and stack frames - Memory models and memory maps - Memory type specifiers - Object classes and storage class - Segmented / non-segmented addressing - Reset vector - Initialization of Block Segment Sections (BSS) and of variables which have been initialised at file level - Step-by-step startup (KEIL debugger)

The system startup file

- Compiler dependent; KEIL C166 tool chain uses a generic assembler file called *Start167.a66*
- Macros are used to configure this file, e.g. …

… *$MOD167* ensures that the assembler generates instructions for the C167 (as opposed to C166, C164)

… *$CASE* enable case sensitive pre-processing of this file

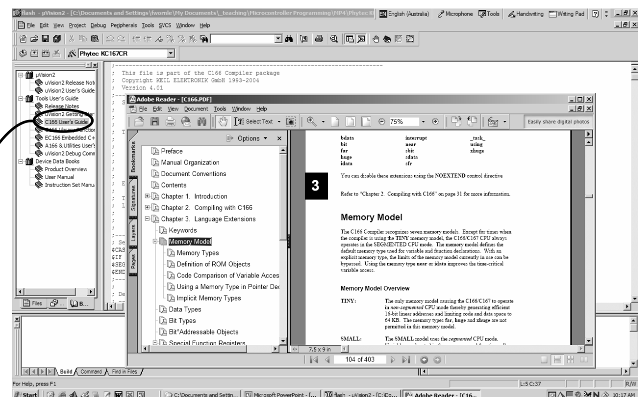
… etc.

The system startup file

```
$MOD167                                                    ; Define C167 mode
;
;------------------------------------------------------------------------------
;  This file is part of the C166 Compiler package
;  Copyright KEIL ELEKTRONIK GmbH 1993-2004
;  Version 4.01
;------------------------------------------------------------------------------
;  START167.A66:  This code is executed after processor reset and provides the
;                 startup sequence for the extended 166 architecture CPU's.
;                 (i.e. C167/C165/C164/C163/C161, ST10-262 ect.)
;
;  To translate this file use A166 with the following invocation:
;
;     A166 START167.A66 SET (<model>)
;
;     <model> determines the memory model and can be one of the following:
;        TINY, SMALL, COMPACT, HCOMPACT, MEDIUM, LARGE, HLARGE, XLARGE
;
;
;  Example:  A166 START167.A66 SET (SMALL)
          (…)
```

Some compilers define so-called *memory models* to define a memory layout of the compiled program

Memory models



Memory models are compiler specific features and are therefore described in the *compiler manuals*

Memory models

- Memory models define *default mappings* for variable declarations without *memory type specifier*
- The memory type specifier of an object (variable or function) tells the *compiler* to which *object class* it is to be assigned
- The *object class* is used by the *linker* to locate an object to an absolute address
- This correspondence is defined in a *memory map;* the programmer thus has complete control over where in memory an object will end up, e.g. internal / external RAM or ROM, EEPROM, etc.

Memory models – object classes

Object classes known to the KEIL compiler / linker

Class Names	Location
NCODE	ROM or EPROM space for **near** code; must fit into **one** 64 KB segment.
NCONST	ROM or EPROM space for **const near** variables; must fit into **one** 16 KB page. With the L166 DPPUSE directive, you can enlarge the 16KB area.
FCONST	ROM or EPROM space for **const far** variables
HCONST	ROM or EPROM space for **const huge** variables
XCONST	ROM or EPROM space for **const xhuge** variables
FCODE	ROM or EPROM space for **far** code
NDATA, NDATA0	RAM space for **near** variables; both classes must fit into one 16 KB page. With the L166 DPPUSE directive, you can enlarge the 16KB area.
SDATA, SDATA0	RAM space for **sdata** variables; address 0C000h-0FFFFh (Page 3)
IDATA, IDATA0	On-chip RAM for **idata** variables: Address range for 166: 0xFA00-0xFDFF Address range for 167: 0xF800-0xFDFF
BIT, BIT0	Bit-addressable RAM for **bit** variables; address 0FD00h-0FDFFh
BDATA, BDATA0	Bit-addressable RAM for **bdata** variables; address 0FD00h-0FDFFh
FDATA, FDATA0	RAM space for **far** variables
HDATA, HDATA0	RAM space for **huge** variables
XDATA, XDATA0	RAM space for **xhuge** variables
ICODE	ROM or EPROM space; located in on-chip code memory

Memory models – object classes

- Object classes are either CODE, CONST or DATA
- The objects within any of these classes can be accessed using Near, Far, Huge or Xhuge addresses (hence, NDATA are variables which are addressed using paged ‘14+2’-bit addresses, etc.)
- A class name ending on ‘0’ indicates that the associated memory is to be erased at system start
- Names beginning with ‘I’ refer to classes within the on-chip RAM or ROM; names beginning with ‘B’ refer to bit-addressable memory; ‘S’ stands for the system page (0xC000 to 0xFFFF, page 3)

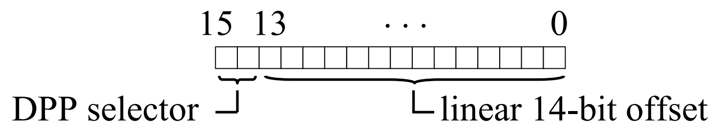
Memory models – memory types

Object classes are assigned based on the memory type of a variable

Memory Type	80C166 Address Space
near	16-bit pointer; 16-bit address calculation allows access to16 KB for variables in NDATA group, for constants in NCONST group, and for system area in SDATA group. With the L166 directive DPPUSE, you can have up to 64KB in the NDATA/NCONST group. If **near** is applied to a function, a CALLA or CALLR (16-bit call) is generated for this function.
idata	On-chip RAM of the 80C166; fastest access to variables.
bdata	Bit-addressable on-chip RAM of the 80C166; supports mixed bit and byte accesses (limited to 256 bytes).
sdata	SYSTEM area (address space 0C000h-0FFFFh); can be used for the definition of PEC-addressable objects.
far	32-bit pointer; 16-bit address calculation allows full access to the whole address space. The size of a single object (array or structure) is limited to 16 KB. If **far** is applied to a function, a CALLS (segmented call) instruction is generated for this function. **far** is the optimum memory type on the 80C166 CPU for accessing the 256KB address space of this CPU.
huge	32-bit pointer; 16-bit address calculation supports objects of up to 64 KB. **huge** is the optimum memory type on the C167/C165 CPU for accessing the 16MB address space.
xhuge	32-bit pointer; 32-bit address calculation supports objects with unlimited size.

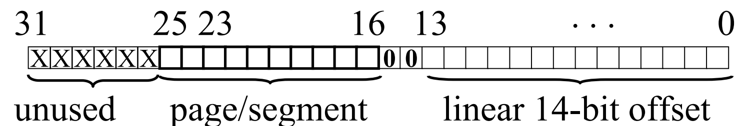
Memory models – memory type

- *Near addresses* contain two bytes (16-bit)
- A near address is interpreted as linear 16-bit address when the CPU is running in *non-segmented mode*; non-segmented mode restricts the address space to the first 64 kByte (default)
- In *segmented mode*, a near address consists of a Data-Page Pointer selector and a linear offset:



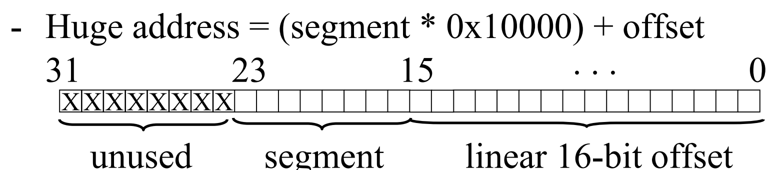
Memory models – memory type

- *Far addresses* contain two words (32-bit)
- A far address consists of a linear 14-bit offset (this limits the maximum object size to 16 kByte) and …
  … for *function* pointers: a segment number (0 – 255)
  … for *variable* addresses: a page number (0 – 1023)
- Far functions: address = (segm * 0x10000) + offset
  Far variables: address = (page * 0x4000) + offset



Memory models – memory type

- *Huge/Xhuge addresses* contain two words (32-bit)
- A huge (data) address consists of a linear 16-bit offset (limits the maximum object size to 64 kByte) and a segment number (0 – 255)
- Huge address = (segment * 0x10000) + offset



- An Xhuge (data) address is simply a linear 32-bit address (unlimited object size, i. e. 16 MB)

Memory models – memory type

- Huge/Xhuge pointers are only useful with variables
- For function calls, *huge, xhuge* and *far* pointers are identical: they consist of an 8-bit segment number and a 16-bit linear address offset
- Example:
  char xhuge myVariable[0x20000];
  This defines an xhuge character array with a total size of 128 kByte; note that access to the elements above 64k is not possible with a far address! (This would require a new segment number to be loaded)

Memory models

- The KEIL compiler provides 8 memory models: TINY, SMALL, COMPACT, HCOMPACT, MEDIUM, LARGE, HLARGE and XLARGE
- These memory models define the *default memory type* to be used for function calls and variables
- These defaults can usually be overridden by an explicit declaration, e.g.

```
char  far   myArray[1024];
void  far   mySubroutine(void);
```


Memory models

- The memory models of the KEIL compiler C166

Memory Model	Default Memory Type For...	
	Variables	Functions
TINY	near	near (up to 64KB code size)
SMALL	near	near (up to 64KB code size)
COMPACT	far	near (up to 64KB code size)
HCOMPACT (not for 166 CPU)	huge	near (up to 64KB code size)
MEDIUM	near	far (unlimited code size)
LARGE	far	far (unlimited code size)
HLARGE (not for 166 CPU)	huge	far (unlimited code size)

- All models but TINY run in *segmented* CPU mode
- Note that not all compilers define memory models (Metrowerks' CodeWarrior uses #*pragma* directives)

Memory models

- TINY:
  non-segmented CPU mode; all code/data access via linear 16-bit addresses → address space: 64 kByte; specifiers *far, huge* and *xhuge* are not permitted

- SMALL:
  segmented CPU mode; variables are in the ***near*** area and function calls generate ***near*** addresses; code and data can be located outside the first 64k; overriding specifiers *far, huge* and *xhuge* are permitted

Memory models

- COMPACT:
  same as SMALL memory model, but data objects of a size larger than 6 bytes (or whatever threshold has been configured using the HOLD directive) are placed in *far memory* and accessed using *far pointers*
- HCOMPACT:
  same as COMPACT memory model, but with *huge* memory and pointers instead of *far*
- Both COMPACT and HCOMPACT are used with small programs which have large data requirements

Memory models

- MEDIUM:
  same as SMALL memory model, but all function calls generate *far* addresses; used with large programs having small data size requirements
- LARGE:
  both, access to oversized data (→HOLD) as well as functions calls generate *far* addresses; used with large programs having large data size requirements
- HLARGE:
  same as LARGE memory model, but with oversized data objects (→HOLD) being placed in *huge* memory

Memory models – memory map

- The *memory map* assigns each *object class* to an absolute address range; together with the specifics of the underlying hardware (type and configuration of the controller, external memory and peripherals) this defines 'where stuff goes'
- Note that the explicit use of *storage class* specifiers (static, automatic, extern, etc.) or *scope modifiers* (const, volatile, etc.) provides further mechanisms to locate objects in memory. For example, an object which has been declared as *const* is located in N/F/HCONST (which is usually mapped to ROM)

Memory models – memory map

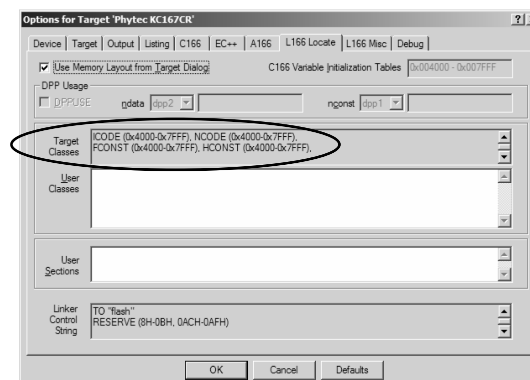
- The *memory map* is specified to the linker (e.g. on the command line, in the menus of the IDE, in form of a linker map file, etc.)
- Example for the KEIL linker (L166):

```
L166 myProg.obj \
        CLASSES (FCODE(0x000000-0x00BFFF, 0x010000-0x03FFFF), \
                 NCONST(0x00C000-0x00DFFF), FCONST(0x00C000-0x00DFFF), \
                 NDATA(0x100000-0x10BFFF), NDATA0(0x100000-0x10BFFF),
                 FDATA(0x10C000-0x13FFFF), FDATA0(0x10C000-0x13FFFF) )
```

- Far data can be found from 0x10C000 onwards, the code is at 0 – 0xBFFF and 0x10000 – 0x3FFFF, constants (near and far) are located above 0xC000

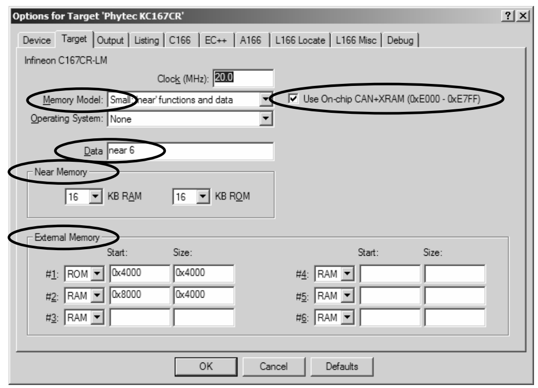
Memory models – memory map

- In the KEIL μVision IDE the memory map is specified in the linker tab of the options menu



Memory models – memory map

- This memory map can also be build automatically from the specifications made in the *Target* tab



The system startup file

```
    (…)
;---------------------------------------------------------------------------
; Setup model-dependend Assembler controls
$CASE
$IF NOT TINY
$SEGMENTED
$ENDIF
;---------------------------------------------------------------------------
    (…)
```

Only the TINY memory model uses the non-segmented CPU mode; for all other models, this directs the assembler to generate segmented addresses and to use machine language op-codes with segmented operands

The system startup file

```
    (…)
; Definitions for BUSCON0 and SYSCON Register:
; --------------------------------------------
;
; MCTC0: Memory Cycle Time (BUSCON0.0 .. BUSCON0.3):
; Note: if RDYEN0 == 1 a maximum number of 7 waitstates can be selected
_MCTC0  EQU   1         ; Memory wait states is 1 (MCTC0 field = 0EH).
;                       ; (Reset Value = 15 additional state times)
;
; RWDC0: Read/Write Signal Delay (BUSCON0.4):
_RWDC0  EQU   1         ; 0 = Delay Time     0.5 States (Reset Value)
;                       ; 1 = No Delay Time  0   States
;
; MTTC0: Memory Tri-state Time (BUSCON0.5):
_MTTC0  EQU   0         ; 0 = Delay Time     0.5 States (Reset Value)
;                       ; 1 = No Delay Time  0   States
    (…)
```

EQUate directives are used to define a series of macros; these macros are later combined to define the contents of the *system configuration* registers

The system startup file

- BUSCON registers configure the *External Bus Controller (EBC);* the external bus can be enabled or disabled, it can be configured to be an 8-bit multiplexed bus, a 16-bit non-multiplexed bus, …, read/write access to the bus can be slowed down using wait states, etc.
- SYSCON registers configure the controller itself; control of the internal XBUS, the watchdog oscillator, the internal ROM, the CPU (e.g. segmented mode or not), the system stack size, the behaviour following a hardware reset, etc.

The system startup file

```
   (…)
; STKSZ: Maximum System Stack Size selection  (SYSCON.13 .. SYSCON.15)
;  Defines the system stack space which is used by CALL/RET and PUSH/POP
;  instructions.  The system stack space must be adjusted according the
;  actual requirements of the application.
$SET (STK_SIZE = 0)
;     System stack sizes:
;       0 = 256 words (Reset Value)
;       1 = 128 words
;       2 =  64 words
;       3 =  32 words
;       4 = 512 words
;       5 = not implemented
;       6 = not implemented
;       7 = no wrapping (entire internl RAM use as STACK, set size with SYSSZ)
; If you have selected 7 for STK_SIZE, you can set the actl system stack size
; with the following SSTSZ statement.
SSTSZ   EQU  200H           ; set System Stack Size to 200H Bytes
   (…)
```

Definition of macros specifying the size of the *system stack*

The system startup file

```
   (…)
; USTSZ: User Stack Size Definition
;  Defines the user stack space available for automatics.  This stack space is
;  accessed by R0.  The user stack space must be adjusted according the actual
;  requirements of the application.
USTSZ    EQU  1000H        ; set User Stack Size to 1000H Bytes.
;
; WATCHDOG: Disable Hardware Watchdog
; --- Set WATCHDOG = 1 to enable the Hardware watchdog
$SET (WATCHDOG = 0)
;
; CLR_MEMORY: Disable Memory Zero Initialization of RAM area
; --- Set CLR_MEMORY = 0 to disable memory zero initilization
$SET (CLR_MEMORY = 1)
;
; INIT_VARS: Disable Variable Initialization
; --- Set INIT_VARS = 0 to disable variable initilization
$SET (INIT_VARS = 1)
   (…)
```

Definition of the *User Stack* size, control over the *Watchdog* timer, *clearing of memory* and *initialisation of variables* during startup, …

The system startup file

- Other macros control the configuration of the CAN bus interface, the activation or deactivation of the fast internal XRAM, the power down modes of the chip, the main clock source, the activation (or not) of the internal PLL circuit which allows frequency multiplication, not to forget the individual activation or deactivation of all on-chip peripherals (A/D converter, PWM unit, CAPCOM unit, General Purpose Timers, secondary serial interface (ASC1), the I^2C bus interface, etc.

The system startup file

```
   (…)
;
; BUSCON1/ADDRSEL1 .. BUSCON4/ADDRSEL4 Initialization
; ================================================
;
;
; BUSCON1/ADDRSEL1
; --- Set BUSCON1 = 1 to initialize the BUSCON1/ADDRSEL1 registers
$SET (BUSCON1 = 1)
;
; Define the start address and the address range of Chip Select 1 (CS1#)
; This values are used to set the ADDRSEL1 register
%DEFINE (ADDRESS1) (0000H)       ; Set CS1# Start Address (default 100000H)
%DEFINE (RANGE1)   (1024K)       ; Set CS1# Range (default 1024K = 1MB)
;
    (…)
```

BUSCONx/ADDRSELx (x = 1 – 4) allows the association of up to 4 chip select signals (CSx\) with a corresponding external memory window

The system startup file

```
    (…)
$IF NOT TINY
ASSUME  DPP3:SYSTEM
ASSUME  DPP2:NDATA
$ENDIF

NAME    ?C_STARTUP
PUBLIC  ?C_STARTUP

$IF MEDIUM OR LARGE OR HLARGE OR XLARGE
Model   LIT  'FAR'
$ELSE
Model   LIT  'NEAR'
$ENDIF

EXTRN   main:Model
    (…)
```

Public symbol *?C_STARTUP* is defined as global reference for the linker to locate the code of the startup module

Symbol *main* is declared a public (EXTRN) jump label, which can either be 'far' or 'near' (depending on the chosen memory model)

Eventually, the actual startup module is defined: Data Page Pointers (DPP) are initialized and a number of public symbols are defined

The system startup file

```
    (…)
PUBLIC              ?C_USRSTKBOT

?C_USERSTACK        SECTION DATA PUBLIC 'NDATA'
$IF NOT TINY
NDATA               DGROUP  ?C_USERSTACK
$ENDIF
?C_USRSTKBOT:
                    DS      USTSZ          ; Size of User Stack
?C_USERSTKTOP:
?C_USERSTACK        ENDS

?C_MAINREGISTERS  REGDEF  R0 - R15
    (…)
```

The *User Stack* is defined (size USTSZ bytes) and associated with object class NDATA; labels for top and bottom of the stack are defined

The main registers (R0 – R15) are defined

The system startup file

```
    (…)
?C_SYSSTACK      SECTION DATA PUBLIC 'IDATA'
$IF NOT TINY
SDATA            DGROUP  ?C_SYSSTACK
$ENDIF
_BOS:                                   ; bottom of system stack
                 DS      SSTSZ          ; Size of User Stack
_TOS:                                   ; top of system stack

?C_SYSSTACK      ENDS
    (…)
```

The *System Stack* is defined (size SSTSZ bytes) and associated with object class IDATA; labels for top and bottom of the stack are defined (_TOS, _BOS)

Used by the CPU for *context switching* and upon entry/exit of an *Interrupt Service Routine (ISR)*

The system startup file

```
    (…)
?C_STARTUP_CODE SECTION CODE 'ICODE'
    (…)
?C_RESET          PROC TASK C_STARTUP INTNO RESET = 0
?C_STARTUP:       LABEL Model
$IF (WATCHDOG = 0)
                  DISWDT                ; Disable watchdog timer
$ENDIF
    (…)
```

the following *procedure* is of type *task* (interrupt / trap); invoked by vector 0 (RESET)

'far' or 'near'

first actual instruction

Procedure ?C_RESET is defined, containing a *task* C_STARTUP; this interrupt/trap routine is invoked through interrupt vector 0 (RESET, abs. address: 0)

If requested, the *Watchdog* timer is switched off

The system startup file

```
   (…)
BCON0L          SET             (_MTTC0 << 5) OR (_RWDC0 << 4)
BCON0L          SET  BCON0L OR  ((NOT _MCTC0) AND 0FH)
BCON0L          SET  BCON0L AND (NOT (_RDYEN0 << 3))
BCON0L          SET  BCON0L OR  (_RDY_AS0 << 3)
BCON0H          SET             (_ALECTL0 << 1) OR (_RDYEN0 << 4)
BCON0H          SET  BCON0H OR  (_CSREN0 << 6) OR (_CSWEN0 << 7)
$IF (BTYP_ENABLE == 1)
BCON0L          SET  BCON0L OR  (_BTYP0 << 6)
BCON0H          SET  BCON0H OR  (_BUSACT0 << 2)
$ENDIF

$IF (BTYP_ENABLE == 0)
                BFLDL  BUSCON0,#03FH,#BCON0L
                BFLDH  BUSCON0,#0D2H,#BCON0H
$ELSE
                BFLDL  BUSCON0,#0FFH,#BCON0L
                BFLDH  BUSCON0,#0D6H,#BCON0H
$ENDIF
   (…)
```

2nd and 3rd actual instruction

The BUSCON register is configured as specified by all associated macro definitions (_MTTC0, etc.)

The system startup file

- The SYSCON registers are initialised
- Then the 4 pairs of register which control chip select signals CSx\ are set-up in accordance with the corresponding macro specifications (BUSCONx and ADDRSELx, where x = 1 – 4)
- The stack pointer SP is initialised
- The Data Page Pointers (DPP) are initialized
- The Context Pointer (CP) is loaded with the base address of the registers (?C_MAINREGISTERS)

The system startup file

```
   (…)
;-----------------------------------------------------------------------------
;
; The following code is necessary to set RAM variables to 0 at start-up
; (RESET) of the C application program.
;
$IF (CLR_MEMORY = 1)
EXTRN          ?C_CLRMEMSECSTART : WORD

Clr_Memory:
$IF TINY
               MOV  R8,#?C_CLRMEMSECSTART
               JMPR cc_Z,EndClear

RepeatClear:
$IF (WATCHDOG = 1)
               SRVWDT                  ; SERVICE WATCHDOG
$ENDIF
   (…)
```

clear RAM (if requested)

clear Watchdog timer

Reset RAM space to ‘0’; this slows down startup, but may saves a lot of manual initialisations

The system startup file

```
   (…)
;-----------------------------------------------------------------------------
;
;  The following code is necessary, if the application program contains
;  initialized variables at file level.
;
$IF (INIT_VARS = 1)
EXTRN                ?C_INITSECSTART : WORD

Init_Vars:
$IF TINY
                MOV  R8,#?C_INITSECSTART
RepeatInit:
$IF (WATCHDOG = 1)
                SRVWDT                          ; SERVICE WATCHDOG
$ENDIF
                MOV  R2,[R8+]
                JMPR cc_Z,EndInit
   (…)
```

copy initialised variables from ROM into RAM (if requested)

The contents of globally initialized variables have to be transferred from ROM to RAM

The system startup file

```
    (…)
;-----------------------------------------------------------------------------

$IF TINY
                  JMP  main
$ELSE
                  JMP  FAR main
$ENDIF

?C_RESET          ENDP
?C_STARTUP_CODE   ENDS

$IF (INIT_VARS = 1)
EXTERN            ?C_ENDINIT:WORD
$ENDIF
                  END
```

end of the RESET procedure (ENDP)

end of section ?C_STARTUP_CODE (ENDS)

end of the startup file (END)

Eventually, the startup code concludes with a jump (or far jump) to *main;* this is where the actual user code ('our program') starts…

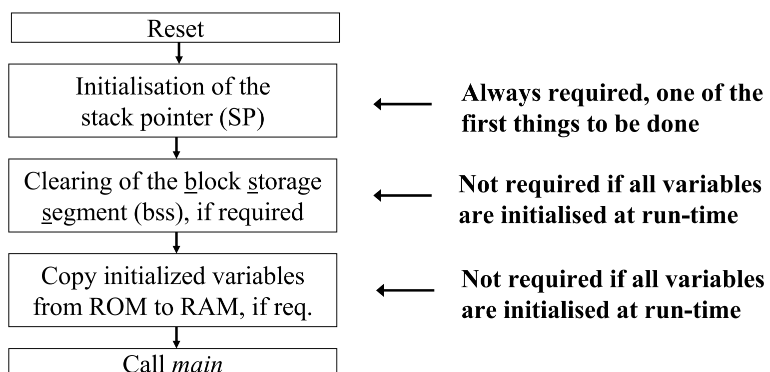
The system startup file

- Other compilers use different but similar startup files: Metrowerks' CodeWarrior uses a C-language file called *start12.c*, GNU gcc uses a very compact assembler language file called *crt0.s*

- Compiled versions of these modules can often be found in the system libraries; in this case, they do not have to be included in the build process – provided the application is linked against these libraries (e.g. libgcc.a for GNU gcc, etc.)

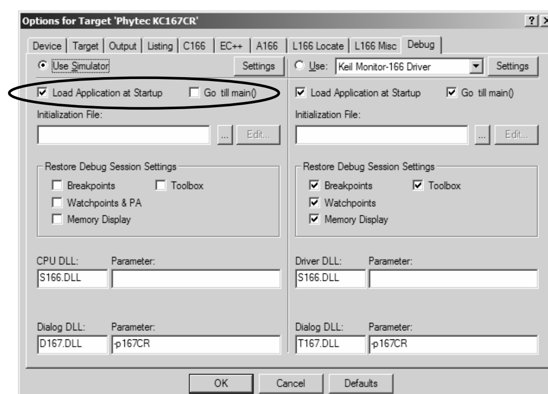
The system startup file

- Fundamental tasks common to all startup modules:



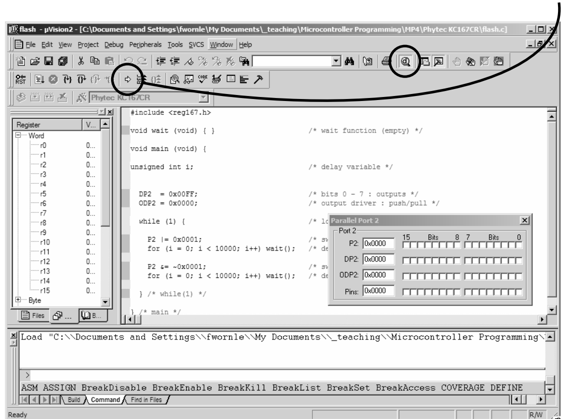
Step-by-step execution of the startup code (KEIL)

- Configure the simulator to load the application and halt, i. e. the startup code is not run automatically



Step-by-step execution of the startup code (KEIL)

- Start debugger and go to current line (yellow arrow)

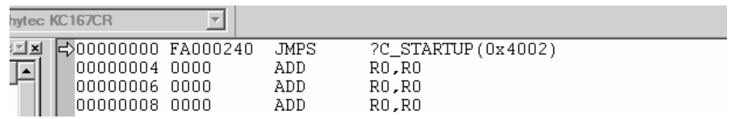


Step-by-step execution of the startup code (KEIL)

- First instruction is at absolute address 0x0000 (RESET vector); the assembler has turned

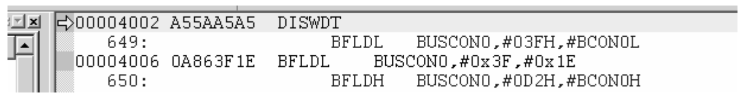
  ?C_RESET PROC TASK C_STARTUP INTNO 0

  into an absolute inter-segment jump (JMPS) to address 0x4002 (note: little endian format); this is the starting address of routine C_STARTUP



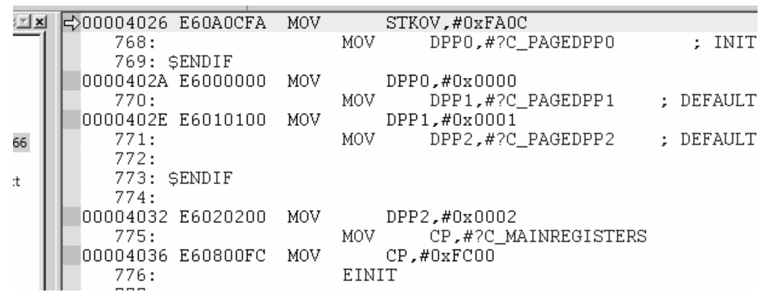
Step-by-step execution of the startup code (KEIL)

- Routine C_STARTUP has been placed at 0x4002 (note: ROM has been defined from 0x4000 onwards)
- Addresses 0x4000 – 0x4001 contain a global variable (?C_INITSECSTART), used when copying the contents of initialized variables from ROM to RAM
- First instruction of C_STARTUP disables the Watchdog timer (DISWDT); then BUSCON0 is initialized (0x4006 – 0x400D)



Step-by-step execution of the startup code (KEIL)

- The stack overflow pointer is initialised, DPP0 – DPP2 are loaded with default page numbers (0 – 2), the context pointer (CP) is initialised and the CPU is put in *normal mode* (end of initialization, EINIT)



Step-by-step execution of the startup code (KEIL)

- The Block Storage Segment (BSS) is cleared and the values of initialized variables are copied from ROM to RAM; finally, there is an inter-segment (far) jump to *main*

```
   1028:                 JMP     FAR main
0000412C FA003241  JMPS      main(0x4132)
      3: void wait (void) { }                        /* wait func
      4:
00004130 CB00      RET
      5: void main (void) {
      6:
      7: unsigned int i;                             /* delay var
      8:
      9:
     10:   DP2  = 0x00FF;                            /* bits 0
00004132 E6E1FF00  MOV       DP2,#0x00FF
     11:   ODP2 = 0x0000;                            /* output dr
```

First instruction of the *main* program