Embedded Software Development
2024. 11. 11.

Special C language elements

Méréstechnika és
Információs Rendszerek
Tanszék

inline functions

- Inline functions: the compiler “picks out” the inside of the function and actual function call does not happen, instead, the code found in the function is used and substituted into our code
  - Faster than normal functions since no overhead of function call
  - It is only worth when the function contains only few instructions
  - Even if the function is marked as inline the compiler may use it in a different way (inline feature of the function may be ignored by the compiler)
  - Static keyword is usually used with inline function since that are restricted to the same compilation unit (e.g. C file) in which they are defined
    - Note: the functions in C are by default global. If we want to limit the scope of the function, we use the keyword static before the function
  - Generally they are found in the header files
- Example:

```
uint32_t adder_fn(uint32_t x, uint32_t y) {
    return (x+y);
}
```

```
static inline __attribute__((always_inline)) uint32_t adder_fn(uint32_t x, uint32_t y){
    return (x+y);
}
```

inline functions

- Without inline: 23 instr   with inline: 11 instr

```
117              int_num = adder_fn(x_add, y_add);
000013b8:      ldr      r3,[pc,#0x4c] ; 0x1404
000013ba:      ldr      r2,[r3]
000013bc:      ldr      r3,[pc,#0x4c] ; 0x1408
000013be:      ldr      r3,[r3]
000013c0:      mov      r0,r2
000013c2:      mov      r1,r3
000013c4:      bl       0x0000130c
000013c8:      mov      r2,r0
000013ca:      ldr      r3,[pc,#0x44] ; 0x140c
000013cc:      str      r2,[r3]
```

Function call

Return from function

```
               adder_fn:
0000130c:      push     {r7}
0000130e:      sub      sp,sp,#0xc
00001310:      add      r7,sp,#0x0
00001312:      str      r0,[r7,#0x4]
00001314:      str      r1,[r7]
 75               return (x+y);
00001316:      ldr      r2,[r7,#0x4]
00001318:      ldr      r3,[r7]
0000131a:      add      r3,r2
 76            }
0000131c:      mov      r0,r3
0000131e:      adds     r7,#0xc
00001320:      mov      sp,r7
00001322:      pop.w    {r7}
00001326:      bx       lr
 83            {
```

```
117              int_num = adder_fn(x_add, y_add);
0000139e:      ldr      r3,[pc,#0x4c] ; 0x13e8
000013a0:      ldr      r2,[r3]
000013a2:      ldr      r3,[pc,#0x4c] ; 0x13ec
000013a4:      ldr      r3,[r3]
000013a6:      str      r2,[r7,#0x4]
000013a8:      str      r3,[r7]
 75               return (x+y);
000013aa:      ldr      r2,[r7,#0x4]
000013ac:      ldr      r3,[r7]
000013ae:      add      r3,r2
117              int_num = adder_fn(x_add, y_add);
000013b0:      ldr      r2,[pc,#0x40] ; 0x13f0
000013b2:      str      r3,[r2]
```

inline functions

- Even if the function is marked as inline the compiler may use it in a different way
  - Can be forced, e.g.: __attribute__((always_inline))
  - In general leave the compiler to do its job, forcing the compiler is acceptable only if speed is the largest concern
- In some cases the compiler recognizes that a function cannot be inline

Container classes

- auto:
  - Default container type in functions and blocks (not needed to be specified)
  - Available only inside the code block and disappears at the end of the block
- static:
  - Inside a function: Stores its value until the end of the program (even among function calls)
  - With global variable: visible only in the given compilation unit (in that C file) (note: extern type is the opposite – see later)
- register:
  - The variable is stored in a certain register
  - Use if a variable has to be accessed fast and frequently
  - Rarely used, leave it for the comopiler…

```
register int buttons __asm__("r4");
```

```
122                     buttons = BSP_ButtonGet(0);
000013b4:     movs      r0,#0x0
000013b6:     bl        0x00000288
000013ba:     mov       r3,r0
000013bc:     mov       r4,r3
```

Container classes

- extern:
  - It marks that a certain variable or function is found in an other compilation unit, i.e., other C file.
  - Compilation units, i.e., all C files must belong to the same project
  - During compilation the compiler assigns a general label for the variable or function and the linker searches in which object file that certain variable or function can be found
  - It can be <u>initialized at one place</u>. At <u>other places only declarations</u> are found
  - Example:

  - The extern variable can be referred at both places
  - It is used generally in case of shared variables
  - When a function of C syntax found in an external file and called from a C++ file then *extern "C"* must be used during declaration

bitfield structures

- If a variable does not require at least 8 bit it is possible to assign values bitwise
- Advantages:
  - Memory saving (especially important if only a small amount of memory is available)
  - Can be applied to a function register and manipulate its content bitwise at C level (WARNING! Take care of compiler settings: do not change them)
- Since different compilers may handle bitfield structures in a different way therefore double-checking is necessary
- When defining the fields of the structure use colons to set the size in bits

```
struct data_array1_strct{
char data_11;
char data_12;
char data_13;
char data_14;
char data_15;
} data_array1;
```

-> data is stored by 1 byte for each element, i.e., total 5 bytes

```
struct data_array2_strct{
char data_21:1;
char data_22:1;
char data_23:1;
char data_24:1;
char data_25:1;
} data_array2;
```

-> data is stored by 1 bit for each element, i.e., total 5 bits

bitfield structures

```
struct data_array1_strct{
char data_11;
char data_12;
char data_13;
char data_14;
char data_15;
} data_array1;

struct data_array2_strct{
char data_21:1;
char data_22:1;
char data_23:1;
char data_24:1;
char data_25:1;
} data_array2;


data_array1.data_11 = 11;
data_array1.data_12 = 12;
data_array1.data_13 = 13;
data_array1.data_14 = 14;
data_array1.data_15 = 15;
data_array2.data_21 = 21;
data_array2.data_22 = 22;
data_array2.data_23 = 23;
data_array2.data_24 = 24;
data_array2.data_25 = 25;

data_array1_size = sizeof (data_array1) ;
data_array2_size = sizeof (data_array2) ;
```

- Example: two sturctures: in structure `data_array2` field size is 1-bit
- Size of `data_array1` is 5 byte, size of `data_array2` is 1 byte (5 bit, but 1 byte is minimal).
- Structure `data_array2` is able to store only 1-bit data (the last bit is kept the rest is cut off)

data_array1	struct a...	536871084 (...
data_11	char	0xb
data_12	char	0xc
data_13	char	0xd
data_14	char	0xe
data_15	char	0xf
data_array2	struct a...	0x2000009c
data_21	char	0x1
data_22	char	0x0
data_23	char	0x1
data_24	char	0x0
data_25	char	0x1
data_array1_size	uint32_t	0x5
data_array2_size	uint32_t	0x1

&(data_array2 : 0x2000009C

0x2000009C 00000015

It can be seen that in the memory really 10101b = 15hex value can be found at address 0x200009C

union type

- Different type of variables can be assigned to a memory part (once the structure is defined it has to be filled up with data and handled accordingly)
- Useful when the data type is unknown during compilation time since using union type it will not be necessary to reserve different variables for the unknown data
- Example:

```c
union UnionType {
    int i;
    float f;
    char str[5];
} union_var;

    union_var.i = 5;
    union_var.f = 5.0;
    strcpy(union_var.str, "5.0");
```

union_var	union U...	0x20000...	0x200000bc
(x)= i	int	0 (Deci...	0x200000bc
(x)= f	float	0.0 (Dec...	0x200000bc
str	char[5]	0x20000...	0x200000bc
(x)= str[0]	char	0 ('\0') (...	0x200000bc
(x)= str[1]	char	0 ('\0') (...	0x200000bd
(x)= str[2]	char	0 ('\0') (...	0x200000be
(x)= str[3]	char	0 ('\0') (...	0x200000bf
(x)= str[4]	char	0 ('\0') (...	0x200000c0

union_var	union U...	0x20000...	0x200000bc
(x)= i	int	5 (Deci...	0x200000bc
(x)= f	float	7.0E-45 ...	0x200000bc
str	char[5]	0x20000...	0x200000bc
(x)= str[0]	char	5 ('\005'...	0x200000bc
(x)= str[1]	char	0 ('\0') (...	0x200000bd
(x)= str[2]	char	0 ('\0') (...	0x200000be
(x)= str[3]	char	0 ('\0') (...	0x200000bf
(x)= str[4]	char	0 ('\0') (...	0x200000c0

union_var	union U...	0x20000...	0x200000bc
(x)= i	int	1084227...	0x200000bc
(x)= f	float	5.0 (Dec...	0x200000bc
str	char[5]	0x20000...	0x200000bc
(x)= str[0]	char	0 ('\0') (...	0x200000bc
(x)= str[1]	char	0 ('\0') (...	0x200000bd
(x)= str[2]	char	160 (' ') ...	0x200000be
(x)= str[3]	char	64 ('@')...	0x200000bf
(x)= str[4]	char	0 ('\0') (...	0x200000c0

union_var	union U...	0x20000...	0x200000bc
(x)= i	int	3157557...	0x200000bc
(x)= f	float	4.42468...	0x200000bc
str	char[5]	0x20000...	0x200000bc
(x)= str[0]	char	53 ('5') (...	0x200000bc
(x)= str[1]	char	46 ('.') (...	0x200000bd
(x)= str[2]	char	48 ('0') (...	0x200000be
(x)= str[3]	char	0 ('\0') (...	0x200000bf
(x)= str[4]	char	0 ('\0') (...	0x200000c0

Union + bitfield

- In embedded environment at C language level it is easy to handle a register at both bit and byte level as well
- Example (Simplicity Studio diagnostic.h):
  - Inside union type variable:
    - There exist a bitfield structure used to access the configuration bits in a bitwise manner
    - There exists a 32-bit variable named *word* used to access the whole 32-bit register content
    - HalCrashAfsrType.bits.WRONGSIZE= 1; the same as

      HalCrashAfsrType.word|=1 << 3;

      but more elegant and simple → more clear code, less possibility of errors

```
typedef union {
  struct {
    uint32_t MISSED         : 1;  // B0
    uint32_t RESERVED       : 1;  // B1
    uint32_t PROTECTED      : 1;  // B2
    uint32_t WRONGSIZE      : 1;  // B3
    uint32_t                : 28; // B4-31
  } bits;

  uint32_t word;
} HalCrashAfsrType;
```

Structured handling of register arrays

- 1st step: definition of a structure according to the register arrays - Example: register set for ADC (C code + datasheet):

```
typedef struct
{
  __IOM uint32_t CTRL;
  __IOM uint32_t CMD;
  __IM uint32_t  STATUS;
  __IOM uint32_t SINGLECTRL;
  __IOM uint32_t SCANCTRL;
  __IOM uint32_t IEN;
  __IM uint32_t  IF;
  __IOM uint32_t IFS;
  __IOM uint32_t IFC;
  __IM uint32_t  SINGLEDATA;
  __IM uint32_t  SCANDATA;
  __IM uint32_t  SINGLEDATAP;
  __IM uint32_t  SCANDATAP;
  __IOM uint32_t CAL;

  uint32_t       RESERVED0[1];
  __IOM uint32_t BIASPROG;
} ADC_TypeDef;
```

Offset	Name
0x000	ADCn_CTRL
0x004	ADCn_CMD
0x008	ADCn_STATUS
0x00C	ADCn_SINGLECTRL
0x010	ADCn_SCANCTRL
0x014	ADCn_IEN
0x018	ADCn_IF
0x01C	ADCn_IFS
0x020	ADCn_IFC
0x024	ADCn_SINGLEDATA
0x028	ADCn_SCANDATA
0x02C	ADCn_SINGLEDATAP
0x030	ADCn_SCANDATAP
0x034	ADCn_CAL
0x03C	ADCn_BIASPROG

Application of volatile type is important otherwise the optimizer may remove non-used fields that results a shift of the whole structure

```
/* following defines should be used for structure members */
#define    __IM    volatile const    /*! Defines 'read only' structure member permissions */
#define    __OM     volatile        /*! Defines 'write only' structure member permissions */
#define    __IOM    volatile        /*! Defines 'read / write' structure member permissions */
```

Structured handling of register arrays

- 2nd step: search the base address of register array of the certain peripheral


```
#define ADC0_BASE          (0x40002000UL) /**< ADC0 base address  */
```

- 3rd step: set a pointer to the appropriate memory address pointing to the certain type of structure:

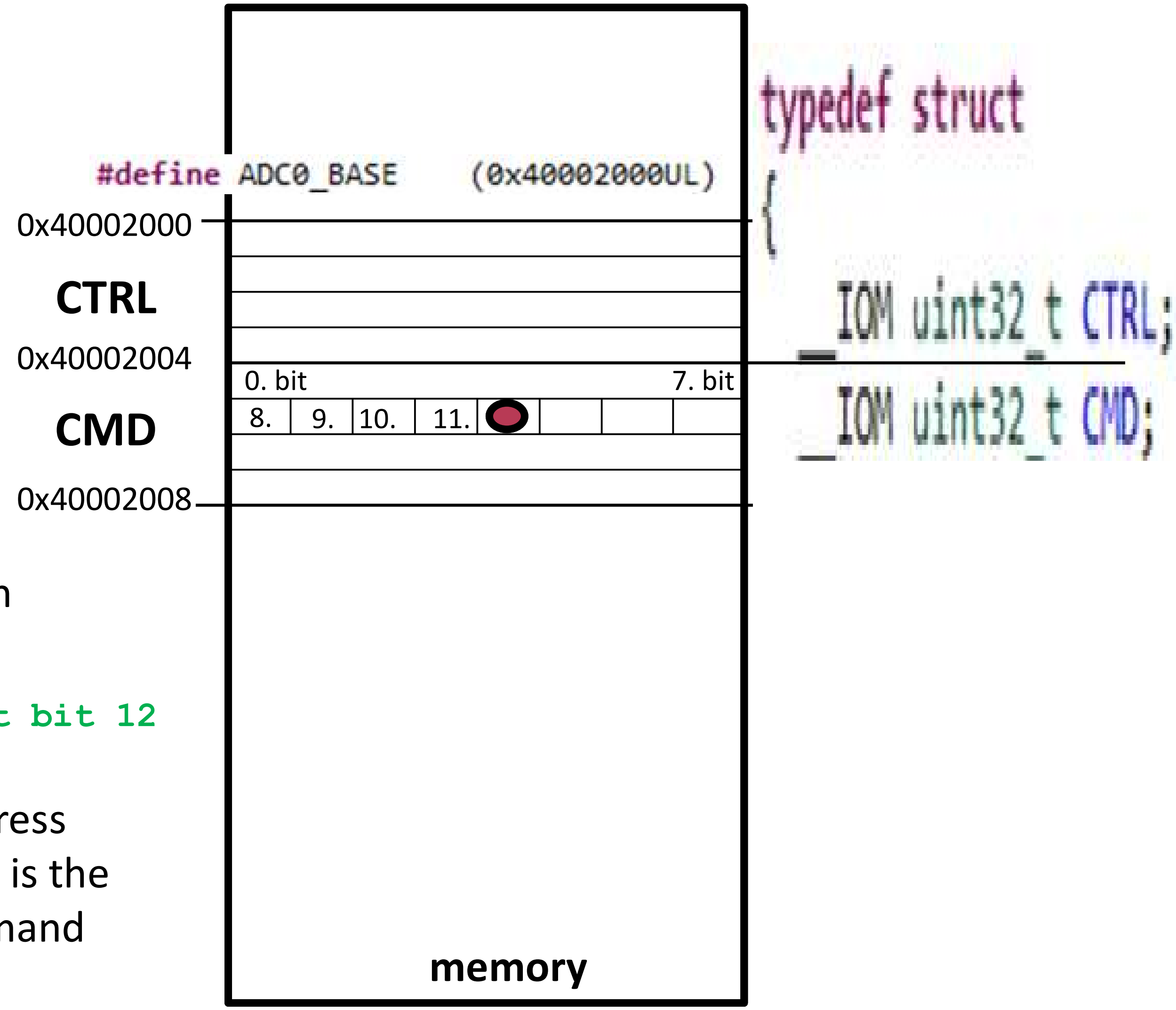
```
#define ADC0          ((ADC_TypeDef *) ADC0_BASE)
```

- 4th step: application of certain element of the structure:

```
ADC0->CMD |= 1<<12; // set bit 12 into high
```

  - So bit 12 of register with address 0x40002004 is set into 1 (this is the memory address of the command register of ADC, or you can also see it as the bit 4 of register at address 0x40002005)

Structured handling of register arrays



- 4th step: application of certain element of the structure:

```
ADC0->CMD |= 1<<12; // set bit 12 into high
```

- So bit 12 of register with address 0x40002004 is set into 1 (this is the memory address of the command register of ADC)
- Registers are 32-bit (4 bytes)

Attributes of functions and variables

- In C language keyword __attribute__ ((…)) is used to assign special features to functions or variables. Examples (not valid for all processors or compilers):
  - __attribute__ ((interrupt ("IRQ"))); IT function
  - __attribute__((always_inline)): function is used always inline
  - __attribute__((weak)): function can be redefined.
    - E.g.: IT handling, the default IT function is weak, so a function with the same name can be defined anywhere in the code to be the IT function (this way the default function is overdefined)
  - __attribute__((section("*name*"))): if section called *name* is given in the linker file then variable will be placed there
  - __attribute__ ((__cleanup__(__iRestore))): when a variable diasappears a function is called

Compilation directives(pragma)

- #pragma or _pragma: compilation directives/keywords
- Either general or HW-specific instructions can be used, e.g.:
  - #pragma once: a function is included only once
  - #pragma interrupt: marks an IT function
  - #pragma align(4): start address should be always an integer multiple of 4 bytes
    - Can be especially important in case of DSP
  - #pragma pack: fields of a structure are ordered directly one after the other
- Compiler specific, documentation has to be checked
- Several similar functions can be implemented just like by keyword __attribute__ (e.g.: interrupt, pack…)

Idiom recognition

- Idiom recognition
  - The look of the command is recognized by the compiler and can compile it according to the instructions of the certain processor
  - Examples (depends on the compiler):
    - Saturation (Cortex SSAT asm command): Y = (x<-8)? -8 : (x>7 ? 7:x)
    - Circular buffer (DSP): a+=w[j]*x[i % N]
      - Modulo operation is not performed, instead, the HW supported circular buffer is used
  - No need to use special functions therefore the program can be compiled on other processors as well but despite of this fact the code can be efficient and well fit for the certain processor
  - It is not sure that all compilers can recognize them
  - The programmer guy must know what are the possibilities
  - In case of FPGAs it is also important to use general HW description to recognize the syntheser what the developer wants to implement

Use of integer data type

- In C language the minimum required number representation has to be defined for many data types (e.g. unsigned integer must cover 0 … 65535 but it can be larger…).
  - Embedded systems: many architectures exist therefore type *int* can be 16-bit or even 32-bit
- Problem: in embedded systems it is important to know the exact data-width (16-bit or 32-bit, etc.)
  - Mapping variables into registers
  - Estimation of computation needs
- C99 standard: use of inttypes
  - **#include <stdint.h>**
  - Defines types with exact data-width, e.g.:
    - int16_t : 16-bit signed integer
    - uint32_t : 32-bit unsigned integer (e.g. long unsigned int)

define

- Special symbols: # and ##
- # symbol: certain character set is substituted as string (stringizing operator)
- ## merges two character set (Token-Pasting / merging Operator)
  - Example:
    - #define set(var, num, value) var##num = #value
    - Calling the function in your code: set(def_var, 3, 2)
    - Processed by the preprocessor to what?
      - def_var3 = "2"; ->found only in the pre-processed code not in your code
- Be careful since it may result in a messy code

enum data type

- enum data type application
  - List is mapped into integer numbers
    - Default start value is 0 but other value can also be defined
  - In C no type check is used but it is done in C++
  - Example:

```c
typedef enum {
  usartStopbits0p5 = USART_FRAME_STOPBITS_HALF,       /**< 0.5 stopbits. */
  usartStopbits1   = USART_FRAME_STOPBITS_ONE,        /**< 1 stopbits. */
  usartStopbits1p5 = USART_FRAME_STOPBITS_ONEANDAHALF, /**< 1.5 stopbits. */
  usartStopbits2   = USART_FRAME_STOPBITS_TWO         /**< 2 stopbits. */
} USART_Stopbits_TypeDef;
```

```c
#define _USART_FRAME_STOPBITS_SHIFT          12
#define _USART_FRAME_STOPBITS_MASK           0x3000UL
#define _USART_FRAME_STOPBITS_HALF           0x00000000UL
#define _USART_FRAME_STOPBITS_DEFAULT        0x00000001UL
#define _USART_FRAME_STOPBITS_ONE            0x00000001UL
#define _USART_FRAME_STOPBITS_ONEANDAHALF    0x00000002UL
#define _USART_FRAME_STOPBITS_TWO            0x00000003UL
#define USART_FRAME_STOPBITS_HALF            (_USART_FRAME_STOPBITS_HALF << 12)
#define USART_FRAME_STOPBITS_DEFAULT         (_USART_FRAME_STOPBITS_DEFAULT << 12)
#define USART_FRAME_STOPBITS_ONE             (_USART_FRAME_STOPBITS_ONE << 12)
#define USART_FRAME_STOPBITS_ONEANDAHALF     (_USART_FRAME_STOPBITS_ONEANDAHALF << 12)
#define USART_FRAME_STOPBITS_TWO             (_USART_FRAME_STOPBITS_TWO << 12)
```

Application of library functions

- It must be known that a function:
  - Uses peripherals at what level
  - Needs what resources
  - Whether requires initialization (e.g. before sending data)
- Blocking/non-blocking functions
  - Whether the function returns or not before the end of running
  - E.g. sending data via serial port:
    - Function returns after the entire data set has been sent
    - Or the whole array containing the data to be transmitted is handled and sending is done in the background while running can be continued in the main program

Blocking data sending: entire data set has to be sent before return of the function:

Non-blocking sending: after initialization of sending, the function returns and data is being sent in the background: