Whitepaper: Co-operative Scheduler (COS) for openCM

Version 0.1

Ernst Forgber

Hochschule Hannover, Germany
University of Applied Sciences and Arts
Faculty 1

November 28, 2016

Abstract

This whitepaper gives an introduction to COS, a simple stack-less, co-operative tasking system for micro-controllers. COS is implemented in C, has low memory demand per task and is easily ported to different micro-controllers

Contents

List of Figures

List of Tables

1 Introduction

This is the implementation of a simple stack-less, co-operative scheduling system (COS). A language extension of the gnu C-compiler 'gcc' is used to implement the co-routine pattern to realize a task (see R. Dunkels: 'protothreads' [1], Michael Dorin: 'Building 'instant-up' real-time operating systems' [2] and Ernst Forgber 'Multitasking mit AVR RISC-Controllern' [3]). The scheduler may run on any system providing a gnu gcc-compiler.

2 Installation

This is COS for platforms openCM (ARM Cortex-M3) and Arduino. On these platforms, no further hardware resources as timers or interrupts are required. COS uses `millis`() as system clock. See documentation of file `CosScheduler.cpp` for an overview of the functionality COS provides.

Please extract the files of the archive and copy them to your 'libraries' directory. After that, the 'libraries' directory should have this structure:

```
libraries/
    CosScheduler/
                examples/
                html/
                utility/
```

Restart the IDE. To view the documentation of COS, go to 'html' sub-directory and open file 'index.html' in your browser.

Make sure that in file `libraries/CosScheduler/utility/cos_configure.h` the correct platform is selected by un-commenting the corresponding definition:

```
/*****************************************************/
/******* select the platform COS will be running on ***/
/******* un-comment ONLY ONE of the following options */
/*****************************************************/
#define COS_PLATFORM          PLATFORM_OPEN_CM_9_04
//#define COS_PLATFORM        PLATFORM_ARDUINO
//#define COS_PLATFORM        PLATFORM_RENESAS_RX63N
```

3 Principle of Operation

COS is an implementation of a simple stack-less, co-operative tasking system. It uses a language extension of the gnu C-compiler 'gcc' to implement the co-routine pattern.

A 'co-routine' has several exit points and will resume where it exited before. The implementation uses macros, building up a 'switch'-statement with line numbers a 'case' marks and 'return' statements at co-operative scheduling points.

When reaching a scheduling point, the task function executes a 'return' statement and will forget all its local variables, except of those labeled 'static'. The scheduling macros will ensure that before returning, the task will store its current code line number in the task-struct. The next time, the scheduler calls the task-function, execution will resume at that stored line of code, implemented as a 'case' mark in the 'switch' statement.

The scheduling macros have to be called by the task-function directly. They must not be used in nested function calls.

A task-function is a callback-function. It either runs to its very end and will then be deleted from the task-list, or it reaches a scheduling point (see macros below) and will execute a co-routine return storing the current code line number for the next activation.

A system clock is mandatory, in order to start the task callback-functions on time. The function `_gettime_Ticks`(**void**) has to be implemented on the platform, COS is running on. Time is measured in 'ticks'. The system time in Ticks will be returned by `_gettime_Ticks`(**void**). Time measurement may be implemented by a timer interrupt, or if available, it may be provided by an operating system.

The scheduler will call the task callback-functions with respect to task-state, task-priority and time of last activation.

Among the tasks in state `TASK_STATE_READY`, the one with highest priority will be activated if it is time to run, i.e. if: `timeNow`−`timeLastActivation` > `sleepTime_Ticks`.

The scheduler uses time differences, therefore timer wrap-around will not cause problems, time differences will be correctly computed, as long as the time interval is shorter as a complete turn-around of the tick counter.

Before the task-function is called, the scheduler will set the sleep-time `sleepTime_Ticks` of the task to 0. The task-function has to specify its sleep-time each time it is activated, see examples below.

The task-list is never empty, there is at least the idle-task with its callback-function **static void** `idleTask`(`CosTask_t` ∗`task_pt`).

All task callback-functions return nothing and have a single parameter `CosTask_t`∗.

The scheduler runs in an endless loop. The task-list will never be deleted and there is no end of program.

Before running the scheduler, the function **int** `COS_InitTaskList`(**void**) has to be called. It will initialize the task-list and register the idle-task.

The scheduler may run in one of two modes: priority based of round-robin scheduling.

Priority based scheduling requires the task-list to be sorted due to priority. In this mode, a coarse estimation of CPU-load may be achieved by means of two tasks: The CPU-load-task has maximum priority and will reset a counter to 100. The idle-task with lowest priority decrements that counter. The task periods have to be adjusted to run the idle-task 100 times, while the CPU-load-task runs once, provided there are no other tasks running. The counter value is an estimate of CPU-load: With growing CPU-load, the idle-task may be suppressed and will not be able to decrement the counter to 0 any more. At maximum load, idle will be totally suppressed by other tasks and the counter value will remain 100, meaning 100% CPU-load.

In round-robin mode, the CPU-load estimation won't work, but will do no harm either.

4 Code Examples

In directory `libraries/CosScheduler/examples` several examples for COS usage are given. Please copy the `*.ino` file to a new 'sketch', save it, compile and run the example code.

4.1 Mini Example: Blinking LED

In the first example, two tasks are used to blink a LED and to print time information on a serial terminal. The code is written for the openCM platform. Programs for Adruino and openCM are quite similar. For this example the main difference affects serial output: Arduino uses `Serial.print`(...), while openCM uses `SerialUSB.print`(...).

Figure 1 shows a data-flow diagram of the program. Both task print output on a serial terminal. The LED-task switches a LED on an off.

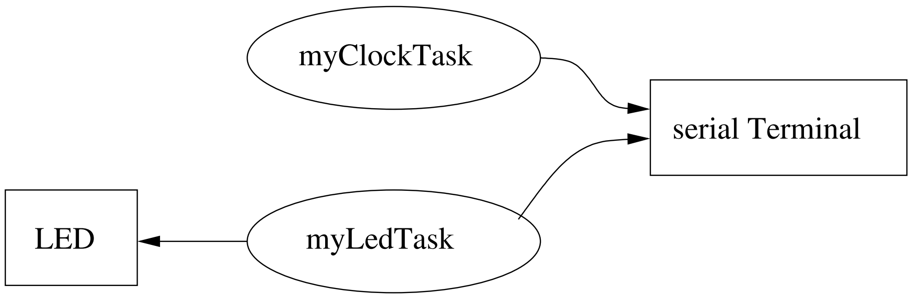

Figure 1: Data-flow diagram of the blinking LED example.

The following listing shows the code for openCM:

```
#include <CosScheduler.h>

const int pin = BOARD_LED_PIN;
CosTask_t *tA, *tB;         /*!< pointer to tasks */


void myClockTask(CosTask_t *pt)
{   static uint8_t second=0, minute=0, hour=0;
    COS_TASK_BEGIN(pt);

    SerialUSB.print("Clock started\r\n");
    while(1)
    {
        SerialUSB.print(hour);   SerialUSB.print(":");
        SerialUSB.print(minute); SerialUSB.print(":");
        SerialUSB.print(second); SerialUSB.print("\r\n");
        COS_TASK_SLEEP(pt,_milliSecToTicks(1000));
        second++;
        if(second>=60)
        {  second = 0;
           minute++;
           if(minute>=60)
           {  minute = 0;
              hour++;
              if(hour>=24)
              {  hour=0;
              }
           }
        }
    }
    COS_TASK_END(pt);
32 }
33
34 /*-----------------------------------------------------*/
35
36  void myLedTask(CosTask_t *pt)
37 {   COS_TASK_BEGIN(pt);
38
39     SerialUSB.print("LED blink started\r\n");
40     while(1)
41     {   COS_TASK_SLEEP(pt,_milliSecToTicks(200));
42         digitalWrite(pin, HIGH);
43         COS_TASK_SLEEP(pt,_milliSecToTicks(200));
44         digitalWrite(pin, LOW);
45     }
46     COS_TASK_END(pt);
47 }
48
49 /*----------------------------------------------------*/
50
51 void setup()
52 {
53     pinMode(pin, OUTPUT);
54     CosInitTaskList();
55     tA = CosCreateTask(1, NULL, myClockTask);
56     tB = CosCreateTask(2, NULL, myLedTask);
57 }
58
59 void loop()
60 {   delay(2000);
61     CosVersionInfo();
62     CosPrintTaskList();
63     CosRunScheduler();  // endless loop...
64 }
```

This is a short description of the code:

code line 1: In order to use COS, the header file `CosScheduler.h` is included.

code line 3: The LED pin on openCM is defined as macro, on Arduino use pin 13 instead.

code line 4: Pointers to tasks usually are global, so a tasks may know each other.

code line 7: A task-callback-function returns nothing. It gets a pointer to its task struct.

code line 8: Local task variables are persistent when defined **static**.

code line 9: This macro has to be used at the beginning of each task.

code line 12: The task runs in an 'endless' loop, only interrupted at co-operative scheduling macros.

code line 17: This scheduling macro lets the task sleep and invokes the scheduler. When the sleep time has expired, the task becomes ready again. The Scheduler will activate the task and it will resume operation at code line 18.

code line 31: This macro has to be used at the end of each task. Even though it seems to be unreachable code due to the **while**(1) loop, the macro is necessary to complete the program code of a task.

code line 54: The `setup`() function has to initialize the task-list.

code line 55: In order to add a task to the task-list, `CosCreateTask`() has to be called.

code line 62: For debug purposes, it may be helpful to print the task-list on the terminal.

code line 63: The scheduler is started and will run in an endless loop. It will permanently search the task-list and will activate tasks that are ready to run according to their priority.

4.2 Producer - Consumer

The second example shows two tasks working as producer and consumer. Again, the code is given for openCM. For Arduino, change `SerialUSB.print`() to `Serial.print`().

A data-flow diagram is given in figure 2. Inter-task communication is implemented by a FIFO with several data slots.

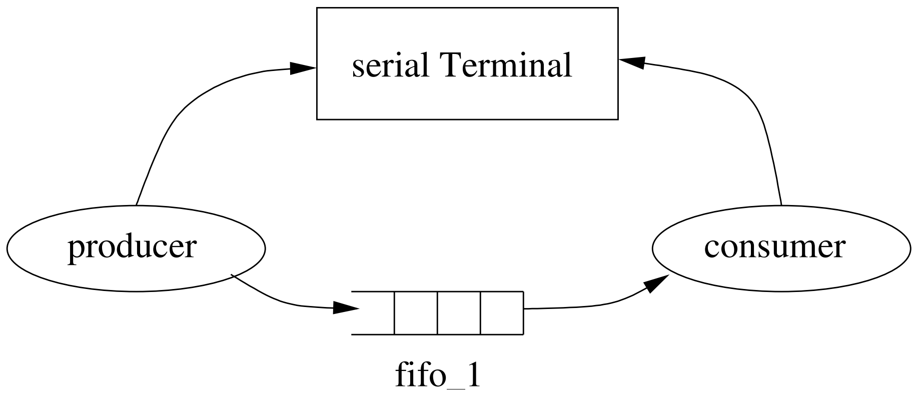

Figure 2: Data-flow diagram of the producer-consumer example.

The following listing shows the code for openCM:

```
#include <CosScheduler.h>

/*! new 3D vector data type */
typedef struct
{ double x; /**< x-coordinate */
  double y; /**< y-coordinate */
  double z; /**< z-coordinate */
}Vector_t;

CosTask_t *tProducer, *tConsumer;    /*!< pointer to tasks */
CosFifo_t fifo_1; /*!< global fifo for inter-task communication */

/*!
 **************************************************************
 * @par Description:
 *  The task sends a value of type 'Vector_t' through a FIFO to
 *  consumerTask() in a loop. The producer synchronizes to the
 *  consumer.
 * @see consumerTask()
 * @arg  -
 * @param  pt              - IN/OUT, pointer to task
 ****************************************************************/
 void producerTask(CosTask_t *pt)
{   static int16_t x=0;
    static Vector_t vec = {1,1,1};
    COS_TASK_BEGIN(pt);

    SerialUSB.print("\r\nProd started\r\n");
    for (x=0; x<100; x++)
    {     vec.x += 1.5;
          vec.y *= 1.1;
          vec.z -= 0.3;
          SerialUSB.print("prod sends: vec=[");
          SerialUSB.print(vec.x); SerialUSB.print(", ");
          SerialUSB.print(vec.y); SerialUSB.print(", ");
          SerialUSB.print(vec.z); SerialUSB.print("]");
          SerialUSB.print("\r\n");
          COS_FifoBlockingWriteSingleSlot(pt, &fifo_1, &vec);
          COS_TASK_SLEEP(pt,_milliSecToTicks(200));
    }
    SerialUSB.print("Prod ends\n");
    COS_TASK_END(pt);
}

/*!
 ***************************************************************
 * @par Description:
 * Receives a value of type 'Vector_t' from producerTask() through
 * a FIFO. The consumerTask() synchronizes with  producerTask().
 * @see consumerTask()
 * @arg  -
 * @param  pt              - IN/OUT, pointer to task
 ****************************************************************/
 void consumerTask(CosTask_t *pt)
{
    static Vector_t vec={0,0,0};
    COS_TASK_BEGIN(pt);

    SerialUSB.print("\r\nCons started\r\n");
    while(1)
    {     COS_FifoBlockingReadSingleSlot(pt, &fifo_1, &vec);
          SerialUSB.print("cons: vec=[");
          SerialUSB.print(vec.x); SerialUSB.print(", ");
          SerialUSB.print(vec.y); SerialUSB.print(", ");
          SerialUSB.print(vec.z); SerialUSB.print("]");
          SerialUSB.print("\r\n");
          COS_TASK_SLEEP(pt,_milliSecToTicks(1000));
    }
    SerialUSB.print("Cons ends\r\n");
    COS_TASK_END(pt);
71 }
72 /*--------------------------------------------------*/
73
74 void setup()
75 {
76     CosInitTaskList();
77     COS_FifoCreate(&fifo_1, sizeof(Vector_t), 5);
78     tProducer = CosCreateTask(4, NULL, producerTask);
79     tConsumer = CosCreateTask(5, NULL, consumerTask);
80 }
81
82 void loop()
83 {   delay(2000);
84     SerialUSB.print("\r\nopenCM test: send a vector via FIFO\r");
85     delay(2000);
86     CosVersionInfo();
87     CosPrintTaskList();
88     CosRunScheduler();  // endless loop
89 }
```

This is a short description of the code:

code line 1: In order to use COS, the header file `CosScheduler.h` is included.

code line 4-8: A new data type is defined. The FIFO will be able to store values of this type.

code line 10: Pointers to tasks usually are global, so a tasks may know each other.

code line 11: The FIFO is defined, but not yet initialized. It has to be global, in order to be visible to both tasks.

code line 23: The producer task-callback-function returns nothing. It gets a pointer to its task struct.

code line 24-25: Local task variables are persistent when defined **static**.

code line 26: This macro has to be used at the beginning of each task.

code line 29: The producer runs in a loop. It writes data of type `Vector_t` to the FIFO. As long as free data slots are available in the FIFO, the write operation will not block.

code line 38: Writing to the FIFO is done by calling a co-operative scheduling macro. If the FIFO is full, the scheduler will perform a task-switch.

code line 39: This scheduling macro lets the task sleep and invokes the scheduler. When the sleep time has expired, the task becomes ready again. The Scheduler will activate the task and it will resume operation at code line 28.

code line 42: This macro has to be used at the end of each task. The macro is necessary to complete the program code of a task. When it is executed, it deletes the task from the task-list and frees some memory.

code line 56: The consumer task has a local variable, to execute a blocking read from the FIFO.

code line 61: If there is data in the FIFO, this macro will read a slot and store the data in variable `vec`. If the FIFO is empty, the scheduler will perform a task-switch.

code line 67: The consumer has a longer sleep-time than the producer. Hence, the producer will fill up the FIFO until it is full and subsequently synchronize to the slower consumer read operations.

code line 76: The `setup()` function has to initialize the task-list.

code line 77: The FIFO is initialized for the proper data type and required number of data slots.

code line 78: In order to add a task to the task-list, `CosCreateTask()` has to be called.

code line 86: For debug purposes, it may be helpful to print the task-list on the terminal.

code line 88: The scheduler is started and will run in an endless loop. It will permanently search the task-list and will activate tasks that are ready to run according to their priority.

4.3 COS Demo: Semaphore, FIFO

The thirde example shows a program with 5 tasks, two tasks working as producer and consumer. The code is given for openCM and Arduino.

A data-flow diagram is given in figure 3. Inter-task communication is implemented by a FIFO with several data slots.

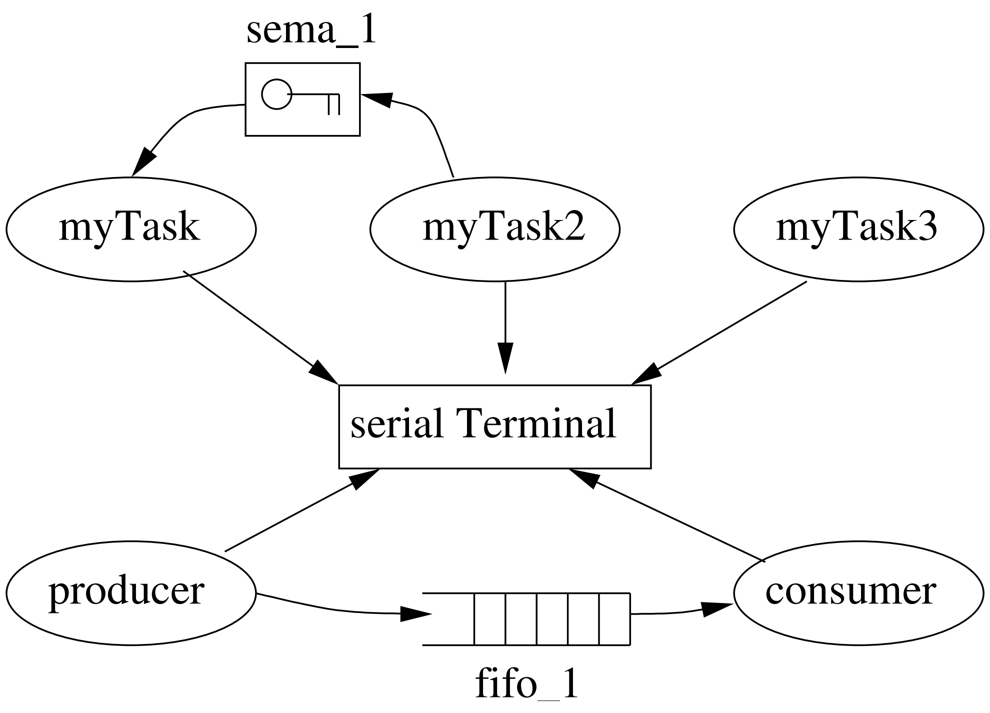

Figure 3: Data-flow diagram of the producer-consumer and semaphore example.

```
#include <CosScheduler.h>

#if COS_PLATFORM == PLATFORM_ARDUINO
    #define PRINT_TO_TERMINAL(x) Serial.print(x)
#endif // COS_PLATFORM
#if COS_PLATFORM == PLATFORM_OPEN_CM_9_04
    #define PRINT_TO_TERMINAL(x) SerialUSB.print(x)
#endif // COS_PLATFORM

```

```



CosTask_t *tA, *tB, *tC;                    /*!< pointer to tasks */
CosTask_t *tProducer, *tConsumer;           /*!< pointer to tasks */
CosFifo_t fifo_1; /*!< global fifo for inter-task communication */
CosSema_t sema_1; /*!< global semaphor for task synchornisation */


/*!
 *****************************************************************
 * @par Description:
 * This task has several scheduling points. It carries out steps
 *   0,..,4. Before the task terminates, it waits (blocking wait)
 *   at a semaphore, synchronizing to task myTask2().
 *
 * @see myTask2()
 * @arg  -
 * @param  pt              - IN/OUT, pointer to task struct
 *****************************************************************/
 void myTask(CosTask_t *pt)
{
    COS_TASK_BEGIN(pt);

    PRINT_TO_TERMINAL("T started\r\n");
    PRINT_TO_TERMINAL("T: step0\n");
    COS_TASK_SCHEDULE(pt);
    PRINT_TO_TERMINAL("T:s1\n");
    COS_TASK_SCHEDULE(pt);
    PRINT_TO_TERMINAL("T:s2\n");
    COS_TASK_SCHEDULE(pt);
    PRINT_TO_TERMINAL("T:s3\n");
    COS_TASK_SCHEDULE(pt);
    PRINT_TO_TERMINAL("T:s4\n");

    COS_SEM_WAIT(&sema_1,pt);
    PRINT_TO_TERMINAL("T:end\r\n");
    COS_TASK_END(pt);
}

/*!
 *****************************************************************
 * @par Description:
 *  This task increments a counter and prints its value on the
 *  termial. It runs in a loop with additional sleep-time.
 *  Before it terminates, is sends a signal to a semaphore, thus
 *  synchronizing to task myTask().
 *
 * @see myTask()
 * @arg  -
 * @param  pt              - IN/OUT, pointer to task struct
 *****************************************************************/
```

```
void myTask2(CosTask_t *pt)
{   static int16_t count =0;
    COS_TASK_BEGIN(pt);

    PRINT_TO_TERMINAL("T2 started\r\n");
    while(count < 10)
    {    PRINT_TO_TERMINAL("T2 cnt=");
         PRINT_TO_TERMINAL(count++);
         PRINT_TO_TERMINAL("\r\n");
         COS_TASK_SLEEP(pt,_milliSecToTicks(500));
    }
    COS_SEM_SIGNAL(&sema_1);
    PRINT_TO_TERMINAL("T2 end\r\n");
    COS_TASK_END(pt);
}

/*!
 ****************************************************************
 * @par Description:
 *  This task increments a counter and prints its value on the
 *  termial. It runs in a loop with additional sleep-time.
 *
 * @see myTask()
 * @arg  -
 * @param  pt              - IN/OUT, pointer to task struct
 ****************************************************************/
void myTask3(CosTask_t *pt)
{   static int16_t count =0;
    COS_TASK_BEGIN(pt);

    PRINT_TO_TERMINAL("T3 started\r\n");
    while(count < 10)
    {    PRINT_TO_TERMINAL("T3 cnt=");
         PRINT_TO_TERMINAL(count++);
         PRINT_TO_TERMINAL("\r\n");
         COS_TASK_SLEEP(pt,_milliSecToTicks(100));
    }
    PRINT_TO_TERMINAL("T3 ends\r\n");
    COS_TASK_END(pt);
}

/*!
 ****************************************************************
 * @par Description:
 *  The task runs in a loop and sends integer values via FIFO to
 *  the consumer task. The producer synchronizes to the consumer.
 *
 * @see consumerTask()
 * @arg  -
 * @param  pt              - IN/OUT, pointer to task struct
 ****************************************************************/
void producerTask(CosTask_t *pt)
```

```
{   static int16_t x=0;
    COS_TASK_BEGIN(pt);

    PRINT_TO_TERMINAL("\r\nProd started\r\n");
    for (x=0; x<10; x++)
    {    PRINT_TO_TERMINAL("prod sends:");
         PRINT_TO_TERMINAL(x);
         PRINT_TO_TERMINAL("\r\n");
         COS_FifoBlockingWriteSingleSlot(pt, &fifo_1, &x);
         COS_TASK_SLEEP(pt,_milliSecToTicks(200));
    }
    PRINT_TO_TERMINAL("Prod ends\n");
    COS_TASK_END(pt);
}

/*!
 ****************************************************************
 * @par Description:
 * The consumer task runs in an endless loop an receives integer
 * values via FIFO from the producer task. The consumer
 * synchronizes to the producer.
 *
 * @see consumerTask()
 * @arg  -
 * @param  pt              - IN/OUT, pointer to task struct
 ****************************************************************/
 void consumerTask(CosTask_t *pt)
{   static int16_t x=0;
    COS_TASK_BEGIN(pt);

    PRINT_TO_TERMINAL("\r\nCons started\r\n");
    while(1)
    {    COS_FifoBlockingReadSingleSlot(pt, &fifo_1, &x);
         PRINT_TO_TERMINAL("cons x=");
         PRINT_TO_TERMINAL(x);
         PRINT_TO_TERMINAL("\r\n");
         COS_TASK_SLEEP(pt,_milliSecToTicks(1000));
    }
    PRINT_TO_TERMINAL("Cons ends\r\n");
    COS_TASK_END(pt);
}


/*---------------------------------------------------*/
void setup()
{
    #if COS_PLATFORM == PLATFORM_ARDUINO
        Serial.begin(9600);
    #endif // COS_PLATFORM

    if(0!=CosInitTaskList())
    {   PRINT_TO_TERMINAL("error in COS_InitTaskList!\r\n");
166      }
167      COS_SemCreate(&sema_1, 0);
168      COS_FifoCreate(&fifo_1, sizeof(int16_t), 5);
169
170      tA = CosCreateTask(1, NULL, myTask);
171      tB = CosCreateTask(2, NULL, myTask2);
172      tC = CosCreateTask(3, NULL, myTask3);
173      tProducer = CosCreateTask(4, NULL, producerTask);
174      tConsumer = CosCreateTask(5, NULL, consumerTask);
175  }
176  /*----------------------------------------------------*/
177  void loop()
178  {    delay(2000);
179      CosVersionInfo();
180      CosPrintTaskList();
181      CosRunScheduler();
182  }
```

Some additional explanations to the program code:

code line 3: The macro `COS_PLATFORM` is defined in file `utility/cos_configure.h`. It is used to specify the hardware platform, COS is running on.

code line 7: A new macro handles terminal output for different platforms.

code line 16: A semaphore may be used to synchronize tasks. In COS, only counting semaphores are available.

code line 36: This co-operative scheduling macro invokes the scheduler without blocking the task. If no other task with higher priority is ready to run, the task will resume immediately.

code line 45: This macro performs a blocking wait on a semaphore. Since the semaphore is initialized to be empty (line 167), the task will block until task `myTask2` will send its signal (line 73).

code line 47: This macro terminates the task, deletes it from the task-list and frees some memory.

code line 73: This macro un-blocks task `myTask`, waiting on the semaphore (line 45).

code line 88: This task runs without any interconnection to the other tasks.

code line 122: The producer sends integer values via FIFO to the consumer. The producer will keep writing values until the FIFO is full and then will synchronize to the read operations of the consumer.

code line 146: The consumer reads integer values from the FIFO. Since the consumer has a longer sleep-time than the producer, the FIFO will eventually be full and the consumer will slow down the producer to it data consumption rate.

code line 164: Before starting the scheduler, `setup`() has to initialize the task-list.

code line 167: The semaphore is initialized to be empty, so any 'wait' will block the calling task.

code line 168: The FIFO is initialized to store a maximum of 5 16-bit signed integers.

References

[1] Adam Dunkels et al. (2006), ”Protothreads: Simplifying Event-Driven Programming of Memory-Constraint Embedded Systems”,
`http://dl.acm.org/citation.cfm?doid=1182807.1182811`

[2] Michael Dorin (2008), ”Building ’instant-up’ real-time operating systems”,
`http://www.embedded.com`

[3] Ernst Forgber (2014), ”Multitasking mit AVR RISC-Controllern”, Franzis Verlag, ISBN 9783645652704.