* ACOBS
ACtiveOBject System

Multi-threaded FW framework (OS) for embedded ARM systems

Torsten Jaekel, June 2014

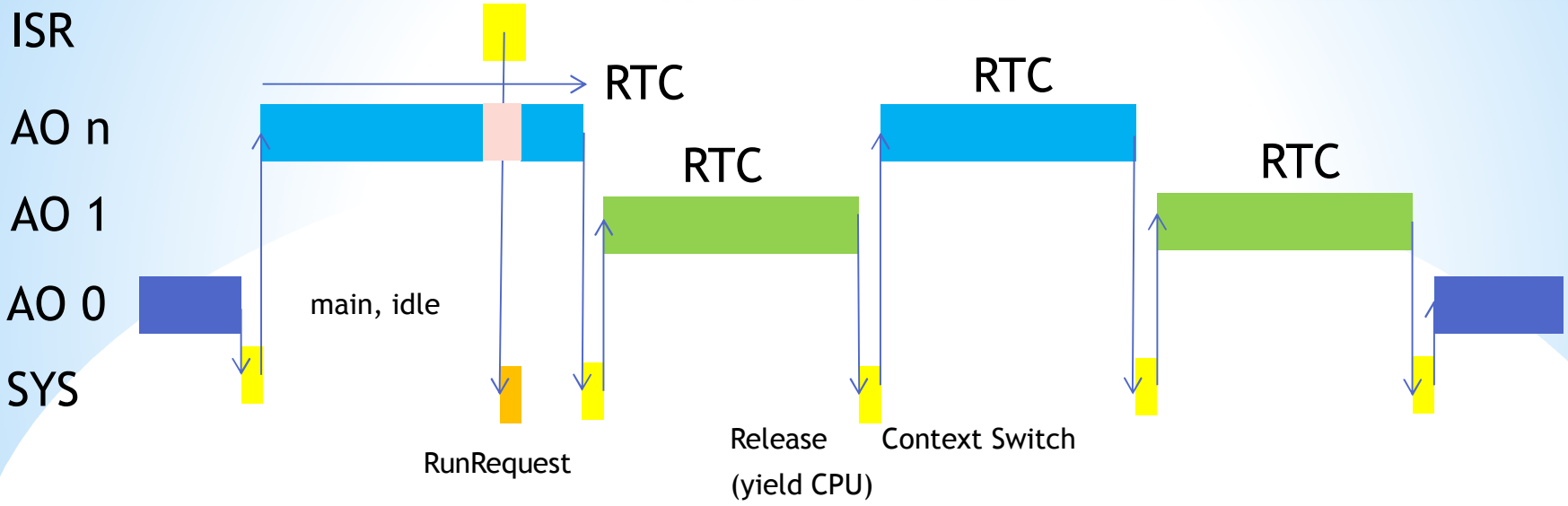
*ACOBS - Overview

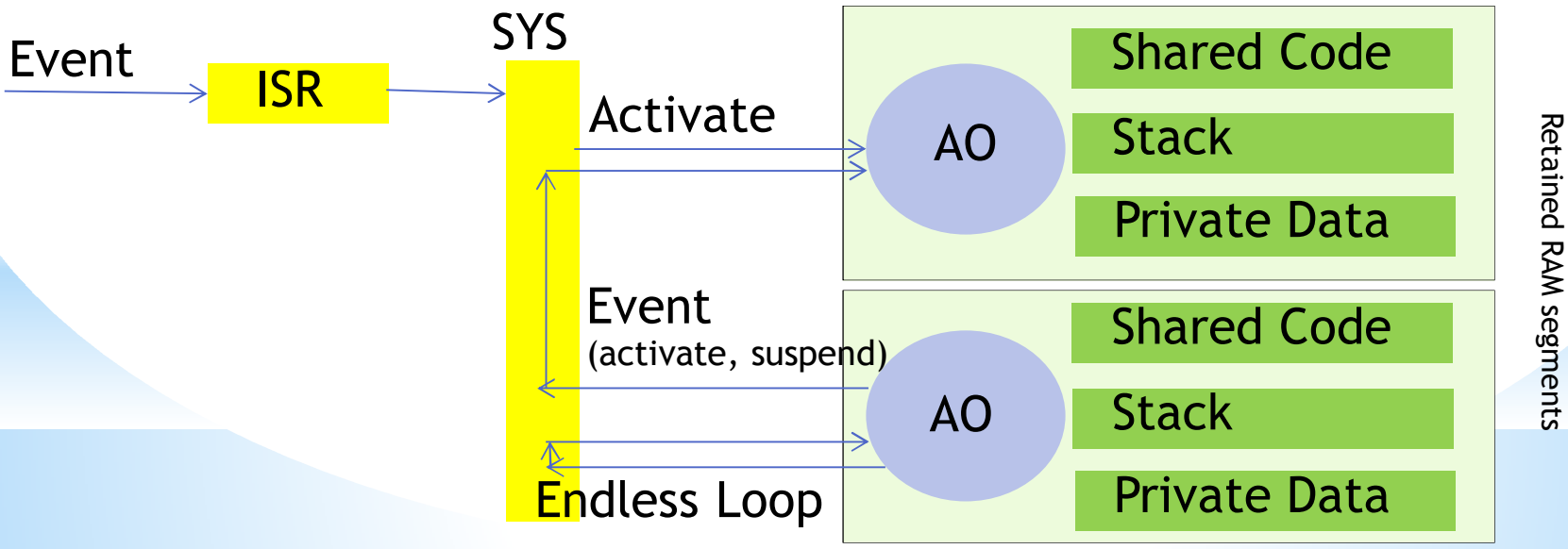
ACOBS

ACtive OBject (operating) System

- **Simplified FW System for Multi-Threading on ARM embedded systems**
  - ACOBS is not a full blown Operating System
  - It is:
    - A framework to support Event Driven Programming and Multi-Threading
    - It is Non-Preemptive (cooperative)
    - An Active Object (AO) behaves like a concurrent task
    - It is based on *Run-To-Completion* (RTC) approach
    - Idea is inspired by "*QP*": http://www.state-machine.com/qp/
  - Very small foot print (approx. 4KB for core functions)
  - Intended for Cortex-M processors
  - No need for *Critical Sections*, *Semaphores*, *Task Message Queues*, *Inter-Task Synchronization* etc.:
    a very simple multi-task (AO) Scheduler and Context Switcher
    (simple assembly code, all other in C)
  - Object Oriented Design and Programming in C

*ACOBS - Overview





* Active Objects

*Active Object

Active Object (AO)

- When active – they run through an "*atomic code sequence*", as Run-to-Completion (RTC)
- They yield the CPU via dedicated System Calls which enters the AO scheduler (switch the context to the next active or requested AO in list)
- AOs have to be "fair" (*cooperative*): they have to yield the CPU at a certain time (RTC sequence done) - large code sequences have to be split into "*atomic code sequences*"
- Can be suspended by itself or by another AO
- Can be activated via a "run request", e.g. triggered by an INT seen or by another AO ("event to handover/activate another AO")
- AOs can exit and can be re-launched (start over, re-initialize an AO, e.g. for SW error recovery, fatal FW issues)
- ISRs are very short, mainly just acknowledge INT and set flag for AO "RunRequest" (non-preemptive)
- main() is the "Idle AO" (always active, running as background/housekeeping task, needed to keep scheduler alive)
  main() can enter WFI if not any AO is in "RunRequest" or "Active" state
- A background cyclic timer interrupt (SysTick) could activate the "idle AO" or the next Event INT will kick off the scheduler again

*Active Object

AO Scheduler

- *Save and Restore Context* and activate next AO (Round Robin list, currently no priorities - can be extended, e.g. Event AO with higher prio and running first, IPC AO with lower prio – RTC for event handling with higher prio)
- Task switch just on dedicated *System Calls* (non-preemptive): "*Deterministic Real-Time Worst Case Latency*"
- No need for event queuing: every event can be associated with an AO (also sharing the code with private data), the AOs are queued implicitly
- The AO context switch is very fast, the scheduler is very simple (Round Robin, just iterate to next active or requested AO in list, directly indexed, not using concatenated lists, simplified dynamic memory management: "*MemPools*")

Power Optimized AOs

- All entities of an AO can be placed on segmented, retained memory regions (AO stack, local variables, "private" data and code hosted on inactive memory segments, when AO is inactive - minimize powered RAM segments)

OOP

- *Single inheritance* in C – derive specific AOs from a hardware independent base class
- *Simple polymorphism* – "virtual" functions, call AO functions via generic base class

* ACOBS API

*AO Instantiation

Define AOs during compile time

- No dynamic AO creation and instantiation, defined static with active or suspended state during compile time
- Define the stack location and size (better trim of SRAM size, decoupled, re-start AOs on fatal issues, "long jump" inside AO code possible)
- Inherit from AO Base Class (scheduler acts on generic base class)

Example

- Derive a new "class" for the AO, inherit the AO Base Class (typedef, structs)
- Define the data array for the derived AO (initialized member variables)
- Create the stack region (and size) for local variables
- Define (and initialize) the *Scheduler Context Storage* table:
  set start SP, entry address (start PC), stored registers ...
  This array is used to store and restore AO context with current and cold start settings
- Define the scheduler *AO Management* table:
  assign a unique AO ID (implicit via array index)
  define the cold start state (Active, RunRequest) and the reference to *Scheduler Context Storage* table entry
- Code the AO function body:
  cold start sequence = "Constructor"
  endless task loop – with a system call to yield CPU after RTC
  yield the CPU when an "atomic sequence" has been done

*AO Instantiation

```
#include "active_object.h“
#include "fifo_ao.h“                                                    //AO definition with inherited AO base class

static void FIFO_AO(void);                                              //prototype
static unsigned long FIFO_AO_Stacks[(EFIFOChannel)MCP_IPC_NUM_CMD_FIFOS][FIFO_AO_STACK_SIZE + SP_LR_OFF];

static TFIFOAOData FifoAOData[(EFIFOChannel)MCP_IPC_NUM_CMD_FIFOS] = {
   {
      /* initialize the base */
      {MCP_IPC_CONV_FIFO2IND(AP_SEC2MCP) + 1},                          //base class member
      /* initialize the FIFO AO members */
      AP_SEC2MCP,                                                       //new sub-class members
      MCP2AP_SEC
   },
   //…
};

static TAOContext FIFO_AO_Context[(EFIFOChannel)MCP_IPC_NUM_CMD_FIFOS + 1] = {
   {
      /* main_AO_Context  - main() is always the idle AO */
      /* AOContext */
      0,              /* PC of main will be stored here */
      0,
      0,
      0,              /* main will store SP here, SP is already set */
      0,0,0,0,
      0,              /* not used on main object */
      0,              /* not used on main object */
      0               /* data Ptr, not used here */
   },
   {
      /* AOContext */
      /* cold start parameters – as constant copy, used to restart */
      FIFO_AO,0,0,&FIFO_AO_Stacks[MCP_IPC_CONV_FIFO2IND(AP_SEC2MCP)][FIFO_AO_STACK_SIZE],
      0,0,0,0,
      /* the current AO state, stored context */
      FIFO_AO,                                                                  /* Entry */
      &FIFO_AO_Stacks[MCP_IPC_CONV_FIFO2IND(AP_SEC2MCP)][FIFO_AO_STACK_SIZE],   /* Stack Pointer */
      &FifoAOData[MCP_IPC_CONV_FIFO2IND(AP_SEC2MCP)]                            /* Data Pointer */
   },
   //…
};
```

*AO Instantiation

```
/* global, used by AO scheduler */

TAOMngt /* EXPORT */ AO_Mngt[(EFIFOChannel)MCP_IPC_NUM_CMD_FIFOS + NUM_EVENT_AO + 1 + 1] = {
   /* main AO - we have to have always and it has to be active */
   {
      /* AO_ID: 0 - main – main has to be always AO ID 0 */
      /* state */
      Active,
      /* request */
      NoRunRequest,
      /* AOContext */
      &FIFO_AO_Context[0]                    /* main (idle AO) has ID 0 */
   },
   /* now the actual AOs – the index on array is the AO ID */
   {
      /* AO_ID: 1 - AP_SEC2MCP */
      /* state */
      Active,
      /* request */
     NoRunRequest,
     /* AOContext */
     &FIFO_AO_Context[MCP_IPC_CONV_FIFO2IND(AP_SEC2MCP) + 1]
    },
    //….
};
```

Remarks

- The AO management table is global and used by name in scheduler
- The order (index) in list is used as unique AO ID (main has to be the first, ID 0)
- An AO can be active at startup time (their AO “constructor” will be called) or inactive (launched via request, “constructor” called when requested)

AO Task Code

```
void FIFO_AO(void)
{
    /*
     * Local Variables – bear stack size in mind
     */
    TFIFOAOData *dataPtr;                                   //reference to “private” data
    TpIPCMsg pIPCmsg, pRSPmsg;
    EMCP_API_MsgErr ipcMsgErr;

    /*
     * Constructor - initialize the AO - register the AO on Context and set dataPtr
     */
    dataPtr = (TFIFOAOData *)AO_DEF_START;  //cold start – init context storage and data reference pointer

   /*
    * Endless Task Loop of the thread
    */
   while (1)
   {
       ipcMsgErr = mcpIPC_receiveCmd(dataPtr->cmdCh, &pIPCmsg);
       if (ipcMsgErr != NO_ERR)
       {
          /* incomplete command - wait and yield CPU */
          AO_suspend_self();                                //system call, yield the CPU

          /* it needs a new interrupt to be triggered and to continue here ! */
         continue;
       }
       else
       {
          /*
           * now do as RTC
           */
          //....
       }
   } //end while Task Loop
}
```

Remarks

- Code as reentrant (thread-safe) shared code
- Use “private” data members for different states
- Use AO ID or indexes to differentiate what to do

*ISR Code - Activate AO

```
void EXT00_IRQ_Handler(void)
{
    /* clear the interrupt */
    mcp_fifoIntClear(AP_SEC2MCP);

    /* activate the associated AO */
    FIFO_setIntRcvd(AP_SEC2MCP);                          //do some AO specific stuff and call system function to request AO
}

void FIFO_setIntRcvd(EFIFOChannel ch)
{
    /* increment the counter on the lower layer FIFO, here for SM FIFO, on APB FIFO it is empty call */
    sFIFO[ch].setIntRcvd(ch);                             //"virtual", polymorph member function call

     /* call system function to request to activate AO */
    AO_runRequest((int)MCP_IPC_CONV_FIFO2IND(ch) + 1);    //system call - request to activate AO
}
```

Remarks

- The ISR will just acknowledge INT reason (clear it) and do System Call to set flag for scheduler – *keep it as short as possible*
- On next System Call with CPU yielded – the scheduler will find requested AO and let it run
- The ISR itself does not change the context – *no race condition, no need for semaphores and critical sections*
- INT latency (real time constraint) is determined by AO code design (RTC sequences) – *deterministic behavior*

*ACOBS - Summary

- ACOBS uses standard C
- Very few lines of assembly code for context save and restore
- Multi-threading based on Active Objects and “Run-to-Completion”
- Ported and tested on ARM Cortex-M0/M0+, M3/M4
- Supporting some OOP features such as single inheritance
- ACOBS is intended for deterministic real-time firmware