week	lecture	Topics
8	The Build Process – Advanced Concepts	- Sections, modules, programs - The linker - Interpreting the assembler listing - Interpreting the linker map file - Near data and far data - Library files - Development utilities - Objects file formats: ELF, COFF, DWARF-1/2, S-Records, Intel HEX

Build process fundamentals

- A good understanding of the build process is based on the detailed knowledge of how compiler and linker interact to produce an executable program
- These details are generally compiler (assembler, linker/locator) specific and also depend on the underlying hardware (i. e. the microcontroller)
- A number of *fundamental concepts* are common to most development tool chains (KEIL C166, GNU gcc, etc.); knowing about these opens the door to a wealth of useful information provided by these tools

Memory sections

- Many microcontroller programs are too long or too complex to be written as a single module or source code file; *modular programming* allows individual sections of a program to be reused in other projects
- A module is a black-box code fragment with a specific input / output interface; more specifically, a module consists of *relocatable object code*
- Similar modules are often collected in *library files (archive files);* they can be linked to programs with the same basic input / output requirements

Memory sections

- On *compiler* level, the code is split in procedures and functions (sub-routines, sub-programs)
- On the level of *assembler* and *linker,* programs are organised in *sections* (absolute or relocatable blocks of code or data). *Relocatable sections* have section name, type, class, group and various attributes
- Sections with the same name, but from different modules, are considered *partial sections*
- The linker combines all *partial sections* to *sections*

Memory sections

- *Absolute sections* need to be located to the specified memory address; they can not be combined with other sections (example: interrupt vector table)
- A *module* (source file) contains one or more *code sections*; the definition of a module determines the scope of its local symbols; the name of a module is assigned by the programmer
- A *program* consists of a single absolute module, merging all absolute and relocatable sections from all input modules

Memory sections

- The *linker* processes all module object files of a project; it combines all partial sections with the same name and type (they can be from the same or from different modules) and it assigns absolute memory locations to these combined sections; finally, all external references to symbols in other modules are resolved by the linker/locator – the end product is a single absolute object module *(the 'program')*
- The results of the link process can be logged to a file – on the C167 this is called the *map-file* (ext. *.m66)

Example (KEIL C166 tool chain)

```
#include <stdio.h>                      /* sprintf(), sscanf */
#include <reg167.h>


#define MY_BUF 80                       /* size of local buffer */
static char far userbuf[MY_BUF];        /* far data */

/* constant near data (kept in ROM, not copied over to RAM) */
static const char  *msg = "\r\n x y and z are (respectively): ";

void main(void) {
unsigned int  x, y, z;                  /* automatic variables */

  while(1) {

     /* display message */
     sprintf(userbuf, "10 20 30\n");
     sscanf(userbuf, "%d %d %d", &x, &y, &z);
     sprintf (userbuf, "%s %d %d %d", msg, x, y, z);

  } /* while */

} /* main */
```

The COMPACT memory model has been chosen (far data, near code)

Interpreting the assembler listing

- When compiling/assembling a module (source code file) the compiler/assembler can be instructed to provide *module specific information* in a listing file:

```
ASSEMBLY LISTING OF GENERATED OBJECT CODE

             ; FUNCTION main (BEGIN  RMASK = @0x7FFF)
                                                   ; SOURCE LINE # 11
0000 2806          SUB       R0,#06H
                                                   ; SOURCE LINE # 14
0002             ?C0003:
                                                   ; SOURCE LINE # 17
0002 E6FA0000 R    MOV       R10,#POF (SC?7?C)
0006 E6FB0000 R    MOV       R11,#PAG (SC?7?C)
000A E6F80000 R    MOV       R8,#POF (userbuf)
000E E6F90000 R    MOV       R9,#PAG (userbuf)
0012 CA000000 E    CALLA     cc_UC,sprintf
      (...)
```

At this stage, all addresses are offsets, relative to the beginning of this module

<u>R</u>elocatable or <u>E</u>xternal expressions that must be calculated by the linker/locator

Absolute addresses still unknown

Interpreting the assembler listing

Automatics **are stored on the user stack; subtracting '6' off the user stack pointer (R0) reserves space for 3 16-bit integer variables (x, y, z); note that the stack grows 'downwards' (towards lower addresses)**

The *POF / PAG operators* determine the *Page Offset / Page* of a variable

```
ASSEMBLY LISTING OF GENERATED OBJECT CODE

            ; FUNCTION main (BEGIN  RMASK = @0x7FFF)
                                               ; SOURCE LINE # 11
0000 2806           SUB       R0,#06H
                                               ; SOURCE LINE # 14
0002          ?C0003:
                                               ; SOURCE LINE # 17
0002 E6FA0000 R    MOV       R10,#POF (SC?7?C)
0006 E6FB0000 R    MOV       R11,#PAG (SC?7?C)
000A E6F80000 R    MOV       R8,#POF (userbuf)
000E E6F90000 R    MOV       R9,#PAG (userbuf)
0012 CA000000 E    CALLA     cc_UC,sprintf
     (...)
```

Source code line #17: ***sprintf(userbuf, "10 20 30\n");***

Interpreting the assembler listing

The values of *automatic variables x, y,* and *z* are *pushed onto the user stack;* this is how C implements parameter passing to (library) functions

```
     (...)
005E D4400400      MOV       R4,[R0+#04H]  ; z
0062 8840          MOV       [-R0],R4
0064 D4400400      MOV       R4,[R0+#04H]  ; y
0068 8840          MOV       [-R0],R4
006A D4400400      MOV       R4,[R0+#04H]  ; x
006E 8840          MOV       [-R0],R4
0070 F2F50200 R    MOV       R5,msg+02H
0074 F2F40000 R    MOV       R4,msg
0078 8850          MOV       [-R0],R5
007A F0C4          MOV       R12,R4
007C E6FA0000 R    MOV       R10,#POF (SC?7?0)
0080 E6FB0000 R    MOV       R11,#PAG (SC?7?0)
0084 E6F80000 R    MOV       R8,#POF (userbuf)
0088 E6F90000 R    MOV       R9,#PAG (userbuf)
008C CA000000 E    CALLA     cc_UC,sprintf
0090 06F00800      ADD       R0,#08H
                                            ; SOURCE LINE # 21
0094 0DB6          JMPR      cc_UC,?C0003
            ; FUNCTION main (END    RMASK = @0x7FFF)
```

The automatically generated local symbol *SC?7?0* will later be replaced by the address of the format string "%d %d %d" (kept in ROM)

Interpreting the assembler listing

The debugger allows the executable to be *dis-assembled;* this reveals all absolute addresses

Local symbol *SC?7?0* has been resolved to ROM address 0x001E; this is where the format string is stored

Example: Pointer variable *msg*

Variable *msg* has been defined as pointer to a constant character:
static const char *msg = "\r\n x y and z are (respectively): ";
In the COMPACT memory model this translates into a *far pointer,* with both *page number* (loaded into R5) and *page offset* (loaded into R4)

Note: Address still to be located

```
     (...)
0070 F2F50200 R    MOV       R5,msg+02H
0074 F2F40000 R    MOV       R4,msg
0078 8850          MOV       [-R0],R5
```

Variable *msg* itself is kept in *near memory* (NDATA0) and must thus be addressed via the *Data Page Pointers (DPP)*:

Disassembler listing:

```
     (...)
00000A00  F2F50290   MOV      R5,DPP2:0x1002
00000A04  F2F50290   MOV      R4,DPP2:0x1000
00000A08  00788850   MOV      [-R0],R5
```

Interpreting the *assembler listing*

- The listing usually includes a symbol table:

```
NAME                                    CLASS    SPACE  TYPE   OFFSET   SIZE
------------------------------------------------------------------------------
    (…)
BUSCON2. . . . . . . . . . . . . . . .  sfr             uint   FF16H    2
SSCRB. . . . . . . . . . . . . . . . .  sfr             uint   F0B2H    2
SSCBR. . . . . . . . . . . . . . . . .  sfr             uint   F0B4H    2
sscanf . . . . . . . . . . . . . . . .  extern   NCODE  funct  -----
sprintf. . . . . . . . . . . . . . . .  extern   NCODE  funct  -----
main . . . . . . . . . . . . . . . . .  public   NCODE  funct  -----
  x. . . . . . . . . . . . . . . . . .  auto            uint      0H    2
  y. . . . . . . . . . . . . . . . . .  auto            uint      2H    2
  z. . . . . . . . . . . . . . . . . .  auto            uint      4H    2
msg. . . . . . . . . . . . . . . . . .  static   NDATA0 ptr       0H    4
userbuf. . . . . . . . . . . . . . . .  static   FDATA0 array     0H    80
```

In the COMPACT memory model, variable *msg* is a FAR pointer (page, offset → 4 bytes); it is kept in NDATA0 because it is a 'small' object (less than 6 bytes, see: HOLD directive)

Automatics* are referred to by their offset to the *user stack pointer

Interpreting the *assembler listing*

- The listing should also present the memory usage:

```
MODULE INFORMATION:   INITIALIZED  UNINITIALIZED
  CODE SIZE        =        150     --------
  NEAR-CONST SIZE  =   --------     --------
  FAR-CONST SIZE   =         65     --------
  HUGE-CONST SIZE  =   --------     --------
  XHUGE-CONST SIZE =   --------     --------
  NEAR-DATA SIZE   =          4     --------
  FAR-DATA SIZE    =         80     --------
  XHUGE-DATA SIZE  =   --------     --------
  IDATA-DATA SIZE  =   --------     --------
  SDATA-DATA SIZE  =   --------     --------
  BDATA-DATA SIZE  =   --------     --------
  HUGE-DATA SIZE   =   --------     --------
  BIT SIZE         =   --------     --------
  INIT'L SIZE      =          8     --------
END OF MODULE INFORMATION.
```

The COMPACT memory model assigns all constant data objects (with a size of more than 6 bytes) to class FCONST; in our program this applies to the message string, the format string, etc.

There is some near data, because pointer variable *msg* only takes up 4 bytes, which is below the HOLD threshold

Interpreting the *linker map file*

```
INPUT MODULES INCLUDED:
  start167.obj (?C_STARTUP)
         COMMENT TYPE 128: A166 V4.23
  Test.obj (TEST)
         COMMENT TYPE 128: C166 V4.23
  C:\PROGRAM FILES\KEIL\R423\C166\LIB\C167C.LIB (?C_ENDINIT)
         COMMENT TYPE 128: A166 V4.21c
  C:\PROGRAM FILES\KEIL\R423\C166\LIB\C167C.LIB (SPRINTF)
         COMMENT TYPE 128: A166 V4.21c
  C:\PROGRAM FILES\KEIL\R423\C166\LIB\C167C.LIB (SCANF)
         COMMENT TYPE 128: A166 V4.21c
  C:\PROGRAM FILES\KEIL\R423\C166\LIB\C167C.LIB (?C_PCASTS)
         COMMENT TYPE 128: A166 V4.21c
  C:\PROGRAM FILES\KEIL\R423\C166\LIB\C167C.LIB (?C?PRNFMT)
         COMMENT TYPE 128: A166 V4.21c
  C:\PROGRAM FILES\KEIL\R423\C166\LIB\C167C.LIB (ISSPACE)
         COMMENT TYPE 128: C166 V4.21d
  C:\PROGRAM FILES\KEIL\R423\C166\LIB\C167C.LIB (GETCHAR)
         COMMENT TYPE 128: C166 V4.21d
  C:\PROGRAM FILES\KEIL\R423\C166\LIB\C167C.LIB (UNGET)
         COMMENT TYPE 128: C166 V4.21d
  C:\PROGRAM FILES\KEIL\R423\C166\LIB\C167C.LIB (PUTCHAR)
         COMMENT TYPE 128: A166 V4.21c
  C:\PROGRAM FILES\KEIL\R423\C166\LIB\C167C.LIB (GETKEY)
         COMMENT TYPE 128: C166 V4.21d
```

All *Input Modules* are listed with full file name and *entry point label* (in brackets)

All library functions (e.g. *sprintf* and *scanf)* as well as all the sub-routines thes functions appear to call (e.g. *getchar*, etc.) are taken from the COMPACT memory model library C167C.LIB

Interpreting the *linker map file*

- The *memory map* presents another very useful piece of information; every section which is part of the final program is listed with its *section name*, *memory class, type* and *address range* (amongst other details)
- The *section name* is commonly a pre-defined label (e.g. *?C_STARTUP_CODE)* or an automatically generated specifier (e.g. *?PR?TEST* for the code section of the *test* program)
- The prefixes *?PR?, ?ND0?, ?FD0?, etc.* are used to indicate the memory class of a section

Interpreting the *linker map file*

```
MEMORY MAP OF MODULE:  Test (?C_STARTUP)

START     STOP      LENGTH    TYPE  RTYP  ALIGN  TGR  GRP  COMB  CLASS    SECTION NAME
=====================================================================================
000000H   000003H   000004H   ---   ---   ---    ---  ---  ---   * INTVECTOR TABLE *
000004H   00000DH   00000AH   XDATA REL   WORD   ---  ---  GLOB  ---      ?C_INITSEC
00000EH   00001DH   000010H   CONST ABS   WORD   ---  ---  PRIV  ---      ?C_CLRMEMSEC
00001EH   00005EH   000041H   DATA  REL   BYTE   ---  ---  PUBL  FCONST   ?FC?TEST
000060H   00009FH   000040H   DATA  REL   WORD   ---  ---  PUBL  FCONST   ?FC??PRNFMT
0000A0H   0001CDH   00012EH   CODE  REL   WORD   ---  ---  PRIV  ICODE    ?C_STARTUP_CODE
0001CEH   00065BH   00048EH   CODE  REL   WORD   ---    2  PRIV  NCODE    ?PR?SCANF
00065CH   00098FH   000334H   CODE  REL   WORD   ---    2  PUBL  NCODE    ?C_LIB_CODE
000990H   000A25H   000096H   CODE  REL   WORD   ---    2  PUBL  NCODE    ?PR?TEST
000A26H   000A57H   000032H   CODE  REL   WORD   ---    2  PRIV  NCODE    ?PR?PUTCHAR
000A58H   000A85H   00002EH   CODE  REL   WORD   ---    2  PUBL  NCODE    ?PR?GETCHAR
000A86H   000A9FH   00001AH   CODE  REL   WORD   ---    2  PUBL  NCODE    ?PR?ISSPACE
000AA0H   000AABH   00000CH   CODE  REL   WORD   ---    2  PUBL  NCODE    ?PR?GETKEY
000AACH   000AB3H   000008H   CODE  REL   WORD   ---    2  PUBL  NCODE    ?PR?UNGET
008000H   008FFFH   001000H   DATA  REL   WORD   ---    1  PUBL  NDATA    ?C_USERSTACK
009000H   009003H   000004H   DATA  REL   WORD   ---    1  PUBL  NDATA0   ?ND0?TEST
009004H   009004H   000001H   DATA  REL   BYTE   ---    1  PUBL  NDATA0   ?ND0?GETCHAR
009006H   009055H   000050H   DATA  REL   WORD   ---  ---  PUBL  FDATA0   ?FD0?TEST
00FA00H   00FBFFH   000200H   ---   ---   ---    ---  ---  ---   * SYSTEM STACK *
00FC00H   00FC1FH   000020H   DATA  ---   BYTE   ---  ---  ---   *REG*    ?C_MAINREGISTERS
```


Interpreting the *linker map file*

Memory location *0x0000* contains the *RESET vector;* the *(far) constants* of the program have been located at *0x001E* and the *startup code* resides at *0x00A0;* the code sections of the *library functions* start at *0x01CE* (notice the immense size of *scanf*… !)

```
MEMORY MAP OF MODULE:  Test (?C_STARTUP)

START     STOP      LENGTH    TYPE  RTYP  ALIGN  TGR  GRP  COMB  CLASS    SECTION NAME
=====================================================================================
000000H   000003H   000004H   ---   ---   ---    ---  ---  ---   * INTVECTOR TABLE *
000004H   00000DH   00000AH   XDATA REL   WORD   ---  ---  GLOB  ---      ?C_INITSEC
00000EH   00001DH   000010H   CONST ABS   WORD   ---  ---  PRIV  ---      ?C_CLRMEMSEC
00001EH   00005EH   000041H   DATA  REL   BYTE   ---  ---  PUBL  FCONST   ?FC?TEST
000060H   00009FH   000040H   DATA  REL   WORD   ---  ---  PUBL  FCONST   ?FC??PRNFMT
0000A0H   0001CDH   00012EH   CODE  REL   WORD   ---  ---  PRIV  ICODE    ?C_STARTUP_CODE
0001CEH   00065BH   00048EH   CODE  REL   WORD   ---    2  PRIV  NCODE    ?PR?SCANF
00065CH   00098FH   000334H   CODE  REL   WORD   ---    2  PUBL  NCODE    ?C_LIB_CODE
```


Interpreting the *linker map file*

The *user stack* grows downwards from RAM address *0x8FFF to 0x8000;* the *system stack* is relatively small – it grows from *0xFBFF down to 0xFA00* (this is in internal RAM)*;* a partial section for initialized near data is *?ND0?TEST* – this is our far pointer *msg* (4 bytes)

```
000990H   000A25H   000096H   CODE  REL   WORD   ---    2  PUBL  NCODE    ?PR?TEST
000A26H   000A57H   000032H   CODE  REL   WORD   ---    2  PRIV  NCODE    ?PR?PUTCHAR
000A58H   000A85H   00002EH   CODE  REL   WORD   ---    2  PUBL  NCODE    ?PR?GETCHAR
000A86H   000A9FH   00001AH   CODE  REL   WORD   ---    2  PUBL  NCODE    ?PR?ISSPACE
000AA0H   000AABH   00000CH   CODE  REL   WORD   ---    2  PUBL  NCODE    ?PR?GETKEY
000AACH   000AB3H   000008H   CODE  REL   WORD   ---    2  PUBL  NCODE    ?PR?UNGET
008000H   008FFFH   001000H   DATA  REL   WORD   ---    1  PUBL  NDATA    ?C_USERSTACK
009000H   009003H   000004H   DATA  REL   WORD   ---    1  PUBL  NDATA0   ?ND0?TEST
009004H   009004H   000001H   DATA  REL   BYTE   ---    1  PUBL  NDATA0   ?ND0?GETCHAR
009006H   009055H   000050H   DATA  REL   WORD   ---  ---  PUBL  FDATA0   ?FD0?TEST
00FA00H   00FBFFH   000200H   ---   ---   ---    ---  ---  ---   * SYSTEM STACK *
00FC00H   00FC1FH   000020H   DATA  ---   BYTE   ---  ---  ---   *REG*    ?C_MAINREGISTERS
```


Interpreting the *linker map file*

- Another list indicates which sections have been combined in so-called *section groups:*

```
GROUP LIST OF MODULE:  Test (?C_STARTUP)

GROUP NAME                          TYPE  TGR  GRP  CLASS      SECTION NAME
===========================================================================
NDATA                               DATA  ---    1  NDATA      ?C_USERSTACK
                                                    NDATA0     ?ND0?TEST
                                                    NDATA0     ?ND0?GETCHAR

NCODE                               CODE  ---    2  NCODE      ?PR?TEST
                                                    NCODE      ?C_LIB_CODE
                                                    NCODE      ?PR?SCANF
                                                    NCODE      ?PR?ISSPACE
                                                    NCODE      ?PR?GETCHAR
                                                    NCODE      ?PR?UNGET
                                                    NCODE      ?PR?PUTCHAR
                                                    NCODE      ?PR?GETKEY
```

Interpreting the *linker map file*

- Finally, there is a number of *symbol tables;* at this stage, all symbols have been resolved

```
PUBLIC SYMBOLS OF MODULE:  Test (?C_STARTUP)

      VALUE       PUBLIC SYMBOL NAME              REP   TGR  CLASS   SECTION
      =======================================================================
(…)
      000000H     ?C_PAGEDPP0                     CONST ---  ---     ---
      000001H     ?C_PAGEDPP1                     CONST ---  ---     ---
      000002H     ?C_PAGEDPP2                     CONST ---  ---     ---
      000688H     ?C_PCASTS                       LABEL ---  NCODE   ?C_LIB_CODE
      0000A0H     ?C_STARTUP                      LABEL ---  ICODE   ?C_STARTUP_CODE
      00FA00H     ?C_SYSSTKBOT                    CONST ---  ---     ---
      008000H     ?C_USRSTKBOT                    VAR   ---  NDATA   ?C_USERSTACK
      000000H     RESET                           INTNO ---  ---     ---
      000AA0H     _getkey                         LABEL ---  NCODE   ?PR?GETKEY
(…)
      000990H     main                            LABEL ---  NCODE   ?PR?TEST
      000A26H     putchar                         LABEL ---  NCODE   ?PR?PUTCHAR
      000280H     scanf                           LABEL ---  NCODE   ?PR?SCANF
(…)
```


Example: Pointer variable *msg* (contd.)

The memory model can be overridden locally using the *near* modifier:
static const char near *msg = "\r\n x y and z are (respectively): ";
The compiler now inserts a jump to *library function ?C_PCASTS* which analyses bit 14 and 15 of pointer *msg* to determine which DPP register needs to be loaded; it then fetches the contents of this register from SFR memory (DPP are kept at 0xFE00 – 0xFE07) and stores it in R5

***?C_PCASTS* is an *external* function**

```
       (…)
0070 F2F40000 R    MOV       R4,msg
0074 CA000000 E    CALLA     cc_UC,?C_PCASTS
0078 8850          MOV       [-R0],R5
```

The reason why the compiler produces this seemingly inefficient code is that we are linking our program against a *COMPACT memory model library* (function calls *sprintf* and *sscanf);* the functions of this library expect their call-up parameters to be far pointers!

Example: Pointer variable *msg* (contd.)

- The symbol table of the modified program:

```
NAME                                   CLASS    SPACE  TYPE   OFFSET  SIZE
---------------------------------------------------------------------------
    (…)
BUSCON2. . . . . . . . . . . . . . . . sfr             uint   FF16H   2
SSCRB. . . . . . . . . . . . . . . . . sfr             uint   F0B2H   2
SSCBR. . . . . . . . . . . . . . . . . sfr             uint   F0B4H   2
sscanf . . . . . . . . . . . . . . . . extern   NCODE  funct  -----
sprintf. . . . . . . . . . . . . . . . extern   NCODE  funct  -----
main . . . . . . . . . . . . . . . . . public   NCODE  funct  -----
  x. . . . . . . . . . . . . . . . . . auto            uint      0H   2
  y. . . . . . . . . . . . . . . . . . auto            uint      2H   2
  z. . . . . . . . . . . . . . . . . . auto            uint      4H   2
msg. . . . . . . . . . . . . . . . . . static   NDATA0 ptr       0H   2
userbuf. . . . . . . . . . . . . . . . static   FDATA0 array     0H   80
```

The near pointer *msg* only takes up 2 bytes of storage in the NDATA0 area (16-bit DPP-format address: bits 14, 15 define the *Data Page Pointer* (DPP0 – DPP3), bits 0 – 13 are the actual *page offset)*

Example: Pointer variable *msg* (contd.)

- Memory usage has also changed:

```
MODULE INFORMATION:   INITIALIZED  UNINITIALIZED
  CODE SIZE        =        150      --------
  NEAR-CONST SIZE  =         34      --------
  FAR-CONST SIZE   =         31      --------
  HUGE-CONST SIZE  =   --------      --------
  XHUGE-CONST SIZE =   --------      --------
  NEAR-DATA SIZE   =          2      --------
  FAR-DATA SIZE    =         80      --------
  XHUGE-DATA SIZE  =   --------      --------
  IDATA-DATA SIZE  =   --------      --------
  SDATA-DATA SIZE  =   --------      --------
  BDATA-DATA SIZE  =   --------      --------
  HUGE-DATA SIZE   =   --------      --------
  BIT SIZE         =   --------      --------
  INIT'L SIZE      =          6      --------
END OF MODULE INFORMATION.
```

The message string (length: 33 characters + end-of-string byte) can now be found in the NCONST memory class

The near data has decreased to 2 bytes, as the *msg* pointer is now a '14+2' bit near address (DPP number, offset)

Library files

- The *libraries* are collections of object code which have been compiled in a particular memory model
- Most compilers automatically search the *standard libraries* (e.g. *libc.a* on UNIX based systems) to resolve any external symbols which cannot be found in any of the listed modules
- More exotic libraries have to be specified explicitly by the programmer (e.g. on UNIX based systems, the command line option *–lm* causes the application to be linked against the maths library *libm.a*)

Library files

- Any library function a program may call upon has to be declared prior to its first usage; this is commonly done by the accompanying *header file (*.h)*
- Example:

```
#include <stdio.h>                    /* printf() */
void main(void) {
  printf("Hello world\n");
  while(1);                           /* forever… */
} /* main */
```

The declaration for *printf* can be found in header file *stdio.h;* the two brackets ('<' and '>') instruct the compiler (more precisely: the pre-processor of the compiler) to look for this file on the standard path

Library files

- The objects within a library have been compiled using a variety of *code generation options;* most importantly, a particular *memory model* has been chosen, defining the way data is accessed (near / far addressing) and how sub-routines are called (near / far calls)
- It is therefore necessary to link the program against those libraries which have been compiled for the same memory model as that of the program
- Compilers usually come with a *complete set of libraries* for each one of their memory models

Library files

- Example: The KEIL C166 compiler suite provides a full set of libraries in the /lib folder; a list of objects contained in each one of these libraries can be obtained using the *librarian utility /bin/lib166:*

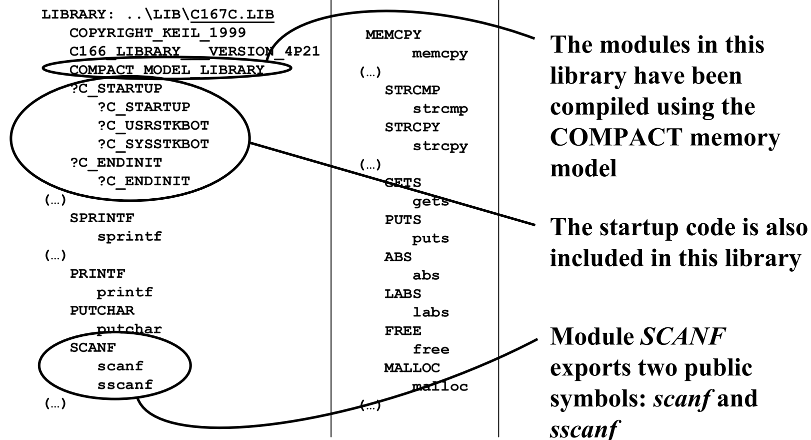
```
C:\Keil\C166\BIN>lib166 LIST ..\lib\C167C.lib TO .\c167.txt PUBLICS

LIB166 LIBRARY MANAGER V4.24
COPYRIGHT KEIL ELEKTRONIK GmbH 1987 - 2002

C:\Keil\C166\BIN>
```

- This extracts the full list of objects which have been archived in library file *C167C.lib* (in folder /lib), including all *PUBLIC symbols;* the resulting list is written to output file *c167.txt*

Library files



The modules in this library have been compiled using the COMPACT memory model

The startup code is also included in this library

Module *SCANF* exports two public symbols: *scanf* and *sscanf*

Build process fundamentals

- Understanding the information provided by the compiler/assembler/linker log files is imperative to successful embedded software development
- Every compiler uses its very own nomenclature (as these are log-files, there are no standards); however, in essence they all resemble each other
- Frequently, there are development tools which assist the programmer by extracting relevant sections from the log-files (e.g. UNIX *binutils* – this collection of tools is widely used both, on personal computer systems as well as for microcontroller programming

Build process fundamentals

- The originally UNIX based *GNU gcc compiler suite* [3] has been ported to a large number of environments, including HC12 microcontrollers *(m6812-elf-gcc)* and the ATMEL centred WinAVR tools *(avr-gcc);* gcc does not define memory models but instead works with *linker script* files
- *Linker script* files describe the which sections will be part of the final program and in which order they appear; the compiler / linker comes with a built-in default linker script which commonly works for the majority of simple applications

Build process fundamentals

- The most important sections on UNIX based systems are *.text* (contains code and constants), *.data* and *.bss* (initialized and un-initialized data, respectively)
- Other sections describe *entry to* and *exit from* a program *(.init, .fini),* the constructors and destructors of C++ programs *(.ctors, .dtors),* ROM variables *(.rodata)*, shared overlayed data sections *(.common),* EEPROMable sections *(.eeprom)*, the startup code section *(.install)*, an interrupt vector table *(.vector)*, debugging information *(.debug)*, documenting comments *(.comment)*, etc.

Build process fundamentals

- (Hugely simplified) example: *gcc linker script*

```
/* Default linker script, for normal executables */
OUTPUT_FORMAT("elf32-avr","elf32-avr","elf32-avr")
OUTPUT_ARCH(avr:4)

MEMORY
{
  text   (rx)   : ORIGIN = 0, LENGTH = 8K
  data   (rw!x) : ORIGIN = 0x800060, LENGTH = 0xffa0
}

SECTIONS
{
  .text : { *(.text.*) }  > text
  .data : { *(.data)   }  > data
  .bss  : { *(.bss)    }  > data
}
```

MEMORY defines the memory map; two memory blocks have been set-up: *text* is a read-only (ROM) block from 0 to 0x2000 (8k) and *data* refers to the read-writeable area (RAM) above 0x800060

Only the three most essential sections have been defined here: *.text* for the code and constants, .data for initialized variables and *.bss* for un-initialized data; note that .data and *.bss* are stored in memory block *data*

Build process fundamentals

The *GNU Binary Utilities* [1] is a collection of tools to assist GNU gcc programmers in developing software

- *make* – flexible rule-based project builder which helps maintaining large software systems; warning: it takes some time to become a 'make expert'…
- *objdump* – extract and display information from an object file; central to the debugging process
- *ar* – create and maintain *archives (libraries)*
- *objcopy* –translate object files to a different format
  …

Build process fundamentals

- Example: *Extracting information from a library file*

```
C:\WinAVR\bin>avr-objdump -a C:\WinAVR\avr\lib\libc.a -xd > libc.txt

C:\WinAVR\bin>
```

- This extracts the full list of objects contained in the standard library file *libc.a* of the WinAVR compiler *avr-gcc;* option 'd' causes the code section to be *disassembled;* the results are stored in file *libc.txt*
- Loading *libc.txt* into a text editor allows the details of all objects in the library file to be studied; this is useful when code does not behave as expected

Build process fundamentals

- Example: *Extracting information from a library file*

```
(…)
scanf.o:     file format elf32-avr
rw-rw-rw- 0/0    976 Mar 31 08:31 2004 scanf.o
architecture: avr, flags 0x00000011:
HAS_RELOC, HAS_SYMS
start address 0x00000000

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         0000002c  00000000  00000000  00000034  2**0
                  CONTENTS, ALLOC, LOAD, RELOC, READONLY, CODE
  1 .data         00000000  00000000  00000000  00000060  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  2 .bss          00000000  00000000  00000000  00000060  2**0
                  ALLOC
(…)
```

This extract of file libc.txt shows the entry of object module *scanf.o* and the list of its sections *(.text, .data* and *.bss);* looking at the size of the code section *(.text → 0x2C = 44 bytes)* might suggest that *scanf* is only a small module – however, this is not true: the actual code of scanf is in *vscanf.o*

Build process fundamentals

- Example: *Extracting information from a library file*

```
(…)
SYMBOL TABLE:
00000000 l    d  .text  00000000
0000003f l       *ABS*  00000000 __SREG__
0000003e l       *ABS*  00000000 __SP_H__
0000003d l       *ABS*  00000000 __SP_L__
00000000 l       *ABS*  00000000 __tmp_reg__
00000001 l       *ABS*  00000000 __zero_reg__
0000002c l       .text  00000000 Letext
00000000 l    d  .data  00000000
00000000 l    d  .bss   00000000
00000000         *UND*  00000000 __do_copy_data
00000000         *UND*  00000000 __do_clear_bss
00000000 g     F .text  000002c scanf
00000000         *UND*  00000000 __prologue_saves__
00000000         *UND*  00000000 __iob
00000000         *UND*  00000000 vfscanf
00000000         *UND*  00000000 __epilogue_restores__
(…)
```

Modules *scanf*, *fscanf* and *sscanf* all share the same computational core: *vfscanf.o*

The symbol table reveals that *scanf* depends on a larger module called *vfscanf.o;* this is the actual computational part of *scanf*

Build process fundamentals

- Example: *Extracting information from a library file*

```
(…)
Disassembly of section .text:

00000000 <scanf>:
   0:    a0 e0          ldi    r26, 0x00     ; 0
   2:    b0 e0          ldi    r27, 0x00     ; 0
(…)
  1c:    79 2f          mov    r23, r25
  1e:    80 91 00 00    lds    r24, 0x0000
            20: R_AVR_16    __iob
  22:    90 91 00 00    lds    r25, 0x0000
            24: R_AVR_16    __iob+0x1
  26:    00 d0          rcall    .+0         ; 0x28
            26: R_AVR_13_PCREL    vfscanf
  28:    e2 e0          ldi    r30, 0x02     ; 2
  2a:    00 c0          rjmp    .+0          ; 0x2c
            2a: R_AVR_13_PCREL    __epilogue_restores__+0x20
```

Entry point *scanf* refers to a short *prologue* section which saves a number of registers before calling the actual computational core *vfscanf;* the offset of the *rcall* instruction will (and can only) be resolved at link time

Object module file formats [2]

- The linker combines *object modules* which may be available in a variety of file formats such as the …

  … *Executable and Linking Format (ELF)*, developed by UNIX Systems Laboratories – ELF is now the de-facto standard for binaries on all UNIX and UNIX based systems such as GNU/Linux, etc.

  … *Common Object File Format (COFF);* former binary file format on UNIX (System V Release 3)

  … *Debug Information Format (DWARF-1/2);* often in combination with ELF or COFF

Object module file formats

- Example: *Object file formats supported WinAVR*

```
BFD header file version 2.14 20030612 + coff-avr-patch (20030831)

elf32-avr      (header little endian, data little endian)
coff-avr       (header little endian, data little endian)
coff-ext-avr   (header little endian, data little endian)
elf32-little   (header little endian, data little endian)
elf32-big      (header big endian, data big endian)
Srec           (header endianness unknown, data endianness unknown)
Symbolsrec     (header endianness unknown, data endianness unknown)
Tekhex         (header endianness unknown, data endianness unknown)
Binary         (header endianness unknown, data endianness unknown)
Ihex           (header endianness unknown, data endianness unknown)
```

- WinAVR supports 32-bit ELF files (both little endian / big endian), standard and extended COFF files, Motorola S-Records, Intel Hex files, etc.

Object module file formats

- The benefit of *standard file formats* is that the output file becomes platform independent and can be read and processed by tools of different manufacturers; for example, the UNIX tool *objdump* can make sense of the contents of an ELF/DWARF-2 file produced by a compiler for an Analog Devices SHARC Digital Signal Processor, etc.
- More importantly, when an object file adheres to the *DWARF* standard, it can be debugged using a large number of source level debuggers (e.g. GNU gdb, etc.); the code must be hardware independent though!

Object module file formats

- The *Motorola S-Record* and the *Intel HEX* format are two output formats which have been developed with the programming of PROM chips (Programmable Read Only Memory) in mind
- Knowledge of these format specifications can be very helpful when developing embedded software, especially when working with new hardware and/or within a new development environment
- The *S-Record / HEX file* is the lowest level at which aspects of a program can be analysed

Object module file formats

- Example: *Motorola S-Record*

Type	Record length	Address	Code/Data	Checksum

Type
S0 : Header
S1 : 2-byte address field
S2 : 3-byte address field
S3 : 4-byte address field
S9 : address field is a 2-byte entry point address; this is always the last record sent

Record length
Character pair which, when taken as hex value, represents the count of remaining character pairs in this record

Address
2, 3 or 4-byte address

Code/Data
… the actual machine code / data bytes

Checksum
Simple checksum to ensure data consistency

Object module file formats

- Example: *Motorola S-Record*

```
8000 cf 0b ff    lds  #stkbeg       ; initialize the stack pointer
8003 cd 0b df    ldy  #stkbeg-$20   ; initialize the data stack pointer
8006 06 80 09    jmp  _m0
8009 cc 00 08    ldd  #$0008
800c 5b 16       stab $0016

S1138000CF0BFFCD0BDF068009CC00085B16CC003C
```

- This is an *S1-record* with a total of *0x13* = 19 bytes to follow; the starting address is *0x8000* and the checksum (last byte) is *3C*.
- This information can become important to check where the compiler placed different code sections…

The Build Process

- Building software applications for embedded systems requires good knowledge of the development tools as well as the particularities of the targeted hardware
- The use of high-level languages (e.g. C, C++, Java) on microcontrollers can lead to situations where the system appears to behave in a 'weird' way
- Such *weird behaviour* often turns out to be connected to an insufficient knowledge of the build process the compiler/assembler/linker use to generate the program
- *As so often, lifelong learning is the key to success…*

Further reading:

[1]
GNU Binary Utilities, *Free Software Foundation,* www.gnu.org/software/binutils/manual/html_chapter/binutils.html, accessed: January 2005

[2]
ELF/DWARF, *Free Standards Group – Reference Specifications,* www.linuxbase.org/spec/refspecs/, accessed: January 2005

[3]
The GCC Project, *Free Software Foundation,* gcc.gnu.org/, accessed: January 2005