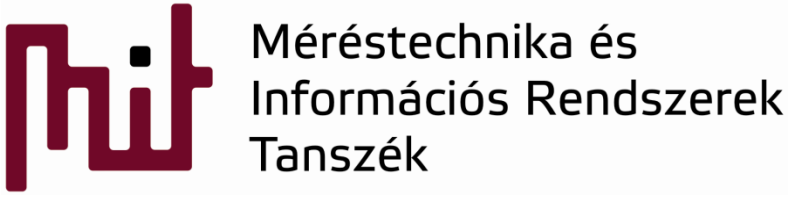
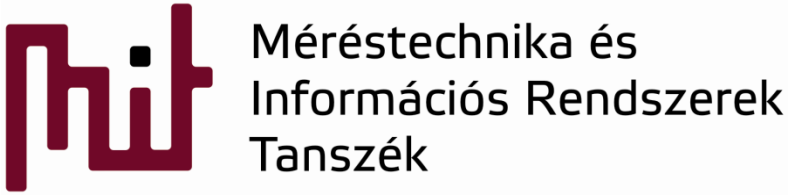
Embedded and Ambient Systems 2020.11.10.

Shared variables, memory management, robust programming



Shared variables



Shared variables

- Problem: a process has access to a certain variable or any data in a **time-overlapped manner**. Typical examples:
  - Between IT and IT
  - Between IT ans main program (certain task)
  - In case of preemptive systems
- Problem can arise if data or any structure containing coherent data can be read/written in non-atomic manner
  - Atomic operation: other process cannot interrupt running of the program during executing a certain operation
  - In high-level languages instructions seem to be uniform can be non-atomic. E.g. increment of a 32-bit number *can* be atomic on a 32-bit architecture, but surely cannot be atomic on an 8-bit architecture, since an 8-bit processor needs more ASM instruction for sure.

Shared variable: example

- Variable `dataarray` is used by both the main program and the timer IT routine

```
// declaration data structure
struct Type{
      uint32_t data1;
      uint32_t data2;
} dataarray;
// timer IT handing
void LETIMER0_IRQHandler(void){
      // writing dataarray
      dataarray.data1 = 0xFF;
      dataarray.data2 = 0xFF;
      LETIMER0->IFC = 0xFF;  // IT flag clear
}
void main(void){
  // infinite loop
while (1) {
      // writing dataarray
      dataarray.data1 = 0x00;
      dataarray.data2 = 0x00;
// in theory progam cannot step here since either 0x00 or 0xFF is the variable value
// condition: if not true that data1 and data2 value are both 0xFF or 0x00
      if (!( (dataarray.data1==0xFF && dataarray.data2==0xFF) ||
                              (dataarray.data1==0x00 && dataarray.data2==0x00)) ){
            error = true;
            BSP_LedToggle(1);
      } } }
```

Shared variable: example

- Problem: data1 and data2 fields of dataarray structure is written in a non-atomic manner:
  - IT routine may even run in the time instant when instruction `dataarray.data1 = 0x00;` has already run in the main program, but, data assignment of data2 has not run yet. In this case the following happens:
    - `dataarray.data1 = 0x00;  // main program`
    - `IT routine begins`
    - `dataarray.data1 = 0xFF;  // IT`
    - `dataarray.data2 = 0xFF;  // IT`
    - `IT routine ends`
    - `dataarray.data2 = 0x00;  // main program`
  - Final result: data1=0xFF and data2=0x00, so data becomes incosistent

Shared variable: example

- Solution: data1 and data2 fields of dataarray structure has to be written in an atomic manner
  - In IT routine it is done automatically, since no other IT routine exists. If an IT routine of higher priority existed then writing in an atomic manner should be assured
  - Writing in an atomic manner has to be assured in the main program. But how?
    - Simple solution: IT should be disabled at the beginning of writing and enabled at the end of writing
    - Function libraries of compilers/processors offer to insert atomic code parts. Its operation is also based on the disabling/enabling of IT but examines also whether an IT had already been enabled at the beginning of the critical section of the code or not

Shared variable: example

- Let's complement the main program by writing the variable in an atomic manner. In the development environment used (Simplicity Studio – Cortex-M3) the CORE_CRITICAL_SECTION(…) macro defines an atomic operation

```
while (1) {
       // critical section: let the writing be an atomic operation
      CORE_CRITICAL_SECTION(
            // writing dataarray
            dataarray.data1 = 0x00;
            dataarray.data2 = 0x00;
      )
// in theory progam cannot step here since either 0x00 or 0xFF is the variable value
// condition: if not true that data1 and data2 value are both 0xFF or 0x00
      if (!( (dataarray.data1==0xFF && dataarray.data2==0xFF) ||
                              (dataarray.data1==0x00 && dataarray.data2==0x00)) ){
            error = true;
            BSP_LedToggle(1);
      } } }
```

Shared variable: example

- Let's complement the main program by writing the variable in an atomic manner. In the development environment used (Simplicity Studio – Cortex-M3) the CORE_CRITICAL_SECTION(…) macro defines an atomic operation

```
while (1) {
      // critical section: let the writing be an atomic operation
     CORE_CRITICAL_SECTION(
          // writing dataarray
          dataarray.data1 = 0x00;
          dataarray.data2 = 0x00;
     )
// in theory progam cannot stop here since either 0x00 or 0xFF is the variable value
// condition: if not true that data1 and data2 value are both 0xFF or 0x00
     if (!( (dataarray.data1==0xFF && dataarray.data2==0xFF) ||
                    (dataarray.data1==0x00 && dataarray.data2==0x00)) ){
          error = true;
          BSP_LedToggle(1);
     } } }
```

This part is still not atomic!!!

Shared variable: example

- One problem still exists: there is a “hidden” reading in the code:
  - During the evaluation of **if** (…) condition the variable value has to be read. The following can happen:
    - We start the evaluation of condition

      (dataarray.data1==0x00 && dataarray.data2==0x00)
    - During the evaluation of (dataarray.data1==0x00) partial condition the value of variables data1 and data2 are both 0x00, therefore this partial condition is true
    - Then an IT event occurs and as a consequence both values of the variables become 0xFF. Unfortunately partial condition (dataarray.data2==0x00) becomes false due to the IT
    - So since the partial condition becomes false (and due to the inversion (!) sign at the beginning of condition) **if** condition becomes true
  - Evaluation of the condition has to be atomic (critical section)

Shared variable: example

- Correct solution: critical section should be applied to the **if** condition

```
while (1) {
      // critical section: let the writing be an atomic operation
     CORE_CRITICAL_SECTION(
          // writing dataarray
          dataarray.data1 = 0x00;
          dataarray.data2 = 0x00;
     )
// in theory progam cannot step here since either 0x00 or 0xFF is the variable value
// condition: if not true that data1 and data2 value are both 0xFF or 0x00
CORE_CRITICAL_SECTION(
     if (!( (dataarray.data1==0xFF && dataarray.data2==0xFF) ||
                         (dataarray.data1==0x00 && dataarray.data2==0x00)) ){
          error = true;
          BSP_LedToggle(1);
     }
     )
} }
```

Shared variable: example

- In our discussed example IT routine cannot be interrupted (no other process exists) therefore it can be considered atomic, therefore application of critical section is not necessary here
- The code works properly but can be still polished

Problem:

- Critical section contains not only the evaluation of condition in terms of atomic execution but also some conditionally executable instructions that make the duration the critical section long-lasting
- As a result IT may be disabled for a long time
- Application of critical section on condition **if** is not advised since IT remains disabled if the code part after the condition cannot run:

```
CORE_CRITICAL_SECTION(
    if (…)
) { … code …}
```

- Solution:
  - Making a local copy of variables inside a critical section to continue work with
  - Code parts that should be evaluated in an atomic manner should be separated and only them to be executed inside the critical section (evaluation of the condition here)

Shared variable: example

- First solution: making a local copy of variables inside a critical section to continue work with

```
// critical section: making a local copy
CORE_CRITICAL_SECTION(
        dataarray_loc.data1 = dataarray.data1;
        dataarray_loc.data2 = dataarray.data2;
)
        // continue work with local variables
        if (!( (dataarray_loc.data1==0xFF && dataarray_loc.data2==0xFF) ||
                        (dataarray_loc.data1==0x00 && dataarray_loc.data2==0x00)) ){
                error = true;
                BSP_LedToggle(1);
        }
```

Shared variable: example

- Second solution: evaluation of the condition in an atomic manner

```
// critical section: evaluation of condition
CORE_CRITICAL_SECTION(
   condition = ! ( (dataarray.data1==0xFF && dataarray.data2==0xFF) ||
            (dataarray.data1==0x00 && dataarray.data2==0x00));
)
if (condition){
       error = true;
       BSP_LedToggle(1);
}
```

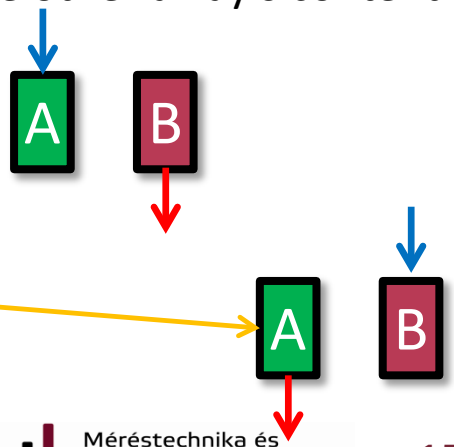
Shared variables: summary

- If data or array of data is accessed (read and/or written) by more than only one entity (IT, condition, etc…) than inconsistent data may appear, if reading and/or writing is not atomic in terms of the entire data/sequence of data
  - It can be even processor-dependent in terms of what data can be handled in an atomic manner (e.g. 8-bit or 32-bit uC)
- At every code parts where critical data can be accessed has to be done in a n atomic manner in a critical section. Typical critical (atomic) sections are:
  - IT enable/disable
  - When use the functions that come with the uP and apply them to critical code parts
  - In embedded operation systems there are synchronization primitives available (e.g. semaphores, messages, etc.)
- In order not to block the running of the program when shared variables are used the followings have to be done:
  - Only those code parts should be inserted into a critical section that contain concrete data
  - A local copy of the variables have to be created in a critical section and continue work with these local variable copies

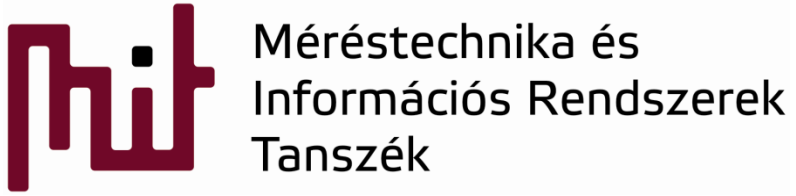
Double buffering

- Typical problem: processing a data sequence while new data keeps arriving (producer/consumer problem)
  - If only one array was applied to store data, overwriting the data would potentially happen during processing
- Double buffering:
  - When an array has been finished to be filled with data, data collection is continued in an other array and at the same time data processing is performed on the filled array. After data processing has finished the role of the two arrays are interchanged.
- Example:
  - Formatted measurement data has to be continuously sent
  - Two arrays are available: message_A and message_B.
  - Pseudo-code: (at one time one array is being filled and the other array's content is being sent, then the reverse operation takes action)

```
If (A_is_active)
    Send(message_B) // sending message is started in the background
    Create(message_A)  // composing the new message
Else
    Send(message_A) // sending message is started in the background
    Create(message_B) // composing the new message
end
```



Memory management



Memory management

- Specialties of embedded systems
  - Limited memory
  - Long-term running without restart
    - Fragmentation of memory is not allowed since that may lead to error, memory cannot be released due to the lack of restart
  - Memory protection: due to limited resource, memory protection is not always available or very limited, however there exist uCs with advanced memory protection features
- Dynamical memory usage is prohibited or be very limited
  - Dynamical memory usage: leads to fragmented memory
  - Static memory usage should be used (declaration of a variable as a constant size array)
  - Dynamical memory usage may be allowed at program initialization only but never during runtime
    - Array size is considered dynamic since its size not known during compilation but static in terms of its size is not changed after initialization

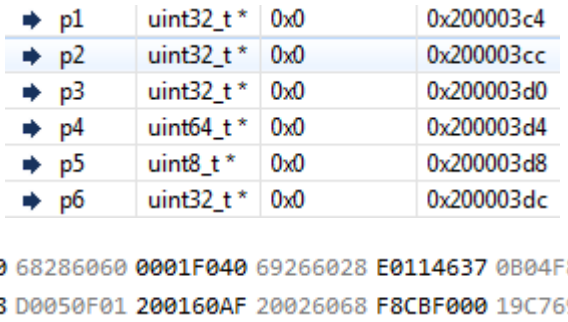
Fragmentation

- Fragmentation: allocation of memory (malloc) and later freeing them enclosures may occur, when block sizes are different the possibility of enclosures are higher
- Example:

```
uint32_t *p1 = (uint32_t *) malloc(4);
*p1 = 0x10000011;
w = *p1;
//.............
uint32_t *p2 = (uint32_t *) malloc(4);
*p2 = 0x10000012;
w = *p2;
//.............
free(p1);
//.............
uint32_t *p3 = (uint32_t *) malloc(4);
*p3 = 0x10000013;
w = *p3;
//.............
free(p3);
//.............
uint64_t *p4 = (uint64_t *) malloc(8);
*p4 = 0x2000002410000014;
w = *p4;
//.............
uint8_t *p5 = (uint8_t *) malloc(1);
*p5 = 0x15;
w = *p5;
//.............
uint32_t *p6 = (uint32_t *) malloc(4);
*p6 = 0x10000016;
w = *p6;
```

Fragmentation

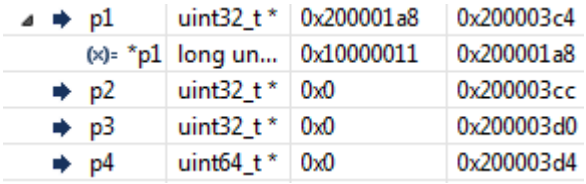
- Starting memory status
  - Only pointer
  - MEM: „memory garbage”



p1	uint32_t *	0x0	0x200003c4
p2	uint32_t *	0x0	0x200003cc
p3	uint32_t *	0x0	0x200003d0
p4	uint64_t *	0x0	0x200003d4
p5	uint8_t *	0x0	0x200003d8
p6	uint32_t *	0x0	0x200003dc

```
0x200001A8  20016020 68286060 0001F040 69266028 E0114637 0B04F851 B1121F12 0F01F110
0x200001C8  F110D0F8 D0050F01 200160AF 20026068 F8CBF000 19C769A0 44306960 BF3E4287
```

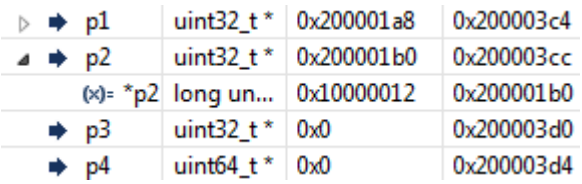
- Allocation p1
  - p1 points to addr. 0x200001a8
  - Value of memory cell is set to 0x10000011



p1	uint32_t *	0x200001a8	0x200003c4
(x)= *p1	long un...	0x10000011	0x200001a8
p2	uint32_t *	0x0	0x200003cc
p3	uint32_t *	0x0	0x200003d0
p4	uint64_t *	0x0	0x200003d4

```
0x200001A8  10000011 68286060 0001F040 69266028 E0114637 0B04F851 B1121F12 0F01F110
0x200001C8  F110D0F8 D0050F01 200160AF 20026068 F8CBF000 19C769A0 44306960 BF3E4287
```

- Allocation p2
  - p1 points to addr. 0x200001b0
  - Value of memory cell is set to 0x10000012

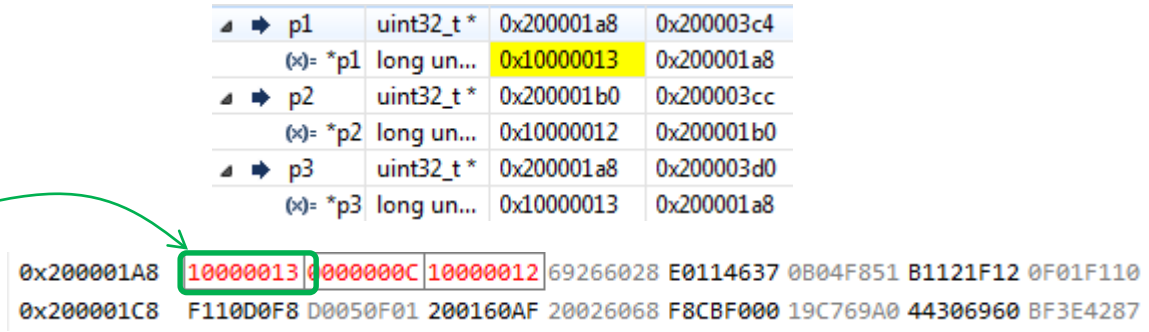


p1	uint32_t *	0x200001a8	0x200003c4
p2	uint32_t *	0x200001b0	0x200003cc
(x)= *p2	long un...	0x10000012	0x200001b0
p3	uint32_t *	0x0	0x200003d0
p4	uint64_t *	0x0	0x200003d4

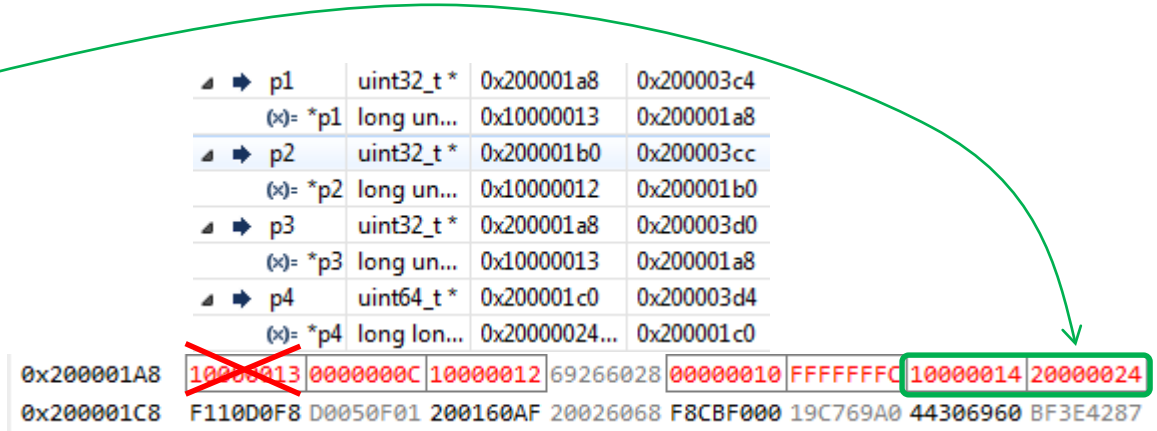
```
0x200001A8  10000011 00000000 10000012 69266028 E0114637 0B04F851 B1121F12 0F01F110
0x200001C8  F110D0F8 D0050F01 200160AF 20026068 F8CBF000 19C769A0 44306960 BF3E4287
```

Fragmentation

- Freeing p1 and allocating p3
  - In the place of pointer p1 which has been free, the content of pointer p3 exactly fits into
  - To the place where p1 points to 0x10000013 stored by p3 is inserted



- Freeing p3 and allocating p4
  - In the place of pointer p3 which has been free, the content of p3 pointer does not fit into (p3: 32 bit, p4: 64 bit)
  - To the place where p4 points to 0x1000001420000024 is inserted
  - The original value of p3 remains there but this is memory garbage since this memory part can be re-allocated later
    - Freeing a memory part not the same as deleting its content, the old value remains there



Memory management

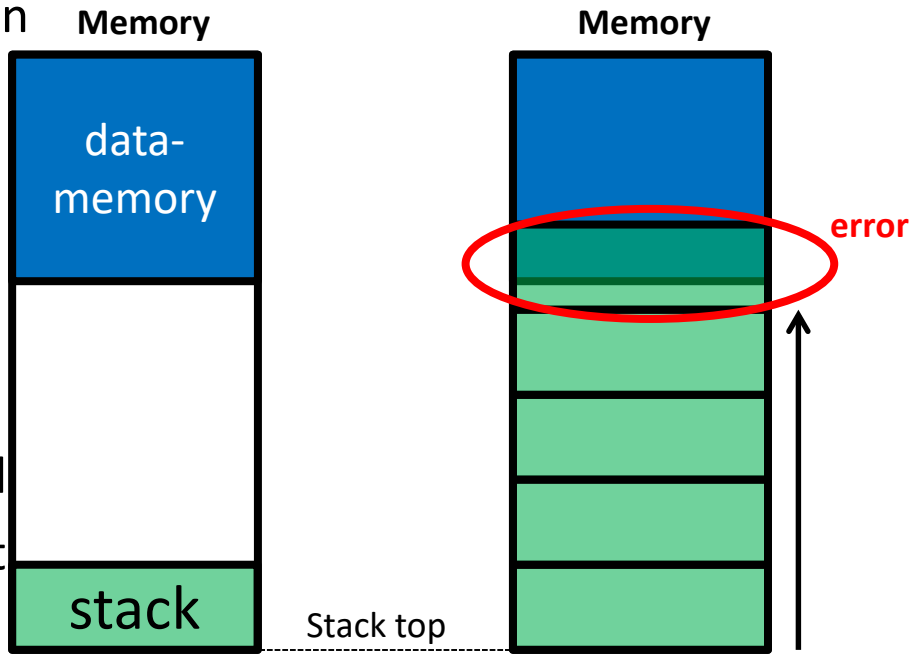
- Memory Protection Unit
  - Processor has different operation modes: Privileged / Unprivileged
  - Privileged operation mode: no limitation
  - Unprivileged operation mode: limited
  - It can be set that in unprivileged operation mode certain memory parts or peripherals should not be accessed
  - If the SW tries to access to a forbidden memory part an exception (which is an IT) is generated
  - Using this method running of even an operation system can be efficiently supported in HW

Stack overflow

- Memory is limited
- The size of the stack is changed dynamically since during function calls/It's the processor status and local variables are stored here
- If the size of stack becomes too large (e.g. local variables require large memory, recursive function calls) an overlapping of heap and stack may happen → they overwrite each other
- The problem may arise in the reverse direction: data may overwrite stack content in case of dynamical memory allocation
- By default no protection

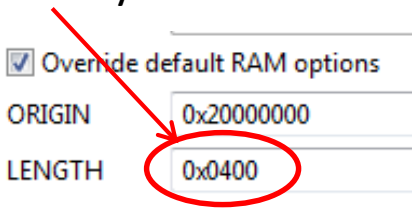
mechanism. Certain compilers may have built-in protection, e.g.:

-fstack-protector option: stack neighbourhood is filled up by given data pattern and after return it is checked whether pattern has been changed or not



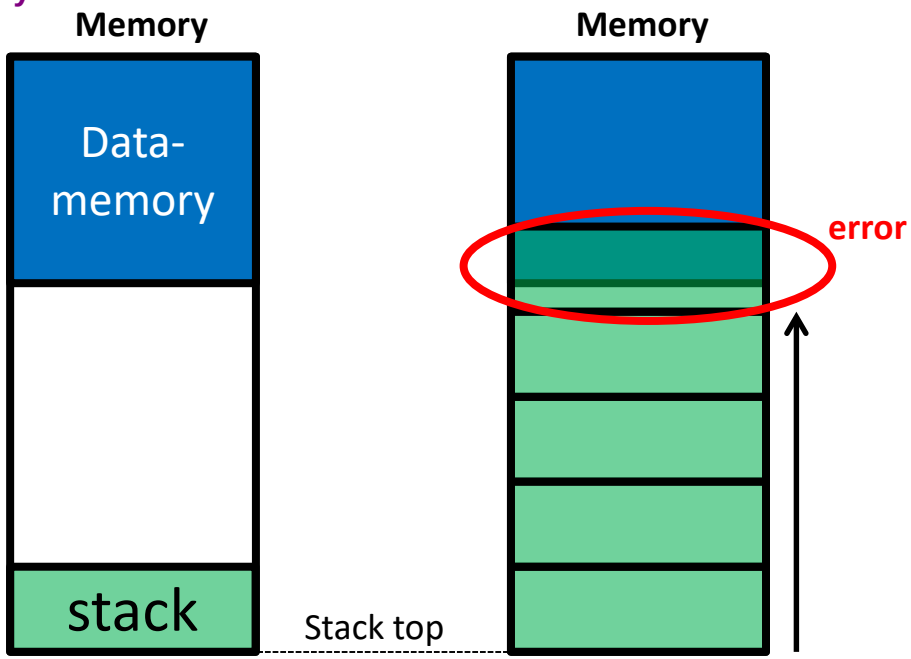
Stack overflow

- Example code: recursive function call, at every call the function allocates an array of 20 elements (80 bytes)
- In heap an array of 60 elements is allocated (240byte)
- The top of the stack is set intentionally smaller as if data memory would be 1 kB (typical data memory size of a very simple uC)
  - Project Properties: C/C++ Build → Settings → Memory Layout




```
#define N_dataarray 60
uint32_t dataarray[N_dataarray]; // in data memory
#define N_STACK 20
void big_stack(uint32_t depth)
{
      uint32_t stack_array[N_STACK];
      uint32_t iii;
      for(iii=0; iii<N_STACK; iii++){
            stack_array[iii] = 1;
      }
      if (depth<6){
            big_stack(++depth);
      }
}

void main(void)
{
   big_stack(0);
}
```



Stack overflow

- Starting

Name	Type	Value	Location
(x)= adattomb[32]	long un...	0xffffffff	0x20000128
(x)= adattomb[33]	long un...	0xffffffff	0x2000012c
(x)= adattomb[34]	long un...	0xffffffff	0x20000130
(x)= adattomb[35]	long un...	0xffffffff	0x20000134
(x)= adattomb[36]	long un...	0xffffffff	0x20000138
(x)= adattomb[37]	long un...	0xffffffff	0x2000013c
(x)= adattomb[38]	long un...	0xffffffff	0x20000140
(x)= adattomb[39]	long un...	0xffffffff	0x20000144
(x)= adattomb[40]	long un...	0xffffffff	0x20000148
(x)= adattomb[41]	long un...	0xffffffff	0x2000014c
(x)= adattomb[42]	long un...	0xffffffff	0x20000150
(x)= adattomb[43]	long un...	0xffffffff	0x20000154
(x)= adattomb[44]	long un...	0xffffffff	0x20000158
(x)= adattomb[45]	long un...	0xffffffff	0x2000015c
(x)= adattomb[46]	long un...	0xffffffff	0x20000160
(x)= adattomb[47]	long un...	0xffffffff	0x20000164
(x)= adattomb[48]	long un...	0xffffffff	0x20000168
(x)= adattomb[49]	long un...	0xffffffff	0x2000016c
(x)= adattomb[50]	long un...	0xffffffff	0x20000170
(x)= adattomb[51]	long un...	0xffffffff	0x20000174
(x)= adattomb[52]	long un...	0xffffffff	0x20000178
(x)= adattomb[53]	long un...	0xffffffff	0x2000017c
(x)= adattomb[54]	long un...	0xffffffff	0x20000180
(x)= adattomb[55]	long un...	0xffffffff	0x20000184
(x)= adattomb[56]	long un...	0xffffffff	0x20000188
(x)= adattomb[57]	long un...	0xffffffff	0x2000018c
(x)= adattomb[58]	long un...	0xffffffff	0x20000190
(x)= adattomb[59]	long un...	0xffffffff	0x20000194

After 5th level

Name	Type	Value	Location
(x)= adattomb[32]	long un...	0xffffffff	0x20000128
(x)= adattomb[33]	long un...	0xffffffff	0x2000012c
(x)= adattomb[34]	long un...	0xffffffff	0x20000130
(x)= adattomb[35]	long un...	0x5	0x20000134
(x)= adattomb[36]	long un...	0xffffffff	0x20000138
(x)= adattomb[37]	long un...	0x14	0x2000013c
(x)= adattomb[38]	long un...	0x1	0x20000140
(x)= adattomb[39]	long un...	0x1	0x20000144
(x)= adattomb[40]	long un...	0x1	0x20000148
(x)= adattomb[41]	long un...	0x1	0x2000014c
(x)= adattomb[42]	long un...	0x1	0x20000150
(x)= adattomb[43]	long un...	0x1	0x20000154
(x)= adattomb[44]	long un...	0x1	0x20000158
(x)= adattomb[45]	long un...	0x1	0x2000015c
(x)= adattomb[46]	long un...	0x1	0x20000160
(x)= adattomb[47]	long un...	0x1	0x20000164
(x)= adattomb[48]	long un...	0x1	0x20000168
(x)= adattomb[49]	long un...	0x1	0x2000016c
(x)= adattomb[50]	long un...	0x1	0x20000170
(x)= adattomb[51]	long un...	0x1	0x20000174
(x)= adattomb[52]	long un...	0x1	0x20000178
(x)= adattomb[53]	long un...	0x1	0x2000017c
(x)= adattomb[54]	long un...	0x1	0x20000180
(x)= adattomb[55]	long un...	0x1	0x20000184
(x)= adattomb[56]	long un...	0x1	0x20000188
(x)= adattomb[57]	long un...	0x1	0x2000018c
(x)= adattomb[58]	long un...	0x20000198	0x20000190
(x)= adattomb[59]	long un...	0x25d	0x20000194

Robust programming

Méréstechnika és
Információs Rendszerek
Tanszék

Checking of indexing

- By default no index checking in C
- Take care of array indexing
- Take care of pointer handling (do not use “tricks” with pointers)

Timeout mechanism

- Errors made by programmer person may happen
  - Malfunction has to be counted for (e.g. processor freeze)
  - Errors have to be recognized and handled
- Timeout mechanism: after certain time interval the processor has to provide some signal of operation:
  - Watchdog timer: usually it is a built-in service
  - Program components, dedicated units to send keep-alive messages
  - There exist HW components that need to be polled by the processor periodically

Safe coding, coding conventions

- Application of safe coding and coding conventions
  - Depending on the industrial field but even from company to company different presciptions may exist how to write the code in a standardised way (MISRA).
  - Examples:
    - Use informative names for variables
    - Variable name should follow the type, e.g.: **f**Length (**f**loat), **i**Count (**i**nteger): Hungarian notation
    - Use brackets after „if”

```
if (a>b)
    a = a – b ;
    b = a + 10;   // it does not belong to „if” although it seems
```

    - In conditional structure when a comparison with a constant is applied the constant should come first:
      - Apply this way: **if (5==i)** and not **if (i==5)**
      - Explanation: if a mistake is made **if (5=i)** results in a compilation error, while **if (i=5)** causes no error but the operation is not as expected
  - Language restrictions may be applied: certain language elements are blocked to be used since its operation is not safe (e.g. avoid the use of functions with variable number of elements in the argument, make clean code)

Stuctured programming

- Stuctured programming
  - Make a clean code do not use overcomplicated solutions
  - Use variables with as low visibility as possible (do not use global variables if not needed)
  - Take care of special variables (e.g. mutual exclusion), these should be distinctive in their names
  - Hierarchical structure should be applied:
    - Application level
    - Functions belonging to the device (e.g. development board, entire module)
    - Functions belonging to the processor
    - Functions belonging to the peripherals
  - Hierarchical structure helps code portability

Stuctured programming

- Stuctured programming
  - Modularity: groups of functions should be in separated files
    - .h files: contain the function and variable declaration but not the implementation
    - .c files: contain the function implementation
    - Use informative file names
  - Consistent names should be applied
  - Function names should be informative and consequent, pl.: ADC_Init(…), Timer_Init(…), UART_Init(…).
  - Use as few low level HW access as possible and use the dedicate
  - Comment everything!!!
  - Warnings should be conidered!!!

Explicit type usage

- Type conversion done in an explicit way
  - Example: number_16bit = number1_8bit * number2_8bit;
    - 16-bit result of multiplication is expected. In this case the compiler may transform the result of multiplication into an 8-bit number and after that casting to 16-bit does not help.
  - Instead: number_16bit = (int16_t) number 1_8bit * (int16_t) number 2_8bit;
    - It will be clear for the compiler that a 16-bit multiplication with 16-bit result is expected

Redundancy

- Redundancy, double or triple checking
  - Check whether a certain intervention had the expected result or not
    - For instance, if the level of a certain pint is set it should be checked either polling it internally or externally
  - Certain task should be performed in different ways and compare the results
    - May happen that different programmer group work on different variants of the code
    - There exist such processor that has two separated cores running the same code (lockstep). In case of some implementations the cores are physically turned relative to each other or delayed (delayed lockstep) in operation to experience external disturbances in different ways