A

a small M core RTOS

1. Introduction

.A is a small M core preemptive rtos .

2. Processors , board supported and architecture description

At now A is for STM32 based processors boards. There are no limitations to use A on different processor manufacturer, although no tests have been done on different processors brands.

Currently A runs on L, F, H and U series processors.

There are the IOC files for each processor board, note that excluding for F family all are Nucleo boards.

2.1 Scheduler

The scheduler is called at each sys tick interrupt, actually running at 1 KHz, so 1 milliSecond period. Each time the tick is fired the scheduler saves the currently running process context (e.g registers, stack pointer and so on) and changes the active process.

Moreover periodically kicks the supervisor process, described later.

Processes calls the scheduler, too, when they execute a suspendable task, like waiting for a timer, an HW peripheral and so on. For example, executing a HAL_Delay(time) function will suspend the current process until time expires. Note that in this case the delay can be higher than the specified one if other processes are active.

2.2 Timers

A gives 8 timers per process, the clock of those timers is the tick at 1KHz.

The timers must be declared in the user process and can be one shot type or periodical type.

2.3 Processes

A has a maximum of 8 processes

2.4 Mailbox

A has 8 mailboxes per process, so one process can send i.e chunk of memory to a different process, can send pointers, data or simply signas to other processes something.

2.5 Memory management

A has a memory management mechanism where user processes can allocate chunk of contiguous memory.

The user process can allocate memory, modify it, send to another process or a driver, dellocate it or whatever the process need to do with this memory area.

This can be useful to avoid the prolipheration of unordered variables.

The number of chunks is defined by #define directive, so changing the number of chunks means recompile A.

2.6 HW Manager

HWmanager allocates and deallocates hw resources assigning them to a process, to avoid that 2 different processes try to use the same resource.

2.7 Supervisor process

Supervisor process runs with pid 0, and is started each time all the other processes has been checked and scheduled for active condition. Suppose you have 4 processes each running for the full 1 mSec time slot, supervisor will run each 5 mSec even if there are processes that need cpu time.

Actually supervisor process checks the status of all the other processes, make a simple garbage collection on memory chunks, periodically polls for external libraries like USB and LwIp, and other similar things.

2.7 User processes and user processes descriptor

To define a user process it must be a known entity for A, and for doing this there is a table that describe it in terms of starting address and stack depth.

The compilation of this table must be done or A will use the default one, that points to weak processes that are in a while(1); infinite cycle.

3. Simple user process example

In the following description we assume the user processes are in their own directory parallel to the A directory

3.1 The simplest process

First of all we define the process table in file *processes_table.c*.

```
#include "main.h"
#include "../../A_os/kernel/A.h"
extern   void process_1(uint32_t process_id);     //This is process1
extern   void process_2(uint32_t process_id);     //This is process2
extern   void process_3(uint32_t process_id);     //This is process3
extern   void process_4(uint32_t process_id);     //This is process4 of the application
USRprcs_t       UserProcesses[USR_PROCESS_NUMBER] =
{
                {
                                .user_process = process_1,
                                .stack_size = 1024,
                },
                {
                                .user_process = process_2,
                                .stack_size = 1024,
                },
                {
                                .user_process = process_3,
                                .stack_size = 1024,
                },
                {
                                .user_process = process_4,
                                .stack_size = 1024,
                }
};
```

Here are defined 4 processes (process_1, process_2, process_3 and process_4) with 1024 bytes of stack.

Then we can define i.e. process 2 like this:

```
#include "main.h"
#include "../../A_os/kernel/A.h"
#include "../../A_os/kernel/A_exported_functions.h"
#include "../../A_os/kernel/system_default.h"

void process_2(uint32_t process_id)
{
uint32_t          wakeup;
        create_timer(TIMER_ID_0,400,TIMERFLAGS_FOREVER | TIMERFLAGS_ENABLED);
        while(1)
        {
                wakeup = wait_event(EVENT_TIMER);
                if (( wakeup & WAKEUP_FROM_TIMER) == WAKEUP_FROM_TIMER)
                {
                        /*
                        Do something
                        */
                }
        }
}
```

The include lines are mandatory, because A.h, A_exported_functions.h and system_default.h are needed to address the specific processor capabilities and the prototypes of the A function.

The process itself is very simple : starts a timer and wait for timer expired to do something interesting.

The create_timer function creates the timer as a forever timer (e.g. periodic) with a timer ID of 0 (TIMER_ID_0 is defined in A.h included file) and to fire each 400 mSec.

Then the process enter its own while loop and waits for the timer fire to do something.

And this is all.

The wait_event function returns why the process has been woken up, so the user process can know why it has been made active. This is useful when more than one source of wake up has been selected in he wait_event call.

3.2 A more complex still simple process

Now we define a modbus based process with some memory management.

The file *processes_table.c* is still valid so you can use the one described above.

The process now is *process_1* and is :

```
#include "main.h"
#include "../../A_os/kernel/A.h"
#include "../../A_os/kernel/A_exported_functions.h"
#include "../../A_os/kernel/system_default.h"
#include <string.h>
#include <stdio.h>

extern  USRprcs_t       UserProcesses[USR_PROCESS_NUMBER];

extern  void A_modbus_init(uint8_t address,uint8_t mode,uint32_t uart);
extern  uint8_t A_modbus_process(uint8_t *buf, uint16_t len);
extern  uint8_t         A_set_in_to_modbus(uint16_t         discrete_in_index,         uint8_t
discrete_in_value);

uint8_t A_write_coil_from_modbus(uint16_t coil_index, uint8_t coil_value)
{
        if ( coil_index == 0 )
        {
                if ( coil_value == 0 )
                {
                        HAL_GPIO_WritePin(LED_1_GPIOPORT,
LED_1_GPIOBIT,GPIO_PIN_RESET);
                }
                else
                {
                        HAL_GPIO_WritePin(LED_1_GPIOPORT,
LED_1_GPIOBIT,GPIO_PIN_SET);
                }
        }
/*
        if ( coil_index == 1 )
        {
```

```
                if ( coil_value == 0 )
                {
                        HAL_GPIO_WritePin(LED_2_GPIOPORT,
LED_2_GPIOBIT,GPIO_PIN_RESET);
                }
                else
                {
                        HAL_GPIO_WritePin(LED_2_GPIOPORT,
LED_2_GPIOBIT,GPIO_PIN_SET);
                }
        }
*/
        if ( coil_index == 2 )
        {
                if ( coil_value == 0 )
                {
                        HAL_GPIO_WritePin(LED_3_GPIOPORT,
LED_3_GPIOBIT,GPIO_PIN_RESET);
                }
                else
                {
                        HAL_GPIO_WritePin(LED_3_GPIOPORT,
LED_3_GPIOBIT,GPIO_PIN_SET);
                }
        }

        return 0;
}

#define UART_INTERCHAR_TIMEOUT 10
#define MODBUS_ADDRESS          1
#define MODBUS_RTU_MODE         0x80
#define MODBUS_BUFFER_LEN       64

#ifdef  STM32H743xx
#define PRC1_UART               HW_UART3
#endif
```

```
void process_1(uint32_t process_id)
{
uint32_t        wakeup;
uint8_t *modbus_rx_buf;
        allocate_hw(PRC1_UART);
        modbus_rx_buf = mem_get(MODBUS_BUFFER_LEN);

        hw_receive_uart(PRC1_UART,modbus_rx_buf,MODBUS_BUFFER_LEN,UART_INT
ERCHAR_TIMEOUT);
        A_modbus_init(MODBUS_ADDRESS,MODBUS_RTU_MODE,PRC1_UART);
        create_timer(TIMER_ID_0,200,TIMERFLAGS_FOREVER | TIMERFLAGS_ENABLED);
        while(1)
        {
                wakeup = wait_event(EVENT_TIMER | EVENT_UART3_IRQ);
                if (( wakeup & WAKEUP_FROM_TIMER) == WAKEUP_FROM_TIMER)
                {
                        A_set_in_to_modbus(0,HAL_GPIO_ReadPin(BUTTON_GPIOPORT,
BUTTON_GPIOBIT));
                        //HAL_GPIO_TogglePin(LED_3_GPIOPORT, LED_3_GPIOBIT);

                }
                if      (( wakeup & WAKEUP_FROM_UART3_IRQ) ==
WAKEUP_FROM_UART3_IRQ)
                {

        A_modbus_process(modbus_rx_buf,hw_get_uart_receive_len(PRC1_UART));
                }
        }
}
```

First of all we allocate a uart port with the allocate_hw() function, so the hw resource is locked and only this process can use this resource.

Then we get a chunk of memory for the modbus rx buffer, here of 64 bytes. Note that the minimum chunk is 256, so actually we allocate a 256 bytes chunk.

Then we define that the rx uart should use the allocated memory chunk with hw_receive_uart() function.

Now we can initialize modbus with a call to A_modbus_init().

Finally we create a timer like in the first example but with a 200 mSec period.

The initialization phase is finished, so we can now loop waiting for events.

Each time we receive a buffer from modbus we can call the A_modbus_process() function to decode the modbus commands that will call our own A_write_coil_from_modbus() function, that will move the appropriate bits (in this example the board leds are powered or shutted down.

4. Function lists and description

4.1 System

uint32_t wait_event(uint32_t events);

Waits for a specific event. The event list is:

```
#define EVENT_DELAY                 (1<<HW_DELAY)
#define EVENT_TIMER                 (1<<HW_TIMER)
#define EVENT_MBX                   (1<<HW_MBX)
#define EVENT_SEMAPHORE             (1<<HW_SEMAPHORE)
#define EVENT_SEMAPHORE_TIMEOUT     (1<<HW_SEMAPHORE_TIMEOUT)
#define EVENT_UART1_IRQ             (1<<HW_UART1)
#define EVENT_UART2_IRQ             (1<<HW_UART2)
#define EVENT_UART3_IRQ             (1<<HW_UART3)
#define EVENT_UART4_IRQ             (1<<HW_UART4)
#define EVENT_UART5_IRQ             (1<<HW_UART5)
#define EVENT_UART6_IRQ             (1<<HW_UART6)
#define EVENT_I2C1_IRQ              (1<<HW_I2C1)
#define EVENT_I2C2_IRQ              (1<<HW_I2C2)
#define EVENT_I2C3_IRQ              (1<<HW_I2C3)
#define EVENT_SPI1_IRQ              (1<<HW_SPI1)
#define EVENT_SPI2_IRQ              (1<<HW_SPI2)
#define EVENT_SPI3_IRQ              (1<<HW_SPI3)
#define EVENT_QSPI_IRQ              (1<<HW_QSPI)
#define EVENT_I2S1_IRQ              (1<<HW_I2S1)
#define EVENT_I2S2_IRQ              (1<<HW_I2S2)
#define EVENT_TIM1_IRQ              (1<<HW_TIM1)
#define EVENT_TIM2_IRQ              (1<<HW_TIM2)
#define EVENT_TIM3_IRQ              (1<<HW_TIM3)
#define EVENT_TIM4_IRQ              (1<<HW_TIM4)
#define EVENT_USB_DEVICE_IRQ        (1<<HW_USB_DEVICE)
#define EVENT_USB_IRQ               (1<<HW_USB_HOST)
```

uint8_t get_current_process(void);

Returns the currently running process.

void A_TimeDebug_High(void);

void A_TimeDebug_Low(void);

Debug functions, if defined they will raise or down a specific pin of the processor.

4.2 Memory

*uint8_t *mem_get(uint32_t size);*

Gets a chunk of memory. The size parameter is the number of bytes needed, the chunks are in 256 bytes multiples.

uint32_t mem_release(uint8_t *data_ptr);

Releases an allocated chunk of memory, If the memory is not owned by the process or the pointer is wrong returns 0xffffffff, returns 0 if success.

4.3 Timers

void task_delay(uint32_t tick_count);

Delays at least the amount of time specified in *tick_count* parameter.

uint32_t create_timer(uint8_t timer_id,uint32_t tick_count,uint8_t flags);

Create a timer owned by the calling process with the specified *timer_id* that will last in *tick_count* mSec and repeats as specified in *flags*.

Flags can be TIMERFLAGS_ONESHOT or TIMERFLAGS_FOREVER, meaning that after a fire if TIMERFLAGS_ONESHOT is selected the timer will stop, will be repeated forever if flags is TIMERFLAGS_FOREVER.

uint32_t start_timer(uint8_t timer_id);

Starts the timer *timer_id*.

uint32_t restart_timer(uint8_t timer_id,uint32_t tick_count,uint8_t flags);

Restarts the timer *timer_id,* useful when timer must be prolonged.

uint32_t stop_timer(uint8_t timer_id);

Stops the timer *timer_id*.

uint32_t destroy_timer(uint8_t timer_id);

Destroy (deallocate) the timer *timer_id*.

uint8_t get_timer_expired(void);

Get the expired timer, when you have more than one timer running,

int32_t A_GetTick(void);

Get current tick number.

4.4 HWmanager

uint32_t allocate_hw(uint32_t peripheral);

Allocate the specific HW resource.

The HW resource list is:

#define HW_DELAY	0
#define HW_TIMER	1
#define HW_MBX	2
#define HW_SEMAPHORE	3
#define HW_SEMAPHORE_TIMEOUT	4
#define HW_UART1	5
#define HW_UART2	6
#define HW_UART3	7
#define HW_UART4	8
#define HW_UART5	9
#define HW_UART6	10
#define HW_I2C1	11
#define HW_I2C2	12
#define HW_I2C3	13
#define HW_SPI1	14
#define HW_SPI2	15
#define HW_SPI3	16
#define HW_QSPI	17
#define HW_I2S1	18
#define HW_I2S2	19
#define HW_TIM1	20
#define HW_TIM2	21
#define HW_TIM3	22
#define HW_TIM4	23
#define HW_TIM5	24
#define HW_TIM6	25
#define HW_USB_DEVICE	26
#define HW_USB_HOST	27

*uint32_t hw_set_usb_rx_buffer(uint8_t *rx_buf);*

Sets the buffer where USB rx packets are written. The pointer can be an area allocated with the mem_get() call or an arbitrary array of uint8_t bytes.

uint32_t hw_send_usb(uint8_t ptr, uint16_t len);*

Send data to USB. The **ptr* pointer can be an area allocated with the mem_get() call or an arbitrary array of uint8_t bytes. The *len* parameter is the number of bytes to transmit.

*uint32_t hw_send_uart(uint32_t uart,uint8_t *ptr,uint16_t len);*

Send data to UART. The *uart* parameter is the uart number to use, the **ptr* pointer can be an area allocated with the mem_get() call or an arbitrary array of uint8_t bytes. The *len* parameter is the number of bytes to transmit.

*uint32_t hw_receive_uart(uint32_t uart,uint8_t *rx_buf,uint16_t rx_buf_max_len,uint8_t timeout);*

Sets the buffer where UART rx packets are written and start reception. The *uart* parameter is the uart number to use, the **rx_buf* pointer can be an area allocated with the mem_get() call or an arbitrary array of uint8_t bytes,the *rx_buf_max_len* parameter is the number of bytes to receive to consider the reception complete, the *timeout* parameter is the timeout to consider the reception complete. The reception is considered complete when *rx_buf_max_len* is reached or *timeout* expires, whicher come first.

void HAL_UART_RxTimeoutCheckCallback(void);

This is a callback to call a user written UART callback function only when timeout occours.

uint16_t hw_get_uart_receive_len(uint32_t uart);

Return the length of the last UART received buffer

4.5 Support functions

*void A_memcpy(uint8_t *dest,uint8_t *source,uint16_t size);*

Fast memcpy implementation

4.6 Semaphores handling

uint32_t get_semaphore(uint8_t semaphore_id,uint8_t semaphore_flag,uint32_t semaphore_timeout);

Get a semaphore with *semaphore_id* setting it with *semaphore_flag* waiting the availability for *semaphore_timeout.*

uint32_t destroy_semaphore(uint8_t semaphore_id);

Destroy the semaphore *semaphore_id.*

uint32_t remove_from_waiting_semaphore(uint8_t semaphore_id);

Remove the current process from the semaphore *semaphore_id.*

5. Additional modules and drivers

5.1 Modbus

Modbus at now is implemented as RTU, so binary form. As described above for process_1 the implementation is very simple.

The functions are:

void A_modbus_init(uint8_t address,uint8_t mode,uint32_t uart);

Initializes modbus setting the modbus address *address* in the *mode* RTU or ASCII (not yet implemented) on uart *uart*.

*uint32_t A_rtu_modbus_process(uint8_t *buf, uint16_t len);*

Analyze the buffer *buf of len len and calls the *A_write_coil_from_modbus(uint16_t coil_index, uint8_t coil_value)* user provided function.

The *A_rtu_modbus_process()* function must be called when a buffer is received.