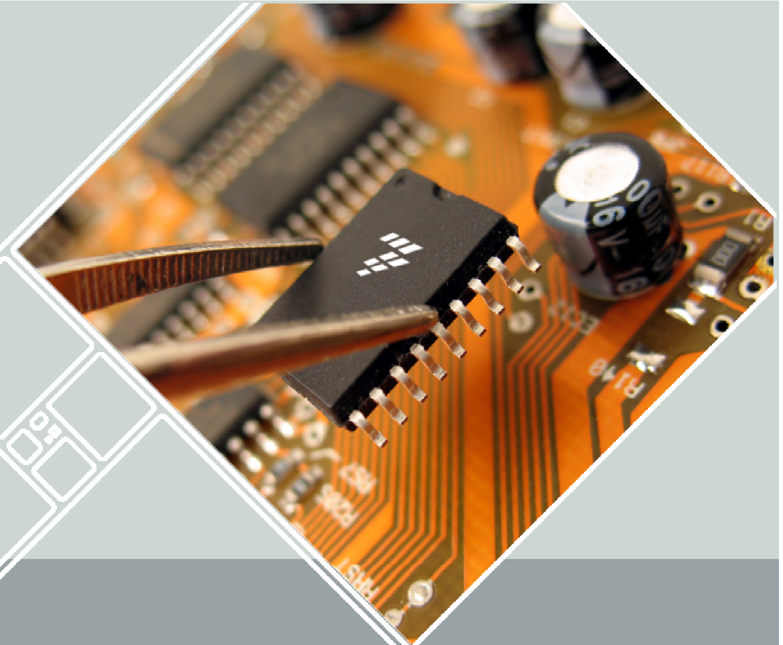


November 11, 2010

C for Embedded Systems Programming

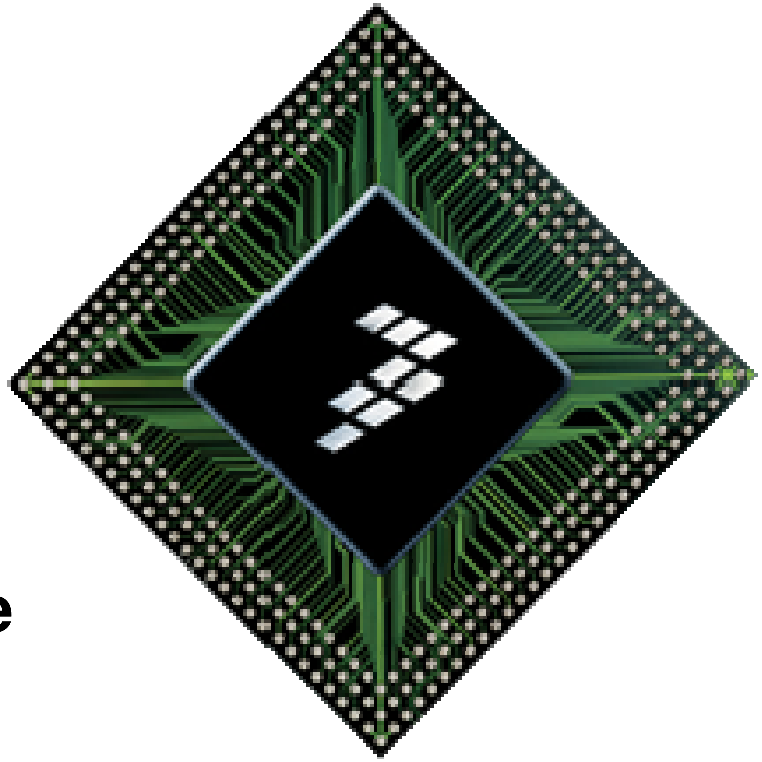
AMF-ENT-T0001

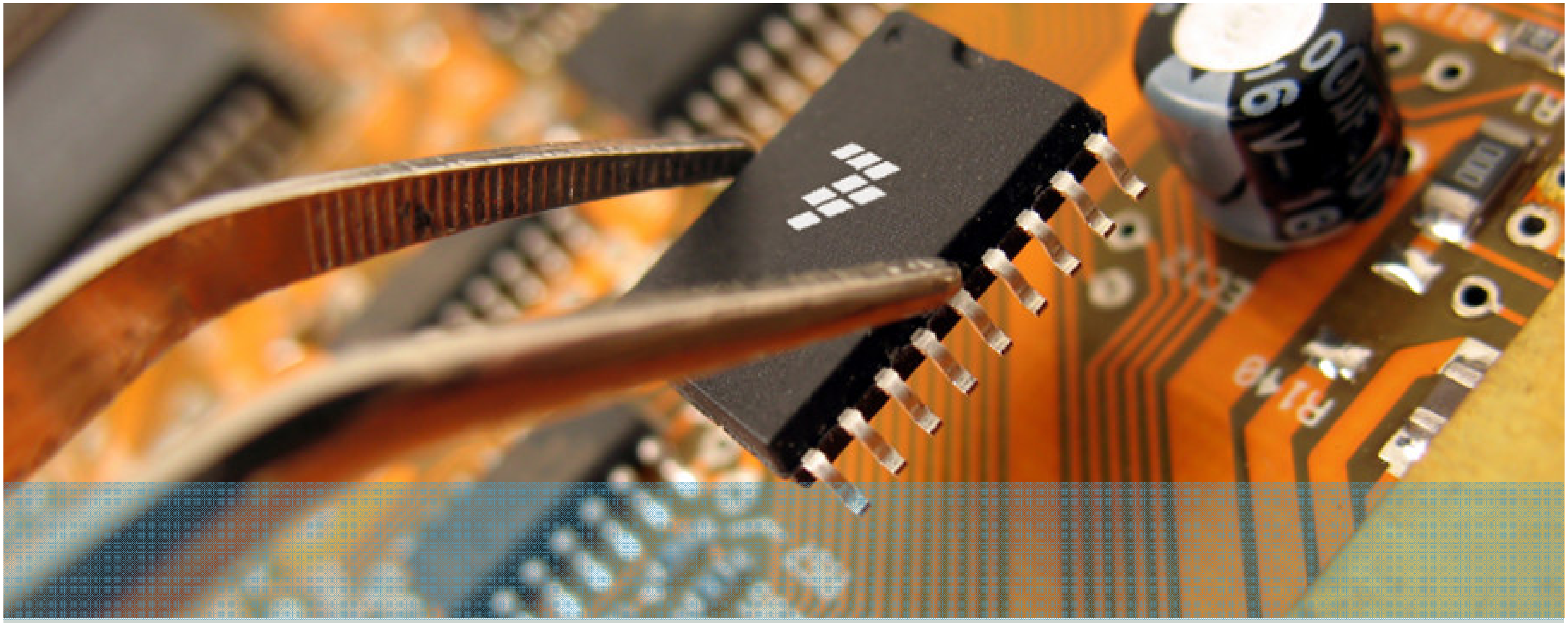
Derrick Klotz

Regional Field Applications Engineer

C Programming for Freescale's 8-bit S08 with Guidelines Towards Migrating to 32-bit Architecture

- ► **Knowing the environment**
  - Compiler and linker
  - .prm and map file
  - Programming models
- ► **Data types for embedded**
  - Choosing the right data type
  - Variable types
  - Storage class modifiers
- ► **Project Software Architecture**
  - Modular File Organization
  - Tips and considerations





Embedded C versus Desktop C

C for Embedded Systems Programming

freescale™
semiconductor

Introduction

► The ‘C’ Programming Language was originally developed for and implemented on the UNIX operating system, by Dennis Ritchie in 1971.

► One of the best features of C is that it is not tied to any particular hardware or system. This makes it easy for a user to write programs that will run without any changes on practically all machines.

► C is often called a middle-level computer language as it combines the elements of high-level languages with the functionalism of assembly language.

► To produce the most efficient machine code, the programmer must not only create an efficient high level design, but also pay attention to the detailed implementation.

Why Change to C?

- **C is much more flexible than other high-level programming languages:**
  - C is a structured language.
  - C is a relatively small language.
  - C has very loose data typing.
  - C easily supports low-level bit-wise data manipulation.
  - C is sometimes referred to as a “high-level assembly language”.
- **When compared to assembly language programming:**
  - Code written in C can be more reliable.
  - Code written in C can be more scalable.
  - Code written in C can be more portable between different platforms.
  - Code written in C can be easier to maintain.
  - Code written in C can be more productive.
- **C retains the basic philosophy that programmers know what they are doing.**
- **C only requires that they state their intentions explicitly.**
- **C program should be Clear, Concise, Correct, and Commented.**

Why Not C?

► These are some of the common issues that we encounter when considering moving to the C programming language:

- Big and inefficient code generation
- Fat code for the standard IO routines (printf, scanf, strcpy, etc…)
- The use of memory allocation: malloc(), alloc(), …
- The use of the stack is not so direct in C
- Data declaration in RAM and ROM
- Compiler optimizations
- Difficulty writing Interrupt Service Routines

► Many of these concerns are the result of failing to acknowledge the available resource differences between embedded microcontrollers and desktop computing environments

Embedded versus Desktop Programming

► **Main characteristics of an Embedded programming environment:**
- Limited ROM.
- Limited RAM.
- Limited stack space.
- Hardware oriented programming.
- Critical timing (Interrupt Service Routines, tasks, …).
- Many different pointer kinds (far / near / rom / uni / paged / …).
- Special keywords and tokens (@, interrupt, tiny, …).

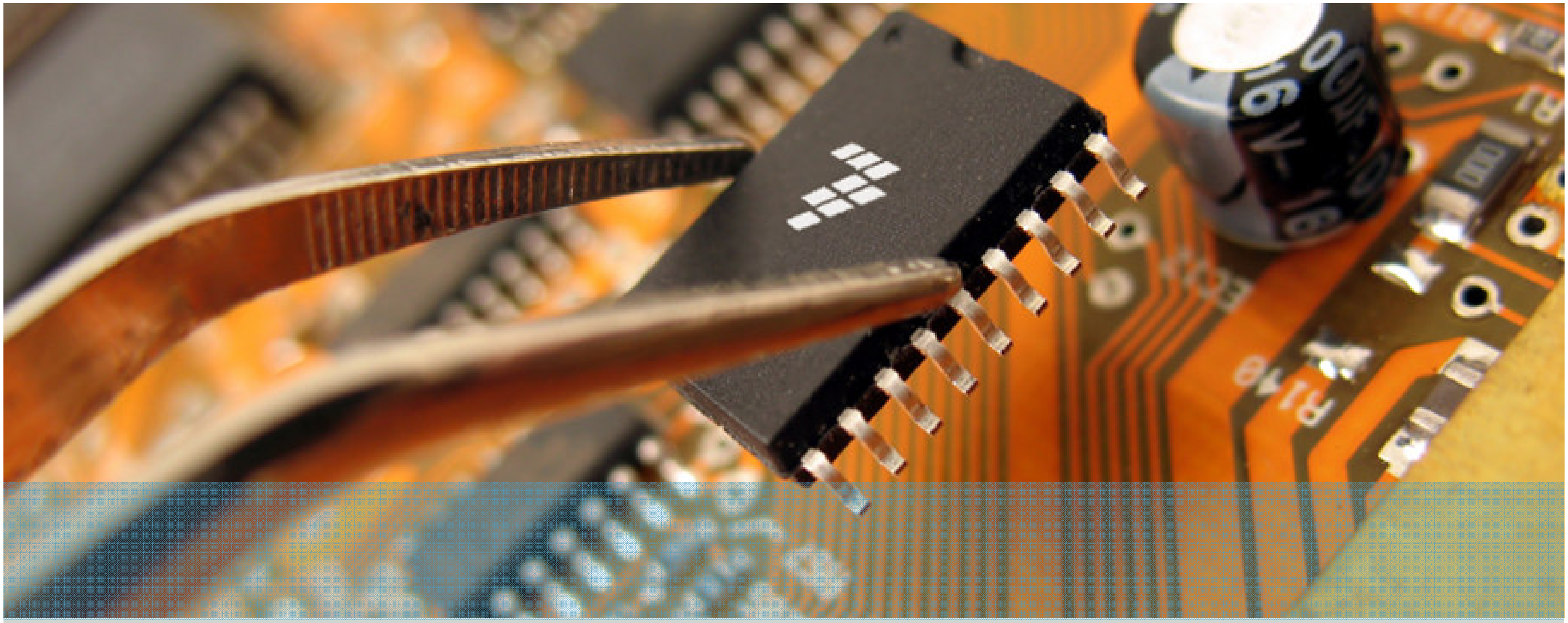
► **Successful Embedded C programs must keep the code small and "tight". In order to write efficient C code there has to be good knowledge about:**
- Architecture characteristics
- The tools for programming/debugging
- Data types native support
- Standard libraries
- Understand the difference between simple code vs. efficient code

Assembly Language versus C

- A compiler is no more efficient than a good assembly language programmer.

- It is much easier to write good code in C which can be converted to efficient assembly language code than it is to write efficient assembly language code by hand.

- C is a means to an end and not an end itself.



Knowing the Environment – Compiler & Linker

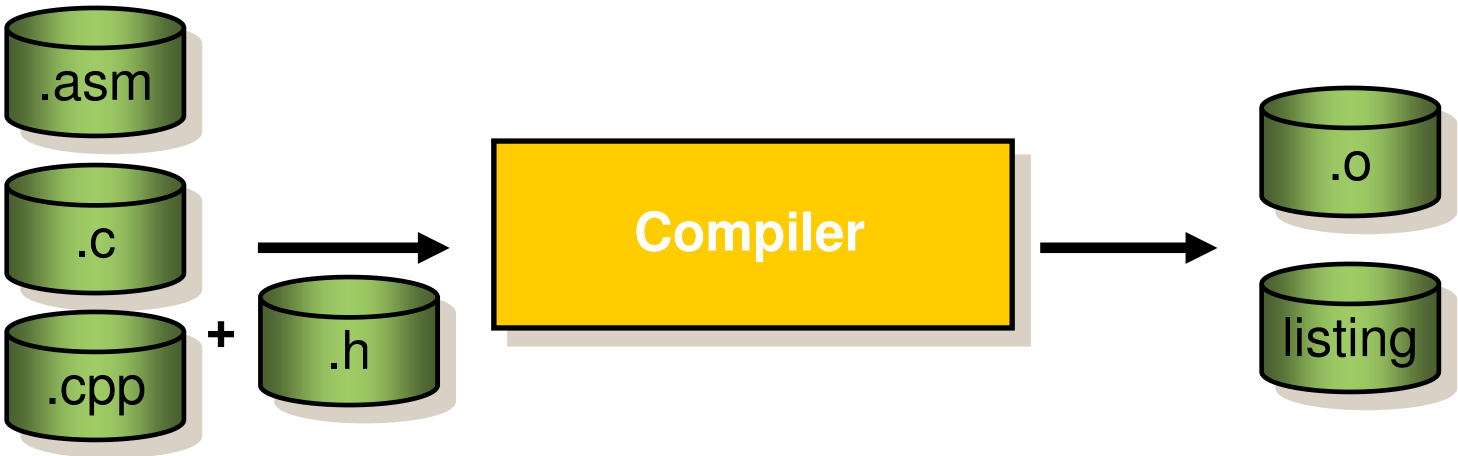
C for Embedded Systems Programming

Compiler's little Details

While choosing a compiler, you must remember that
the **Devil** is in the ***details***

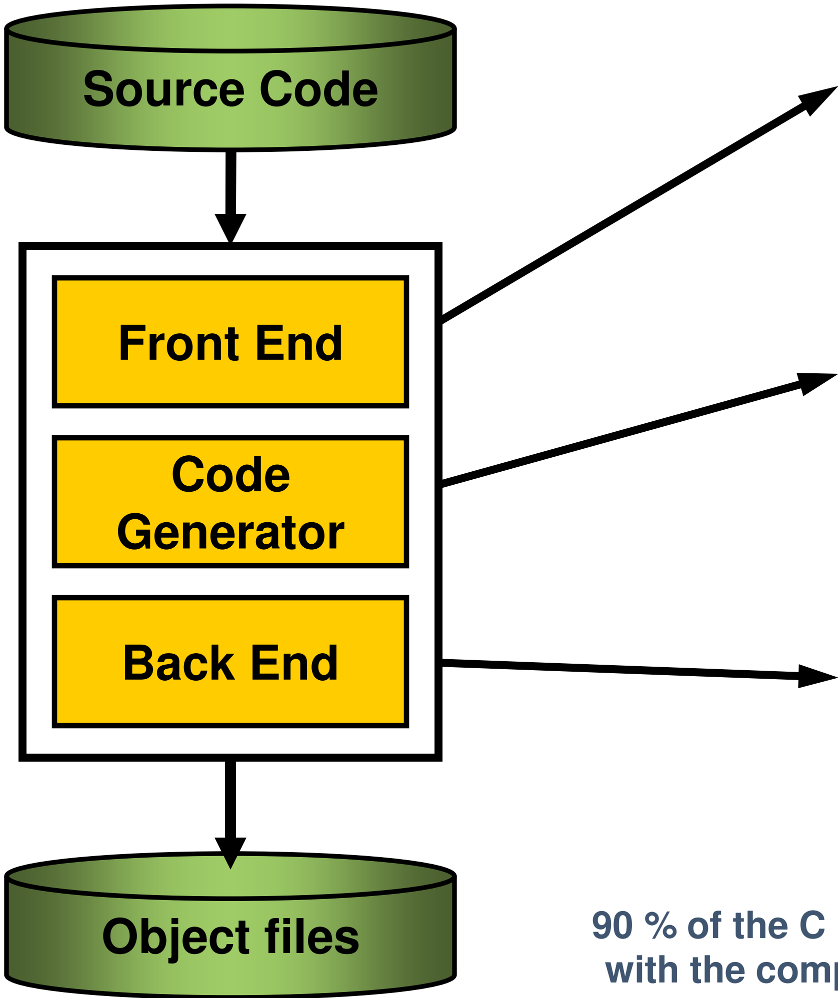
- Nice features that can make a huge difference:
  - ✓ Inline Assembly
  - ✓ Interrupt Functions
  - ✓ Assembly Language Generation
  - ✓ Standard Libraries
  - ✓ Startup code

Compiler Requirements



- Generate ROM-able code
- Generate Optimized code
- Generate Re-entrant code
- Support for Different Members in Microcontroller Family
- Support for Different Memory Models

Compiler – Internal view



Front End – reading th program:

- Identifies the language (C, C++ …)
- Prevents syntax errors
- Takes account of the preprocessing directives (macros and typedef resolutions, conditional compilation etc...)

Code Generator:

- Generate ROM-able code
- Generate optimized code according to the requested compiler options
- Generate re-entrant code

Back End:

- Support for different members in microcontroller family
- Support for different memory models

90 % of the C programming issues are user related, so just as with the compiler front end, when debugging an application, the first step is to carefully read the program

Overview

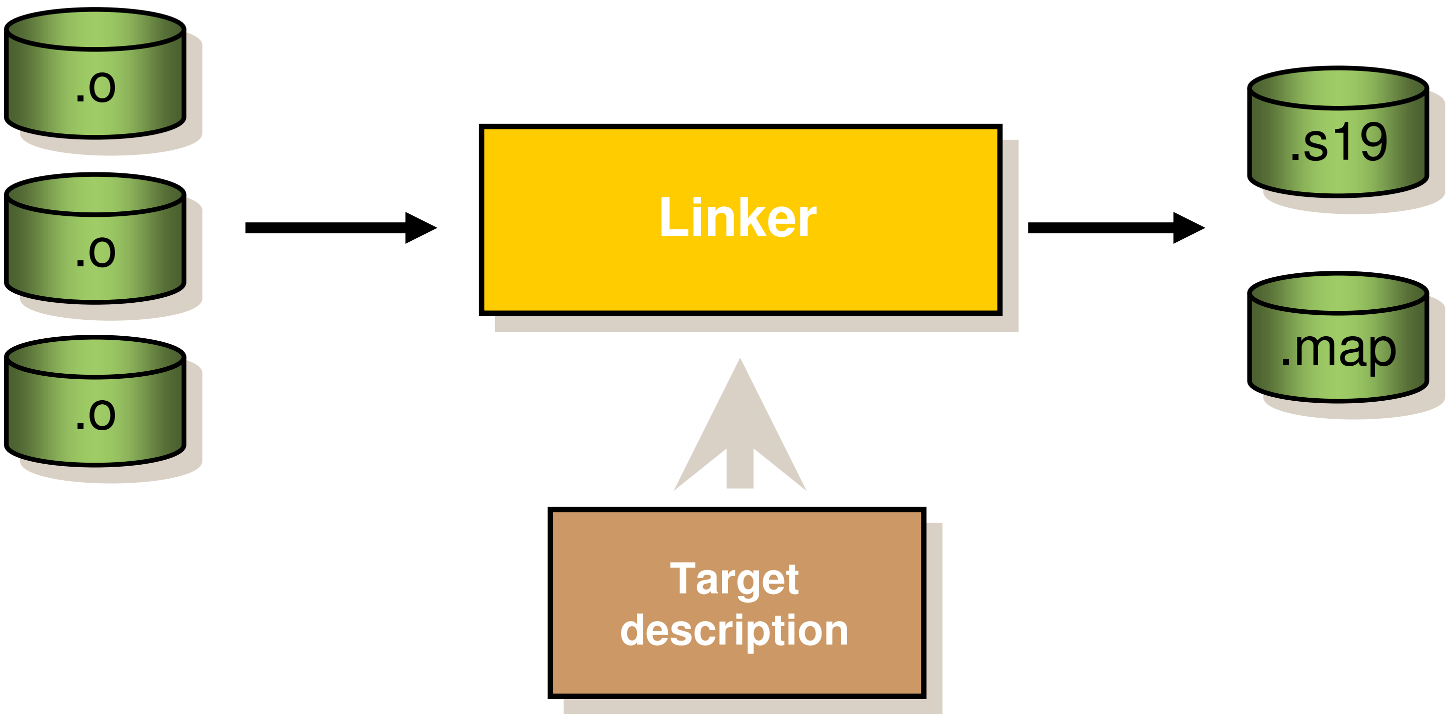
- After compiling process is done, the linker works with the object files generated in order to link the final application

- There are a couple of features that could be achieved by the linker because of the nature of the process

- The linker parameter file could be used to do some useful tricks that will aid during the development process

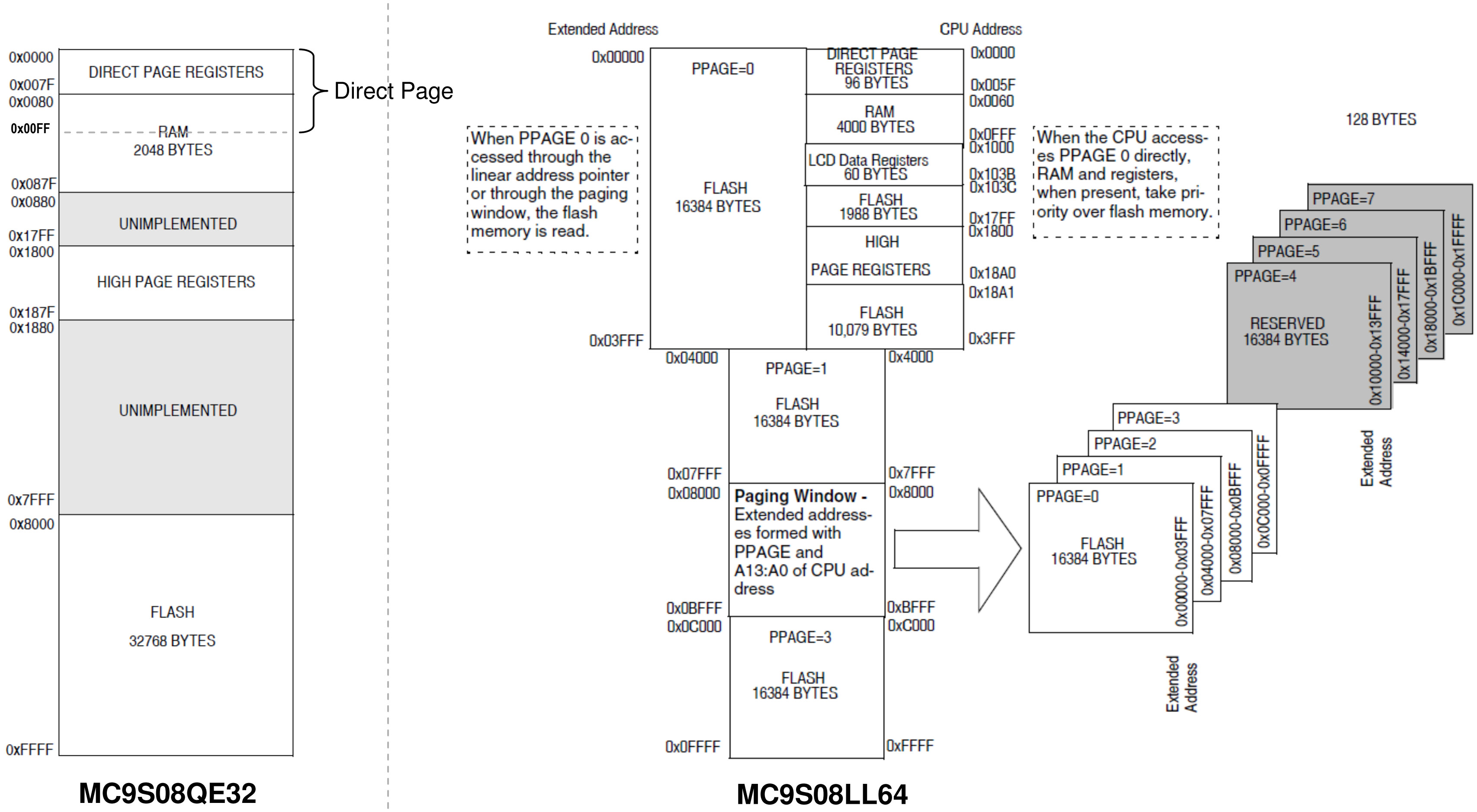
- Information provided by the linker could be used within applications

Linker Requirements

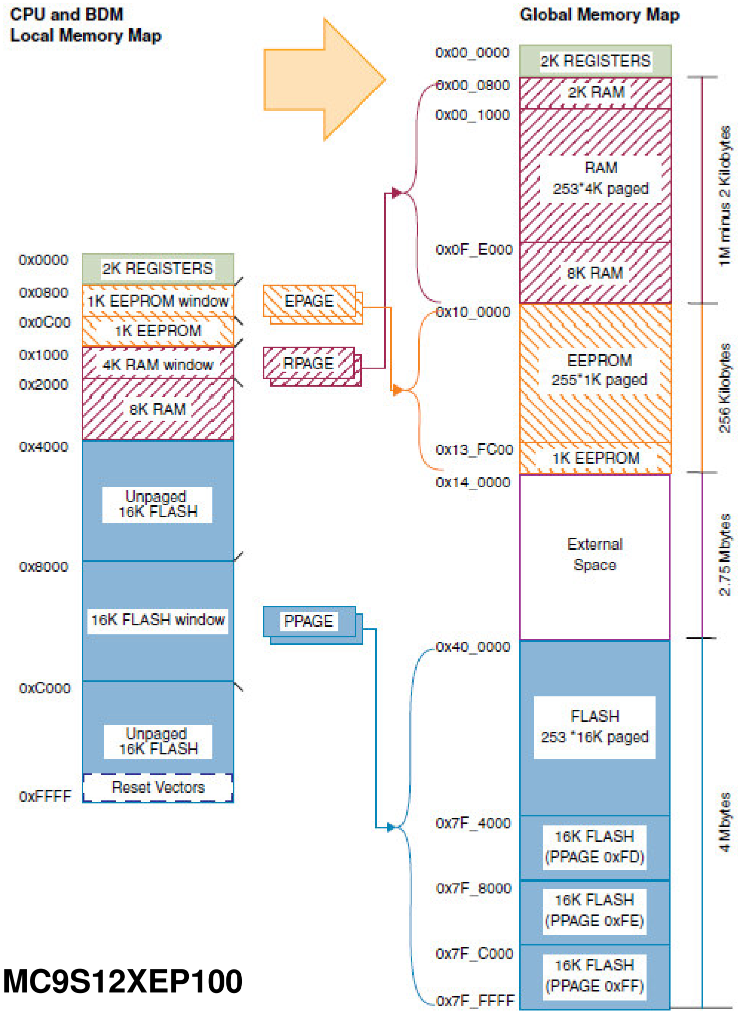


- Merging segments of code
- Allocate target memory (RAM, ROM, stack, special areas)
- Produce files for debugging (symbols, line numbers...)
- Produce files for target (mirror of memory)

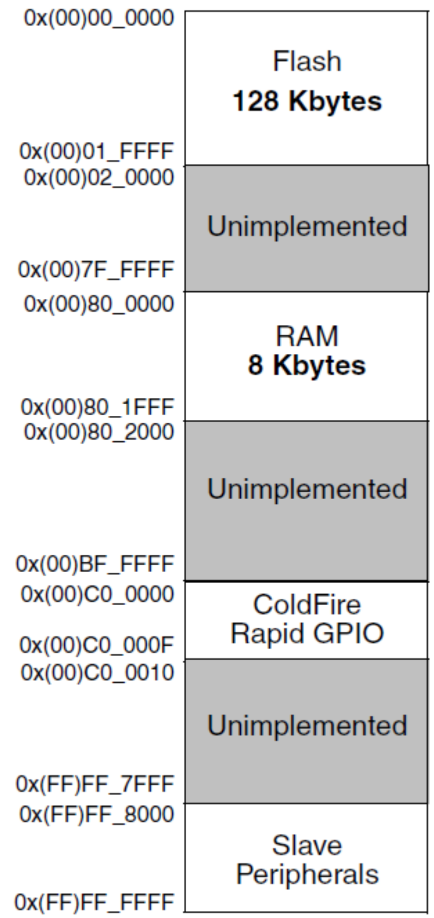
Target Description – S08



Target Description – S12X and ColdFire



MC9S12XEP100



ColdFire MCF51QE128

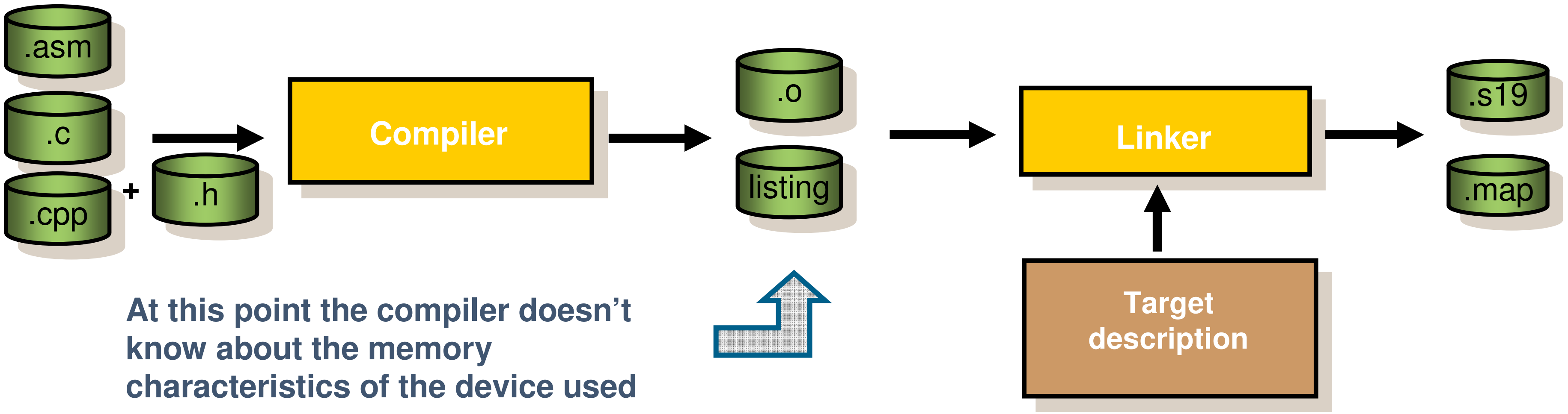
Memory Models

Memory models have a meaning depending on the target used

Model	HC(S)08	HC(S)12 S12X	Data	Function
Tiny	All data, including stack, must fit into the "zero page" pointers have 8-bit addresses unless is explicitly specified		8-bits unless specified with __far	16-bits
Small	All pointers and functions have 16-bit addresses. Code and data shall be located in a 64Kb address space	Data and functions are accessed by default with 16- bit addresses, code and data fit in 64 KB	16-bits unless specified	16-bits
Banked	This model uses the Memory Management Unit (MMU), allowing the extension of program space beyond the 64 KB	Data is also accessed with 16-bit addresses, functions are called using banked calls	16-bits	24-bits
Large		Both code and data are accessed using paged conventions	24-bits	24-bits

What about 32-bit architectures?

Tying Everything Together



At this point the compiler doesn't know about the memory characteristics of the device used

This means that the amount of memory used doesn't matter and the compiler will use a predefined convention to access data and functions

If the compiler doesn't know how the memory is arranged, how can we specify the calling convention that shall be used?

Far and Near

- The keywords far and near in C are intended to refer to data (either code or variables) within the same "memory block" or outside of the "memory block"
- Depending on the architecture, the "memory block" can mean different things

- Applied to functions
- __far and __near specify the calling convention. Far function calls can "cross" pages. Near function calls must stay in the same page

```
void main(void)
{
  MyFunction();
}
```

```
void  __far MyFunction (void);

CALL MyFunction, PAGE(MyFunction)
```

```
void  __near MyFunction (void);

JSR MyFunction
```

- Applied to variables
- A variable declared __near is considered by the compiler to be allocated in the memory section that can be accessed with direct addressing (first 256 bytes for S08, first 64 KB for S12 and S12X) accessing variables in the zero page generates less code and executes faster since the address is only 8-bits

Using “near” and “far”

► For both, code and data, near and far must be used in conjunction with a “#pragma” directive to specify the memory section to place them

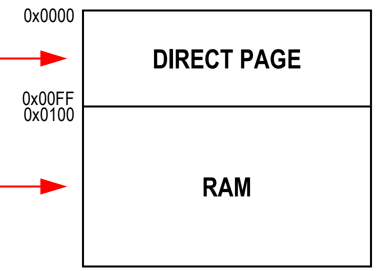
► For variables
- #pragma DATA_SEG <segment name>

► For code
- #pragma CODE_SEG <segment name>

► To understand what the segment name is we need to understand how the linker identifies the memory

Linker

```
#pragma DATA_SEG MY_ZEROPAGE
  char near var;

#pragma DATA_SEG DEFAULT_RAM
  char far var;
```



Compiler

```
char var;
var++;                  ldhx   #var
                        inc    ,x
---------------------------------------------
char near var;
var++;                  inc    var
---------------------------------------------
char far var;
var++;                  ldhx   #var
                        inc    ,x
```

MC9S08LL64 Linker Parameter File (.prm)

SEGMENTS SECTION

• Defines the memory available in the MCU, providing full control over memory allocation. This is essentially a translation of the data sheet.

PLACEMENT SECTION

• Provides the ability to assign each section from the application to specific memory segments. The names identified in this section are used in the source code, for example:

```
#pragma DATA_SEG MY_ZEROPAGE
```

STACKSIZE is one way to reserve a portion of memory for stack usage.

```
NAMES END /* CodeWarrior will pass all the needed files to the linker by command line. But

SEGMENTS /* Here all RAM/ROM areas of the device are listed. Used in PLACEMENT below. */
    Z_RAM                    =  READ_WRITE   0x0060 TO 0x00FF;
    RAM                      =  READ_WRITE   0x0100 TO 0x0FFF;
    /* unbanked FLASH ROM */
    ROM                      =  READ_ONLY    0x18A1 TO 0x7FFF;
    ROM1                     =  READ_ONLY    0x103C TO 0x17FF;
    ROM2                     =  READ_ONLY    0xC000 TO 0xFFAB;
    ROM3                     =  READ_ONLY    0xFFC0 TO 0xFFD1;
    /* banked FLASH ROM */
    PPAGE_0                  =  READ_ONLY    0x008002 TO 0x00903B; /* PAGE partially containe
    PPAGE_0_1                =  READ_ONLY    0x009800 TO 0x0098A0;
    PPAGE_2                  =  READ_ONLY    0x028000 TO 0x02BFFF;
 /* PPAGE_1                  =  READ_ONLY    0x018000 TO 0x01BFFF; PAGE already contained in
 /* PPAGE_3                  =  READ_ONLY    0x038000 TO 0x03BFFF; PAGE already contained in
END

PLACEMENT /* Here all predefined and user segments are placed into the SEGMENTS defined above
    DEFAULT_RAM,                        /* non-zero page variables */
                                        INTO  RAM;

    _PRESTART, STARTUP,                 /* startup code and data structures */
    ROM_VAR,                            /* constant variables */
    STRINGS,                            /* string literals */
    VIRTUAL_TABLE_SEGMENT,              /* C++ virtual table segment */
    NON_BANKED,                         /* runtime routines which must not be banked */
    DEFAULT_ROM,
    COPY                                /* copy down information: how to initialize variable
                                        INTO  ROM; /* ,ROM1,ROM2,ROM3: To use "ROM1,ROM2,ROM

    PAGED_ROM                           /* routines which can be banked */
                                        INTO  PPAGE_2,ROM1,ROM2,ROM3,PPAGE_0,PPAGE_0_1;

    _DATA_ZEROPAGE,                     /* zero page variables */
    MY_ZEROPAGE                         INTO  Z_RAM;
END

STACKSIZE 0x100
VECTOR 0 _Startup /* Reset vector: this is the default entry point for an application. */
```

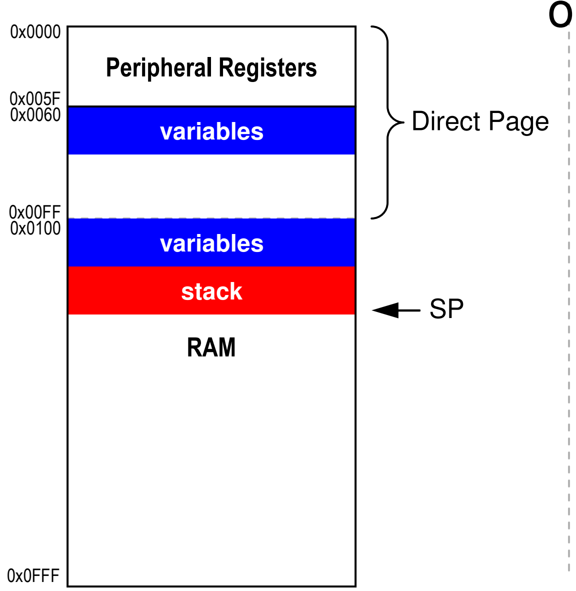
Controlling the Location of the Stack

```
SEGMENTS
  Z_RAM       = READ_WRITE  0x0060 TO 0x00FF;
  RAM         = READ_WRITE  0x0100 TO 0x0FFF;
END
```

```
STACKSIZE 0x100
```

```
#pragma DATA_SEG MY_ZEROPAGE
```

```
#pragma DATA_SEG DEFAULT_RAM
```



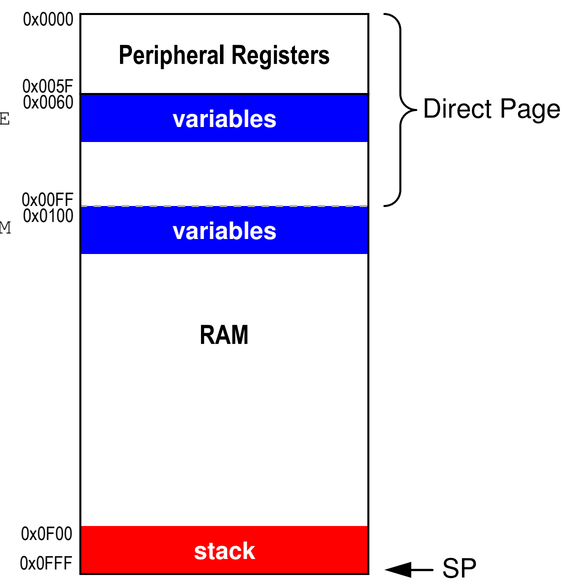
or

```
SEGMENTS
  Z_RAM       = READ_WRITE  0x0060 TO 0x00FF;
  RAM         = READ_WRITE  0x0100 TO 0x0EFF;
  STACK_RAM   = READ_WRITE  0x0F00 TO 0x0FFF;
END

PLACEMENT
  SSTACK      INTO  STACK_RAM;
END
```

```
STACKTOP 0x0FFF
```

```
#pragma DATA_SEG MY_ZEROPAGE
```

```
#pragma DATA_SEG DEFAULT_RAM
```



ColdFire MCF51QE128 Linker Control File (.lcf)

MEMORY SEGMENT
- Describes the available memory

SECTIONS SEGMENT
- Defines the contents of memory sections and global symbols

```
# Sample Linker Command File for CodeWarrior for ColdFire MCF51QE128

# Memory ranges

MEMORY {
   code        (RX)  : ORIGIN = 0x00000410, LENGTH = 0x0001FBF0
   userram     (RWX) : ORIGIN = 0x00800000, LENGTH = 0x00002000
}

SECTIONS {

# Heap and Stack sizes definition
  ___heap_size     = 0x0400;
  ___stack_size    = 0x0400;

# MCF51QE128 Derivative Memory map definitions from linker command files:
# ___RAM_ADDRESS, ___RAM_SIZE, ___FLASH_ADDRESS, ___FLASH_SIZE linker
# symbols must be defined in the linker command file.

# 8 Kbytes Internal SRAM
   ___RAM_ADDRESS = 0x00800000;
   ___RAM_SIZE    = 0x00002000;

# 128 KByte Internal Flash Memory
   ___FLASH_ADDRESS  = 0x00000000;
   ___FLASH_SIZE     = 0x00020000;


.........
```

How can we verify where the Linker put our code and data?

The Map file

The Map File

TARGET SECTION
• Names the target processor and memory model

FILE SECTION
• Lists the names of all files from which objects were used

STARTUP SECTION
• Lists the prestart code and the values used to initialize the startup descriptor "_startupData". The startup descriptor is listed member by member with the initialization data at the right hand side of the member name

SECTION-ALLOCATION SECTION
• Lists those segments for which at least one object was allocated

```
*********************************************************************************************
TARGET SECTION
---------------------------------------------------------------------------------------------
Processor   : Freescale HC08
Memory Model: SMALL
File Format : ELF\DWARF 2.0
Linker      : SmartLinker V-5.0.39 Build 10132, May 13 2010

*********************************************************************************************
FILE SECTION
---------------------------------------------------------------------------------------------
main.obj                                Model: SMALL,          Lang: ANSI-C
start08.obj                             Model: SMALL,          Lang: ANSI-C
mc9s08ll64.obj                          Model: SMALL,          Lang: ANSI-C

*********************************************************************************************
STARTUP SECTION
---------------------------------------------------------------------------------------------
Entry point: 0x191C (_Startup)
_startupData is allocated at 0x1925 and uses 6 Bytes
extern struct _tagStartup {
  unsigned nofZeroOut     1
  _Range   pZeroOut       0x60     1
  _Copy    *toCopyDownBeg 0x1945
} _startupData;

*********************************************************************************************
SECTION-ALLOCATION SECTION
Section Name                    Size  Type     From       To       Segment
---------------------------------------------------------------------------------------------
.init                            132     R     0x18A1     0x1924   ROM
.startData                        14     R     0x1925     0x1932   ROM
.text                             18     R     0x1933     0x1944   ROM
.copy                              2     R     0x1945     0x1946   ROM
MY_ZEROPAGE                        1   R/W       0x60       0x60   Z_RAM
.abs_section_0                     1   N/I        0x0        0x0   .absSeg0
.abs_section_1                     1   N/I        0x1        0x1   .absSeg1
.abs_section_2                     1   N/I        0x2        0x2   .absSeg2
.abs_section_3                     1   N/I        0x3        0x3   .absSeg3
```

The Map File

.init = _PRESTART
.startData = STARTUP
.text = DEFAULT_ROM
.copy = COPY
.stack = SSTACK
.data = DEFAULT_RAM
.common = DEFAULT_RAM

Every variable allocated with an absolute syntax of the kind:

type variablename @0xABCD;

will have a "section" and a "segment" assigned for its own.

All MCU registers are declared this way. This is why we have one abs_section for every MCU register.

SECTION-ALLOCATION SECTION

- Lists those segments for which at least one object was allocated

Linker Parameter file

```
PLACEMENT /* Here all predefined and user segments are placed into the SEGMENTS defined above. */
    DEFAULT_RAM,                        /* non-zero page variables */
                                        INTO  RAM;

    _PRESTART,                          /* startup code */
    STARTUP,                            /* startup data structures */
    ROM_VAR,                            /* constant variables */
    STRINGS,                            /* string literals */
    VIRTUAL_TABLE_SEGMENT,              /* C++ virtual table segment */
    NON_BANKED,                         /* runtime routines which must not be banked */
    DEFAULT_ROM,
    COPY                                /* copy down information: how to initialize variables */
                                        INTO  ROM; /* ,ROM1,ROM2,ROM3: To use "ROM1,ROM2,ROM3" as

    PAGED_ROM                           /* routines which can be banked */
                                        INTO  PPAGE_2,ROM1,ROM2,ROM3,PPAGE_0,PPAGE_0_1;

    _DATA_ZEROPAGE,                     /* zero page variables */
    MY_ZEROPAGE                         INTO  Z_RAM;
END
```

```
*********************************************************************************************
SECTION-ALLOCATION SECTION
Section Name                    Size  Type     From       To       Segment
---------------------------------------------------------------------------------------------
.init                            132    R      0x18A1     0x1924   ROM
.startData                        14    R      0x1925     0x1932   ROM
.text                             18    R      0x1933     0x1944   ROM
.copy                              2    R      0x1945     0x1946   ROM
MY_ZEROPAGE                        1   R/W       0x60       0x60   Z_RAM
.abs_section_0                     1   N/I        0x0        0x0   .absSeg0
.abs_section_1                     1   N/I        0x1        0x1   .absSeg1
.abs_section_2                     1   N/I        0x2        0x2   .absSeg2
.abs_section_3                     1   N/I        0x3        0x3   .absSeg3
```

The Map File

SECTION-ALLOCATION SECTION Summary:

- READ_ONLY = Flash
- READ_WRITE = RAM
- NO_INIT = Registers

VECTOR-ALLOCATION SECTION

- Lists each vector's address and to where it points (ie., interrupt service routine)

OBJECT-ALLOCATION SECTION

- Contains the name, address and size of every allocated object and groups them by module

```
.stack                         256    R/W      0x100       0x1FF    RAM
.vectSeg188_vect                 2      R     0xFFFE      0xFFFF    .vectSeg188

Summary of section sizes per section type:
READ_ONLY (R):           A8 (dec:      168)
READ_WRITE (R/W):       101 (dec:      257)
NO_INIT (N/I):           CD (dec:      205)

*********************************************************************************************
VECTOR-ALLOCATION SECTION
    Address      InitValue   InitFunction
---------------------------------------------------------------------------------------------
     0xFFFE         0x191C    _Startup

*********************************************************************************************
OBJECT-ALLOCATION SECTION
     Name                Module                  Addr   hSize   dSize     Ref    Section   RLIB
---------------------------------------------------------------------------------------------
MODULE:                -- main.obj --
- PROCEDURES:
     main                                        1933      12      18       1   .text
- VARIABLES:
     n                                             60       1       1       2   MY_ZEROPAGE
MODULE:                -- start08.obj --
- PROCEDURES:
     loadByte                                    18A1       E      14       5   .init
     Init                                        18AF      6D     109       1   .init
     _Startup                                    191C       9       9       0   .init
- VARIABLES:
     _startupData                                1925       6       6       4   .startData
- LABELS:
     __SEG_END_SSTACK                             200       0       0       1
MODULE:                -- mc9s08ll64.obj --
- PROCEDURES:
- VARIABLES:
     _PTAD                                          0       1       1       0   .abs_section_0
     _PTADD                                         1       1       1       0   .abs_section_1
     _PTBD                                          2       1       1       0   .abs_section_2
     _PTBDD                                         3       1       1       0   .abs_section_3
```

The Map File

UNUSED-OBJECTS SECTION:
- Shows all of the variables declared but not used after the optimizer did its job

COPYDOWN SECTION
- Lists each pre-initialized variable and its value

OBJECT-DEPENDENCIES SECTION
- Lists for every function and variable that uses other global objects the names of these global objects

DEPENDENCY TREE
- Using a tree format, shows all detected dependencies between functions. Overlapping local variables are also displayed at their defining function

STATISTICS SECTION
- Delivers information like the number of bytes of code in the application

```
*********************************************************************************************
UNUSED-OBJECTS SECTION
---------------------------------------------------------------------------------------------

*********************************************************************************************
COPYDOWN SECTION
---------------------------------------------------------------------------------------------
------- ROM-ADDRESS: 0x1945 ---- SIZE       2 ---
Filling bytes inserted
 0000

*********************************************************************************************
OBJECT-DEPENDENCIES SECTION
---------------------------------------------------------------------------------------------
Init                         USES _startupData loadByte
_Startup                     USES __SEG_END_SSTACK Init main
main                         USES _PTCD _PTCDD _SRS n

*********************************************************************************************
DEPENDENCY TREE
*********************************************************************************************
 main and _Startup Group
 |
 +- main
 |
 +- _Startup
    |
    +- Init
    |  |
    |  +- loadByte
    |
    +- main                  (see above)

*********************************************************************************************
STATISTIC SECTION
---------------------------------------------------------------------------------------------
ExeFile:
Number of blocks to be downloaded: 4
Total size of all blocks to be downloaded: 168
```

Configuring the Map File

- The information displayed in the Map File can be configured by adding a MAPFILE in the .prm file. Only the modules listed will be displayed. For example:

```
MAPFILE FILE SEC_ALLOC OBJ_UNUSED COPYDOWN
```

- If no MAPFILE line is added, all information is included by default

Specifier	Description
ALL	Generates a map file containing all available information
COPYDOWN	Writes information about the initialization value for objects allocated in RAM (COPYDOWN section)
FILE	Includes information about the files building the application (FILE section)
OBJ_ALLOC	Includes information about the allocated objects (OBJECT ALLOCATION section)
SORTED_OBJECT_LIST	Generates a list of all allocated objects, sorted by address (OBJECT LIST SORTED BY ADDRESS section)
OBJ_UNUSED	Includes a list of all unused objects (UNUSED OBJECTS section)
OBJ_DEP	Includes a list of dependencies between the objects in the application (OBJECT DEPENDENCY section)

Specifier	Description
NONE	Generates no map file
DEPENDENCY_TREE	Shows the allocation of overlapped variables (DEPENDENCY TREE section)
SEC_ALLOC	Includes information about the sections used in the application (SECTION ALLOCATION section)
STARTUP_STRUCT	Includes information about the startup structure (STARTUP section)
MODULE_STATISTIC	Includes information about how much ROM/RAM specific modules (compilation units) use
STATISTIC	Includes statistic information about the link session (STATISTICS section)
TARGET	Includes information about the target processor and memory model (TARGET section)

Lab1

```
byte n;
byte var;

void main(void)
{
  EnableInterrupts;

  // initialize LEDs
  PTCD = 0xFF;            // set default value for Port C
  PTCDD = 0b00111100;     // set LED port pins as outputs

  for (;;)
  {
    __RESET_WATCHDOG();   /* feeds the dog */

    for (n=0;;n++)
    {
            PTCD++;       // blink LEDs
            var++;
    }

  } /* loop forever */
  /* please make sure that you never leave main */
}
```

Lab1

```c
#pragma DATA_SEG MY_ZEROPAGE
byte near n;

#pragma DATA_SEG DEFAULT_RAM
byte far var = 7;

void main(void)
{
  EnableInterrupts;

  // initialize LEDs
  PTCD = 0xFF;            // set default value for Port C
  PTCDD = 0b00111100;     // set LED port pins as outputs

  for (;;)
  {
    __RESET_WATCHDOG();   /* feeds the dog */

    for (n=0;;n++)
    {
            PTCD++;       // blink LEDs
            var++;
    }

  } /* loop forever */
  /* please make sure that you never leave main */
}
```

Start08.c – Startup Routine

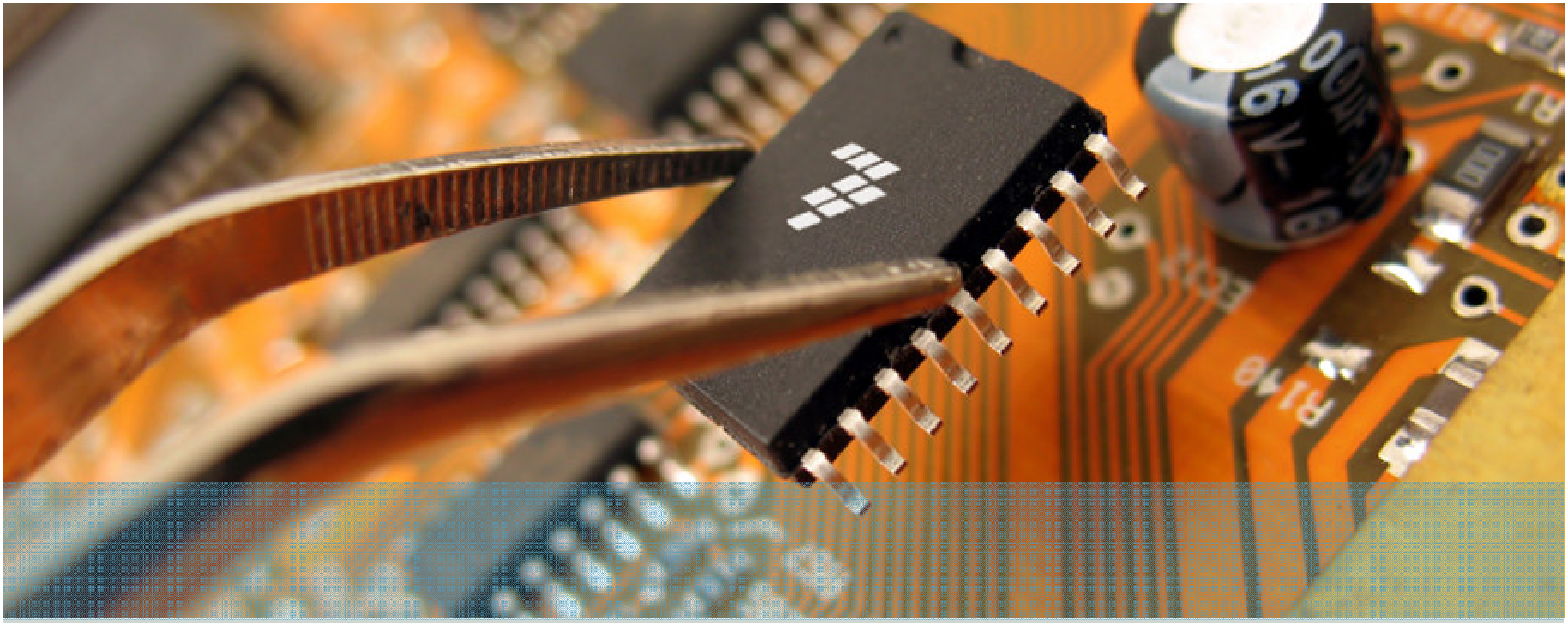
- Do we need the startup code? What does it do?

1. Stack Pointer/Frame setup ✓
2. Global Memory initialized to zero (Zero-Out) ✓
3. ~~Global Variables initialized (Copy-Down)~~
4. ~~Global Constructor calls (C++)~~
5. Call `main()` ✓

Start all Static and Global variables at zero

There is a simpler way …

```
  asm {
        clra                                    ; get clear data
        ldhx    #MAP_RAM_last                   ; point to last RAM location
        stx     MAP_RAM_first                   ; first RAM location is non-zero
        txs                                     ; initialize SP
ClearRAM:
        psha                                    ; clear RAM location
        tst     MAP_RAM_first                   ; check if done
        bne     ClearRAM                        ; loop back if not
  }

  INIT_SP_FROM_STARTUP_DESC();                  // initialize SP
  __asm jmp     main;                           // jump into main()
```



Variable Data Types

C for Embedded Systems Programming

freescale™
semiconductor

Variables

- The *type* of a variable determines what kinds of values it may take on.
- In other words, selecting a type for a variable is closely connected to the way(s) we'll be using that variable.
- There are only a few basic data types in C:

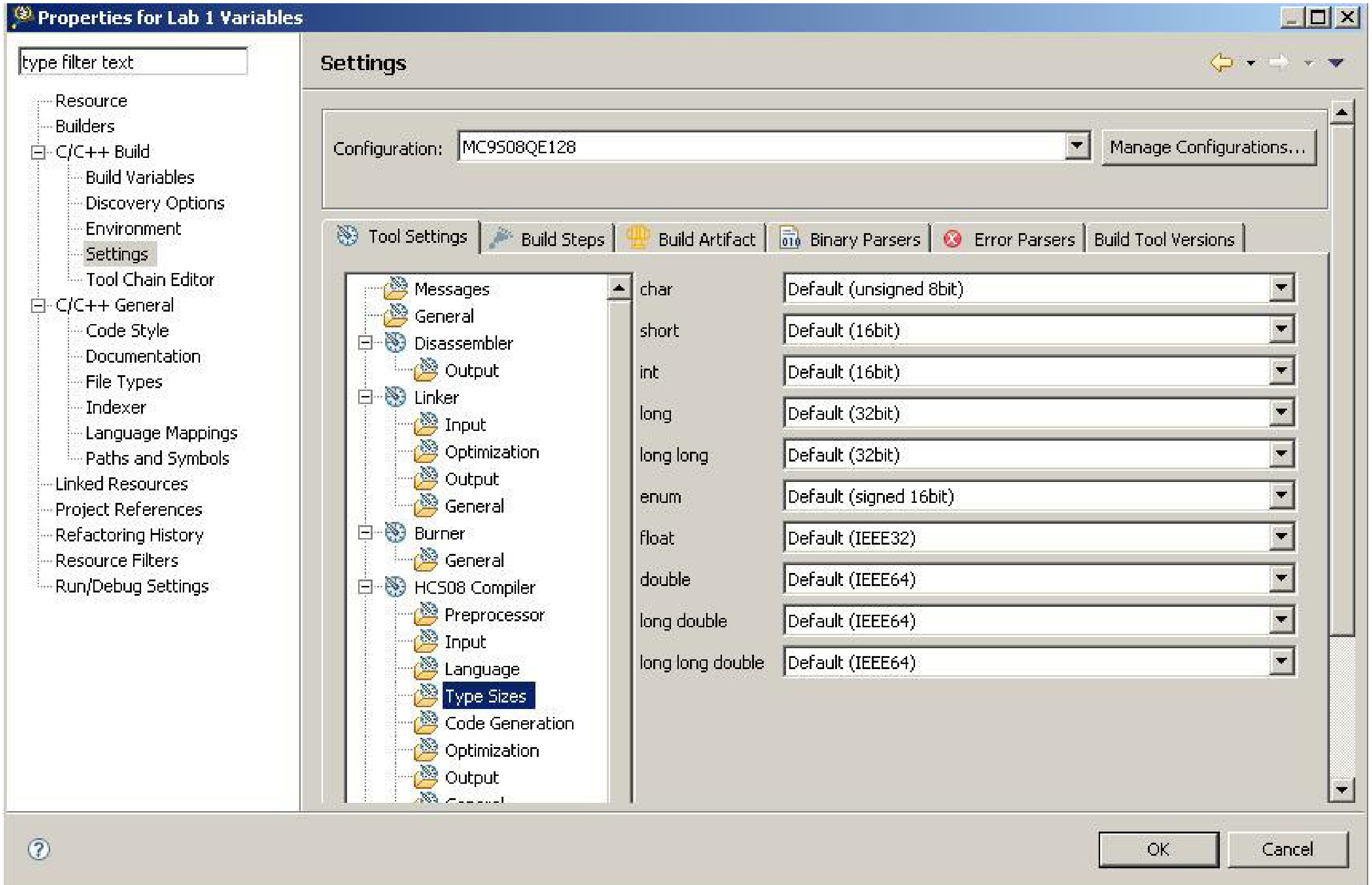
Type	Default format	Default value range		Formats available with option -T
		min	max	
char (unsigned)	8 bit	0	255	8 bit, 16 bit, 32 bit
signed char	8 bit	-128	127	8 bit, 16 bit, 32 bit
unsigned char	8 bit	0	255	8 bit, 16 bit, 32 bit
signed short	16 bit	-32768	32767	8 bit, 16 bit, 32 bit
unsigned short	16 bit	0	65535	8 bit, 16 bit, 32 bit
enum (signed)	16 bit	-32768	32767	8 bit, 16 bit, 32 bit
signed int	16 bit	-32768	32767	8 bit, 16 bit, 32 bit
unsigned int	16 bit	0	65535	8 bit, 16 bit, 32 bit
signed long	32 bit	-2147483648	2147483647	8 bit, 16 bit, 32 bit
unsigned long	32 bit	0	4294967295	8 bit, 16 bit, 32 bit
signed long long	32 bit	-2147483648	2147483647	8 bit, 16 bit, 32 bit
unsigned long long	32 bit	0	4294967295	8 bit, 16 bit, 32 bit

Note:
All scalar types are signed by default, except char

Size of int type is machine dependant

CodeWarrior™ Data Types

► The ANSI standard does not precisely define the size of its native types, but CodeWarrior does…



Data Type Facts

- The greatest savings in code size and execution time can be made by choosing the most appropriate data type for variables
  - For example, the natural data size for an 8-bit MCU is an 8-bit variable
  - The C preferred data type is ‘int’
  - In 16-bit and 32-bit architectures it is possible to have ways to address either 8- or 16-bits data efficiently but it is also possible that they are not addressed efficiently
  - Simple concepts like choosing the right data type or memory alignment can result into big improvements
  - Double precision and floating point should be avoided wherever efficiency is important

Data Type Selection

- ► Mind the architecture
  - The same C source code could be efficient or inefficient
  - The programmer should keep in mind the architecture's typical instruction size and choose the appropriate data type accordingly
- ► Consider:

A++;

	8-bit S08	16-bit S12X	32-bit ColdFire
char near A;	inc A	inc A	move.b A(a5),d0 addq.l #1,d0 move.b d0,A(a5)
unsigned int A;	ldhx @A inc 1,x bne Lxx inc ,x Lxx:	incw A	addq.l #1,_A(a5)
unsigned long A;	ldhx @A jsr _LINC	ldd A:2 ldx A jsr _LINC std A:2 stx A	addq.l #1,_A(a5)

Data Type Selection

- There are 3 Rules for data type selection:
  - Use the smallest possible type to get the job done
  - Use unsigned type if possible
  - Use casts within expressions to reduce data types to the minimum required

- Use typedefs to get fixed size
  - Change according to compiler and system
  - Code is invariant across machines
  - Used when a fixed number of bits is needed for values

- Avoid basic types (‘char’, ‘int’, ‘short’, ‘long’) in application code

8-bit machine

```
/* Fixed size types */
typedef unsigned char uint8_t;
typedef int           int16_t;
typedef unsigned long uint32_t;
```

32-bit machine

```
/* Fixed size types */
typedef unsigned char uint8_t;
typedef short         int16_t;
typedef unsigned int  uint32_t;
```

Data Type Naming Conventions

- ► Avoid basic types (‘char’, ‘int’, ‘short’, ‘long’) in application code
- ► But how?

Basic CodeWarrior Stationary:

```
/* Types definition */
typedef unsigned char byte;
typedef unsigned int word;
typedef unsigned long dword;
typedef unsigned long dlong[2];
```

Processor Expert CodeWarrior Stationary:

```
#ifndef __PE_Types_H
#define __PE_Types_H

/* Types definition */
typedef unsigned char bool;
typedef unsigned char byte;
typedef unsigned int word;
typedef unsigned long dword;
typedef unsigned long dlong[2];

// other stuff

#endif /* __PE_Types_H */
```

Another Popular Technique:

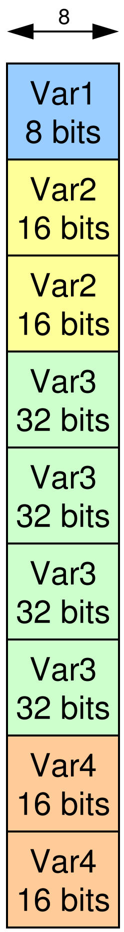
```
typedef unsigned char   __uint8__;
typedef unsigned char   __byte__;
typedef signed char     __int8__;
typedef unsigned short  __uint16__;
typedef signed short    __int16__;
typedef unsigned long   __uint32__;
typedef signed long     __int32__;
```

Recall:

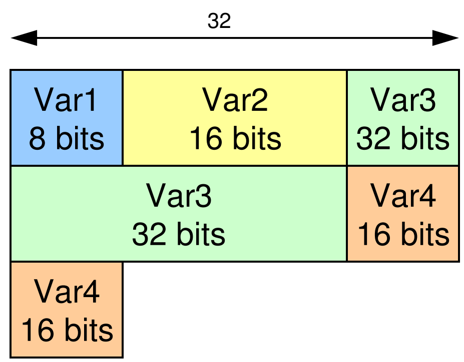
C retains the basic philosophy that programmers know what they are doing
C only requires that they state their intentions explicitly

C provides a lot of flexibility; this can be good or bad

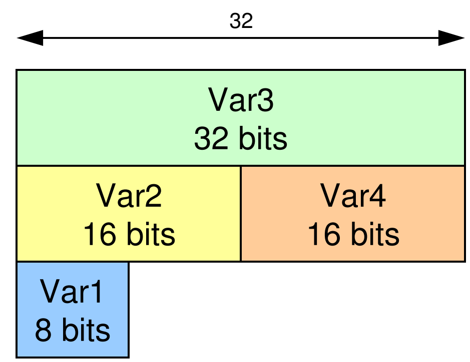
Memory Alignment

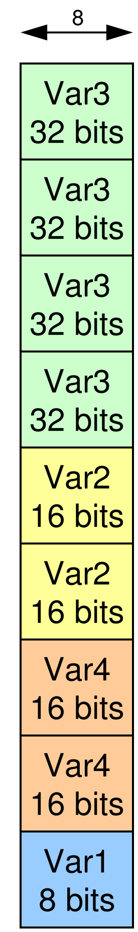



```
typedef struct
{
  byte Var1;
  word Var2;
  dword Var3;
  word Var4;
} tData;
```




```
typedef struct
{
  dword Var3;
  word Var2;
  word Var4;
  byte Var1;
} tData;
```





- Memory alignment can be simplified by declaring first the 32-bit variables, then 16-bit, then 8-bit.
- Porting this to a 32-bit architecture ensures that there is no misaligned access to variables, thereby saving processor time.
- Organizing structures like this means that we're less dependent upon tools that may do this automatically – and may actually help these tools.

Variable Types

- Global
  - Global storage and global scope
- Static
  - Global storage and local scope
- Local
  - Local storage and local scope

Storage Class Modifiers

► The following keywords are used with variable declarations, to specify specific needs or conditions associated with the storage of the variables in memory:

static

volatile

const

These three key words, together, allow us to write not only *better* code, but also ***tighter*** and more ***reliable*** code

Static Variables

- When applied to variables, static has two primary functions:
  - A variable declared static within the body of a function maintains its value between function invocations
  - A variable declared static within a module, but outside the body of a function, is accessible by all functions within that module
- For Embedded Systems:
  - Encapsulation of persistent data
  - Modular coding (data hiding)
  - Hiding of internal processing in each module
- Note that static variables are stored globally, and not on the stack

Static Variable Example

Before entering `MyFunction()` the 1st time, **`myVar = 0`**

This is part of the ANSI C startup "copy down" procedure.

```
FILE1.c

#include <FILE2.h>
//includes functions in file FILE2.c

void main (void)
{
   MyFunction();     //included in FILE2.c

   MyFunction();     //included in FILE2.c
}
```

```
FILE2.c

void MyFunction (void)
{
//Definition of MyFunction in FILE2.C

  static byte myVar = 0;
//local variable declared static

  myVar = myVar + 1;
}
```

Before entering `MyFunction()` the 2nd time, **`myVar = 1`**

`myVar` is a local variable but keeps its value because it is static.

Static Functions

►Functions declared static within a module may only be called by other functions within that module

►Features:
- Good structured programming practice
- Can result in smaller and/or faster code

►Advantages:
- Since the compiler knows at compile time exactly what functions can call a given static function, it may strategically place the static function such that it may be called using a short version of the call or jump instruction

Static Functions – An Example

main.c

```
extern byte ExternalFunction (byte);


void main (void)
{
byte n;

n = ExternalFunction (byte);
}
```

stuff.c

```
// Local prototypes  =================
//
static byte InternalFunction1 (byte);
static byte InternalFunction2 (byte);

// External functions  ===============
//
byte ExternalFunction (byte)
{
  InternalFunction1(param1);
  InternalFunction2(param2);
  return_something;
}

// Local functions  ==================
//
static byte InternalFunction1 (byte)
{
  do_something;
}

static byte InternalFunction2 (byte)
{
  do_something_else;
}
```

Volatile Variables

► A volatile variable is one whose value may be change outside the normal program flow

► In embedded systems, there are two ways this can happen:
- Via an interrupt service routine
- As a consequence of hardware action

► It is considered to be *very good practice* to declare all peripheral registers in embedded devices as volatile

► The standard C solution:

```
#define PORTA (*((volatile unsigned char*) (0x0000)))
```

This macro defines PORTA to be the content of a pointer to an unsigned char.
This is portable over any architecture but not easily readable.
And it doesn't take advantage of the S08's bit manipulation capabilities.

► The CodeWarrior solution:

```
extern volatile PTADSTR _PTAD @0x00000000;
```

CodeWarrior Declaration for PORT A Data Register

```
/*** PTAD - Port A Data Register; 0x00000000 ***/
typedef union {
  byte Byte;
  struct {
    byte PTAD0       :1;                                       /* Port A Data Register Bit 0 */
    byte PTAD1       :1;                                       /* Port A Data Register Bit 1 */
    byte PTAD2       :1;                                       /* Port A Data Register Bit 2 */
    byte PTAD3       :1;                                       /* Port A Data Register Bit 3 */
    byte PTAD4       :1;                                       /* Port A Data Register Bit 4 */
    byte PTAD5       :1;                                       /* Port A Data Register Bit 5 */
    byte PTAD6       :1;                                       /* Port A Data Register Bit 6 */
    byte PTAD7       :1;                                       /* Port A Data Register Bit 7 */
  } Bits;
} PTADSTR;
extern volatile PTADSTR _PTAD @0x00000000;
#define PTAD                            _PTAD.Byte
#define PTAD_PTAD0                      _PTAD.Bits.PTAD0
#define PTAD_PTAD1                      _PTAD.Bits.PTAD1
#define PTAD_PTAD2                      _PTAD.Bits.PTAD2
#define PTAD_PTAD3                      _PTAD.Bits.PTAD3
#define PTAD_PTAD4                      _PTAD.Bits.PTAD4
#define PTAD_PTAD5                      _PTAD.Bits.PTAD5
#define PTAD_PTAD6                      _PTAD.Bits.PTAD6
#define PTAD_PTAD7                      _PTAD.Bits.PTAD7

#define PTAD_PTAD0_MASK                 0x01
#define PTAD_PTAD1_MASK                 0x02
#define PTAD_PTAD2_MASK                 0x04
#define PTAD_PTAD3_MASK                 0x08
#define PTAD_PTAD4_MASK                 0x10
#define PTAD_PTAD5_MASK                 0x20
#define PTAD_PTAD6_MASK                 0x40
#define PTAD_PTAD7_MASK                 0x80
```

Volatile Variables are Never Optimized

```c
unsigned char PORTA @ 0x00;
unsigned char SCI1S1 @ 0x1C;
unsigned char value;

void main (void)
{
  PORTA = 0x05;     // PORTA = 00000101
  PORTA = 0x05;     // PORTA = 00000101
  SCI1S1;
  value = 10;
}
```

without volatile keyword

```
mov     #5,PORTA
lda     #10
sta     @value
```

```c
volatile unsigned char PORTA @ 0x00;
volatile unsigned char SCI1S1 @ 0x1C;
unsigned char value;

void main (void)
{
  PORTA = 0x05;     // PORTA = 00000101
  PORTA = 0x05;     // PORTA = 00000101
  SCI1S1;
  value = 10;
}
```

with volatile keyword

```
mov     #5,PORTA
mov     #5,PORTA
lda     SCI1S1
lda     #10
sta     @value
```

Const Variables

- It is safe to assume that a parameter along with the keyword "const" means a "read-only" parameter.

- Some compilers create a genuine variable in RAM to hold the const variable. Upon system software initialization, the "read-only" value is copied into RAM. On RAM-limited systems, this can be a significant penalty.

- Compilers for Embedded Systems, like CodeWarrior™, store const variables in ROM (or Flash). However, the "read-only" variable is still treated as a variable and accessed as such, although the compiler protects const definitions from inadvertent writing. Each const variable must be declared with an initialization value.

Keyword “Const”

For Embedded Systems:

- Parameters defined const are allocated in ROM space
- The compiler protects ‘const’ definitions of inadvertent writing
- Express the intended usage of a parameter

```
const unsigned short a;
unsigned short const a;
const unsigned short *a;
unsigned short * const a;
```

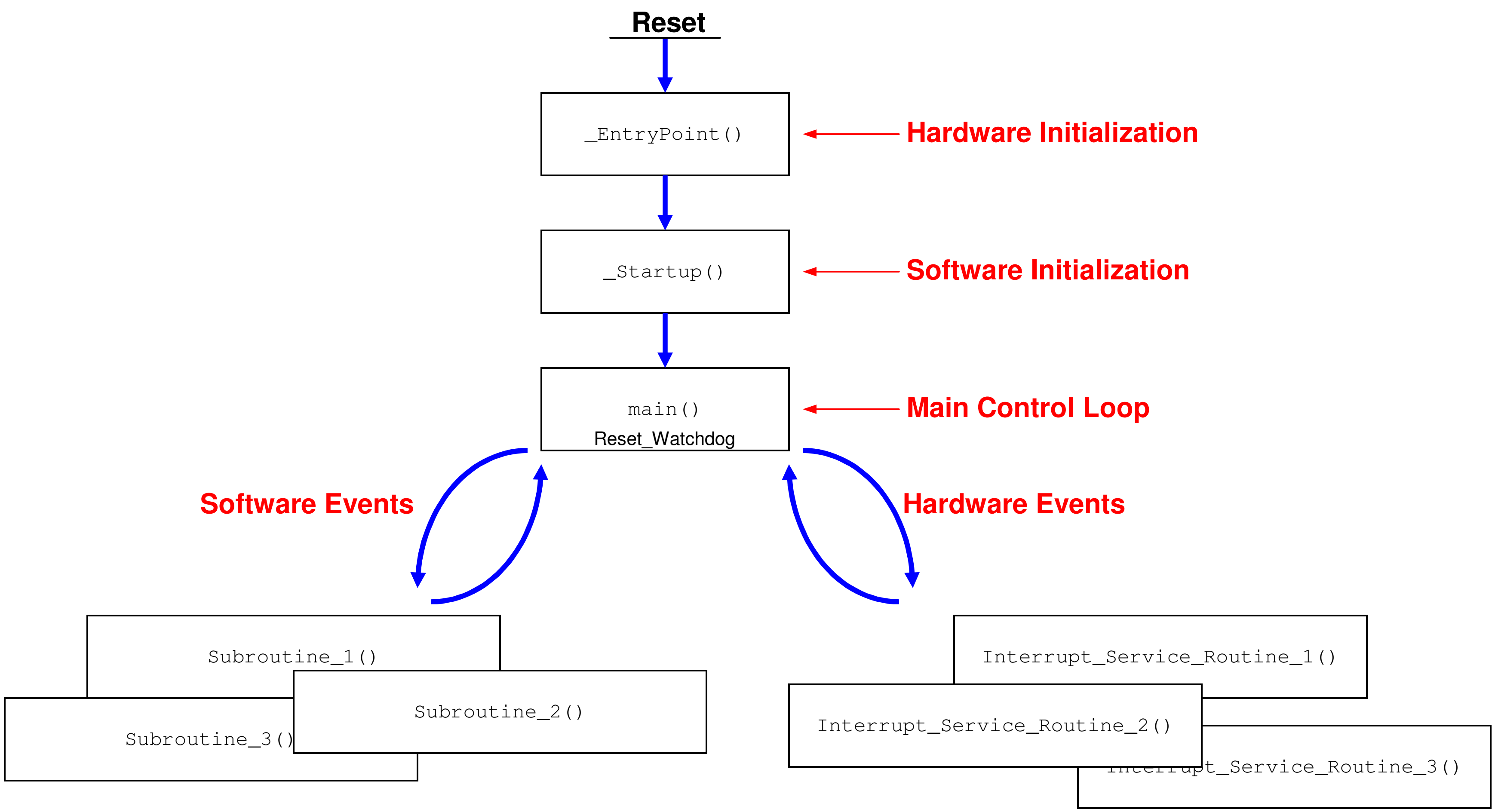
Lab2

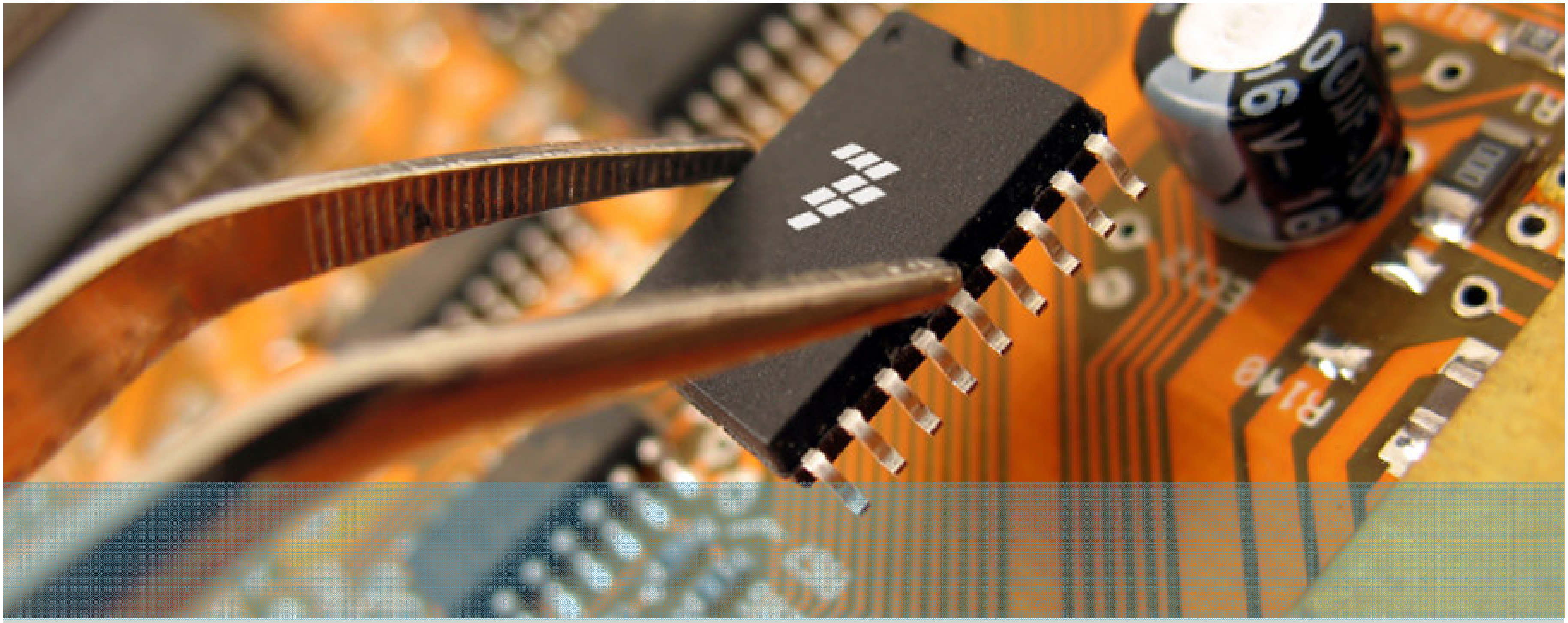
```
#pragma DATA_SEG MY_ZEROPAGE
byte near n;

#pragma DATA_SEG DEFAULT_RAM
byte far var = 7;

void main(void)
{
  EnableInterrupts;

  // initialize LEDs
  PTCD = 0xFF;            // set default value for Port C
  PTCDD = 0b00111100;     // set LED port pins as outputs

  for (;;)
  {
    __RESET_WATCHDOG();   /* feeds the dog */

    for (n=0;;n++)
    {
            PTCD++;       // blink LEDs
            var++;
    }

  } /* loop forever */
  /* please make sure that you never leave main */
}
```

Lab2

```
#pragma DATA_SEG DEFAULT_RAM
byte far var;

void main(void)
{
  #pragma DATA_SEG MY_ZEROPAGE
  static byte near n;

  EnableInterrupts;

  // initialize LEDs
  PTCD = 0xFF;            // set default value for Port C
  PTCDD = 0b00111100;     // set LED port pins as outputs

  for (;;)
  {
    __RESET_WATCHDOG();   /* feeds the dog */

    for (n=0;;n++)
    {
            PTCD++;       // blink LEDs
            var++;
    }

  } /* loop forever */
  /* please make sure that you never leave main */
}
```

Very Basic Software Flow



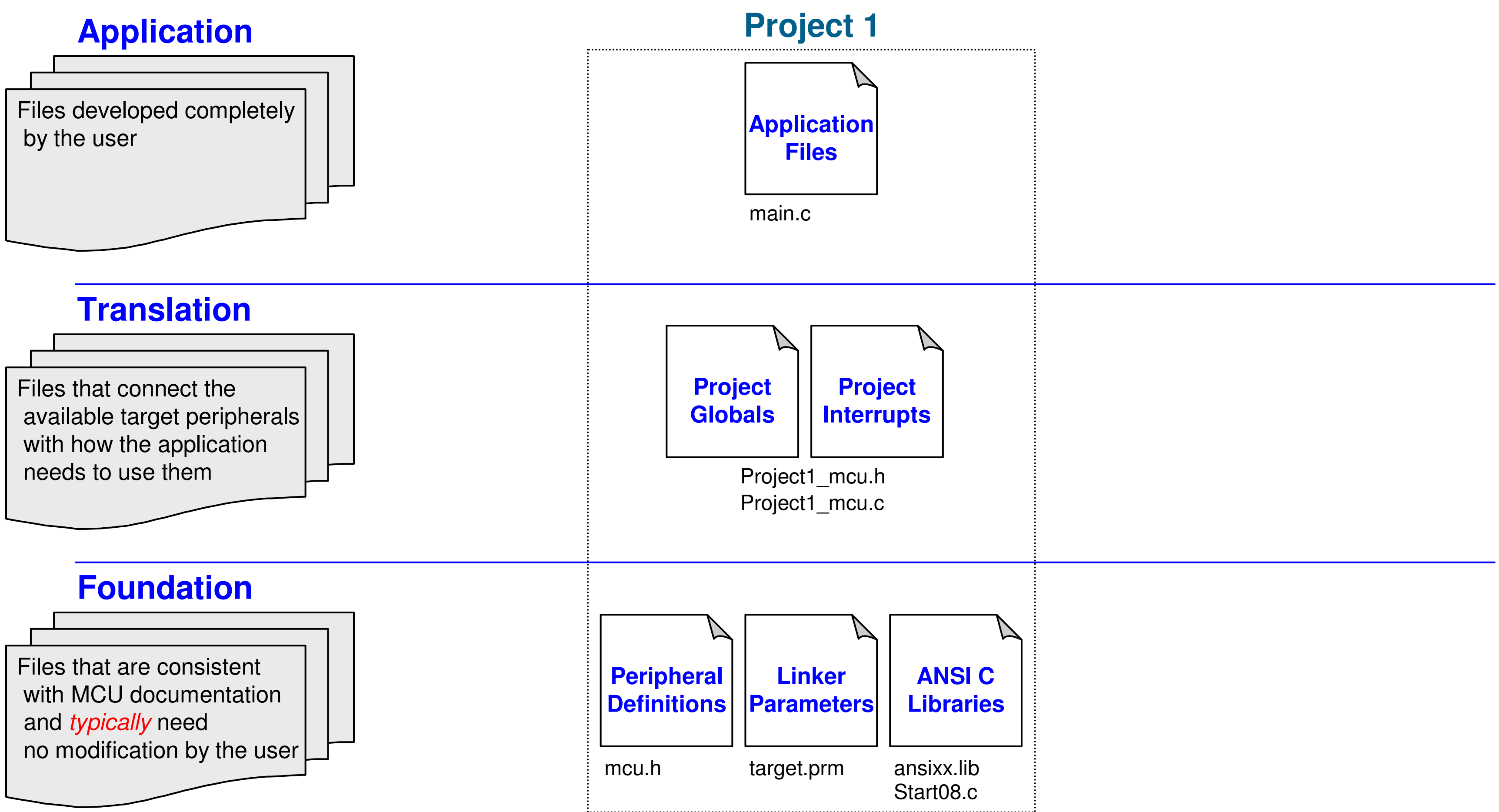


Project Software Architecture

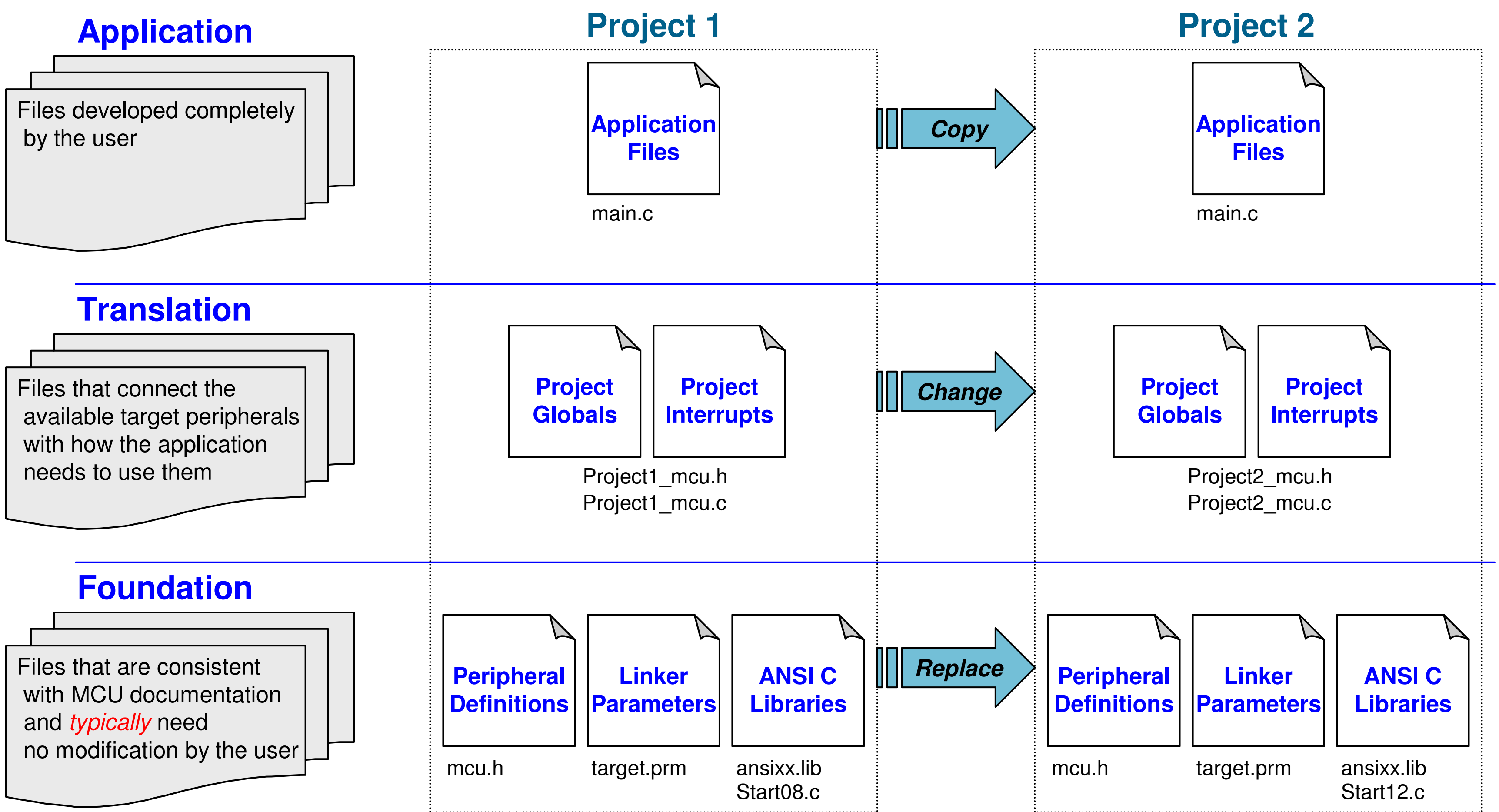
C for Embedded Systems Programming

freescale™ semiconductor

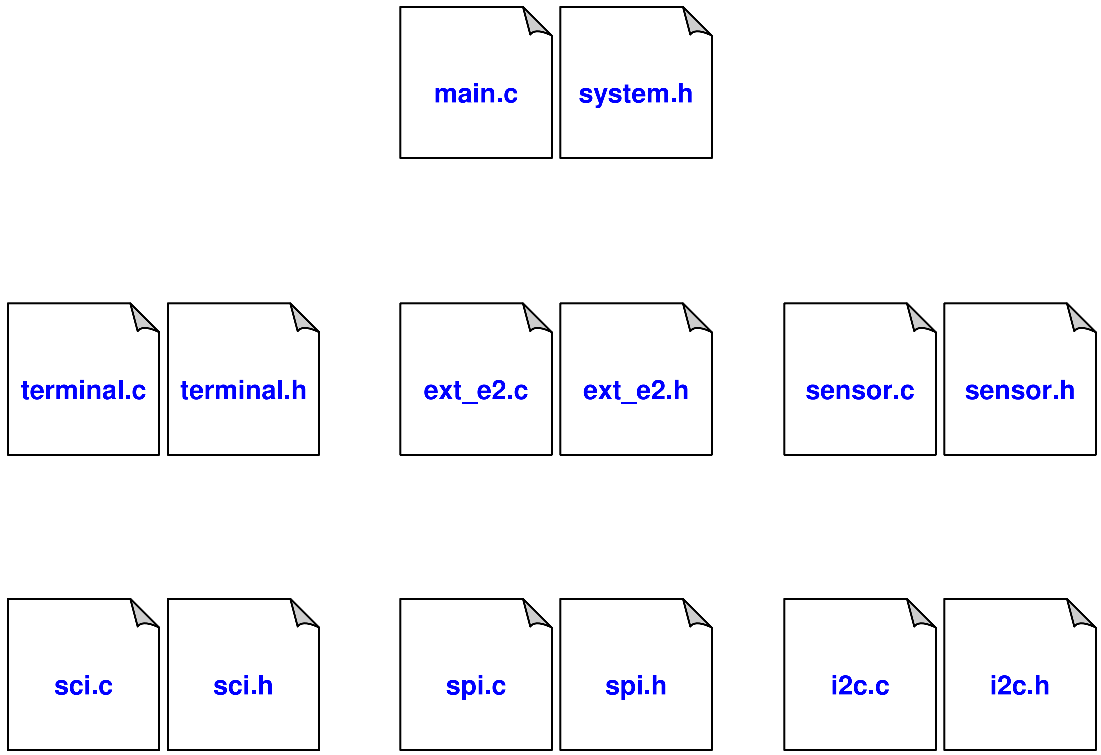
Project Software Architecture



Project Software Architecture



Modular File Organization



Example system.h

system.h

```
/*********************************************************************************************\
* Project Name
\*********************************************************************************************/
#ifndef _SYSTEM_H_
#define _SYSTEM_H_

#include <hidef.h>                /* for EnableInterrupts macro */
#include "derivative.h"           /* include peripheral declarations */

/*********************************************************************************************\
* Public type definitions
\*********************************************************************************************/

/********************************************************************************************
**
**  Variable type definition: BIT_FIELD
*/
typedef union
{
  byte Byte;
  struct {
    byte _0          :1;
    byte _1          :1;
    byte _2          :1;
    byte _3          :1;
    byte _4          :1;
    byte _5          :1;
    byte _6          :1;
    byte _7          :1;
  } Bit;
} BIT_FIELD;


/********************************************************************************************
**
**  Variable type definition: tword
*/
typedef union
{
  unsigned short Word;
  struct
  {
    byte hi;
    byte lo;
  } Byte;
} tword;
```

Example system.h

system.h

```
/**************************************************************************************************\
* Project includes
\**************************************************************************************************/
#include "mma845x.h"              // MMA845xQ macros
#include "iic.h"                  // IIC macros
#include "sci.h"                  // SCI macros
#include "spi.h"                  // SPI macros
#include "terminal.h"             // Terminal interface macros

/**************************************************************************************************\
* Public macros
\**************************************************************************************************/
/**************************************************************************************************
**
**  General System Control
**
**  0x1802  SOPT1     System Options Register 1
**  0x1803  SOPT2     System Options Register 2
**  0x1808  SPMSC1    System Power Management Status and Control 1 Register
**  0x1809  SPMSC2    System Power Management Status and Control 2 Register
**  0x180B  SPMSC3    System Power Management Status and Control 3 Register
**  0x180E  SCGC1     System Clock Gating Control 1 Register
**  0x180F  SCGC2     System Clock Gating Control 2 Register
**  0x000F  IRQSC     Interrupt Pin Request Status and Control Register
*/

#define init_SOPT1    0b01000010
/*                      1100001U = reset
**                      |||xxx||
**                      |||   |+-- RSTPE      =0  : RESET pin function disabled
**                      |||   +--- BKGDPE     =1  : Background Debug pin enabled
**                      ||+------- STOPE      =0  : Stop Mode disabled
**                      |+-------- COPT       =1  : Long COP timeout period selected
**                      +--------- COPE       =0  : COP Watchdog timer disabled
*/

#define init_SOPT2    0b00000010
/*                      00000000 = reset
**                      |x||x|||
**                      | || ||+-- ACIC1      =0  : ACMP1 output not connected to TPM1CH0 input
**                      | || |+--- IICPS      =1  : SDA on PTB6; SCL on PTB7
**                      | || +---- ACIC2      =0  : ACMP2 output not connected to TPM2CH0 input
**                      | |+------ TPM1CH2PS  =0  : TPM1CH2 on PTA6
**                      | +------- TPM2CH2PS  =0  : TPM2CH2 on PTA7
**                      +--------- COPCLKS    =0  : COP clock source is internal 1kHz reference
*/
```

Example system.h

system.h

```
/*********************************************************************************************
**
**  Port I/O
**
**  0x0000  PTAD      Port A Data Register
**  0x0001  PTADD     Port A Data Direction Register
**  0x0002  PTBD      Port B Data Register
**  0x0003  PTBDD     Port B Data Direction Register
**  0x0004  PTCD      Port C Data Register
**  0x0005  PTCDD     Port C Data Direction Register
**  0x0006  PTDD      Port D Data Register
**  0x0007  PTDDD     Port D Data Direction Register
**  0x1840  PTAPE     Port A Pull Enable Register
**  0x1841  PTASE     Port A Slew Rate Enable Register
**  0x1842  PTADS     Port A Drive Strength Selection Register
**  0x1844  PTBPE     Port B Pull Enable Register
**  0x1845  PTBSE     Port B Slew Rate Enable Register
**  0x1846  PTBDS     Port B Drive Strength Selection Register
**  0x1848  PTCPE     Port C Pull Enable Register
**  0x1849  PTCSE     Port C Slew Rate Enable Register
**  0x184A  PTCDS     Port C Drive Strength Selection Register
*/

#define init_PTAD     0b10000000
//                      00000000 = reset
#define init_PTADD    0b10000000
//                      00000000 = reset
#define init_PTAPE    0b00000110
//                      00000000 = reset
#define init_PTASE    0b00000000
//                      00000000 = reset
#define init_PTADS    0b00000000
//                      00000000 = reset
/*                      ||||||||
**                      |||||||+-- X_OUT
**                      ||||||+--- Y_OUT        - MMA845x INT1
**                      |||||+---- Z_OUT        - MMA845x INT2
**                      ||||+----- DIS_MCU
**                      |||+------ BKGD
**                      ||+------- RESET
**                      |+-------- EXTRA_AD
**                      +--------- LEDB_BB      - Blue LED cathode
*/
#define INT1_IS_ACTIVE              (PTAD_PTAD1 == 0)
#define LED_BlueOn                  (PTAD_PTAD7 = 0)
#define LED_BlueOff                 (PTAD_PTAD7 = 1)
```

Example system.h

system.h

```
#define init_PTCD     0b10111011
//                      00000000 = reset
#define init_PTCDD    0b10110111
//                      00000000 = reset
#define init_PTCPE    0b01000000
//                      00000000 = reset
#define init_PTCSE    0b00000000
//                      00000000 = reset
#define init_PTCDS    0b00000000
//                      00000000 = reset
/*                      ||||||||
**                      |||||||+-- LEDG_BB      - Green LED cathode
**                      ||||||+--- SLEEP        - MMA845x CS
**                      |||||+---- G_SEL_2      - MMA845x SA0
**                      ||||+----- G_SEL_1
**                      |||+------ LEDR_BB      - Red LED cathode
**                      ||+------- BUZZER_BB
**                      |+-------- PUSH_BUTTON
**                      +--------- LED_OB       - Yellow LED cathode
*/
#define LED_GreenOn                 (PTCD_PTCD0 = 0)
#define LED_GreenOff                (PTCD_PTCD0 = 1)
#define SENSOR_SHUTDOWN             (PTCD_PTCD1 = 0)
#define SENSOR_ACTIVE               (PTCD_PTCD1 = 1)
#define LED_RedOn                   (PTCD_PTCD4 = 0)
#define LED_RedOff                  (PTCD_PTCD4 = 1)
#define LED_YellowOn                (PTCD_PTCD7 = 0)
#define LED_YellowOff               (PTCD_PTCD7 = 1)
#define SA0_PIN                     (PTCD_PTCD2)

/***********************************************************************************************\
* Public memory declarations
\***********************************************************************************************/

/***********************************************************************************************\
* Public prototypes
\***********************************************************************************************/

#endif  /* _SYSTEM_H_ */
```

Example sci.h

sci.h

```
/***********************************************************************************************\
* Project name
*
* Filename: sci.h
*
\***********************************************************************************************/
#ifndef _SCI_H_
#define _SCI_H_

/***********************************************************************************************\
* Public macros
\***********************************************************************************************/

#define BUFFER_RX_SIZE        10
#define BUFFER_TX_SIZE        200

/***********************************************************************************************
**
**  Serial Communications Interface (SCI)
**
**  0x0020  SCIBDH    SCI Baud Rate Register High
**  0x0021  SCIBDL    SCI Baud Rate Register Low
**  0x0022  SCIC1     SCI Control Register 1
**  0x0023  SCIC2     SCI Control Register 2
**  0x0024  SCIS1     SCI Status Register 1
**  0x0025  SCIS2     SCI Status Register 2
**  0x0026  SCIC3     SCI Control Register 3
**  0x0027  SCID      SCI Data Register
**
**  SCI target baudrate = 115.2k
**  MCU bus frequency = 9.216MHz
**
**  SCI Baud Rate Register = 5
**
**  SCI baudrate = bus / (16 * BR)
**               = 9.216MHz / (16 * 5)
**               = 115.2k
*/

#define init_SCIBDH   0x00
#define init_SCIBDL   0x05
```

Example sci.h

sci.h

```
#define ASCII_BS      0x08
#define ASCII_LF      0x0A
#define ASCII_CR      0x0D
#define ASCII_DEL     0x7F

/***********************************************************************************************\
* Public type definitions
\***********************************************************************************************/

/***********************************************************************************************\
* Public memory declarations
\***********************************************************************************************/

#pragma DATA_SEG MY_ZEROPAGE

extern byte BufferRx[BUFFER_RX_SIZE];

#pragma DATA_SEG DEFAULT

/***********************************************************************************************\
* Public prototypes
\***********************************************************************************************/

void SCIControlInit (void);
void SCISendString (byte *pStr);
void SCI_CharOut(byte data);
void SCI_NibbOut(byte data);
void SCI_ByteOut(byte data);
void SCI_putCRLF (void);
byte SCI_CharIn(void);
byte SCI_ByteIn(void);
void SCI_s12dec_Out (tword data);
void SCI_s8dec_Out(tword data);
void SCI_s12int_Out (tword data);
void SCI_s12frac_Out (tword data);

byte isnum (byte data);
byte ishex (byte data);
byte tohex (byte data);
void hex2ASCII (byte data, byte* ptr);


#endif  /* _SCI_H_ */
```

Example terminal.h

terminal.h

```
/***********************************************************************************************\
* Project Name
*
* Filename: terminal.h
*
\***********************************************************************************************/
#ifndef _TERMINAL_H_
#define _TERMINAL_H_

/***********************************************************************************************\
* Public macros
\***********************************************************************************************/

/***********************************************************************************************\
* Public type definitions
\***********************************************************************************************/

/***********************************************************************************************\
* Public memory declarations
\***********************************************************************************************/

/***********************************************************************************************\
* Public prototypes
\***********************************************************************************************/

void TerminalInit (void);
void ProcessTerminal (void);
void OutputTerminal (byte BlockID, byte *ptr);

#endif  /* _TERMINAL_H_ */
```

Example terminal.c

terminal.c

```
/*************************************************************************************************\
* Project Name
*
* Filename: terminal.c
*
\*************************************************************************************************/

#include "system.h"

/*************************************************************************************************\
* Private macros
\*************************************************************************************************/

/*************************************************************************************************\
* Private type definitions
\*************************************************************************************************/

/*************************************************************************************************\
* Private prototypes
\*************************************************************************************************/

/*************************************************************************************************\
* Private memory declarations
\*************************************************************************************************/

#pragma DATA_SEG MY_ZEROPAGE

#pragma DATA_SEG DEFAULT_RAM

/*************************************************************************************************\
* Public functions
\*************************************************************************************************/

/*************************************************************************************************\
* Private functions
\*************************************************************************************************/
```

freescale™
semiconductor