SEGGER

emBO++ 2023

Using Embedded Real-time Operating Systems with C++

Introduction

Til Stork

Product manager at SEGGER for RTOS embOS

>20 years experience in the embedded industry

>16 years experience in RTOS development

A few years experience in C++ development.
(Real time operating systems are usually developed in C and assembler)

Using C and assembler for RTOS development (and not C++)

- C is portable and widely support by development tools
- Migration from assembler to C was easier than to C++
- A small fraction of an RTOS needs to be written in assembler (context switch, C is not able to access CPU register)
- C and its disadvantages are well known (e.g. MISRA-C guidelines)
- Some C++ feature require dynamic memory allocation which is usually avoided in embedded systems

Bare metal programming

```c
static unsigned long Time;

void SysTick_Handler(void) {
  Time++;
}

static void Task1000(void) {
  DoSomething();
}

static void Task100(void) {
  DoSomething();
}

static void Task10(void) {
  DoSomething();
}

int main(void) {
  OS_InitHW();
  //
  // Start superloop
  //
  while (1) {
    if ((Time % 1000) == 0) {
      Task1000();
    }

    if ((Time % 100) == 0) {
      Task100();
    }

    if ((Time % 10) == 0) {
      Task10();
    }
  }
  return 0;
}
```

A timer interrupt (SysTick_Handler()) increments the variable Time periodically (e.g. every millisecond).

The super-loop checks for certain timeouts and call according actions Task10(), Task100() and Task1000().

For example at Time == 200 Task10() and Task100() but not Task1000() will be executed.

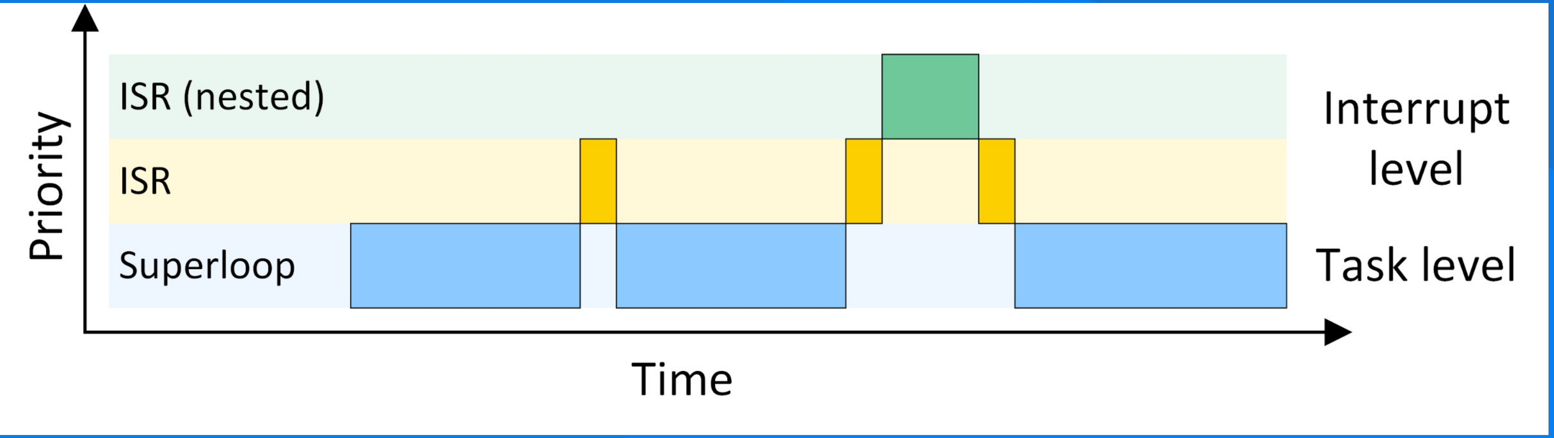
Task100() has higher priority than Task10().
But maybe that is not intended?

Disadvantages:
What happens when Task100() takes > 10 ticks to complete? No real-time behavior.

No energy saving because the loop always executes.

Bare metal programming

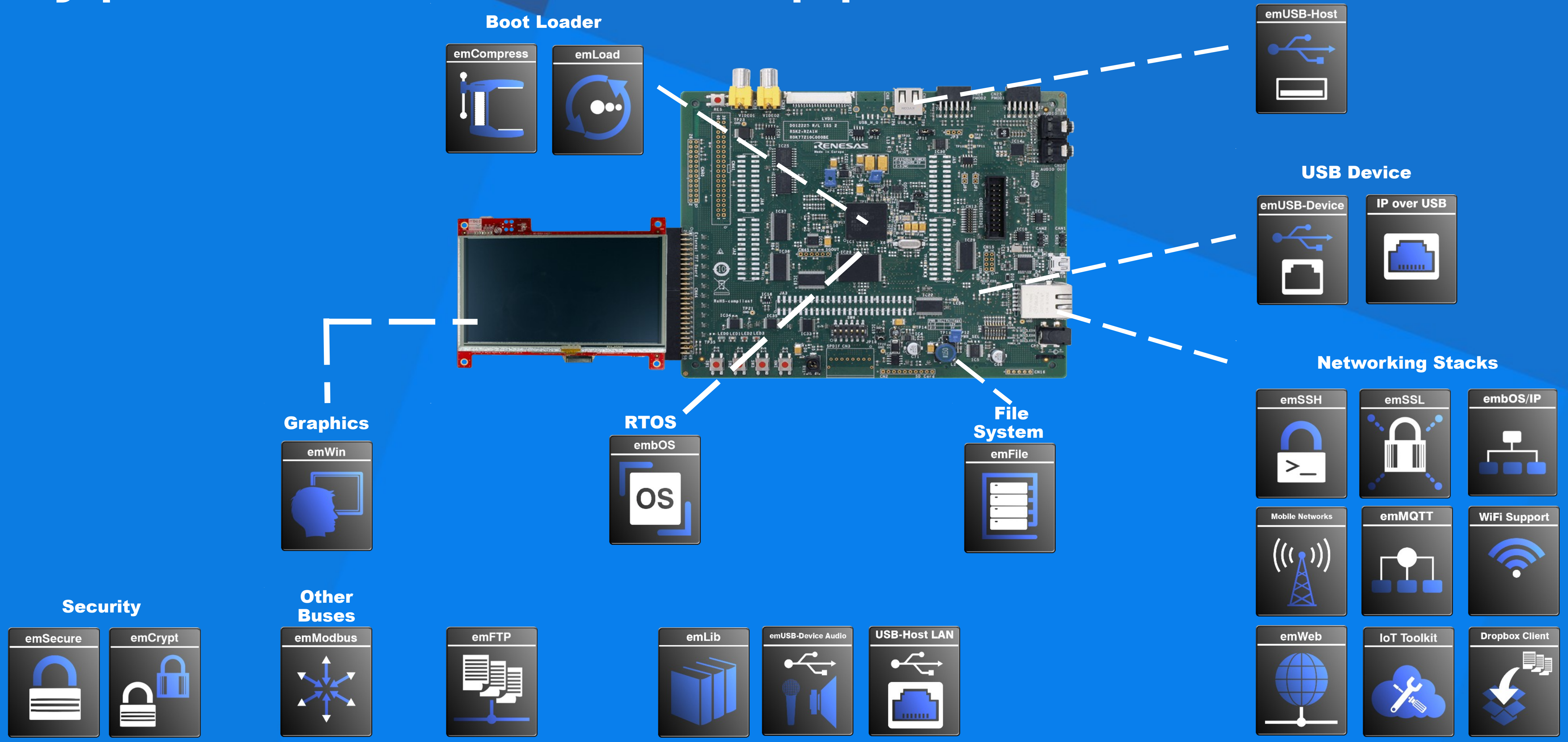
Super-loop diagram



Interrupts service routines (ISRs) can preempt the super-loop

Only ISRs can execute application jobs with higher real-time requirements.

A typical embedded application

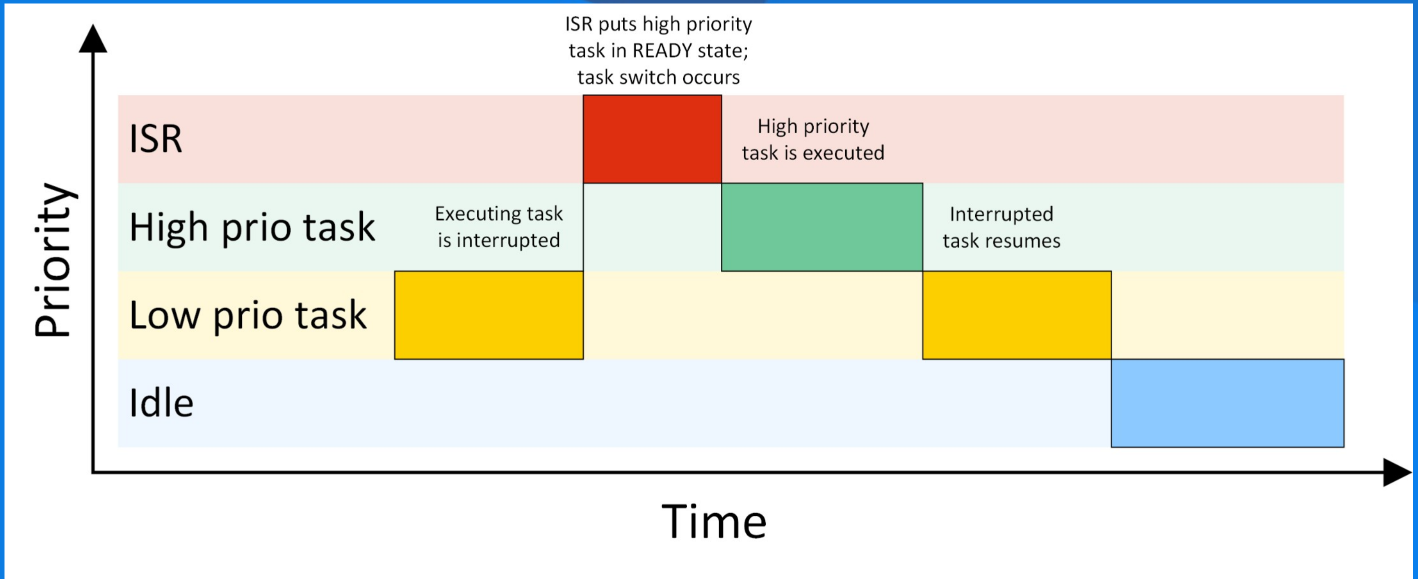


RTOS programming

```c
static void Task1000(void) {
  OS_TIME t0;

  t0 = OS_TIME_GetTicks();
  while (1) {
    DoSomething();
    t0 += 1000;
    OS_TASK_DelayUntil(t0);
  }
}

static void Task100(void) {
  OS_TIME t0;

  t0 = OS_TIME_GetTicks();
  while (1) {
    DoSomething();
    t0 += 100;
    OS_TASK_DelayUntil(t0);
  }
}

static void Task10(void) {
  OS_TIME t0;

  t0 = OS_TIME_GetTicks();
  while (1) {
    DoSomething();
    t0 += 10;
    OS_TASK_DelayUntil(t0);
  }
}

int main(void) {
  OS_Init();    // Initialize embOS
  OS_InitHW();  // Initialize required hardware
  OS_TASK_CREATE(&TCB1000, "Task 1000", 100, Task1000, Stack1000);
  OS_TASK_CREATE(&TCB100,  "Task 100",  101, Task100,  Stack100);
  OS_TASK_CREATE(&TCB10,   "Task 10",   102, Task10,   Stack10);
  OS_Start();   // Start embOS
  return 0;
}
```

- The application can be divided into separate tasks
- Each task has its own priority
- This makes it much easier to fulfill real-time requirements

Preemptive multitasking diagram



- High priority tasks can preempt low priority tasks.
- The higher priority task Task10() can preempt the low priority tasks Task100() and Task1000() whenever necessary.
- ISRs can preempt any tasks.

C++/RTOS challenges

- Thread safety
- Heap fragmentation
- Using embOS API in C++ constructor of global objects

Thread safety

```
class MyClass {
public:
  MyClass() {
  }

  ~MyClass() {
  }

  int x;
};

static void Task_A(void) {
  MyClass *m;

  while (1) {
    m = new MyClass;
    m->x = 1;
    delete m;
  }
}

static void Task_B(void) {
  MyClass *m;

  while (1) {
    m = new MyClass;
    m->x = 2;
    delete m;
  }
}
```

- Task_A can preempt Task_B at any time, also during an heap operation

- The heap operation is indirectly called from the new operator

```
void __heap_lock()
malloc()
operator new(unsigned int)()
void Task_A()
```

- The heap operation malloc() must be atomic

- Implementation of __heap_lock() and __heap_unlock() are provided by the RTOS vendor and e.g. disable/enable interrupts (or a mutex is used)

Heap fragmentation

```
class MyClass {
public:
  MyClass() {
  }

  ~MyClass() {
  }

  int x;
};

static void Task_A(void) {
  MyClass *m;

  while (1) {
    m = new MyClass;
    m->x = 1;
    delete m;
  }
}

static void Task_B(void) {
  MyClass *m;

  while (1) {
    m = new MyClass;
    m->x = 2;
    delete m;
  }
}
```

- Not a specific C++ issue, also happens with C and malloc() / free()
- No heap memory could be left
- Memory allocation doesn't guarantee real-time behavior
- Depends on the used toolchain/standard library
- SEGGER has a solution:
  SEGGER's new real-time allocator

C/C++ runtime initialization

The firmware does not start at main() but at a Reset() function which performs the runtime initialization.

The runtime initialization:

1. Executes the segment initialization (copy data to .data, zero .bss, etc.)

```
static int foo = 42;
static int bar;

int main(void) {
  return 0;
}
```

2. Calls static C++ constructors before main()

3. Calls main()

RTOS initialization

The RTOS API can be used after RTOS initialization only.
embOS API assume that internal OS variables are initialized.

```
int main(void) {
  OS_Init();                                          // Initialize embOS
  ...
  OS_TASK_CREATE(&TCBHP, "Task A", 100, HPTask, StackHP);  // Create task
  OS_MUTEX_Create(&mutex);                            // Create mutex
  OS_Start();                                         // Start embOS
  return 0;
}
```

Using embOS API in C++ constructor of global objects

```
#include "RTOS.h"
#include "stdlib.h"

class foo {
public:
  foo(void) {
    OS_MUTEX_Create(&m);  // Called before OS_Init()
  }
private:
  OS_MUTEX m;
};

static foo MyFoo;

int main(void) {
  OS_Init();   // Initialize embOS
  OS_Start();  // Start embOS
  return 0;
}
```

- The runtime initialization calls the foo() constructor before main().
- The constructor calls OS_MUTEX_Create() which is illegal before OS_Init().

The solution is toolchain specific but the basic idea is always similar:

- The startup code must not call the C++ constructors
- This can be disabled by e.g. a linker option (IAR EWARM) or modifying the linker file (SEGGER Embedded Studio)
- The application calls all static constructors after OS_Init()

```
#include "RTOS.h"
#include "stdlib.h"
#include "iar_dynamic_init.h"

class foo {
public:
  foo(void) {
    OS_MUTEX_Create(&m); // Called after OS_Init()
  }
private:
  OS_MUTEX m;
};

static foo MyFoo;

int main(void) {
  OS_Init();                        // Initialize embOS
  __iar_dynamic_initialization();   // Late initialization of constructors
  OS_Start();                       // Start embOS
  return 0;
}
```

IAR

```
#include "RTOS.h"
#include "stdlib.h"

class foo {
public:
  foo(void) {
    OS_MUTEX_Create(&m); // Called after OS_Init()
  }
private:
  OS_MUTEX m;
};

static foo MyFoo;

int main(void) {
  OS_Init();    // Initialize embOS
  init_ctors(); // Late initialization of constructors
  OS_Start();   // Start embOS
  return 0;
}
```

Embedded Studio

Developing RTOS application in C++

- These days more and more developers are using C++ instead of C
- embOS sources must still be built with a C compiler
- Although the RTOS is not written in C++, the application can be written in C++ and compiler with a C++ compiler in a combined project
- ‘extern “C ”’ statement must be added to C prototypes

```
#ifdef __cplusplus
  extern "C" {
#endif

void OS_TASK_Create(...);

#ifdef __cplusplus
  }
#endif
```

embOS: C API calls

```c
static OS_MUTEX mutex;

static void Task_A(void) {
  OS_MUTEX_Create(&mutex);

  while (1) {
    OS_MUTEX_LockBlocked(&mutex);
    BSP_ToggleLED(0);
    OS_MUTEX_Unlock(&mutex);
  }
}

static void Task_B(void) {
  while (1) {
    OS_MUTEX_LockBlocked(&mutex);
    BSP_ToggleLED(0);
    OS_MUTEX_Unlock(&mutex);
    mutex.UseCnt = 42; // Illegal access
  }
}
```

- OS_MUTEX is a struct
- The application must not modify the struct member directly
- C does not limit the member access
- Programming mistakes can cause issues
- Could cause incompatibility issues when e.g. internal members are renamed

embOS++: C++ wrapper classes

```
class OS_CLASS_Mutex {
public:
  OS_CLASS_Mutex() {
    OS_MUTEX_Create(&m);
  };

  ~OS_CLASS_Mutex() {
    OS_MUTEX_Delete(&m);
  };

  void Lock(void) {
    OS_MUTEX_Lock(&m);
  }

  void Unlock(void) {
    OS_MUTEX_Unlock(&m);
  }

private:
  OS_MUTEX m;
};
```

```
static OS_CLASS_Mutex mutex;

static void Task_A(void) {
  while (1) {
    mutex.Lock();
    BSP_ToggleLED(0);
    mutex.Unlock();
  }
}

static void Task_B(void) {
  while (1) {
    mutex.Lock();
    BSP_ToggleLED(0);
    mutex.Unlock();
    mutex.m.UseCnt = 42; // Compiler error
  }
}
```

Advantages

- Access to private members will cause compiler error
- All embOS RTOS object classes could be located in a separate namespace like SEGGER::OS.
  This reduces the risk of a naming conflict with 3rd party software.
- This might be available for embOS

C++ multi-threading interface

- The C++11 standard introduced unified multi-threading interface (std::thread, std::mutex, …)

```
void Task_A(string msg) {
  cout << "Task says: " << msg;
}

int main(void) {
  thread t1(Task_A, "Hello");
  t1.join();
}
```

- It is up to the compiler vendor how to implement it
- Multi-threading requires an operating system is present
- OS is beyond the C++ standard definition
- Interface not designed for RTOS

Just for fun: Implicit usage of interrupt API

- Using C API to call BSP_ToogleLED() with disabled interrupts

```
static void Task_A(void) {
  while (1) {
    OS_INT_Disable();
    BSP_ToggleLED(0);
    OS_INT_Enable();
  }
}
```

- Using a C++ object in a block scope to disable/enable interrupts

```
class critical {
public:
   critical() {
      OS_INT_Disable();
   };

   ~critical() {
      OS_INT_Enable();
   };
};

static void Task_A(void) {
  while (1) {
    critical section;
    BSP_ToggleLED(0);
  }
}
```

```
           while (1) {
           critical c;
00001800     9000        str r0, [sp]
00001802     F001F87A    bl 0x000028FA <critical::critical()>
           —— OS_Start2Tasks.cpp — 55 ——
           BSP_ToggleLED(0);
00001806     2000        movs r0, #0
00001808     F002F815    bl 0x00003836 <BSP_ToggleLED>
0000180c     9800        ldr r0, [sp]
           —— OS_Start2Tasks.cpp — 56 ——
           }
0000180e     F001F87C    bl 0x0000290A <critical::~critical()>
           —— OS_Start2Tasks.cpp — 53 ——
           while (1) {
00001812     E7F4        b 0x000017FE
           —— OS_Start2Tasks.cpp — 55 ——
           BSP_ToggleLED(0);
           }
           }
```

Q&A