```
/*
**  Myapp.c - Sample code for STM32 HyperPanel OS ============== 02/08/19 =  **
**                                                                           **
**  The Nucleo STM32L152RE board has to be extended with an Arduino shield   **
**  with 3 LED's (REG, GREEN and YELLOW), 3 buttons and infrared receiver.   **
**  We are using USART 2 that is routed throught the USB debug link as a     **
**  console. One can use minicom as an asynchronous terminal emulator.       **
**                                                                           **
**  This simple code is located into the Application Container, it is run    **
**  by the VMK sub-operation system. On the other hand, the I/O container    **
**  runs the ASY (Asynchronous Lines) driver, the GPIO driver (General       **
**  Purpose IO lines), the LED driver, the KBDITF driver (keyboard interface,**
**  including remote control unit RCU) and a DAC (Digital to Analog          **
**  converter) driver. Those five drivers are run by the VMIO sub-OS.        **
**                                                                           **
**  This sample code illustrates the unrivaled OS architecture whereas user  **
**  tasks are just hidden FSM's. This sample code is made of one very simple **
**  user FSM and one user task. The FSM may be requested to continuously     **
**  sends one same data buffer on serial line ASY1, to the DAC, or both. The **
**  user task manages a user interface (UI) through the ASY0 serial line.    **
**  This UI is event driven and receives events from the ASY0 serial line    **
**  (user command from the minicom keyboard), from the 3 buttons of the      **
**  Arduino shield and from an Infrared RCU. The UI also receives each       **
**  second a time-out event in order to refresh the displayed time. The user **
**  task accepts at any time any incoming data on the ASY1 serial line and   **
**  if the FSM is not currently transmitting data the UI task "echoes" on    **
**  ASY1 the incoming data. If the FSM has been requested to transmit on     **
**  ASY1 then incoming data is not echoed. The user FSM is using non         **
**  blocking writes only while the user task only uses blocking writes for   **
**  ASY0 and non blocking writes for ASY1. The user task is doing non        **
**  blocking reads on both ASY0 and ASY1.                                    **
**                                                                           **
**  The sample code also illustrates that one FSM and one task can very      **
**  easily exchange events, that may be used as requests, responses or       **
**  synchronization. Here the task sends SND_START and SND_STOP request      **
**  events and the FSM responds with R_SND_START and R_SND_STOP events.      **
**                                                                           **
**  The two container's OS and this sample application do not need more RAM  **
**  than the STM32L152RE 80 KB of RAM memory.                                **
**                                                                           **
**  ========================================================================= **
**                                                                           **
**  This application is using the driver/protocol/service API to send        **
**  requests to the drivers. Those requests are in fact event messages that  **
**  are received by the automatons located inside the I/O container. The     **
**  ASY driver is one automaton AUT_ASY that has one logical way for each    **
**  USART. The GPIO driver has one logical way for the GPIO controller plus  **
**  one logical way for each PIN, here we are using three pins, one for      **
**  each button. The LED driver has one logical way for the LED controller   **
**  plus one logical way for each subchannel (each LED is a subchannel). The **
**  The KBDITF driver has one single logical way for all "keyboard" events,  **
**  including RCU. The DAC driver has one logical way per DAC device.        **
**                                                                           **
**  The ASY driver includes all the functions of an advanced asynchronous    **
**  line driver. It handlers multiple controller's, those may be             **
**  heterogenous, supports all kind of line parameters, supports all kinds   **
**  of input and output flow control, is able to buffer data while the       **
**  applicative software is busy, is able to handler DMA transfers and is    **
**  able to handle very precisely line idle times.                           **
**                                                                           **
**  The LED driver includes an automatic blinking function. The VMK code     **
**  has only to provide a "blinking" pattern and the LED driver repeats the  **
**  pattern without disturbing the VMK code. So this simple code does not    **
**  contain anything related to blink pattern management.                    **
**                                                                           **
**  The GPIO driver includes a "de-bounce" function for input pins that are  **
**  connected to "dirty" buttons. Also, this simple code does not have to    **
**  manage "de-bouncing" of buttons. The GPIO driver is able to              **
```

```
**  simultaneously poll a great number of GPIO's and use interrupts for a    **
**  limited set (hardware limit) for wich zero latency is needed. The use    **
**  of polling or IRQ's is dynamically done by the applicative software      **
**  When IRQ is selected, then hardware signal filtering may be enabled.     **
**                                                                           **
**  The KBD driver is an interface to both GPIO's used for push-buttons and  **
**  also to a RCU receiver driver that is able to handle all the known       **
**  infra-red RCU protocols (NEC 2000, RC5, RC6, XMP, ...).                  **
**                                                                           **
**  The DAC driver is able to manage several DAC's concurrently, it supports **
**  a wide range of sampling frequencies, is able to LUT (Look Up Tables)    **
**  for values transcoding. This may be use for example to adjust signal     **
**  level.                                                                   **
**                                                                           **
**  ========================================================================= **
**                                                                           **
**  The "space_appcont_xyz.c" file contains the definition of a table named  **
**  "task_grp" that has two entries of type "Task_grp". The first entry      **
**  contains the name "fsm" and the address of the procedure "init_app_fsm". **
**  The second entry contains the name "tsk" and the address of procedure    **
**  "init_app_tsk". At the very end of its initialization, the VMK sub-OS    **
**  starts the tasks groups, it calls the group start procedure's, so in     **
**  our case the "init_app_fsm" procedure is called and then the             **
**  "init_app_tsk" is called. Those call are done using the VMK stack, its   **
**  address and size are definied uin the linker's command file :            **
**  RAM_VMK_STACK        : ORIGIN = 0x2000C800 , LENGTH = 2K                 **
**                                                                           **
**  The first task group "fsm" contains one and only one "system" task (a    **
**  hardware context) which is associated to a "task_id" is 0. Do not be     **
**  confused, it is NOT a user task. In fact it corresponds to a hardware    **
**  context that is mainly a dedicated system stack. This context is         **
**  systematically created at VMK initialisation. The stack address and size **
**  are defined in the command link file:                                    **
**  RAM_VMOS_STACK       : ORIGIN = 0x2000D000 , LENGTH = 2K                 **
**  The C treatments of all the APP FSM's are executed in context 0, so      **
**  they use the task 0 stack. With this simple demo program, the            **
**  "init_vmk_fsm" procedure only delares one FSM using the "iniaut"         **
**  function and then sends one initialization event to this FSM.            **
**                                                                           **
**  The second task group "tsk" contains all the user tasks, the task id's   **
**  are greater or equal to 1. Each has its own stack, the address and size  **
**  of it are given to the "create_task" procedure. With this demo code,     **
**  the "init_app_tsk" procedure only creates one user task and its stack is **
**  simply a C global variable "mystack[]". Also, the task entry point has   **
**  to be given to "create_task", in our case it is the "loop_app_tsk" C     **
**  procedure                                                                **
**                                                                           **
**  ========================================================================= **
**                                                                           **
**  The shield has 3 leds RED, GREEN and YELLOW. It has 3 switch buttons,    **
**  one 56 KHz Infrared Receiver. It has also 3 connectors 6 pins, 10 pins   **
**  and 10 pins. The electrical signals that are used by "myapp.c" are       **
**  depicted below                                                           **
**                                                                           **
**                          ASY1_RX  ASY1_TX    GND                          **
**                             X        X        X                          **
**                             X        X        X                          **
**                          ASY1_RX  ASY1_TX    GND                          **
**                                                                           **
**                                              GND                          **
**           X        X        X        X        X                          **
**           X        X        X        X        X                          **
**                                              GND                          **
**                                                                           **
**                                   DAC_OUT    GND                          **
**           X        X        X        X        X                          **
**           X        X        X        X        X                          **
**                                   DAC_OUT    GND                          **
```

```
**                                                                      **
**  All the drivers have a "module" name that is given to "drv_init_driver"  **
**  that returns and identifier we call a "MODD". All the devices have      **
**  a name that is given to "drv_alloc_device" that returns and identifier  **
**  we call a "device IOD". With devices that have "sub-channels" each      **
**  sub-channel have one or more name that is given to "drv_alloc_subchan"  **
**  that returns and identifier we call a "sub-channel IOD". The present    **
**  application is using the following names and identifiers :              **
**                                                                      **
**  +-----------------+---------------------+-----------+-----------------+  **
**  | Name       MODD | Name        dev IOD | schan IOD | Physical        |  **
**  +-----------------+---------------------+-----------+-----------------+  **
**  | ASY      myio.c | ASY\DEV0   asy_iod0 |           | USB cable (*)   |  **
**  |                 | ASY\DEV1   asy_iod1 |           | ASY1_RX ASY1_TX |  **
**  |                 | ASY\DEV2   asy_iod2 |           | ASY1_RX ASY1_TX |  **
**  +-----------------+---------------------+-----------+-----------------+  **
**  | GPIO     myio.c | GPIO\DEV0  gpio_iod | gpio_iod1 | Switch 1        |  **
**  |                 |                     | gpio_iod2 | Switch 2        |  **
**  |                 |                     | gpio_iod3 | Switch 3        |  **
**  +-----------------+---------------------+-----------+-----------------+  **
**  | LED      myio.c | LED\DEV0    led_iod | led_iod0  | RED Led         |  **
**  |                 |                     | led_iod1  | GREEN Led       |  **
**  |                 |                     | led_iod2  | YELLOW Led      |  **
**  +-----------------+---------------------+-----------+-----------------+  **
**  | KBDITF   myio.c | KDBIDF\DEV0 kbd_iod |           | Infrared Rcv    |  **
**  +-----------------+---------------------+-----------+-----------------+  **
**  | DAC      myio.c | DAC\DEV0    dac_iod |           | DAC_OUT         |  **
**  +-----------------+---------------------+-----------+-----------------+  **
**                                                                      **
**  (*) The RX and TX signal of ASY0 are connected to an external chip      **
**      that is a serial <--> Usb converter. Therefore data that is send    **
**      using ASY0 goes throught the USB cable                          **
**                                                                      **
**  The returned identifers are all global variables. With devices that do  **
**  not have sub-channels (ASY, KDBITF and DAC) the software must use the   **
**  device IOD in order to use the device (asy_iod0, asy_iod1, asy_iod2,    **
**  kbd_iod and dac_iod). With devices that have sub-channels, then the sub- **
**  channel IOD is used in order to interact with the sub-channel           **
**  (gpio_iod1, gpio_iod2, gpio_iod3 for switching buttons and led_iod0,    **
**  led_iod1, led_iod2 for Leds).                                           **
**                                                                      **
**  ======================================================================  **
**  commenter a partir de open_asy
*/


/* Includes files and external references ..................................*/

#include <hypos.h>                      // HyperpanelOS interfaces

#include <drv_rcu.h>                    // RCU procedures prototypes
#include <drv_i2c.h>                    // i2c_send, i2c_receive

#include "./myio.h"                     // Prototypes for "myio_xxx" procs
#include "./mytsk.h"                    // Prototypes for "loop_app_tsk"
#include "./myapp.h"                    // Constant and structs for myapp.c

#include <keym.txt>                     // Applicative key codes


/* Internal global variables for the AUT_USND automaton ----------------------*/

unsigned short      state_usnd[VLNB] ; // State variables for AUT_USND
unsigned char       sp_usnd [VLNB]   ; // Stack pointers for AUT_USND
S_evt               evt_usnd         ; // Current event for AUT_USND
Usnd_ctx            usnd_ctx[VLNB]   ; // Context tables for AUT_USND
```

```
/* Internal global variables for the user task ------------------------------*/

int                 idto             ; // Timer identifier
int                 flag_rest        ; // Flag for restart
int                 errcode          ; // Treatment error code

int                 asy_iod0         ; // IOD of devive ASY\DEV0
int                 asy_iod1         ; // IOD of device ASY\DEV1
int                 asy_iod2         ; // IOD of device ASY\DEV2

int                 gpio_iod         ; // IOD of device GPIO\DEV0
int                 gpio_iod1        ; // IOD of subchannel 0 of GPIO\DEV0
int                 gpio_iod2        ; // IOD of subchannel 1 of GPIO\DEV0
int                 gpio_iod3        ; // IOD of subchannel 2 of GPIO\DEV0

int                 led_iod          ; // IOD of device LED\DEV0
int                 led_iod0         ; // IOD of subchannel 0 of LED\DEV0
int                 led_iod1         ; // IOD of subchannel 1 of LED\DEV0
int                 led_iod2         ; // IOD of subchannel 2 of LED\DEV0

int                 kbd_iod          ; // IOD of device KBDITF\DEV0
int                 rcusnd_id        ; // ID for RCU SND
int                 rcurcv_id        ; // ID for RCU RCV

int                 dac_iod          ; // IOD of device DAC\DEV0

unsigned char       *buf_aud         ; // Address of DEV0 audio buffer

unsigned int        page             ; // Menu current page, UI state variable
char                bufchr[LGCHR+1]  ; // Menu line input (integer number)
int                 lgchr            ; // Count of characters in bufchr

int                 ev_val           ; // Returned by "wait_myevent"
int                 opt_val          ; // Returned by "ev_opt"
int                 condcode         ; // Condition for FSM engine
int                 condret          ; // Condition of return

unsigned int        cfg[VMAX]        ; // Current serial+gpio configuration
int                 iods[VMAX]       ; // IOD to be used with each parameter

unsigned char      *buf_snd[ASYMAX]  ; // Address of transmit buffer
unsigned char      *buf_rcv[ASYMAX]  ; // Address of received buffer
int                 tok_rcv[ASYMAX]  ; // Count of receive tokens
int                 cnt_snd[ASYMAX]  ; // Count of ongoing sent messages

unsigned char      *buf_rcu          ; // Address of RCU send buffer

unsigned char       i2c_tx_frame[256]; // Buffer containing the IC2 frame to
                                       // transmit
unsigned char       i2c_tx[256]      ; // Buffer containing bytes to tranmit
unsigned char       i2c_rx[256]      ; // Buffer containing the received bytes
I2c_io              i2c_iolist[2]    ; // Buffer containing struct describing
                                       // frames to be transmitted
Lcd_text            i2c_text[2] =      // Text to display on the screen LCD
                      {                //
                        {              // Color coding: bbbb bggg gggr rrrr
                          0x001F     , // Foreground color: red
                          0xFFFF     , // Background color: white
                          1          , // Horizontal coordinate
                          1          , // Vertical coordinate
                          "Hello"      // Text
                        }            , //
                        {              // Color coding: bbbb bggg gggr rrrr
                          0xF800     , // Foreground color: blue
                          0x07FF     , // Background color: yellow
                          20         , // Horizontal coordinate
                          15         , // Vertical coordinate
                          "world"      // Text
```

```
                        }               //
                      }               ; //

/* Static data for the AUT_USND FSM ----------------------------------------*/

                                        // Events codes ----------------------
#define SND_START    101                // Start request
#define SND_STOP     102                // Stop  request
#define R_SND_START  103                // Response to Start request
#define R_SND_STOP   104                // Response to Stop  request

                                        // State numbers ---------------------
#define STOPPED        0                // Sending is not started
                                        // VLASY1/VLASY2/VLDAC states --------
#define STARTED        1                // Sending is running
#define STOPPING       2                // SND_STOP has been received
                                        // VLSNDRCU --------------------------
#define SNDWAIT        3                // Waiting for end of sending
#define TOWAIT         4                // Sleeping, waiting for a TO_RCU
#define LASTWAIT       5                // Waiting for the last end of sending

#define STATENB        6                // Transition table size


static S_trans const trans_usnd[] =
  {

/*****************************************************************************
 * Transition table for the AUT_USND VMK FSM                                 *
 * -----------------------------------------                                 *
 *                                                                           *
 * This VMK FSM receives one SND_START event that holds one data buffer      *
 * address and one IOD device. This buffer is send again and again. When the *
 * SND_STOP event is received then no more transmission is asked to the      *
 * device driver. Each locical way has one dedicated context that is a       *
 * "Usnd_ctx" structure. We have a context table, the "usnd_ctx[VLNB]" array.*
 *                                                                           *
 * The automaton treatements are the same ones for ASY1/ASY2 and DAC because *
 * those 3 devices are using the same device API from "myio.c". The RCU      *
 * receiver device uses the dedicated RCU system API and we chose to clearly *
 * separate the FSM treatements rather that making tests or switch in each   *
 * FSM treatement. Therefore the present transition table can be divided in  *
 * 3 blocks:                                                                 *
 *                                                                           *
 * - The first STOPPED state is common to VLASY1/VLASY2/VLDAC/VLSNDRCU       *
 * - STARTED and STOPPING are common to VLASY1/VLASY2/VLDAC                  *
 * - SNDWAIT, TOWAIT and LASTWAIT are dedicated to VLSNDRCU                  *
 *                                                                           *
 * - STOPPED   No transmission is done. We are waiting for a SND_START       *
 *             event request in order to start sending.                      *
 *                           ----------------------------------------------- *
 *             . SND_START   This event is a request to start sending. A     *
 *                           dedicated subroutine "req_snd_start" is used    *
 *                           to send it. This event contains the automaton/  *
 *                           logical way of the sender, its event waiting    *
 *                           queue number, the device IOD to be used for     *
 *                           sending, the address of the data buffer that    *
 *                           will be repeatedly sent and the data bytes      *
 *                           count. The "usnd_start" treatement is called,   *
 *                           it stores those 6 values in the context and     *
 *                           then it issues two calls to the "myio_send".    *
 *                           Those calls are non blocking since the device   *
 *                           has been allocated (drv_alloc_device) using the *
 *                           NBLOCKING_IO option bit. So the device generates*
 *                           immediately (0 latency) two REQ_WRITE event     *
 *                           requests and transmission corresponding to the  *
 *                           first REQ_WRITE immediately starts. The last    *
```

```
 *                           action of 'usnd_start" is to return a           *
 *                           R_SND_START response event to the one that      *
 *                           sent us the SND_START. This response event will *
 *                           re-schedule the "req_snd_start" procedure that  *
 *                           was waiting for it. Then we go to the STARTED   *
 *                           state.                                          *
 *                           ----------------------------------------------- *
 *             . C_RCU       This condition is set if the SND_START event    *
 *                           request is for VLSNDRCU. The consequence of     *
 *                           this condition is to go to state SNDWAIT        *
 *                           ----------------------------------------------- *
 *             . SND_STOP    This event is a request to stop sending. It     *
 *                           should never be received in the present state   *
 *                           since we did not yet received the SND_START.    *
 ******************************************************************************/

/* STOPPED (0) State : Waiting for the SND_START  event .....................*/
#define L_STOPPED  3

SND_START   , usnd_start  , 0     , STARTED , // Request to start transmitting
C_RCU       , urcu_send   , 0     , SNDWAIT , // .. We have a RCU
SND_STOP    , trien       , 0     , STOPPED , // Incorrect request here


/*****************************************************************************
 * States for VLASY1/VLASY2/VLDAC                                            *
 * ------------------------------                                            *
 *                                                                           *
 * - STARTED   The same data buffer is sent again and again. Each time the   *
 *             VMIO driver send us a RESP_WRITE, we issue a new "myio_send", *
 *             that sends one REQ_WRITE to the VMIO driver.                  *
 *                           ----------------------------------------------- *
 *             . SND_START   This event should never be received in the      *
 *                           STARTED state since we already have been        *
 *                           requested to start transmitting.                *
 *                           ----------------------------------------------- *
 *             . SND_STOP    This event is a request to stop sending. A      *
 *                           dedicated subroutine "req_snd_stop" is used to  *
 *                           send _t. This event contains the automaton and  *
 *                           logical way number of the sender and also the   *
 *                           internal queue number that we will have to use  *
 *                           for writing the response event. The "usnd_stop" *
 *                           treatment is called, it stores those 3 values   *
 *                           in the "Usnd_ctx" context structure. Then we go *
 *                           to the STOPPING state, we have two RESP_WRITE   *
 *                           events to be waited for.                        *
 *                           ----------------------------------------------- *
 *             . RESP_WRITE  This event is response event to one REQ_WRITE.  *
 *                           It is sent by the VMIO driver just after the    *
 *                           end of transmission of the last byte of the     *
 *                           message or at beginning of transmission of the  *
 *                           last byte of the message, depending on the      *
 *                           hardware capabilities. Just before sending us   *
 *                           the driver had started sending the next         *
 *                           message. So here we have "plenty" of time to    *
 *                           issue a new "myio_send" (the just started       *
 *                           message transmission duration). The "usnd_resp" *
 *                           treatment is called, it issues a call to the    *
 *                           "myio_send" function. A REQ_WRITE is written    *
 *                           in the REQ_WRITE driver's internal queue for    *
 *                           write requests with a 0 latency. So the driver  *
 *                           is now transmitting the very beginning of the   *
 *                           previous REQ_WRITE and has another REQ_WRITE in *
 *                           its write request queue. The automaton next     *
 *                           state is STARTED (no state change).             *
 *                                                                           *
 * - STOPPING  We received a SND_STOP event and we are waiting for the       *
 *             two remaining RESP_WRITE from the driver. Upon the second     *
```

```
 *             RESP_WRITE receive, we will go to the STOPPED state. When     *
 *             we enter the STOPPING state then the "ctx->nbreq" counter of  *
 *             awaited RESP_WRITE events is always equal to 2.               *
 *                           ------------------------------------------------ *
 *             . SND_START   This event is a request to start sending. It    *
 *                           should never be received in the present state   *
 *                           but only from the STOPPED state.                *
 *                           ------------------------------------------------ *
 *             . SND_STOP    This event is a request to stop sending. It     *
 *                           should never be received in the present state   *
 *                           but only from the STARTED state.                *
 *                           ------------------------------------------------ *
 *             . RESP_WRITE  Completion of one "req_write" that has been     *
 *                           issued from the STARTED state. This event is    *
 *                           sent by the VMIO driver. The "usnd_resp"        *
 *                           treatment is called, we decrement by 1 the      *
 *                           "ctx->nbreq" counter. If the counter value is   *
 *                           now 0 then we set the C_END condition           *
 *                           ------------------------------------------------ *
 *             . C_END       This not an event but a condition, that is set  *
 *                           when all the awaited RESP_WTITE events have     *
 *                           been received. The "usnd_end" treatment is      *
 *                           called: It send a R_SND_STOP response event,    *
 *                           using the "aur/vl/nfa" values that have been    *
 *                           stored in the context when the SND_STOP event   *
 *                           was received. The automaton next state is the   *
 *                           STOPPED state. The R_SND_STOP is received by    *
 *                           the "req_snd_stop" subroutine that was waiting  *
 *                           for it.                                         *
 ***************************************************************************/

/* STARTED (1) State : Waiting for the SND_STOP  event ......................*/
#define L_STARTED (L_STOPPED + 3)

SND_START   , trien       , 0     , STARTED , // Incorrect request here
SND_STOP    , usnd_stop   , 0     , STOPPING, // Request to stop transmitting
RESP_WRITE  , usnd_resp   , 0     , STARTED , // End of one transmission


/* STOPPING (2) State : Waiting for the 2 RESP_WRITE remaining events .......*/
#define L_STOPPING (L_STARTED + 4)

SND_START   , trien       , 0     , STOPPING, // Incorrect request here
SND_STOP    , trien       , 0     , STOPPING, // Incorrect request here
RESP_WRITE  , usnd_resp   , 1     , STOPPING, // End of one transmission
C_END       , usnd_end    , 0     , STOPPED , // . No more awaited RESP_WRITE


/***************************************************************************
 * States for VLSNDRCU                                                     *
 * -------------------                                                     *
 * - SNDWAIT   A RCU message is currently sent and we are waiting for the  *
 *             RESP_WRITE end of transmission event.                       *
 *                           ------------------------------------------------ *
 *             . SND_START   This event should never be received in the    *
 *                           SNDWAIT state since we already have been      *
 *                           requested to start transmitting.              *
 *                           ------------------------------------------------ *
 *             . SND_STOP    This event is a request to stop sending. A    *
 *                           dedicated subroutine "req_snd_stop" is used to  *
 *                           send it. This event contains the automaton and  *
 *                           logical way number of the sender and also the   *
 *                           internal queue number that we will have to use  *
 *                           for writing the response event. The "usnd_stop" *
 *                           treatment is called, it stores those 3 values   *
 *                           in the "Usnd_ctx" context structure. We go to   *
 *                           the LASTWAIT state in order to wait for the     *
 *                           last RESP_WRITE event.                        *
```

```
 *                           ------------------------------------------------ *
 *             . RESP_WRITE  This event is generated by the RCU system        *
 *                           driver and its indicates that the last message   *
 *                           sending is completed. We start a TO_RCU time-    *
 *                           out because we want to wait for 5 seconds        *
 *                           before the next sending. The treatment that      *
 *                           starts the TO_RCU time-out is "urcu_tostart".    *
 *                           Then we go to TOWAIT in order to wait for the    *
 *                           TO_RCU event.                                    *
 *                                                                            *
 * - TOWAIT    A time-out is currently running and we are waiting for the     *
 *             TO_RCU event that will be sent by the VMOS to indicate that    *
 *             the waiting period is now over.
 *                           ------------------------------------------------ *
 *             . SND_START   This event should never be received in the       *
 *                           SNDWAIT state since we already have been         *
 *                           requested to start transmitting.                 *
 *                           ------------------------------------------------ *
 *             . SND_STOP    This event is sent to us by the applicative      *
 *                           task, using the "req_snd_stop" subroutine. We    *
 *                           use the "urcu_stop" treatement that clears the   *
 *                           running TO_RCU and immediately sends the         *
 *                           R_SND_STOP response event to the task.           *
 *                           ------------------------------------------------ *
 *             . TO_RCU      This event is generated by the VMK, it notifies  *
 *                           the end of the time-out. We execute the          *
 *                           "urcu_send" treatment that request to the RCU    *
 *                           driver to start transmitting a new message.      *
 *                                                                            *
 * - LASTWAIT  We received a SND_STOP event and we are waiting for the        *
 *             the RESP_WRITE that corresponds to the end of transmission     *
 *             of a message transmission that was ongoing when the SND_STOP   *
 *             event was received.
 *                           ------------------------------------------------ *
 *             . SND_START   This event should never be received in the       *
 *                           LAST state since we did not yet notified that    *
 *                           we would be halted.                              *
 *                           ------------------------------------------------ *
 *             . SND_STOP    We should not receive this event since we        *
 *                           already have been requested to stop and we did   *
 *                           not answered to it for now.                      *
 *                           ------------------------------------------------ *
 *             . RESP_WRITE  This is the event we are waiting for, sent by    *
 *                           the RCU system driver, that tells us that the    *
 *                           last RCU message transmission has been           *
 *                           completed. We execute the "usnd_end" treatment   *
 *                           that sends the R_SND_STOP response event, using  *
 *                           the references that the "usnd_stop" treatement   *
 *                           had stored in the context. The next state is     *
 *                           the STOPPED state.                               *
 ******************************************************************************/

/* SNDWAIT (3) State : Waiting for the RESP_WRITE  .........................*/
#define L_SNDWAIT (L_STOPPING + 3)

SND_START   , trien       , 0     , SNDWAIT , // Incorrect request here
SND_STOP    , usnd_stop   , 0     , LASTWAIT, // Request to stop transmitting
RESP_WRITE  , urcu_tostart, 0     , TOWAIT  , // End of one transmission

/* TOWAIT (4) State : Waiting for TO_RCU time-out event .....................*/
#define L_TOWAIT (L_SNDWAIT + 3)

SND_START   , trien       , 0     , TOWAIT  , // Incorrect request here
SND_STOP    , urcu_stop   , 0     , STOPPED , // Request to stop transmitting
TO_RCU      , urcu_send   , 0     , SNDWAIT , // Time-out notification

/* LASTWAIT (5) State : Waiting for the last RESP_WRITE  ...................*/
#define L_LASTWAIT (L_TOWAIT + 3)
```

```
SND_START   , trien       , 0     , LASTWAIT, // Incorrect request here
SND_STOP    , trien       , 0     , LASTWAIT, // Request to stop transmitting
RESP_WRITE  , usnd_end    , 0     , STOPPED , // End of one transmission

  }                                 ; // ---------------------------------

static USHORT const lim_usnd[] =      // List of state limit's in trans_usnd
  {                                   // ---------------------------------
    0         , L_STOPPED    ,        // 1st line for STOPPED, STARTED
    L_STARTED , L_STOPPING   ,        // 1st line for STOPPED, end of table
    L_SNDWAIT , L_TOWAIT     ,
    L_LASTWAIT,
  }                                 ; // ---------------------------------


/* Lists of tags for the "open_xyz" procedures ..............................*/

static const char *asy_taglist0 = "BAUDS=115200\nNBBITS=8\n"
                                  "STOPBIT=1\nPARITY=NONE"                  ;

static const char *asy_taglist1 = "BAUDS=9600\nNBBITS=8\n"
                                  "STOPBIT=1\nPARITY=NONE\nBUFSIZE=0\n"
                                  "FLOWCTL=NONE\nTMODE=NONE\nTEMPO=1"       ;

static const char *gpio_taglist = "FUNC=GPIO\nOUTPUT=HI_Z\nWEAKPULL=UP\n"
                                  "EVENTS=REPORT"                           ;

static const char *dac_taglist  = "DEPTH=8\nSFREQ=8000\nNBCH=2\nENDIAN=MSB"  ;


#include "./myapp_scr.txt"

/* Beginning of the code ------------------------------------------------------

------------------  User FSM -----------------------------------------------
init_app_fsm        VMK user Finite State Machine's creation function

usnd_start          SND_START event treatment
usnd_stop           SND_STOP  event treatment for ASY1/ASY2/DAC
usnd_resp           RESP_WRITE event treatment
usnd_end            C_END condition treatment
urcu_send           C_RCU and TO_RCU treatment, send one RCU message
urcu_tostart        RESP_WRITE treatment, starts the TO_RCU time-out
urcu_stop           SND_STOP event treatment for RCU
req_snd_start       Send a SND_START event to the AUT_USND automaton
req_snd_stop        Send a SND_STOP  event to the AUT_USND automaton

------------------  User task main loop and termination ----------------------
loop_app_tsk        Entry point for create_task - Main event loop

------------------  User task devices initialization and termination --------
open_asy            Open of two serial ports
close_asy           Close of two serial ports
open_gpio           Open the GPIO driver
close_gpio          Close the GPIO driver
open_led            Open the LED driver
close_led           Close the LED driver
open_kbd            Open the KBDITF driver
close_kbd           Close the KBDITF driver
open_dac            Open the DAC driver
close_dac           Close the DAC driver

------------------  UI event driven treatments -----------------------------
empt                Do nothing
pmen                Print the menu on ASY\DEV0 and the time
ptim                Print the time in seconds
```

```
pgrp                Print the status strings
pcmd                Print the last selected option/character
pchr                Print echo of one ASCII character
pcap                Prints the RCU symbols captured values
perr                Print the "errcode" value
tcfg                Togles a "cfg[]" value
tinp                Togles VINPUT and VNUMRCU
tssy                Togles VONMIN/VONMAX/VSTMIN/VSTMAX
tchr                Add character to "bufchr
scfg                Sets a "cfg[]" value
sval                Do a drv_detval with a "cfg[]" value
sled                LED blinking start/stop
sint                Converts the ASCII decimal string from "buhchr" to integer
rasy                ASY1/ASY2 continuous sending start/stop
rdac                DAC  continuous sending start/stop
rrcu                RCU transmitter sending start/stop
ewreq               Write I2C EEPROM (offset specified)
eroreq              Read  I2C EEPROM (offset specified)
erreq               Read  I2C EEPROM (next bytes)
ldreq               Display text on screen LCD
lcreq               Clear screen LCD
ltcreq              Get touch coordinates
sapt                Set ASY1/2 pattern
sdpt                Set DAC pattern
srcu                Sets new RCU
scap                Starts a RCU symbols capture
sret                Affects parameter to "condret"
scon                Affects "condret" to "condcode"
rest                Restart

------------------  UI event's parsing ---------------------------------------
ev_opt              Convert EV_... to OPT...
ev_asy0_rcv         EV_ASY0_RCV     has been received - Determine command
ev_asy0_snd         EV_ASY0_SND     has been received - Determine command
ev_asy12_rcv        EV_ASY1/2_RCV   has been received - Determine command
ev_asy12_snd        EV_ASY1/2_SND   has been received - Determine command
ev_gpio012_in       EV_GPIO0/1/2_IN has been received - Determine command
ev_i2c_end          EV_I2C_END      has been received - Determine command
ev_kbd_rcv          EV_KBD_RCV      has been received - Determine command
ev_msec             EV_MSEC         has been received - Determine command

------------------  UI waiting events subroutines ----------------------------
wait_myevents       Wait for all my events
*/



/*  Procedure init_app_fsm ----------------------------------------------------

    Purpose : Creates one VMK Finite State Machine (automaton), then sends
              one initialization event to it. This code is executed within
              VMK hardware context/stack. The C treatments of this automaton
              will be executed with the VMOS hardware context/stack (taskid 0).
*/

int init_app_fsm(Task_grp *g,char *mod, char*conf)
  {
    int             ret             ; // Returned code

/*************************************************************************
 * Step 1 : Declare our FSM (automaton) to the VMK. There is only one call   *
 * ------   to the "iniaut" procedure to be done and the AUT_USND FSM is     *
 *          immediately "on line". In our case we do not use the             *
 *          hierarchical capabilities of the VMK FSM engine and so we        *
 *          provide 1 for "Depth of state stacks". We also do not use in our *
 *          code the "pushcond" procedure (see vmk1.c 2.7.3 pushcond proced) *
 *          and so we set to HNULL the "acond" input parameter               *
 *************************************************************************/
```

```
    iniaut(AUT_USND   ,                 // Automaton number
           trans_usnd ,                 // Transition table address
           lim_usnd   ,                 // First transition num. for each state
           STATENB    ,                 // Transition table size
           VLNB       ,                 // External Logical Ways Number
           VLNB       ,                 // Internal Logical Ways Number
           1          ,                 // Depth of state stacks
           0          ,                 // Depth of the conditions stack
           state_usnd ,                 // State variables storage address
           sp_usnd    ,                 // Stack pointers storage address
           HNULL      ,                 // Conditions stack storage address
           &evt_usnd  ,                 // Where to copy the incoming event
           &ret         )             ; // Procedure returned error code


    return 0                          ; // Exit without any error
  }


/*  Procedure usnd_start ----------------------------------------------------

    Purpose : SND_START event treatment. We store the received event parameters
              in the "Usnd_ctx" context and we call "myio_send" 2 times.
*/
static int usnd_start(int par)
  {
    Usnd_ctx        *ctx              ; // Context address
    int             i                 ; // Loop counter
    int             ret               ; // Returned code
    S_evt           evt               ; // R_SND_START response event

/*****************************************************************************
 * Step 1 : Store the parameters of the SND_START event in the logical way   *
 * ------   context as follows. We are using here two global variables that  *
 *          have been set by the VMK before calling the present procedure.   *
 *          The "voielog" global variable is the value of the current        *
 *          Logical Way. The "evt_usnd" (S_evt structure) contains a copy of *
 *          the just received event (so SND_START here):                     *
 *          - First, we set a pointer "ctx" on the context that is           *
 *            associated to the current LW, to the address of the "voielog"  *
 *            entry of the "usnd_ctx[]" table of context structures.         *
 *          - All the events are containing the reference of the sender of   *
 *            the event. This reference is made of 3 fields "emet" (The      *
 *            automaton number), "evl" (logical way number) and "node" (The  *
 *            node number, only used for a networked spread OS). We keep in  *
 *            the context the values of "emet" and "evl". The SND_START      *
 *            event also contains in the "res2" field the Internal Queue     *
 *            number where a response event should be written. We keep that  *
 *            queue number in "ctx->nfa".                                    *
 *          - The SND_START event contains three parameters, that have been  *
 *            written by "req_snd_start":                                    *
 *            . The "reserve" field contains the IOD of the device that will *
 *              be used to send the data. We store it in "ctx->iod".         *
 *            . The "adresse" field contains the address of the data buffer  *
 *              that will be repeatedly sent. We store it in "ctx->buf".     *
 *            . The "longueur" field contains two packed values. Bits b0..15 *
 *              is the number of data bytes or symbols that are in the       *
 *              buffer, we store that count in "ctx->buflg". With the RCU    *
 *              transmitter, bits b16..31 are the value of the "rpt" param   *
 *              that will be given to "send_rcu", we keep that value in the  *
 *              context "rpt" field.                                         *
 *          - With RCU, we will use a time-out. We have a field context that *
 *            stores the running time-out identifier and the 0xFFFFFFFF      *
 *            empty value while the time-out is not running. We set this     *
 *            "ideto" field to the 0xFFFFFFFF empty value                    *
 *****************************************************************************/
```

```
    ctx        = usnd_ctx              // We set "ctx" to the address of
               + voielog             ; // usnd_ctx[voielog]

    ctx->emet  = evt_usnd.emet       ; // Automaton   number of sender
    ctx->evl   = evt_usnd.evl        ; // Logical Way number of sender
    ctx->nfa   = evt_usnd.res2       ; // Queue number       of sender

    ctx->iod   = evt_usnd.reserve    ; // IOD of device to be used
    ctx->buf   = evt_usnd.adresse    ; // Data buffer address
    ctx->buflg = evt_usnd.longueur     // Number of data bytes (bits with RCU)
               & 0xFFFF              ; // are bits b0..15
    ctx->rpt   = evt_usnd.longueur     // With the RCU transmitter, bits
               >> 16                 ; // b16..31 is the count of repeats
    ctx->idto  = 0xFFFFFFFF          ; // TO_RCU identifier = empty


/*****************************************************************************
 * Step 2 : Special case for VLSNDRCU. We do not execute step 3 that starts  *
 * ------   two send requests. We just set condition C_RCU because RCU is    *
 *          using a dedicated set of states, starting with the SND_WAIT      *
 *          state. The C_RCU condition will execute the "urcu_send"          *
 *          treatment that requests to start a RCU transmit. The RESP_WRITE  *
 *          end of transmission event will be received in the SND_WAIT state.*
 *          After calling "relaut" we have to jump to step 4 because we need *
 *          to respond with a R_SND_START event.                             *
 *****************************************************************************/

    if (voielog EQ VLSNDRCU)           // If this is a RCU transmitter request,
      BEGIN                            // we set C_RCU that will execute the
        relaut(C_RCU)                ; // "urcu_send" treatment and will go
        goto step4                   ; // to the SNDWAIT next step. Here we
      END_IF                           // jump to step4 to send the R_SND_START
                                       // to the SND_START sender.

/*****************************************************************************
 * Step 3 : We issue two write request, two calls to the "myio_send"         *
 * ------   procedure, using the same data buffer "ctx->buf". The reason to  *
 *          do that is that we want to reduce to 0 the latency between two   *
 *          successive writes started by the driver. Please remind that the  *
 *          drivers we are using have been allocated using the NBLOCKING_IO  *
 *          option bit (see calls that have been made to "drv_alloc_device").*
 *          As a result, the two calls to "myio_send" that we do here are    *
 *          non blocking, each of those just sends a REQ_WRITE event to the  *
 *          VMIO driver and we DO NOT WAIT here for the RESP_WRITE :         *
 *          - We set "ctx->nbreq" to 2, this value is the count of           *
 *            RESP_WRITE event we will be waiting for.                       *
 *          - The first call to "myio_send" sends a REQ_WRITE to the VMIO    *
 *            driver. The sole fact to writing this event in the VMIO        *
 *            automaton engine event input queue interrupts with no latency  *
 *            our code and the VMIO is entered. The VMIO stack is installed. *
 *            The VMIO FSM engine calls a C procedure of the driver, that is *
 *            its treatement for the REQ_WRITE event. As the transmission    *
 *            medium is currently idle, the physical "send" of the data      *
 *            block immediately begins. When the driver C procedure returns, *
 *            the VMIO exits, our stack is restored and we finish the        *
 *            execution of the code of the "myio_send" procedure.            *
 *          - So when the first call to "myio_send" returns, transmission of *
 *            the first byte of the message has been started, and probably   *
 *            not more has been done (second byte transmission probably not  *
 *            yet started).                                                  *
 *          - Then we call "myio_send" for the second time. At that instant, *
 *            for sure transmission of the first data block is ongoing and   *
 *            it will not be completed before a "long" time. The "myio_send" *
 *            sends a second REQ_WRITE that is written in the VMIO event     *
 *            input queue. This event writing immediately interrupts the     *
 *            "myio_send" code, the VMIO stack is installed and the driver's *
 *            REQ_WRITE handling treatment is executed for the second time.  *
 *            But now the first transfer is still ongoing, and so the        *
```

```
 *            driver's REQ_WRITE handling treatment will just keep in an     *
 *            internal queue the values of the second REQ_WRITE parameters,  *
 *            including the buffer address and the number of data bytes to   *
 *            be sent.                                                       *
 *          So just after we exit from the second call to "myio_send", the   *
 *          current situtation can be depicted as follows:                   *
 *          - Transmission of the first message is ongoing, and a "long"     *
 *            time will occur before it's completion.                        *
 *          - The second sending request is stored in an internal driver's   *
 *            waiting queue only dedicated to write requests.                *
 *          And here is the reason for doing those two non-blocking write    *
 *          requests: The hardware will raise an "End Of Transmission"       *
 *          Interrupt Request exactly (or even some time before) when the    *
 *          last byte of the first message has been sent. In our case, the   *
 *          interrupt handling processing is NEVER deferred, and this one    *
 *          IMMEDIATELY start the second message sending. Therfore, latency  *
 *          between transmission of the last byte of the first message and   *
 *          the first byte of the second message cannot be shorter since it  *
 *          is 0. This is true with all the HypOS drivers.                   *
 ***************************************************************************/

    ctx->nbreq = 2                  ; // We are going to do 2 writes

    for(i = 0; i LT 2; i++)           // Send 2 write requests to the VMIO
      myio_send (ctx->iod   ,         // IOD of device
                 ctx->buf   ,         // Buffer Address
                 ctx->buflg ,         // Number of bytes to be sent
                 &ret        )       ; // Count (not used here)


/***************************************************************************
 * Step 4 : Send the R_SND_START response event. The SND_START event has been *
 * -------  sent by "req_snd_start" procedure, that had stored in the event   *
 *          the caller's Internal Queue Number (Value is 2 to 15 and has been *
 *          written in the "res2" event field). Also, the "putevt_vmk"        *
 *          procedure also stores in the "emet" and "evl" fields of the       *
 *          event the automaton and LW numbers of the event's sender. So here *
 *          we can return to the sender a R_SND_START response event. This    *
 *          event is awaited inside the "req_snd_start" procedure.            *
 ***************************************************************************/

    step4                           : // Send the R_SND_START event

    Memset(&evt,0,sizeof(evt))      ; // Clear the event structure

    evt.aut      = ctx->emet        ; // Target automaton number
    evt.vl       = ctx->evl         ; // Target logical way number
    evt.code     = R_SND_START      ; // Event code

    putevt_vmk(ctx->nfa   ,           // VMK Internal queue 2 to 15
               &evt        )        ; // Event to be written

    return 0                        ; // Exit without any error
  }


/*  Procedure usnd_stop ----------------------------------------------------

    Purpose : SND_STOP event treatment. We store in the context the references
              "emet" and "evl" of the one that sent us the SND_STOP event and
              also in "nfa" the Internal Queue number that will be used to
              return the R_SND_STOP response event.
*/

static int usnd_stop(int par)
  {
    Usnd_ctx        *ctx            ; // Context address
```

```
/*************************************************************************
 * Step 1 : Store the reference of the one who sends us the SND_STOP event.  *
 * ------   We will need it later (in usnd_end) to return a R_SND_STOP       *
 *          response event to the SND_STOP event sender:                     *
 *          - First, we set a pointer "ctx" on the context that is           *
 *            associated to the current LW. The VMK, before calling the      *
 *            present procedure, has set the "voielog" global variable to    *
 *            the current LW number. We use it as an index in our context    *
 *            table "usnd_ctx[]" and so "ctx" is set to &usnd_ctx[voielog].  *
 *          - The VMK has also copied in the "evt_usnd" global variable      *
 *            (a S_evt structure) the just received event (so a SND_STOP).   *
 *            we extract from it the sender automaton number "emet", the     *
 *            sender logical way number "evl" and the Internal Queue number  *
 *            "res2" where the R_SND_STOP event will have to be written. All *
 *            those 3 values are needed by the "usnd_end" procedure that     *
 *            will be called later.                                          *
 *************************************************************************/

    ctx        = usnd_ctx              // We set "ctx" to the address of
               + voielog             ; // usnd_ctx[voielog]

    ctx->emet  = evt_usnd.emet       ; // Automaton   number of sender
    ctx->evl   = evt_usnd.evl        ; // Logical Way number of sender
    ctx->nfa   = evt_usnd.res2       ; // Queue number       of sender

    return 0                         ; // Exit without any error
  }


/*  Procedure usnd_resp ----------------------------------------------------

    Purpose : RESP_WRITE event treatment. As one transmission has been
              completed, we start a new one, calling "myio_send" one time.
*/
static int usnd_resp(int par)
  {
    Usnd_ctx        *ctx             ; // Context address
    int              ret             ; // Returned code

/*************************************************************************
 * Step 1 : Decrement by 1 the number of awaited RESP_WRITE event, this      *
 * ------   count being the "nbreq" field of the Logical Way context:        *
 *          - The VMK, before calling this procedure, has set the "voielog"  *
 *            global variable the value to the current Logical Way and so    *
 *            we just use it as an index in the "usnd_ctx[]" context table.  *
 *            So we set "ctx" to &usnd_ctx[voielog].                         *
 *          - Then we decrement by 1 the "ctx->nbreq" counter.               *
 *************************************************************************/

    ctx        = usnd_ctx              // We set "ctx" to the address of
               + voielog             ; // usnd_ctx[voielog]

    ctx->nbreq --                    ; // Decrement the number of still
                                       // awaited RESP_WRITE events.

    if ( par ) goto step3            ; // If we are in the STOPPING state


/*************************************************************************
 * Step 2 : At that point the automaton is in the STARTED state (the state   *
 * ------   variable value is STARTED). The Driver's Interrupt handler has   *
 *          just started transmitting a new frame before sending us the      *
 *          RESP_WRITE for the previous frame. So we have "plenty" of time   *
 *          (The transmit duration of the just started frame) to give to the *
 *          driver another new frame. We call the "myio_send" procedure to   *
 *          do this, we remind that this call is a non-blocking one, it does *
 *          not waits for any RESP_WRITE event. So the situation is as       *
 *          follows:                                                         *
```

```
 *          - The driver is currently transmitting the very beginning of the *
 *            frame specified by the previous call to "myio_send".           *
 *          - The "myio_send" procedure we call here puts a new REQ_WRITE    *
 *            event in the VMIO event input queue. The fact of writing the   *
 *            event immediately interrupts the "myio_send" code, the VMIO    *
 *            stack is installed and the driver's REQ_WRITE handling         *
 *            treatment is immediately executed. But currently a frame is    *
 *            being sent, and so the driver's REQ_WRITE handling treatment   *
 *            will just store in an internal queue the values of the new     *
 *            REQ_WRITE parameters, including the buffer address and the     *
 *            number of data bytes to be send. Then we exit from the VMIO,   *
 *            the stack is restored, we resume the execution of the          *
 *            "myio_send" code and then we return from "myio_send".          *
 *          So after returning from "myio_send", the new situation is:       *
 *          - Transmission of the previous frame is "ongoing", we are still  *
 *            near the very beginning of this frame.                         *
 *          - The send request we just did is stored in an internal driver's *
 *            waiting queue that is only dedicated for write requests.       *
 *          We increment by 1 the "ctx->nbreq" counter and its value is now  *
 *          equal to 2. We are done and we exit from the present procedure.  *
 ****************************************************************************/

    myio_send (ctx->iod   ,            // IOD of device
               ctx->buf   ,            // Buffer Address
               ctx->buflg ,            // Number of bytes to be sent
               &ret        )         ; // Counter (not used here)

    ctx->nbreq ++                     ; // Count goes from 1 to 2

    goto end                          ; // we are done


/****************************************************************************
 * Step 3 : Here, the automaton is in the STOPPING state (the state variable *
 * ------   value is equal to STOPPING). We have decremented the counter of  *
 *          remaining awaited RESP_WRITE. We will go two times through this  *
 *          step. The first time the new counter value is 1 and we do        *
 *          nothing. The second time the new couner value is 0 and we set    *
 *          the C_END condition, thus resulting to an immediate call of      *
 *          "usnd_end" treatment following the exit of the present procedure.*
 *          The "usnd_end" procedure sends a R_SND_STOP event to the one     *
 *          that sent us the SND_STOP request. This R_SND_STOP event is      *
 *          awaited by the "req_snd_stop" procedure that is currently        *
 *          unscheduled.                                                     *
 ****************************************************************************/

    step3                             : // Current state is STOPPING

    if (ctx->nbreq LE 0)                // If no more RESP_WRITE are awaited
      relaut(C_END)                   ; // then execute the "usnd_end"
                                        // treatment

    end                               :

    return 0                          ; // Exit without any error
  }


/*  Procedure usnd_end ------------------------------------------------------

    Purpose : C_END condition treatment. A response event is returned to the
              one that sends us the SND_STOP event.
*/

static int usnd_end(int par)
  {
    Usnd_ctx        *ctx              ; // Context address
    S_evt           evt               ; // Response event structure
```

```
/*****************************************************************************
 * Step 1 : Set a pointer "ctx" on the context structure for the current      *
 * ------   logical way. The VMK, before calling the present procedure, has   *
 *          set the "voielog" global variable to the current LW number. We    *
 *          use it as an index in our context table "usnd_ctx[]" and so "ctx" *
 *          is set to &usnd_ctx[voielog]                                      *
 *****************************************************************************/

    ctx        = usnd_ctx              // We set "ctx" to the address of
               + voielog             ; // usnd_ctx[voielog]


/*****************************************************************************
 * Step 2 : Send the R_SND_STOP response event. The SND_STOP event has been  *
 * -------  sent by "req_snd_stop" procedure and this one has requested to   *
 *          be unscheduled until a R_SND_STOP event is received. Here, we    *
 *          send that response event. The "usnd_stop" procedure was the      *
 *          treatment of the SND_STOP event and it has stored in the context *
 *          the information that we need here to build the response event:   *
 *          - The sender automaton/logical way numbers have been stored in   *
 *            the "emet" and "evl" fields of the context.                    *
 *          - The sender's waiting queue number has been written in "nfa".   *
 *          After setting the event's field, we just call "putevt_vmk"       *
 *          procedure. After we return from the present procedure, the VMK   *
 *          will re-schedule the "req_snd_stop" procedure that is waiting for *
 *          this event.                                                      *
 *****************************************************************************/

    Memset(&evt,0,sizeof(evt))       ; // Clear the event structure

    evt.aut      = ctx->emet         ; // Target automaton number
    evt.vl       = ctx->evl          ; // Target logical way number
    evt.code     = R_SND_STOP        ; // Event code

    putevt_vmk(ctx->nfa   ,            // VMK Internal queue 2 to 15
               &evt        )         ; // Event to be written


    return 0                         ; // Exit without any error
  }


/*  Procedure urcu_send ------------------------------------------------------

    Purpose : C_RCU and TO_RCU treatment. We request the RCU system driver to
              start transmitting a new message.
*/

static int urcu_send(int par)
  {
    Usnd_ctx        *ctx             ; // Context address
    int             ret              ; // Error code from send_rcu

/*****************************************************************************
 * Step 1 : Set a pointer "ctx" on the context structure for the current      *
 * ------   logical way. The VMK, before calling the present procedure, has   *
 *          set the "voielog" global variable to the current LW number. We    *
 *          use it as an index in our context table "usnd_ctx[]" and so "ctx" *
 *          is set to &usnd_ctx[voielog]                                      *
 *****************************************************************************/

    ctx        = usnd_ctx              // We set "ctx" to the address of
               + voielog             ; // usnd_ctx[voielog]

    send_rcu( ctx->iod              ,  // Not an IOD but RCU system identifier
         (unsigned short*) ctx->buf ,  // Message to be sent address
         ctx->buflg                 ,  // Size of the message
```

```
        ctx->rpt                  ,  // Repeat count for the message
        RESP_WRITE                ,  // Notification event code
        &ret                      ) ; // Error code

    return 0                        ; // Exit without any error
  }


/*  Procedure urcu_tostart ----------------------------------------------------

    Purpose : RESP_WRITE treatment for RCU. We start a TO_RCU time-out. When
              the end of period notification event will be received, then the
              "urcu_send" treatment will be executed to start next transmission
*/

static int urcu_tostart(int par)
  {
    Usnd_ctx        *ctx            ; // Context address

/*****************************************************************************
 * Step 1 : Set a pointer "ctx" on the context structure for the current     *
 * ------   logical way. The VMK, before calling the present procedure, has  *
 *          set the "voielog" global variable to the current LW number. We   *
 *          use it as an index in our context table "usnd_ctx[]" and so "ctx"*
 *          is set to &usnd_ctx[voielog].                                    *
 *****************************************************************************/

    ctx        = usnd_ctx           // We set "ctx" to the address of
               + voielog            ; // usnd_ctx[voielog]


/*****************************************************************************
 * Step 2 : We start the timer and keep its identifier in the "ctx->idto"    *
 * ------   field context. The reason to keep this identifier is because in  *
 *          the event we would clear the time-out, the "clear_uto" needs it. *
 *          Otherwise we would not need to keep it.                          *
 *          ------------------------------------------------------------- *
 *          The VMK uses NFA_STD_PS as the internal VMK queue for ALL the    *
 *          FSM's, therefore we must use NFA_STD_PS as the third parameter   *
 *          to the "set_to" procedure.                                       *
 *****************************************************************************/

    set_to(TIMER            ,         // One shot timer
           5000             ,         // Timer duration in milliseconds
           NFA_STD_PS       ,         // Wait queue for the expiration event
           TO_RCU           ,         // End of period notification event code
           0                ,         // Reserve field for notification event
           AUT_USND         ,         // FSM number for notification event
           voielog          ,         // Logical way for notification event
           (int*) &ctx->idto  )     ; // Returned time-out identifier

    return 0                        ; // Exit without any error
  }


/*  Procedure urcu_stop -------------------------------------------------------

    Purpose : SND_STOP treatment for RCU. This treatment is executed when we
              have a running time-out, so when we wait before sending the next
              RCU message. We just clear the running time-out and then we send
              the R_SND_STOP response event.
*/

static int urcu_stop(int par)
  {
    Usnd_ctx        *ctx            ; // Context address
    S_evt           evt             ; // Response event structure
```

```
/*****************************************************************************
 * Step 1 : Set a pointer "ctx" on the context structure for the current      *
 * ------   logical way. The VMK, before calling the present procedure, has   *
 *          set the "voielog" global variable to the current LW number. We    *
 *          use it as an index in our context table "usnd_ctx[]" and so "ctx" *
 *          is set to &usnd_ctx[voielog]                                      *
 *****************************************************************************/

    ctx        = usnd_ctx              // We set "ctx" to the address of
               + voielog             ; // usnd_ctx[voielog]

/*****************************************************************************
 * Step 2 : Clear the runnning time-out. The time-out identifier has been    *
 * ------   stored in "ctx->idto" by the "urcu_tostart" treatment, we just   *
 *          have to call the "clear_uto" procedure                           *
 *****************************************************************************/

    clear_to(ctx->idto ,               // Time-out identifier
             AUT_RCU   ,               // Time-out destination FSM
             voielog    )            ; // Time-out destination LW


/*****************************************************************************
 * Step 3 : Send the R_SND_STOP response event. The SND_STOP event has been  *
 * -------  sent by "req_snd_stop" procedure and this one has requested to be *
 *          unscheduled until a R_SND_STOP event is received. Here, we send  *
 *          that response event. The response to the received SND_STOP event *
 *          is built as follows:                                             *
 *          - The "aut" destination is "evt_usnd.emet"                       *
 *          - The "vl"  destination is "evt_usnd.evl"                        *
 *          - The "code" response event is R_SND_STOP                        *
 *          The queue number to put the event response in is "evt_usnd.res2". *
 *          After setting the event's field, we just call "putevt_vmk"       *
 *          procedure. After we return from the present procedure, the VMK   *
 *          will re-schedule the "req_snd_stop" procedure that is waiting for *
 *          this event.                                                      *
 *****************************************************************************/

    Memset(&evt,0,sizeof(evt))       ; // Clear the event structure

    evt.aut      = evt_usnd.emet     ; // Target automaton number
    evt.vl       = evt_usnd.evl      ; // Target logical way number
    evt.code     = R_SND_STOP        ; // Event code

    putevt_vmk(evt_usnd.res2 ,         // VMK Internal queue 2 to 15
               &evt            )     ; // Event to be written


    return 0                         ; // Exit without any error
  }


/*  Procedure req_snd_start ------------------------------------------------

    Purpose : Subroutine to send a SND_START event to the AUT_USND automaton
              and then wait for the R_SND_START response event. This procedure
              is the API to the FSM and is intended to be used from the user
              task code.
*/

static void req_snd_start(int vl, unsigned int iod, unsigned char *buf, int lg)
  {
    S_evt           evt              ; // Response event structure

/*****************************************************************************
 * Step 1 : Initialize the "evt" structure fields:                           *
 * ------   - Set the event destination, that is made of the automaton       *
 *            destination number AUT_USND and the logical way destination    *
```

```
 *            number "vl".                                                  *
 *          - We have are arbitrally chosen to use the "res2" field to      *
 *            provide to AUT_USND our own internal queue number. The        *
 *            "usnd_start" treatement of AUT_USND copies the "res2" field   *
 *            in the "nfa" field of its "Usnd_ctx" context and then this    *
 *            value will be used by the AUT_USND C treatments as the first  *
 *            parameter to the "putevt_vmk" function. We do not have to call*
 *            any API procedure to get our own Internal Queue Number since  *
 *            the VMK provides us a global variable "numfa" that holds that *
 *            value. So we just have to set "evt.res2" to "numfa" value.    *
 *          - The event's code is SND_START. Indeed, this value MUST exist  *
 *            in the AUT_USND transtion table (trans_usnd[]).               *
 *          - The other values we want to communicate as parameters to the  *
 *            AUT_USND FSM are the iod to be used to send data, the buffer  *
 *            to be sent and the count of bytes of the buffer. We           *
 *            respectively use the fields "reserve", "adresse" and          *
 *            "longueur" of the event to convey those.                      *
 *          ------------------------------------------------------------- *
 *          The parsing of this event is done within "usnd_start", that     *
 *          stores the parameters in the proper "Usnd_ctx" context structure.*
 ***************************************************************************/

    Memset(&evt,0,sizeof(evt))       ; // Clear the event structure

    evt.aut      = AUT_USND          ; // Target automaton number
    evt.vl       = vl                ; // Target logical way number
    evt.res2     = numfa             ; // Our Internal Queue number

    evt.code     = SND_START         ; // Event code
    evt.reserve  = iod               ; // reserve = iod
    evt.adresse  = buf               ; // adresse = telecom buffer address
    evt.longueur = lg                ; // lg      = Number of bytes in buffer
                                       // lg      = rpt<<16 + nbsymb


/***************************************************************************
 * Step 2 : Write in VMK internal waiting queue the event for the FSM.     *
 * ------   The VMK uses NFA_STD_PS as the internal VMK queue for ALL the  *
 *          FSM's, therefore we must use NFA_STD_PS as the first parameter *
 *          to the "putevt_vmk" procedure.                                 *
 ***************************************************************************/

    putevt_vmk(NFA_STD_PS ,            // VMK Internal queue 17
               &evt        )         ; // Event to be written


/***************************************************************************
 * Step 3 : Wait for the R_SND_START response event send by AUT_USND.      *
 * ------   We arbitrarily have chosen that AUT_USND responds to SND_START *
 *          by a R_SND_START response event (see procedure "usnd_start"). So *
 *          here we unschedule our task until we receive R_SND_START. We use *
 *          the "wait_coderes" subroutine to unschedule.                   *
 ***************************************************************************/

    wait_coderes(R_SND_START,0)      ; // Unschedule until R_SND_START is
                                       // received.
  }


/*  Procedure req_snd_stop ------------------------------------------------

    Purpose : Subroutine to send a SND_STOP event to the AUT_USND automaton
              and to wait for the response event
*/

static void req_snd_stop(int vl)
  {
    S_evt           evt              ; // Response event structure
```

```
/*************************************************************************
 * Step 1 : Initialize the "evt" structure fields:                       *
 * ------   - Set the event destination, that is made of the automaton   *
 *            destination number AUT_USND and the logical way destination*
 *            number "vl".                                               *
 *          - We have are arbitrally chosen to use the "res2" field to   *
 *            provide to AUT_USND our own internal queue number. The     *
 *            "usnd_stop" treatement of AUT_USND copies the "res2" field *
 *            in the "nfa" field of its "Usnd_ctx" context and then this *
 *            value will be used by another AUT_USND C treatment (usnd_end)*
 *            as the first parameter to the "putevt_vmk" function. We do not *
 *            have to call any API procedure to get our own Internal Queue *
 *            Number since the VMK provides us a global variable "numfa" *
 *            that holds that value. So we just have to set "evt.res2" to *
 *            "numfa" value.                                             *
 *          - The event's code is SND_STOP. Indeed, this value MUST exist *
 *            in the AUT_USND transtion table (trans_usnd[]).            *
 *          ------------------------------------------------------------ *
 *          The parsing of this event is done within "usnd_stop", that   *
 *          stores the parameters in the proper "Usnd_ctx" context structure.*
 *          The response event R_SND_STOP will be send by "usnd_end".     *
 *          ------------------------------------------------------------ *
 *          One can wonder why we again send to AUT_USND our internal queue *
 *          number since we already sent a queue number to AUT_USND along *
 *          with the SND_START event. The reason to do this is that we do *
 *          not impose that SND_START and SND_STOP to be sent by the same *
 *          task, even it is the case with this sample code.              *
 *************************************************************************/

   Memset(&evt,0,sizeof(evt))       ; // Clear the event structure

   evt.aut      = AUT_USND          ; // Target automaton number
   evt.vl       = vl                ; // Target logical way number
   evt.res2     = numfa             ; // Our Internal Queue number

   evt.code     = SND_STOP          ; // Event code


/*************************************************************************
 * Step 2 : Write in VMK internal waiting queue the event for the FSM.   *
 * ------   The VMK uses NFA_STD_PS as the internal VMK queue for ALL the *
 *          FSM's, therefore we must use NFA_STD_PS as the first parameter *
 *          to the "putevt_vmk" procedure.                               *
 *************************************************************************/

   putevt_vmk(NFA_STD_PS ,            // VMK Internal queue 17
              &evt        )         ; // Event to be written


/*************************************************************************
 * Step 3 : Wait for the R_SND_STOP response event send by AUT_USND.     *
 * ------   We arbitrarily have chosen that AUT_USND responds to SND_STOP *
 *          by a R_SND_STOP response event (see procedure "usnd_end"). So *
 *          here we have to unschedule our task until we receive the     *
 *          R_SND_STOP response event. We use the "wait_coderes" subroutine *
 *          to unschedule.                                               *
 *************************************************************************/

   wait_coderes(R_SND_STOP,0)       ; // Unschedule until T_SND_STOP is
                                      // received
  }


/*  Procedure loop_app_tsk ------------------------------------------------

    Purpose : This is our task main loop.
*/
```

```
int loop_app_tsk(void *param)
  {
    const M_trans   *tra             ; // Transition table line address
    int             ret              ; // Treatement error code

/*****************************************************************************
 * Step 1 : Here we initialize the needed drivers:                           *
 * ------   - The ASY driver, we are using here two serial port contollers.  *
 *          - The GPIO driver, we are using one GPIO controller.             *
 *          - The LED driver, we are using one LED controller.               *
 *          - The keyboard interface KBDITF.                                 *
 *          - The DAC driver, we are using one converter.                    *
 *          We also give the first receive token to ASY\DEV0 that we are     *
 *          using as a "console". The menu is sent to ASY\DEV0 and keyboard  *
 *          characters are received from ASY\DEV0. We also give 2 tokens to  *
 *          the other ASY\DEV1 and ASY\DEV2 lines because we accept all      *
 *          incoming data. When the AUT_USND FSM is idle, the present task   *
 *          echoes on ASY all that is received on this same line. When the   *
 *          AUT_USND is running for ASY1/ASY2, then the present task just    *
 *          discards the data received on ASY1/ASY2 (see "ev_asy_rcv"        *
 *          procedure).                                                      *
 *****************************************************************************/

    start                            : // Label if "restart" is requested

    open_asy()                       ; // Open the three serial ports
    open_gpio()                      ; // Open the GPIO driver
    open_led()                       ; // Open the LED driver
    open_kbd()                       ; // Open the KBDITF driver
    open_dac()                       ; // Open the DAC driver

    myio_givetok(asy_iod0  ,           // I/O descriptor
                 2         ,           // Number of receive token
                 1         ,           // Size of receive buffer
                 &tok_rcv[0] )       ; // Counter of given tokens

    myio_givetok(asy_iod1 ,            // I/O descriptor
                 2        ,            // Number of receive token
                 LGRCV12  ,            // Size of receive buffer
                 &tok_rcv[1])        ; // Counter of given tokens

    myio_givetok(asy_iod2 ,            // I/O descriptor
                 2        ,            // Number of receive token
                 LGRCV12  ,            // Size of receive buffer
                 &tok_rcv[2])        ; // Counter of given tokens


/*****************************************************************************
 * Step 2 : The menus allow the user to modify 19 values that are the 19     *
 * ------   integers of the "cfg[19]" global variable. Here we affect        *
 *          initial values to those ones. One has to be aware that the       *
 *          "open_xyz" procedures are doing "set_vals" with default string   *
 *          tags (asy_taglist0, asy_taglist1, gpio_tag_list) and the values  *
 *          we affect here must be the same ones are the ones defined by the *
 *          tag lists, or the values displayed on the UI will not be the     *
 *          ones effectively used by the drivers.                            *
 *****************************************************************************/

    cfg [VRED   ] = CFG_STOPPED      ; // Pattern on RED   : stopped
    cfg [VGREEN ] = CFG_STOPPED      ; // Pattern on GREEN : stopped
    cfg [VYELLO ] = CFG_STOPPED      ; // Pattern on YELLOW: stopped
    cfg [VASY1  ] = CFG_STOPPED      ; // Serial ASY1 transmit: stopped
    cfg [VASY2  ] = CFG_STOPPED      ; // Serial ASY2 transmit: stopped
    cfg [VDAC   ] = CFG_STOPPED      ; // DAC transmit: stopped
    cfg [VUHFRCU] = CFG_STOPPED      ; // UHF transmit: stopped

    cfg [VASY1PT] = 0                ; // ASY1 transmit pattern 0x00 -> 0xFF
```

```
   cfg [VASY2PT] = 1               ; // ASY2 transmit pattern 0xFF -> 0x00
   cfg [VDACPT ] = 0               ; // DAC  transmit pattern

   cfg [VBAUD1 ] = CFG_9600        ; // Baudrate: 9600 bauds
   cfg [VBITS1 ] = CFG_8BITS       ; // Nb of bits/sym: 8 bits/sym
   cfg [VSTOP1 ] = CFG_1STOP       ; // Nb of stop bits: 1 stop bit
   cfg [VPAR1  ] = CFG_NOPARITY    ; // Parity: none
   cfg [VDMA1  ] = CFG_DMAON       ; // DMA: ON
   cfg [VFLOW1 ] = CFG_NOFLOWCTL   ; // Flow control: none
   cfg [VFLUSH1] = CFG_FULLFLUSH   ; // Timing config: none
   cfg [VTEMPO1] = CFG_1MS         ; // Timing: 1 ms
   cfg [VSPEED1] = CFG_FAST        ; // Speed: normal

   cfg [VBAUD2 ] = CFG_9600        ; // Baudrate: 9600 bauds
   cfg [VBITS2 ] = CFG_8BITS       ; // Nb of bits/sym: 8 bits/sym
   cfg [VSTOP2 ] = CFG_1STOP       ; // Nb of stop bits: 1 stop bit
   cfg [VPAR2  ] = CFG_NOPARITY    ; // Parity: none
   cfg [VDMA2  ] = CFG_DMAON       ; // DMA: ON
   cfg [VFLOW2 ] = CFG_NOFLOWCTL   ; // Flow control: none
   cfg [VFLUSH2] = CFG_FULLFLUSH   ; // Timing config: none
   cfg [VTEMPO2] = CFG_1MS         ; // Timing: 1 ms
   cfg [VSPEED2] = CFG_FAST        ; // Speed: normal

   cfg [VPTIM  ] = CFG_P50MS       ; // Software polling interval: 50 msec
   cfg [VPCNT  ] = CFG_P3          ; // Software polling count: 3 times
   cfg [VMON1  ] = CFG_POLL        ; // Monitoring method: polling

   cfg [VDTIM1 ] = CFG_D50US       ; // Hardware polling interval: 50 usec
   cfg [VDCNT1 ] = CFG_D3          ; // Hardware polling count: 3 times
   cfg [VMON2  ] = CFG_POLL        ; // Monitoring method: polling
   cfg [VDTIM2 ] = CFG_D50US       ; // Hardware polling interval: 50 usec
   cfg [VDCNT2 ] = CFG_D3          ; // Hardware polling count: 3 times
   cfg [VMON3  ] = CFG_POLL        ; // Monitoring method: polling
   cfg [VDTIM3 ] = CFG_D50US       ; // Hardware polling interval: 50 usec
   cfg [VDCNT3 ] = CFG_D3          ; // Hardware polling count: 3 times

   cfg [VINPUT ] = 0               ; // RCU receive input : IR input
   cfg [VNUMRCU] = 0               ; // RCU choice name
   cfg [VONMIN ] = 8000            ; // Leader code : ON minimal usec
   cfg [VONMAX ] = 10000           ; // Leader code : ON maximal usec
   cfg [VSTMIN ] = 12000           ; // Leader code : SYMB minimal usec
   cfg [VSTMAX ] = 15000           ; // Leader code : SYMB maximal usec

/*****************************************************************************
 * Step 3 : The "sval" subroutine is the UI treatment procedure that calls   *
 * ------   the driver "setval" API (it calls "myio_setval" that calls the   *
 *          "drv_setval" OS function). The "sval" treatment takes as input   *
 *          ONE parameter "n" that is an index is "cfg[]". The string        *
 *          character, the tag list to be given to "myio_setval" is simply   *
 *          "str_tag[n][ cfg[n] ]. The "str_tag|n]" is a list of string      *
 *          addresses and we use "cfg[0]". (0,1,...) is the selector for one *
 *          of the strings. For example with n=VBITS, then "str_tag[VBITS]"  *
 *          is the address of "tag_bits" and then if for example "cfg[VBITS]"*
 *          is 1 then "str_tag[1] is the address of "NBBITS=7", that will be *
 *          given to "myio_setval". But the "myio_setval" procedure also     *
 *          needs another parameter that is an IOD. It has to be noted that  *
 *          IODs are not predefined values but are returned by the           *
 *          "myio_open" or the "myio_alloc_sub" procedure. So here we set in *
 *          the proper "iods[]" entry (in iods[n]") the needed IOD value.    *
 *          So in the end, the code of "sval(n)" is simply the instruction   *
 *          "myio_setval( iods[n] , str_tag[n][ cfg[n] ] ).                  *
 *****************************************************************************/

   iods[VRED   ] = led_iod0        ; // Pattern on RED  : IOD
   iods[VGREEN ] = led_iod1        ; // Pattern on GREEN: IOD
   iods[VYELLO ] = led_iod2        ; // Pattern on GREEN: IOD
   iods[VASY1  ] = asy_iod1        ; // Start/stop ASY1 : IOD for AUT_USND
   iods[VASY2  ] = asy_iod2        ; // Start/stop ASY1 : IOD for AUT_USND
```

```
    iods[VDAC   ] = 0               ; // Start/stop DAC  : IOD
    iods[VUHFRCU] = rcusnd_id       ; // Start/stop RCU  : RCU id

    iods[VBAUD1 ] = asy_iod1        ; // Baudrate:
    iods[VBITS1 ] = asy_iod1        ; // Nb of bits/sym:
    iods[VSTOP1 ] = asy_iod1        ; // Nb of stop bits:
    iods[VPAR1  ] = asy_iod1        ; // Parity:
    iods[VDMA1  ] = asy_iod1        ; // DMA:
    iods[VFLOW1 ] = asy_iod1        ; // Flow control:
    iods[VFLUSH1] = asy_iod1        ; // Timing config:
    iods[VTEMPO1] = asy_iod1        ; // Timing:
    iods[VSPEED1] = 0               ; // Speed: no IOD

    iods[VBAUD2 ] = asy_iod2        ; // Baudrate:
    iods[VBITS2 ] = asy_iod2        ; // Nb of bits/sym:
    iods[VSTOP2 ] = asy_iod2        ; // Nb of stop bits:
    iods[VPAR2  ] = asy_iod2        ; // Parity:
    iods[VDMA2  ] = asy_iod2        ; // DMA:
    iods[VFLOW2 ] = asy_iod2        ; // Flow control:
    iods[VFLUSH2] = asy_iod2        ; // Timing config:
    iods[VTEMPO2] = asy_iod2        ; // Timing:
    iods[VSPEED2] = 0               ; // Speed: no IOD

    iods[VPTIM  ] = gpio_iod        ; // Software polling rate: controller
    iods[VPCNT  ] = gpio_iod        ; // Software polling count: controller

    iods[VMON1  ] = gpio_iod1       ; // Monitoring method: sub-channel
    iods[VDTIM1 ] = gpio_iod1       ; // hardware polling rate: sub-channel
    iods[VDCNT1 ] = gpio_iod1       ; // hardware polling count: sub-channel
    iods[VMON2  ] = gpio_iod2       ; // Monitoring method: sub-channel
    iods[VDTIM2 ] = gpio_iod2       ; // hardware polling rate: sub-channel
    iods[VDCNT2 ] = gpio_iod2       ; // hardware polling count: sub-channel
    iods[VMON3  ] = gpio_iod3       ; // Monitoring method: sub-channel
    iods[VDTIM3 ] = gpio_iod3       ; // hardware polling rate: sub-channel
    iods[VDCNT3 ] = gpio_iod3       ; // hardware polling count: sub-channel


/*************************************************************************
 * Step 4 : Here we draw the first menu, that is MMAIN. We call the "pmen"   *
 * ------   procedure with "MMAIN" value as a parameter. This procedure      *
 *          send the "mmain" const string. Then "pmen" calls the "ptim"      *
 *          subroutine that reads the time, convert it to an ASCII string    *
 *          and sends it on ASY0. But the menu also shows the 7 states of    *
 *          "cfg[0] to "cfg[6]". To do that we use "pgrp(GMAIN") that        *
 *          converts the 7 integer values in corresponding strings and then  *
 *          send on ASY0 this text.                                          *
 *************************************************************************/

    pmen(MMAIN)                     ; // Print menu on serial line ASY\DEV0
    pgrp(GMAIN)                     ; // Print variable group on ASY\DEV0


/*************************************************************************
 * Step 5 : We start a timer that will send us an event every 1 second so    *
 * ------   that we can update the time. As the timer is started in CLOCK    *
 *          mode, we will not have to call again "set_tto".                  *
 *************************************************************************/

    set_tto(CLOCK      ,              // Timer mode: clock
            1000       ,              // Duration in milliseconds
            TICK       ,              // Event code
            0          ,              // Event reserve field
            &idto        )          ; // Timer identifier


/*************************************************************************
 * Step 6 : Here is our main event loop. For each loop, we do as follows:    *
 * -------  - First we wait for an event (call "wait_myevents").             *
```

```
*          - Then we call the "ev_opt" procedure that will convert the      *
*            "ev_val" value to another value that exists in the "evtcode"   *
*            field of the menu transition table.                            *
*          - Then we scan the "m_trans[page]" (list of transitions for the  *
*            page number "state") in order to find the transition for the   *
*            "opt_val" event code. The transition "evtcode" field must be   *
*            equal to "opt_val". If no transition matches "opt_val" then    *
*            the scan will stop with the last transition that is indicated  *
*            with a DEFAULT value in "evtcode".                             *
*          - The "tra" pointer is now on the found transition (that may be  *
*            the DEFAULT, the last one. We just call in order the 5         *
*            procedures that are pointed by "tra->trt1", "tra->trt2",       *
*            "tra->trt3", "tra->trt4" and "tra->trt5".                      *
*          - After executing the 5 treatments, we test if any of those has  *
*            set a non 0 value in the "condcode" global variable. If yes    *
*            this is our FSM engine condition, we set the "opt_val"         *
*            variable with "condcode" and we jump to scan again the         *
*            transition table, searching for a code event equal to          *
*            "codcode".                                                     *
*          - If one of the 5 treatments is the "rest" procedure then its    *
*            set to 1 the "flag_rest" procedure and we stop the loop.       *
*          ------------------------------------------------------------------ *
*          One can wonder how the "page" variable is updated. The answer is *
*          that each time the "pmen" procedure is called its "page" sets    *
*          to its input parameter.                                          *
*************************************************************************/

   errcode   = 0                    ; // Init of global error code
   flag_rest = 0                    ; // Init of the end-of-loop flag. It is
                                      // set to 1 by "rest" (if called)

   wait_ev                          : // Start of our event loop
                                      // ------------------------------------
   ev_val  = wait_myevents()        ; // Unschedule until one event is
                                      // received

   opt_val = ev_opt( ev_val )       ; // Compute a value that can be found
                                      // in the "M_trans" structures

   scan                             : // Parse the transition table
   condcode = 0                     ; // Reset the condition code
                                      //
   for (tra = m_trans[page]      ;    // Scan the "m_trans[page]" list of
        tra->evtcode != opt_val &&    // transition until we found the line
        tra->evtcode != DEFAULT  ;    // which "evtcode" is "opt_val" or
        tra ++                    ) ; // the DEFAULT (last) line of the list.

   if( (ret = tra->trt1(tra->p1)) )   // Execute p1
     goto error                     ; // and stop if we have an error

   if( (ret = tra->trt2(tra->p2)) )   // Execute p2
     goto error                     ; // and stop if we have an error

   if( (ret = tra->trt3(tra->p3)) )   // Execute p3
     goto error                     ; // and stop if we have an error

   if( (ret = tra->trt4(tra->p4)) )   // Execute p4
     goto error                     ; // and stop if we have an error

   if( (ret = tra->trt5(tra->p5)) )   // Execute p5
     goto error                     ; // and stop if we have an error

   if ( condcode )                    // If any of the treatment as set a
     {                                // non 0 value in the "codecode"
       opt_val = condcode           ; // global variable, this is our FSM
       goto scan                    ; // condition, the condition becomes
     }                                // the new event code and we do another
                                      // transition
```

```
    if (flag_rest == 0) goto wait_ev ; // Goto wait for the next event or
    else                goto end     ; // stop if "pres" has set "flag_rest"

    error                            : // In case a treatment returned any
    errcode = ret                    ; // error code, we store it in the
    opt_val = OPT_ERROR              ; // "errocode" global variable, we set
    goto scan                        ; // a new OPT_ERROR event code and we
                                       // do another transition with OPT_ERROR


/*****************************************************************************
 * Step 7 : Close the five drivers, ASY, GPIO, LED, DRVITF and DAC           *
 * ------                                                                    *
 *****************************************************************************/

    end                              :

    clear_tto(idto)                  ; // Stop the timer

    if ( cfg[VASY1] )                  // If ASY1 is transmitting, send a
      req_snd_stop(VLASY1)           ; // STOP request to (AUT_USND,VLASY1)

    if ( cfg[VASY2] )                  // If ASY2 is transmitting, send a
      req_snd_stop(VLASY2)           ; // STOP request to (AUT_USND,VLASY2)

#ifdef TOTO
    if ( cfg[VDAC] )                   // If DAC is sending data, send a
      req_snd_stop(VLDAC)            ; // SND_STOP event to (AUT_USND,VLDAC)
#endif

    if ( cfg[VUHFRCU] )                // If RCU is transmitting, send a
      req_snd_stop(VLSNDRCU)         ; // STOP request to (AUT_USND,VLSNDRCU)

    close_asy()                      ; // Close the two serial ports
    close_gpio()                     ; // Close the GPIO driver
    close_led()                      ; // Close the LED driver
    close_kbd()                      ; // Close the KBDITF driver
    close_dac()                      ; // Close the DAC driver

    goto start                       ;

    return 0                         ;
  }


/*  Procedure open_asy ----------------------------------------------------

    Purpose : Initialize the code of the ASY driver and then opens the three
              serial port that are available. The two opened devices
              descriptors (Input/Output Descriptors") are kept in "asy_iod0"
              "asy_iod1" and "asy_iod2" respectively. Line parameters (speed,
              nbbits and parity) are set. Then we allocate for each controller
              one transmit buffer.
*/

static void open_asy(void)
  {
    int             ret              ; // Procedures returned code

/*****************************************************************************
 * Step 1 : Open the three devices ASY\DEV0 DEV1 and DEV2. We call the       *
 * -------  "myio_open" simple API procedure. In fact the subroutine calls   *
 *          "drv_init_driver", "add_uroute", "drv_alloc_device", and then    *
 *          "drv_open_device".                                               *
 *****************************************************************************/

    myio_open(DRVASY,DEV0,"",&asy_iod0) ; // Open "ASY\DEV0"
```

```
   myio_open(DRVASY,DEV1,"",&asy_iod1) ; // Open "ASY\DEV1"
   myio_open(DRVASY,DEV2,"",&asy_iod2) ; // Open "ASY\DEV2"


/*************************************************************************
 * Step 2 : Set line parameters for the two serial line controllers. We have *
 * ------   to call the "myio_setval" procedure. This procedure sends a      *
 *          REQ_SETVAL request event the AUT_ASY VMIO automaton. Then we     *
 *          unschedule until the RESP_SETVAL response event is received. We  *
 *          do not use the same parameters with the UI ASY0 asynchronous     *
 *          line and the AUT_USND ASY1 line. We use asy_taglist0 with ASY0:  *
 *          "BAUDS=115200\nNBBITS=8\nSTOPBIT=1\nPARITY=NONE". With ASY1 and  *
 *          ASY2 we use taglist1: "BAUDS=9600\nNBBITS=8\nSTOPBIT=1\n"        *
 *          "PARITY=NONE\nBUFSIZE=0\nFLOWCTL=NONE\nTMODE=NONE\nTEMPO=1".     *
 *************************************************************************/

   myio_setval(asy_iod0,asy_taglist0)  ; // Set "ASY\DEV0" tags
   myio_setval(asy_iod1,asy_taglist1)  ; // Set "ASY\DEV1" tags
   myio_setval(asy_iod2,asy_taglist1)  ; // Set "ASY\DEV2" tags


/*************************************************************************
 * Step 3 : Allocates memory buffers for transmission, one for DEV0 and two  *
 * ------   other ones for DEV1/DEV2. We store buffers addresses in          *
 *          "buf_snd[0]" and "buf_snd[1/2]". It has to be noted that we do   *
 *          not allocate any receive buffer, because this is done by the ASY *
 *          driver. When the VMIO AUT_ASY receives a REQ_REQ, this event     *
 *          contains a number of allowed receive tokens. The receive buffer  *
 *          allocation is done by AUT_ASY. So the present module does not    *
 *          contains any call to "alloc_buf" regarding receive buffers.      *
 *************************************************************************/

   alloc_buf(myapp_memuid ,           // Memory user id
             1500         ,           // Size in bytes of requested buffer
             0            ,           // Flags
             &buf_snd[0]  ,           // Address of allocated buffer
             &ret          )        ; // Return code

   alloc_buf(myapp_memuid ,           // Memory user id
             256          ,           // Size in bytes of requested buffer
             0            ,           // Flags
             &buf_snd[1]  ,           // Address of allocated buffer
             &ret          )        ; // Return code

   alloc_buf(myapp_memuid ,           // Memory user id
             256          ,           // Size in bytes of requested buffer
             0            ,           // Flags
             &buf_snd[2]  ,           // Address of allocated buffer
             &ret          )        ; // Return code


/*************************************************************************
 * Step 4 : AUT_USND will send a continuous byte stream through ASY\DEV1 or  *
 * ------   ASY\DEV2 when it is enabled, to test throughput. We initialize   *
 *          the send buffer once for all with bytes from 0xFF to 0x00.       *
 *************************************************************************/

   fill256(buf_snd[1],cfg[VASY1PT]) ; // Fill up the 256 bytes send buffer
   fill256(buf_snd[2],cfg[VASY2PT]) ; // Fill up the 256 bytes send buffer
  }


/*  Procedure close_asy ----------------------------------------------------

    Purpose : This procedure closes the three USART, then frees those three
              devices, terminates the ASY module, deletes the route that has
              been created for the IND_REPORT events, and finally frees the 3
              telecom buffers that were allocated by "open_asy".
```

```
*/

static void close_asy(void)
  {

    int             ret             ; // Return code

/*************************************************************************
 * Step 1 : We call the "myio_close" procedure that calls "drv_close_device",*
 * ------   "drv_free", "drv_end" and "del_uroute".                        *
 *************************************************************************/

    myio_close(asy_iod0)            ; // Closes "ASY\DEV0"
    myio_close(asy_iod1)            ; // Closes "ASY\DEV1"
    myio_close(asy_iod2)            ; // Closes "ASY\DEV2"


/*************************************************************************
 * Step 2 : Free the 3 memory buffers that were allocated by "open_asy".     *
 * ------   For each of them, we just call the "free_buffer" procedure       *
 *************************************************************************/

    free_buf(buf_snd[0], &ret )     ; // Free "buf_snd[0]"
    free_buf(buf_snd[1], &ret )     ; // Free "buf_snd[1]"
    free_buf(buf_snd[2], &ret )     ; // Free "buf_snd[2]"

    buf_snd[0] = buf_snd[1] =         // Clear references that are
    buf_snd[2] =              HNULL ; // now invalid.
  }


/*  Procedure open_gpio ----------------------------------------------------

    Purpose : Initialize the code of the GPIO driver module, then allocates
              the GPIO controller device, initializes its hardware and finally
              creates a user route in order to catch the IND_REPORT events.
              The input/output descriptor is kept in the global variable
              "gpio_iod".
*/

static void open_gpio(void)
  {

/*************************************************************************
 * Step 1 : We call the "myio_open" that calls "drv_init_driver", then       *
 * ------   "add_uroute", then "drv_alloc_device" and "drv_open_device".     *
 *************************************************************************/

    myio_open(DRVGPIO,DEV0,"",&gpio_iod); // Open GPIO\DEV0


/*************************************************************************
 * Step 2 : The GPIO contoller is ready. We are going to allocate the pins   *
 * ------   subchannels, one for each button. We call the "drv_alloc_subchan"*
 *          procedure, this procedure sends a REQ_ALLOC_SUB event to the     *
 *          VMIO AUT_GPIO automaton. This automaton will then send us a      *
 *          RESP_ALLOC_SUB response event. Here we unschedule until this     *
 *          event is received. Buttons 1, 2, 3 have respectively connected   *
 *          to GPIO named  BUT0, BUT1 and BUT2.                              *
 *************************************************************************/

    myio_alloc_sub(gpio_iod,"UNAME=BUT1",&gpio_iod1); // Alloc BUT0 GPIO
    myio_alloc_sub(gpio_iod,"UNAME=BUT2",&gpio_iod2); // Alloc BUT1 GPIO
    myio_alloc_sub(gpio_iod,"UNAME=BUT3",&gpio_iod3); // Alloc BUT2 GPIO


/*************************************************************************
 * Step 3 : The pins are allocated. We need to configure the buttons to get  *
```

```
 * ------   IND_REPORT events on state change. We call the "myio_setval"     *
 *          procedure, this procedure sends a REQ_SETVAL event to the VMIO   *
 *          AUT_GPIO automaton. This automaton will then send us a           *
 *          RESP_SETVAL response event. Here we unschedule until this event  *
 *          is received. We do this for buttons 1, 2 and 3, so using the     *
 *          three sub-channels IOD, "gpio_iod1", "gpio_iod2" and "gpio_iod3".*
 *          We use the same tag list for the 3 GPIO's that are connected to  *
 *          "Normally Open" push buttons. Therefore we set a weak pull-up in *
 *          order to maintain the PIO in high state when the button is open. *
 *          When the button is pressed the PIO state is low (GND). So we     *
 *          set the GPIO as an input ("OUTPUT=HI_Z") with pull-up:           *
 *          "FUNC=GPIO\nOUTPUT=HI_Z\nWEAKPULL=UP\nEVENTS=REPORT\n".          *
 ****************************************************************************/

    myio_setval(gpio_iod1,gpio_taglist) ; // Configure BUTO GPIO
    myio_setval(gpio_iod2,gpio_taglist) ; // Configure BUT1 GPIO
    myio_setval(gpio_iod3,gpio_taglist) ; // Configure BUT2 GPIO
  }


/*  Procedure close_gpio ----------------------------------------------------

    Purpose : This procedure closes the GPIO controller, then frees this
              device, terminates the GPIO module and deletes the route that
              had been created for the IND_REPORT events.
*/

static void close_gpio(void)
  {

/****************************************************************************
 * Step 1 : Free all used subchannels. We call "myio_free_sub" that calls    *
 * ------   "drv_free_subchan" for each of the 3 buttons which IO descriptor *
 *          are "gpio_iod1/2/3". This procedure sends a REQ_FREE_SUB request *
 *          event to the VMIO AUT_GPIO automaton and here we unschedule      *
 *          until the corresponding RESP_FREE_SUB, sent by AUT_GPIO, is      *
 *          received.                                                        *
 ****************************************************************************/

    myio_free_sub(gpio_iod1)         ; // Free BUT1 GPIO
    myio_free_sub(gpio_iod2)         ; // Free BUT2 GPIO
    myio_free_sub(gpio_iod3)         ; // Free BUT3 GPIO


/****************************************************************************
 * Step 2 : Close the GPIO device. We call the "myio_close" subroutine that  *
 * ------   calls "drv_close" device, the "drv_free" and then "drv_end".     *
 *          A REQ_CLOSE request event is sent to AUT_GPIO and we unschedule  *
 *          until RESP_CLOSE is received from AUT_GPIO. A REQ_FREE event is  *
 *          sent and RESP_FREE is awaited. And a REQ_END driver termination  *
 *          request is sent and we unschedule until RESP_END is received.    *
 ****************************************************************************/

    myio_close(gpio_iod)             ; // Closes the GPIO device and the driver
  }


/*  Procedure open_led ------------------------------------------------------

    Purpose : Initializes the LED driver, opens the LED controller, allocates
              the three LED (RED, GREEN, YELLOW) subchannels.
*/

static void open_led(void)
  {

/****************************************************************************
 * Step 1 : We call the "myio_open" that calls "drv_init_driver", then       *
```

```
 * ------   "add_uroute", then "drv_alloc_device" and "drv_open_device".     *
 *          It has to be noted that the LED driver is a "software device",   *
 *          because there is no LED controller dedicated device. Most of the *
 *          time "hidden" GPIO (not visible by the application) are used.    *
 *          But other hardware may be used. For example a front panel will   *
 *          include a dedicated PCB and an I2C slave controller that drives  *
 *          and controls all the front-panel elements such as LEDS but also  *
 *          buttons, a 4x7 display, ... In such a case, the "drv_led" driver *
 *          will use low-level functions for I2C and not GPIO.               *
 ***************************************************************************/

    myio_open(DRVLED,DEV0,"",&led_iod)  ; // Open LED\DEV0


/***************************************************************************
 * Step 2 : The contoller is ready. We are going to allocate the LED        *
 * ------   subchannels, one for each LED. We call the "myio_alloc_sub"     *
 *          subroutine that calls the "drv_alloc_subchan" procedure. This   *
 *          last procedure sends a REQ_ALLOC_SUB event to the VMIO AUT_LED  *
 *          automaton. This automaton will then send us a RESP_ALLOC_SUB    *
 *          response event. Here we unschedule until this event is received. *
 ***************************************************************************/

    myio_alloc_sub(led_iod,"LEDNAME=RED"    ,&led_iod0); // Allocates RED
    myio_alloc_sub(led_iod,"LEDNAME=GREEN"  ,&led_iod1); // Allocates GREEN
    myio_alloc_sub(led_iod,"LEDNAME=YELLOW" ,&led_iod2); // Allocates YELLOW
  }


/*  Procedure close_led ----------------------------------------------------

    Purpose : Closes the LED controller, then frees it and terminates the LED
              driver module.
*/

static void close_led(void)
  {

/***************************************************************************
 * Step 1 : Free all used subchannels. We call "myio_free_sub" that calls   *
 * -------  "drv_free_subchan" for each of the 3 LEDs which IO descriptor   *
 *          are "led_iod0/1/2". This last procedure sends a REQ_FREE_SUB    *
 *          request event to the VMIO AUT_LED automaton and here we         *
 *          unschedule until the corresponding RESP_FREE_SUB, sent by       *
 *          AUT_LED, is received.                                           *
 ***************************************************************************/

    myio_free_sub(led_iod0)          ; // Frees RED
    myio_free_sub(led_iod1)          ; // Frees GREEN
    myio_free_sub(led_iod2)          ; // Frees YELLOW


/***************************************************************************
 * Step 2 : We call the "myio_close" procedure that calls "drv_close_device",*
 * ------   "drv_free", "drv_end" and "del_uroute".                         *
 ***************************************************************************/

    myio_close(led_iod)              ; // Closes LED device and driver
  }


/*  Procedure open_kbd -----------------------------------------------------

    Purpose : Initialize the code of the KBDITF driver and then open the
              device to get key press events from the remote control unit
              (RCU). The opened device descriptor (Input/Output Descriptor) is
              kept in "kbd_iod".
*/
```

```
static void open_kbd(void)
  {

    int             ret             ; // Procedures returned code
    unsigned short  *pt             ; // Write pointer in "buf_rcu

/*************************************************************************
 * Step 1 : Open the "KBDITF\DEV0" device. We call the "myio_open" simple    *
 * ------   API procedure. In fact the subroutine calls "drv_init_driver",   *
 *          "add_uroute", "drv_alloc_device", and then the "drvopen_device". *
 *          The "KBDITF\DEV0" driver is a software driver that does not      *
 *          handles any hardware by itself. With this sample application,    *
 *          the underlying hardware is an infra-red receiver. This hardware  *
 *          is managed by the "drv_rcu" driver, that in our case sends       *
 *          events to the KBDITF module, that sends DRV_KEY_PRESS and        *
 *          DRV_KEY_RELEASE events to us.                                    *
 *************************************************************************/

    myio_open(DRVKBD,DEV0,"",&kbd_iod)  ; // Opens KBDITF\DEV0


/*************************************************************************
 * Step 2 : Query for the RCU id for the UHF transmitter and store it in the *
 * -------  "rcusnd_id" global variable. Also query for the RCU id for the   *
 *          RCU receiver and keep it in the "rcurcv_id" global variable.     *
 *************************************************************************/

    get_rcu_id(1          ,             // 0:Receiver 1:Transmitter
               0          ,             // numdev = the first one
               &rcusnd_id ,             // Corresponding id
               &ret        )          ; // Error code

    get_rcu_id(0          ,             // 0:Receiver 1:Transmitter
               0          ,             // numdev = the first one
               &rcurcv_id ,             // Corresponding id
               &ret        )          ; // Error code


/*************************************************************************
 * Step 3 : Allocates one memory buffer for the RCU send function. Each      *
 * ------   symbol is made of two USHORTS (ON,OFF) durations and so 4 need,  *
 *          4 bytes per symbols and 40 * 4 = 160 bytes for 40 symbols.       *
 *************************************************************************/

    alloc_buf(myapp_memuid ,           // Memory user id
              160          ,           // Size in bytes of requested buffer
              0            ,           // Flags
              &buf_rcu     ,           // Address of allocated buffer
              &ret          )        ; // Return code

    pt = (unsigned short*) buf_rcu   ;


/*************************************************************************
 * Step 4 : Writes 4 symbols, 4 (ON,OFF) couples of durations in usec        *
 * ------                                                                    *
 *************************************************************************/

    *pt++ = 400                 ;
    *pt++ = 600                 ;

    *pt++ = 200                 ;
    *pt++ = 800                 ;

    *pt++ = 200                 ;
    *pt++ = 800                 ;
```

```
    *pt++ =  200                ;
    *pt++ = 1800                ;
  }


/*  Procedure close_kbd ----------------------------------------------------

    Purpose : This procedure closes the keyboard interface, frees the device,
              terminates the KBDITF module, deletes the route that has been
              created for the DRV_KEY_PRESS and DRV_KEY_RELEASE events.
*/

static void close_kbd(void)
  {

    int             ret             ; // Procedures returned code

/*************************************************************************
 * Step 1 : We call the "myio_close" procedure that calls "drv_close_device",*
 * ------   "drv_free", "drv_end" and "del_uroute". A REQ_CLOSE is sent to   *
 *          AUT_KBD and our task is unscheduled until we receive from       *
 *          the AUT_KBD FSM the RESP_CLOSE response event. Then we send a   *
 *          REQ_FREE event request and we unschedule until RESP_FREE is     *
 *          received from AUT_KBD. And we send a REQ_END and unschedule,    *
 *          waiting for the RESP_EVENT response event.                      *
 *************************************************************************/

    myio_close(kbd_iod)             ; // Closes KBDITF\DEV0

/*************************************************************************
 * Step 2 : Free the symbol buffer for the RCU transmit function           *
 * -------                                                                 *
 *************************************************************************/

    free_buf(buf_rcu   , &ret )     ; // Free "buf_rcu"

    buf_rcu = HNULL                 ; // Clear the buffer reference
  }


/*  Procedure open_dac -----------------------------------------------------

    Purpose : DAC module initialization, allocation and opening of DAC device
*/

static void open_dac(void)
  {
    int             ret             ; // Procedures returned code
    int             i               ; // Loop counter

/*************************************************************************
 * Step 1 : We have to initialize the driver module code. To do this, we   *
 * ------   have to send a REQ_INIT request event to its VMIO automaton, in *
 *          our case the VMIO AUT_DAC automaton. To do this, we call the   *
 *          "drv_init_driver" procedure. The AUT_DAC automaton responds with *
 *          a RESP_INIT event, we here unschedule until the reception of   *
 *          this response event. Then we have to allocate the DAC device,  *
 *          we have to send a REQ_ALLOC event to AUT_DAC and to unschedule  *
 *          until the RESP_ALLOC response event is received. This is done by *
 *          the "drv_alloc_device" procedure. Then we have to open the     *
 *          device with a REQ_OPEN request event to AUT_DAC and unschedule  *
 *          until the RESP_OPEN is received from AUT_DAC. All this is done  *
 *          with a single call to "myio_open" that issues DRV API calls. To *
 *          be short, "myio_open" calls:                                   *
 *          - "drv_init_driver" (send REQ_INIT and wait RESP_INIT)         *
 *          - "drv_alloc_device" (send REQ_ALLOC and wait RESP_ALLOC)      *
 *          - "drv_open_device" (send REQ_OPEN and wait RESP_OPEN)         *
 *************************************************************************/
```

```
    myio_open(DRVDAC,DEV0,dac_taglist,&dac_iod)  ; // Open DAC\DEV0


/*************************************************************************
 * Step 2 : Allocate a memory buffer to hold samples to convert.         *
 * ------   Sample freq at 8 kHz. 80 evt/s => 100 samples / req          *
 *************************************************************************/

    alloc_buf(myapp_memuid ,           // Memory user id
              200          ,           // Size in bytes of requested buffer
              0            ,           // Flags
              &buf_aud     ,           // Address of allocated buffer
              &ret          )        ; // Return code


/*************************************************************************
 * Step 3 : Fill buffer with samples. We build a 400 Hz triangle wave.   *
 * ------   ~400 Hz at 8kHz => 20 samples per cycles, 10 up, 10 down     *
 *************************************************************************/

    for (i = 0; i LT 20; i += 2)       /* Upward part of the triangle wave.  */
    {
      buf_aud[i]   = (i*0xFF) / 20   ; /*                                   */
      buf_aud[i+1] = (i*0xFF) / 40   ; /*                                   */
    }

    for (i = 0; i LT 20; i += 2)       /* Downward part of the triangle wave.*/
    {
      buf_aud[20 + i] =
                    0xFF - buf_aud[i]; /*                                   */
      buf_aud[20 + i + 1] =
                 0x7F + (i*0xFF) / 40; /*                                   */
    }

    for (i = 1 ; i LT 5 ; i++ )        /* We copy 4 times the first cycle    */
      Memcpy(buf_aud+40*i ,            /* made of 40 samples to fill the     */
             buf_aud      ,            /* 160 remaining samples of the buffer*/
             40            )        ; /*                                   */
  }


/*  Procedure close_dac ----------------------------------------------------

    Purpose : This procedure closes the DAC device, then frees this device,
              termminates the DAC driver module and finally frees the
              "buf_aud" sample buffer that has been allocated by "open_dac".
*/

static void close_dac(void)
  {

    int             ret             ; // Return code

/*************************************************************************
 * Step 1 : We call the "myio_close" procedure that calls "drv_close_device",*
 * ------  "drv_free" and "drv_end". A REQ_CLOSE is sent to AUT_DAC and   *
 *          our task is unscheduled until we receive from the AUT_DAC FSM *
 *          the RESP_CLOSE response event. Then we send a REQ_FREE event  *
 *          request and we unschedule until RESP_FREE is received from the*
 *          AUT_DAC FSM. And we send a REQ_END and unschedule, waiting for*
 *          the RESP_EVENT response event.                                *
 *************************************************************************/

    myio_close(dac_iod)             ; // Closes DAC\DEV0


/*************************************************************************
```

```
 * Step 2 : We free the sample buffer that was allocated by "open_dac".      *
 * ------   The buffer address is kept in the "buf_aud" global variable      *
 ****************************************************************************/

    free_buf(buf_aud , &ret )        ; // Free "buf_aud"

    buf_aud = HNULL                  ; // Clear the reference
  }


/*  Procedure empt ------------------------------------------------------------

    Purpose : UI treatment - Do nothing, but do it well
*/
static int empt(int n)
  {
    return 0                         ; // Nothing, with no error
  }


/*  Procedure pmen ------------------------------------------------------------

    Purpose : UI treatment - Prints (sends) on the ASY0 serial line one
              application menu, one sequence of ASCII characters.
*/

static int pmen(int n)
  {
    const char      *str             ; // Menu string address
    unsigned char   *snd             ; // Address of buffer to be sent
    int             lg               ; // Number of bytes to be sent
    const char      *ps              ; // Patch address
    char            *pd              ; // Pointer in "buf_snd[0]
    int             i                ; // Loop counter

/****************************************************************************
 * Step 1 : Copy in the transmit buffer the characters that have to be sent  *
 * ------   in order to print the menu. They are simply stored into the      *
 *          static array "menu[]" depending on the current page.             *
 ****************************************************************************/

    str = *menu[n]                   ; // String address of the menu text
    snd = buf_snd[0]                 ; // Buffer to be sent

    lg  = strlen(str)                ; // Number of characters to be copied
                                       // and then to be sent on ASY0
    Memcpy(snd,str,lg)               ; // Copy into "buf_snd[0]" the static
                                       // part of the menu

/****************************************************************************
 * Step 2 : If there is a title or another item to be replaced, search for   *
 * ------   the first "#" and patch at that point:                           *
 *          - If the patch address "patch[n]" is HNULL, nothing to do, go to *
 *            the next step.                                                 *
 *          - Search in "buf_snd[0]" for the first '#' character. We use     *
 *            "pd" as a current character pointer and "i" as a counter.      *
 *          - If we found a '#' before the end of the "lg" characters, we    *
 *            copy "patch" to location "pt". We take care not to copy the    *
 *            nul byte that terminates the "patch" string.                   *
 ****************************************************************************/

    ps = patch[n]                    ; // The patch address
    if (ps EQ HNULL) goto step3      ; // If we do not have a patch, continue

    for(i = 0, pd = (char*)snd ;       // Scan the characters until we find a
        i < lg && *pd != '#'   ;       // '#' or we reach the buffer end, so
        i++, pd++               )  ; // "lg" characters
```

```
    if (i < lg )                        // If we found a '#', copy the "patch"
      Strncpy(pd        ,               // at location "pd" in the buffer
              ps        ,               // We do not copy the nul terminator
              Strlen(ps)  )           ; // of "patch"


/*************************************************************************
 * Step 3 : Send using serial DEV0, so IOD "asy_iod0", the "lg" characters   *
 * ------   that are in the buffer pointed by "buf_snd[0]". We call the      *
 *          "myio_write" subroutine that sends the buffer and unschedules us *
 *          until the effective transmission of the last byte.               *
 *************************************************************************/

    step3                             : // Send the buffer

    myio_write (asy_iod0 ,              // I/O descriptor for ASY\DEV0
                snd      ,              // Address of buffer
                lg         )          ; // Number of bytes to be sent


/*************************************************************************
 * Step 4 : Absolutely all the menus are displaying time. We could wait for  *
 * ------   the next time-out event but we may have the XX:XX:XX displayed   *
 *          up to one second. So we do not wait for the time-out and we call *
 *          the "ptim" procedure that just redraws the time.                 *
 *************************************************************************/

    ptim(0)                           ; // Read time, convert in a string and
                                        // send it on the serial line


/*************************************************************************
 * Step 5 : Update the "page" global variable that is the number of the page *
 * ------   that is currently displayed. This number is needed at step 5 of  *
 *          "loop_app_tsk", it is the state variable of the menu automaton.  *
 *          We also reset the count of characters of the "bufchr" input      *
 *          buffer to 0.                                                     *
 *************************************************************************/

    page = n                          ; // Note the currently displayed menu
                                        // This is the UI state variable
    lgchr = 0                         ; // Force to empty the "bufchr" buffer

    return 0                          ; // No error
  }


/*  Procedure pdtc ----------------------------------------------------------

    Purpose : Display the touch coordinates.
*/

static int pdtc(void)
  {

    unsigned char   *snd              ; // Address of buffer to be sent
    char            *pt               ; // Write pointer

/*************************************************************************
 * Step 1 : Put in the transmit buffer the characters that have to be sent   *
 * ------   in order to print the touch coordinates. We:                     *
 *          - Copy into the transmit buffer the ANSI escape sequence that    *
 *            will save current ANSI terminal pointer position. Then we      *
 *            increment the "number of bytes".                               *
 *          - Copy into the transmit buffer the ANSI escape sequence that    *
 *            will move the console cursor the location where the first      *
 *            character of the touch coordinates will be printed. Then we    *
 *            increment the "number of bytes".                               *
```

```
 *          - Write into the config buffer the touch coordinates. Then we    *
 *            increment the "number of bytes".                               *
 *          - Copy into the transmit buffer the ANSI escape sequence that    *
 *            will restore terminal pointer position. Then we increment the  *
 *            "number of bytes".                                             *
 *          The cursor "normal" position is on the right of the last line    *
 *          of the current menu page "Enter command". But the line number    *
 *          where that prompt lays is not the same one for all the menu      *
 *          pages. So whatever the current menu page is, we request the      *
 *          terminal to save the cursor position and to restore it after     *
 *          current time has been displayed.                                 *
 ***************************************************************************/

    snd = buf_snd[0]                ; // Buffer to be sent address
    pt  = (char*) snd               ; // Init write pointer = beg of buffer

    strcpy(pt, loc_save)            ; // Copy into the transmit buffer
                                      // the ANSI escape sequence
    pt += sizeof(loc_save) - 1      ; // Increment the number of bytes

    strcpy(pt, loc_t_coord)         ; // Move cursor to line = 16 column = 53
    pt += sizeof(loc_t_coord) - 1   ; // Increment the number of bytes

    memcpy(pt, i2c_rx, 3)           ; // Copy touch coordinates
    pt += 3                         ; // Increment the number of bytes

    strcpy(pt, loc_restore)         ; // Copy into the transmit buffer
                                      // the ANSI escape sequence
    pt += sizeof(loc_restore) - 1   ; // Increment the number of bytes

/***************************************************************************
 * Step 2 : Send using serial DEV0, so IOD "asy_iod0". We call the          *
 * ------   "myio_write" procedure that sends a REQ_WRITE request event to  *
 *          the AUT_ASY VMIO automaton. Then we unschedule until the        *
 *          RESP_WRITE response event is received.                          *
 ***************************************************************************/

    myio_write (asy_iod0       ,      // I/O descriptor for ASY\DEV0
                snd            ,      // Address of buffer
                pt - (char*)snd)    ; // Number of bytes to be sent

    return 0                        ; // No error
  }


/*  Procedure ptim ---------------------------------------------------------

    Purpose : UI treatment - Read time, convert it in a ASCII string and then
              prints (sends) it on the ASY0 serial line.
*/

static int ptim(int n)
  {
    unsigned char   *snd            ; // Address of buffer to be sent
    int             d, m, y         ; // Day, Month, Year
    int             h, mi, s, ms    ; // Hour, minutes, seconds, milliseconds
    char            *pt             ; // Write pointer
    int             lg              ; // Number of bytes to be sent

/***************************************************************************
 * Step 1 : We first retrieve date and time since boot. We call the         *
 * ------   "tim_get" procedure to obtain date and time.                    *
 ***************************************************************************/

    tim_get(&d  ,                   // day
            &m  ,                   // Month
            &y  ,                   // Year
            &h  ,                   // Hour (0..23)
```

```
            &mi ,                       // Minutes (0..59)
            &s  ,                       // Seconds (0..59)
            &ms   )                   ; // Milliseconds (0..999)


/*************************************************************************
 * Step 2 : Put in the transmit buffer the characters that have to be sent   *
 * ------   in order to print elapsed time since boot. We are using "lg" as  *
 *          the current number of bytes inside the transmit buffer. We:      *
 *          - Copy into the transmit buffer the ANSI escape sequence that    *
 *            will save current ANSI terminal pointer position. Then we      *
 *            increment the "lg" number of bytes.                            *
 *          - Copy into the transmit buffer the ANSI escape sequence that    *
 *            will move the console cursor the location where the first      *
 *            character of the date will be printed. Then we increment the   *
 *            "lg" number of bytes.                                          *
 *          - Write into the config buffer the "HH:MM:SS" string representing*
 *            the elapsed time. Then we increment the "lg" number of bytes.  *
 *          - Copy into the transmit buffer the ANSI escape sequence that    *
 *            will restore terminal pointer position. Then we increment the  *
 *            "lg" number of bytes.                                          *
 *          The cursor "normal" position is on the right of the last line    *
 *          of the current menu page "Enter command". But the line number    *
 *          where that prompt lays is not the same one for all the menu      *
 *          pages. So whatever the current menu page is, we request the      *
 *          terminal to save the cursor position and to restore it after     *
 *          current time has been displayed.                                 *
 *************************************************************************/

   snd = buf_snd[0]                 ; // Buffer to be sent address
   pt  = (char*) snd                ; // Init write pointer = beg of buffer
   lg  = 0                          ; // Init number of writtent bytes

   strcpy(pt, loc_save)             ; // Copy into the transmit buffer
                                      // the ANSI escape sequence
   pt += sizeof(loc_save) - 1       ; // Increment the number of bytes
   lg += sizeof(loc_save) - 1       ; // Increment the number of bytes

   strcpy(pt, loc_time)             ; // Move cursor to line = 3 column = 55
   pt += sizeof(loc_time) - 1       ; // Increment the number of bytes
   lg += sizeof(loc_time) - 1       ; // Increment the number of bytes

   hsprintf(pt               ,        // Put in the transmit buffer
            "%02d:%02d:%02d" ,        // 8 characters HH:MM:SS
            h, mi, s              ) ; //
   pt += 8                          ; // Update the write pointer
   lg += 8                          ; // Update the number of bytes

   strcpy(pt, loc_restore)          ; // Copy into the transmit buffer
                                      // the ANSI escape sequence
   pt += sizeof(loc_restore) - 1    ; // Increment the number of bytes
   lg += sizeof(loc_restore) - 1    ; // Increment the number of bytes


/*************************************************************************
 * Step 2 : Send using serial DEV0, so IOD "asy_iod0", the "lg" characters   *
 * ------   that are in the buffer pointed by "buf_snd[0]". We call the      *
 *          "myio_write" procedure that sends a REQ_WRITE request event to   *
 *          the AUT_ASY VMIO automaton. Then we unschedule until the         *
 *          RESP_WRITE response event is received.                           *
 *************************************************************************/

   myio_write (asy_iod0  ,           // I/O descriptor for ASY\DEV0
               snd       ,           // Address of buffer
               lg         )        ; // Number of bytes to be sent

   return 0                         ; // No error
  }
```

```
/*  Procedure pgrp ------------------------------------------------------------

    Purpose : UI treatment - Prints (sends) on the serial line the group of
              string statuses. The input parameter "n" value is one the
              GMAIN, GASY1, GASY2, GGPIO, GBUT or VINPUT defines.
*/

static int pgrp(int n)
  {
    unsigned char   *snd             ; // Address of buffer to be sent
    int             i, ind           ; // Loop counter
    int             sz               ; // Length of string item
    char            *pt              ; // Write pointer
    int             lg               ; // Number of bytes to be sent
    const char      *const *tab      ; // List of string addresses
    const char      *str             ; // String address
    char            buf[20]          ; // Decimal integer string
    int             val              ; // Copy of "cfg[ind]"
    int             cnt              ; // Count of "cfg[]" values
    int             col              ; // Horizontal tabulation
    int             len              ; // Item width in characters

/*   GMAIN   GCONF    GASY1   GASY2   GGPIO   GBUT   GRCU
     -----   -------  -----   -----   -----   -----  -----  */
static const int tfirst[7] =
{    VRED,   VASY1PT, VBAUD1, VBAUD2, VPTIM , VMON1, VINPUT };
static const int tcnt  [7] =
{    7   ,   3      , 9     , 9     , 2     , 9    , 6      };
static const int tcol  [7] =
{    0   ,   0      , 0     , 0     , 0     , 0    , 1      };
static const int tlen  [7] =
{    7   ,   7      , 7     , 7     , 7     , 7    , 16     };

/*****************************************************************************
 * Step 1 : Initialization of local variables:                               *
 * ------   - We will use "ind" as an index in the "cfg[]" table of values.  *
 *            The "tfirst[]" const table gives us the index of the first     *
 *            "cfg[]" integer to be displayed. For the MAIN menu we have     *
 *            "n = 0" and "ind" is set to VRED. With the MASY1 menu we have  *
 *            "n = 1" and "ind" is set to VBAUD. And with the MGPIO menu we  *
 *            have "n = 2" and "ind" is set to VPTIM.                        *
 *          - We set "cnt" to the number of "cfg[]" values to be displayed.  *
 *            We set "cnt" to "tcnt[n]" constant table value.                *
 *          - We use "pt" as a write pointer in the "buf_snd" transmit       *
 *            buffer. We initialize "pt" to the "buf_snd[0]" start address.  *
 *          - We use "lg" as a count of written bytes in "buf_snd", we       *
 *            initialize to 0 this count.                                    *
 *****************************************************************************/

    ind   = tfirst[n]                ; // Start index in the "cfg[]" array
    cnt   = tcnt[n]                  ; // Count of "cfg[]" values to be used
    col   = tcol[n]                  ; // Horizontal tabulation
    len   = tlen[n]                  ; // Item length
    snd   = buf_snd[0]               ; // Buffer to be sent address
    pt    = (char*) snd              ; // Init write pointer = beg of buffer
    lg    = 0                        ; // Init number of written bytes


/*****************************************************************************
 * Step 2 : Put in the transmit buffer the characters that have to be sent   *
 * ------   in order to print the "cnt" menu statuses. We are using "pt" as  *
 *          a current write position in the transmit buffer and "lg" is the  *
 *          current number of bytes inside the transmit buffer. We start by  *
 *          saving the current terminal pointer position (input):            *
 *          - Copy into the transmit buffer the ANSI escape sequence that    *
 *            will save current pointer position. Then we increment the "pt" *
```

```
*           current write address and the "lg" number of bytes.            *
*         Then we loop on the "cnt" values. For each value:                *
*         - Copy into the transmit buffer the ASCII escape sequence that   *
*           will move the console cursor to the location where the first   *
*           status character will be drawn. We have a static array of      *
*           address strings for that (loc_cfg[9]). Then we increment the   *
*           "pt" current write address and the "lg" number of bytes.       *
*         - Copy in the transmit buffer the string that is associated to   *
*           "cfg[ind]". We have a "str_cfg[]" list of string list that     *
*           gives addresses of arrays of string addresses. So the value    *
*           of "str_cfg[ind]" is the address of the array of string        *
*           addresses for variable "ind". Then the "cfg[ind]" value is     *
*           used as an index in that table and so " str_cfg[ind][cfg[ind]]"*
*           is the address of the string to be displayed.                  *
*         - All the strings do not have exactly the same number of         *
*           characters. So we increment the "pt" write pointer and the     *
*           "lg" count of written bytes by the item size string.           *
*         Then we restore the terminal pointer state:                      *
*         - Copy into the transmit buffer the ANSI escape sequence that    *
*           will restore pointer position. Then we increment the "pt"      *
*           current write address and the "lg" number of bytes.            *
**************************************************************************/

   strcpy(pt, loc_save)             ; // Copy into the transmit buffer
                                      // the ANSI escape sequence
   pt += sizeof(loc_save) - 1       ; // Increment the write pointer
   lg += sizeof(loc_save) - 1       ; // Increment the number of bytes

   for (i = 0 ; i < cnt ; i++, ind++) // Loop on the "cnt" menu statuses
     {                                //
       str = loc_cfg[i][col]        ; // String escape sequence address
       sz  = strlen(str)            ; // Copy into the transmit buffer
       strcpy(pt,str)               ; // the ANSI escape sequence
       pt += sz                     ; // Increment the write pointer
       lg += sz                     ; // Increment the number of bytes

       tab = str_cfg[ind]           ; // List of string addresses
       val = cfg    [ind]           ; // "val" is the index in "tab"

       if (tab NE HNULL)              // If we do have a list of strings
         str = tab[val]             ; // Select string "val" from the list
       else                           // Otherwise, we convert
         BEGIN                        // the "cfg[ind]" value
           hsprintf(buf,"%d",val)   ; // in an integer decimal string
           str = buf                ; // and "str" points to this string
         END_IF                       //

       memcpy(pt,                     // Fill "len" characters with "-"
              "-----------------",    // wich makes up the
              len                  ); // background
       strcpy(pt,str)               ; // The displayed item string
       pt += len                    ; // Increment the write pointer
       lg += len                    ; // Increment the number of bytes
     }

   strcpy(pt, loc_restore)          ; // Copy into the transmit buffer
                                      // the ANSI escape sequence
   pt += sizeof(loc_restore) - 1    ; // Increment the write pointer
   lg += sizeof(loc_restore) - 1    ; // Increment the number of bytes


/**************************************************************************
 * Step 3 : Send using serial DEV0, so IOD "asy_iod0", the "lg" characters  *
 * ------   that are in the buffer pointed by "buf_snd[0]". We call the     *
 *          "myio_write" procedure that sends a REQ_WRITE request event to  *
 *          the AUT_ASY VMIO automaton. Then we unschedule until the        *
 *          RESP_WRITE response event is received.                          *
 **************************************************************************/
```

```
    myio_write (asy_iod0 ,            // I/O descriptor for ASY\DEV0
                snd      ,            // Address of buffer
                lg        )         ; // Number of bytes to be sent

    return 0                        ; // No error
  }


/*  Procedure pcmd ----------------------------------------------------------

    Purpose : UI treatment - Prints (sends) on the serial line the last command
              received, so one "c" character.
*/

static int pcmd(int c)
  {
    unsigned char   *snd            ; // Address of buffer to be sent
    const char      *str            ; // Address of the ANSI escape sequence
    int             lg              ; // Number of bytes to be sent

/*************************************************************************
 * Step 1 : Put in the transmit buffer the characters that have to be sent   *
 * ------   in order to echo the 'c' command character. We are using "lg"    *
 *          as the current number of bytes inside the transmit buffer. We:   *
 *          - Copy into the transmit buffer the ANSI escape sequence that    *
 *            will move pointer to the right place. This place depends on    *
 *            the current page. We have a static array of address strings    *
 *            for that purpose: loc_cmd[].                                   *
 *          - Copy in the transmit buffer the character to be echoed.        *
 *************************************************************************/

    snd = buf_snd[0]                ; // Address of buffer to be sent
    str = loc_cmd[page]             ; // Address of the ANSI escape sequence
    lg  = strlen( str )             ; // Number of bytes in the buffer

    strcpy((char*)snd, str)         ; // ANSI Escape Sequence
                                      // Move cursor to line, column

    snd[lg] = (char)c               ; // Put the echo in transmit buffer

    lg += 1                         ; // Update number of bytes in the buffer


/*************************************************************************
 * Step 2 : Send using serial DEV0, so IOD "asy_iod0", the "lg" characters   *
 * ------   that are in the buffer pointed by "buf_snd[0]". We call the      *
 *          "myio_write" procedure that sends a REQ_WRITE request event to   *
 *          the AUT_ASY VMIO automaton. Then we unschedule until the         *
 *          RESP_WRITE response event is received.                           *
 *************************************************************************/

    myio_write (asy_iod0  ,           // I/O descriptor for ASY\DEV0
                snd       ,           // Address of buffer
                lg         )        ; // Number of bytes to be sent

    return 0                        ; // No error
  }


/*  Procedure pchr ----------------------------------------------------------

    Purpose : UI treatment - Prints (sends) on the serial line one ASCII
              character (ASCII code is c).
*/

static int pchr(int c)
  {
```

```
    unsigned char   *snd             ; // Address of buffer to be sent
    int             lg               ; // Number of bytes to be sent


/*************************************************************************
 * Step 1 : Send using serial DEV0, so IOD "asy_iod0", the "c" character.    *
 * ------   We copy it in "buf_snd[0] then we call the "myio_write"          *
 *          procedure that sends a REQ_WRITE request event to the AUT_ASY    *
 *          VMIO automaton. Then we unschedule until the RESP_WRITE response *
 *          event is received.                                               *
 *          ---------------------------------------------------------------- *
 *          With a BACKSPACE ('c' is 8) we have a special treatment:         *
 *          - We send a BACKSPACE,                                           *
 *          - We send a SPACE    ,                                           *
 *          - We send a BACKSPACE,                                           *
 *************************************************************************/

    snd    = buf_snd[0]              ; // Address of buffer to be sent

    if (c EQ 8)                        // If we have a BACKSPACE to echo
      {                                //
        if (lgchr == 0) return 0     ; // Do nothing if buffer is empty
        snd[0] = snd[2] = 8          ; // ASCII code for BACKSPACE
        snd[1] =          32         ; // ASCII code for SPACE
        lg     =          3          ; // We send 3 bytes
      }
    else                               // If we have a "normal" ASCII code
      {                                //
        if (lgchr >= LGCHR) return 0 ; // Do noting if buffer is full
        snd[0] = c                   ; // Copy the charcater to be sent
        lg     = 1                   ; // We send one byte
      }

    myio_write (asy_iod0  ,            // I/O descriptor for ASY\DEV0
                snd       ,            // Address of buffer
                lg         )         ; // Number of bytes to be sent

    return 0                         ; // No error
  }


/*  Procedure pcap ------------------------------------------------------------

    Purpose : UI treatment - Prints the RCU symbols captured values
*/

static int pcap(int n)
  {
    unsigned char   *snd             ; // Address of buffer to be sent
    int             sz               ; // Length of string item
    char            *pt              ; // Write pointer
    int             lg               ; // Number of bytes to be sent
    unsigned short  *ad              ; // Address of measures
    int             nb               ; // Number of measures
    int             y, i , k         ; // Loop counters

/*************************************************************************
 * Step 1 : Initialization of local variables:                           *
 * ------   - We use "pt" as a write pointer in the "buf_snd" transmit   *
 *            buffer. We initialize "pt" to the "buf_snd[0]" start address. *
 *          - We use "lg" as a count of written bytes in "buf_snd", we   *
 *            initialize to 0 this count.                                *
 *          - We retreive from the "adresse" field of the incoming       *
 *            DRV_KEY_PRESS event the start address of the measure's table. *
 *            As we are a task, we have in the global variable "task_evt" a *
 *            copy of the just received event.                           *
 *          - We retreive from the "longueur" field of the incoming event *
 *            the count of measures (2 x symbol count).                  *
```

```
 ***************************************************************************/

    snd = buf_snd[0]                 ; // Buffer to be sent address
    pt  = (char*) snd                ; // Init write pointer = beg of buffer
    lg  = 0                          ; // Init number of written bytes
    ad  = (USHORT*) task_evt.adresse ; // Measure's table start address
    nb  = task_evt.longueur          ; // Count of "unsigned short's"


/***************************************************************************
 * Step 2 : Fill the transmit buffer "buf_snd[0]". We have 15 lines we can  *
 * ------   use and we can display 10 USHORTS on each line, so a maximum    *
 *          of 15 x 10 = 150 decimal number of 5 characters each may be     *
 *          drawn. We have two imbricated loops:                            *
 *          - First loop in on lines, from 5 to 19 (so 15 lines). We store  *
 *            the ANSI escape sequence that moves the cursor on line "y",   *
 *            column 5.                                                     *
 *          - Second loop is on 10 USHORT's. At each step we put 6 ASCII    *
 *            characters, a right justified 5 character decimal number      *
 *            followed by a "space" character.                              *
 *          We stop the loop when the "nb" numbers have been processed. Then *
 *          we add the ANSI escape sequence that moves the cursor on line 22 *
 *          column 43.                                                      *
 ***************************************************************************/

    for(y = 5, k = 0; y LE 19; y++)    /* We loop on lines 5 to 19 of the    */
      BEGIN                            /* terminal. We store the ANSI escape */
        hsprintf(pt,"\x1B[%d;5H",y)  ; /* sequence that moves the terminal   */
        sz = strlen(pt)              ; /* cursor on line "y" column 5. Then  */
        lg += sz                     ; /* increment the "lg" count of        */
        pt += sz                     ; /* characters and the "pt" write      */
                                       /* address in "buf_snd[0]"            */
                                       /*                                    */
        for(i = 0; i LT 10; i++, k++)  /* Loop on the 10 USHORT's, so 5      */
          BEGIN                        /* (ON time, SYMB time) that we want  */
                                       /* to display on the current line "y" */
            if (k GE nb) goto end    ; /* If all the USHORT's have been      */
                                       /* treated, we exit from this loop    */
            hsprintf(pt,"%5d ",ad[k]); /* Store 5 decimal digits and one     */
            lg += 6                  ; /* "space" at address "pt", then      */
            pt += 6                  ; /* increment by 6 the number of       */
          END_FOR                      /* written bytes and also the current */
                                       /* "pt" write address                 */
      END_FOR                          /*                                    */
                                       /*                                    */
    end                              : /* We add the ANSI escape sequence    */
    strcpy(pt,"\x1B[22;43H")         ; /* that moves the terminal cursor     */
    lg += 8                          ; /* on terminal line 22, column 43     */


/***************************************************************************
 * Step 3 : Send using serial DEV0, so IOD "asy_iod0", the "lg" characters  *
 * ------   that are in the buffer pointed by "buf_snd[0]". We call the     *
 *          "myio_write" procedure that sends a REQ_WRITE request event to  *
 *          the AUT_ASY VMIO automaton. Then we unschedule until the        *
 *          RESP_WRITE response event is received.                          *
 ***************************************************************************/

    myio_write (asy_iod0 ,             // I/O descriptor for ASY\DEV0
                snd      ,             // Address of buffer
                lg        )          ; // Number of bytes to be sent

    return 0                         ; // No error
  }


/*  Procedure perr ---------------------------------------------------------
```

```
    Purpose : UI treatment - Prints (sends) on the serial line the decimal
              value or "errcode".
*/
static int perr(int c)
  {
    unsigned char   *snd             ; // Address of buffer to be sent
    char            *pt              ; // Write pointer in "snd"
    int             lg               ; // Number of bytes to be sent

    snd = buf_snd[0]                 ; // Address of buffer to be sent
    pt  = (char*) snd                ; // Write pointer in the buffer

    strcpy(pt, loc_error)            ; // ANSI Escape Sequence
                                       // Move cursor to line, column
    lg  = strlen(pt)                 ; // Count of written bytes
    pt += lg                         ; // Update the write pointer

    hsprintf(pt,"Error code %d",       // Add the error message in
             errcode            )    ; // the buffer

    lg += strlen(pt)                 ; // Update the count of written bytes

    myio_write (asy_iod0  ,            // I/O descriptor for ASY\DEV0
                snd       ,            // Address of buffer
                lg         )         ; // Number of bytes to be sent

    return 0                         ; // No error
  }


/*  Procedure tcfg ----------------------------------------------------------

    Purpose : UI treatment - Toggles one "cfg[]" value (0->, 1->0)
*/

static int tcfg(int n)
  {
    cfg[n] ^= 1                      ; // Toggles "cfg[n]" value

    return 0                         ; // No error
  }


/*  Procedure tinp ----------------------------------------------------------

    Purpose : UI treatment - Toggles VINPUT and VNUMRCU
*/

static int tinp(int n)
  {
    if (n EQ VINPUT)
      {
        cfg[VINPUT] ^= 1             ; // 0->1, 1->0
        cfg[VNUMRCU] = vnumrcu[cfg[VINPUT]] ; // 0:0, 1:3
      }
    else
      {
        cfg[VNUMRCU] ++              ; // 0->1, 1->0, 2->3, 3->4, 4->5, 5->6

        if (cfg[VNUMRCU] EQ NBRCU_IR)
          cfg[VNUMRCU] = 0           ;

        if ( cfg[VNUMRCU] EQ
             (NBRCU_IR+NBRCU_UHF) )
          cfg[VNUMRCU] = NBRCU_IR    ;
      }

    return 0                         ; // No error
```

```
  }


/*  Procedure tssy ----------------------------------------------------------

    Purpose : UI treatment - Toggles VONMIN/VONMAX/VSTMIN/VSTMAX
*/

static int tssy(int n)
  {
    int             swpos           ; // Current switch position
    int             numrcu          ; // RCU order number in database
    char            name[20]        ; // Current RCU name
    int             otnom           ; // ON time nominal in usec
    int             stnom           ; // SYMB time nominal in usec
    int             ret             ; // Error code

/*************************************************************************
 * Step 1 : Retreive the RCU database entry number "numrcu" for the RCU that *
 * ------   is currently associated with receiver "rcurcv_id". The "numrcu"  *
 *          value and the "swpos" switch positions are modified when the     *
 *          "change_rcu" procedure is called.                                *
 *************************************************************************/

    get_rcu    (rcurcv_id    ,         // The receiver device identifier
                &swpos       ,         // The current switch position
                &numrcu      ,         // The RCU database order number
                name         ,         // The corresponding RCU name
                &ret          )      ; // Error code

/*************************************************************************
 * Step 2 : Retreive the timings in useconds for the first symbol (0) for    *
 * ------   that RCU. By convention, the first received message symbol is    *
 *          always available from entry 0 of the RCU symbol list.            *
 *************************************************************************/

    get_rcu_sym(numrcu             ,   // The RCU database order number
                0                  ,   // The RCU symbol order number
                (int*)&cfg[VONMIN] ,   // ON   time  minimal in usec
                (int*)&cfg[VONMAX] ,   // ON   time  maximal in usec
                (int*)&cfg[VSTMIN] ,   // SYMB time  minimal in usec
                (int*)&cfg[VSTMAX] ,   // SYMB time  maximal in usec
                &otnom             ,   // ON   time  nominal in usec
                &stnom             ,   // SYMB time  nominal in usec
                &ret                ); // Error code

    return 0                         ; // No error
  }


/*  Procedure tchr ----------------------------------------------------------

    Purpose : UI treatment - Add a new character to the input line
*/

static int tchr(int n)
  {
    if (n == 8)                        // If we have a BACKSPACE
      {
        if ( lgchr <= 0) return 0    ;
        lgchr --                     ;
      }
    else                               // If we have a "normal" ASCII code
      {
        if ( lgchr >= LGCHR) return 0;
        bufchr[lgchr++] = n          ;
      }
```

```
    return 0                        ; // No error
  }


/*  Procedure scfg ------------------------------------------------------------

    Purpose : UI treatment - Sets one "cfg[]" value to "opt_val -1".
*/

static int scfg(int n)
  {
/*****************************************************************************
 * Step 1 : The last user choice is an integer number ranging from 1 to 9    *
 * ------   and its value is available from the "opt_val" global variable.   *
 *          We store that choice in the indicated "cfg[n]" global array. We  *
 *          substract 1 because the value will be used as C table indexes,   *
 *          that starts with 0. The tables that will take "cfg[n]" as        *
 *          indexes are "str_cfg[]" and "str_tag[]".                         *
 *****************************************************************************/

    cfg[n] = opt_val - 1            ; // Set "cfg[n]" to the user choice - 1

    return 0                        ; // No error
  }


/*  Procedure sval ------------------------------------------------------------

    Purpose : UI treatment - Calls "myio_setval", using "cfg[n]" as a taglist
              selector.
*/

static int sval(int n)
  {
    const char *const      *tag    ; // Address of the ANSI escape sequence
    int                    ret     ; // Error code

/*****************************************************************************
 * Step 1 : We are going to call "myio_setval" with a const string as second *
 * ------   parameter. For each variable "n" ("n" is an index in "cfg[]")    *
 *          we have a list of strings, that is in fact an array of string    *
 *          addresses. For example, for N = VBAUD, the corresponding list of *
 *          strings is "tag_baud". We have an array "str_tag[]" that         *
 *          contains all the addresses of the string list, so the            *
 *          "str_tag[VBAUD]" value is the address of "str_tag[VBAUD]". Here  *
 *          we set the "tag" local variable to "str_tag[n]", so to           *
 *          "tag_baud" if "n" value is VBAUD.                                *
 *****************************************************************************/

    tag = str_tag[n]                ; // Address of array of C string addresses


/*****************************************************************************
 * Step 2 : Now we call the "myio_setval" procedure, that calls 'drv_setval" *
 * -------  - The first parameter is the device IOD or the sub-device IOD.   *
 *            At step 3 of "loop_app_tsk" we had stored in "iods[n]" the IOD *
 *            value to be used here. So "iods[n]" is our first parameter.    *
 *          - The second parameter is the C string of index "cfg[n]" in the  *
 *            "tag" list of strings, so "tag[ cfg[n] ]" is the address of    *
 *            the const string to be provided as a second parameter.         *
 *          We have 3 variables that do not imply any call to "drv_setval",  *
 *          VASY1 (3), VDAC (4) and VSPEED (13). Indeed the present          *
 *          procedure must not be called with each of those values. We have  *
 *          HNULL values in "str_tag[VASY1]", in "str_tag[VDAC]" and         *
 *          "str_tag[VSPEED]". In case of any error in the const tables, we  *
 *          here check that "tag" is not HNULL (if yes, it's a bug...).      *
 *****************************************************************************/
```

```
    if (tag != HNULL)                // "tag" must not be HNULL
      ret = myio_setval(iods[n],     // IOD to be used with "drv_setval"
                        tag[cfg[n]]); // C string to be used with "drv_setval"
    else
      vhalt()                      ; // Here we have a bug ...

    return ret                     ; // No error
  }


/*  Procedure sled ------------------------------------------------------------

    Purpose : UI treatment - Starts/Stops LED blinking.
*/

static int sled(int n)
  {
    int            ret             ; // Error code

/*****************************************************************************
 * Step 1 : The LED driver has a stack of blinking patterns for each LED.    *
 * ------   Before pushing a new blinking pattern in the stack, we request   *
 *          to empty the blink pattern stack to the led driver. To do this,  *
 *          we send a REQ_SETVAL "CMD=POPALL" event to the LED driver, so we *
 *          call the "myio_setval" procedure that send the REQ_SETVAL event  *
 *          to the VMIO AUT_LED automaton. Then we ask to be unscheduled     *
 *          until the response event RESP_SETVAL is received.                *
 *****************************************************************************/

    myio_setval(iods[n],               // Always call "drv_setval" with
                led_tagpop)          ; // "CMD=POPALL"


/*****************************************************************************
 * Step 2 : The blink pattern stack for our LED is now empty. We push a new  *
 * ------   blinking pattern: we call "myio_setval" that sends to the VMIO   *
 *          AUT_LED a REQ_SETVAL that carries the new blinking pattern, that *
 *          has to be ended by "CMD=PUSH". And then we ask to be unscheduled *
 *          until the RESP_SETVAL response event, replied by AUT_LED, has    *
 *          been received.                                                   *
 *          The "myio_setval" procedure needs two input parameters:          *
 *          - The first parameter is the device or sub-channel IOD value. In *
 *            our case, the "loop_app_tsk" has stored in "iods[]" the IOD    *
 *            value to be used for each "n" value. So we use here "iods[n]"  *
 *            as first "myio_setval" parameter value.                        *
 *          - The second parameter is a character string, the TAG list. In   *
 *            our case, we have const data (C strings) for all the possible  *
 *            tag lists. We have also string lists, that are just lists of   *
 *            string addresses. For example the VBAUD parameter may have 5   *
 *            possible values (CFG_9600 0, CFG_19200 1, CFG_38400 2,         *
 *            CFG_57600 3 and CFG_115200 4). So we have 5 corresponding      *
 *            taglist strings and the "tag_baud[5]" contains the 5 addresses *
 *            of those 5 strings. But also, we have one more array           *
 *            "str_tag[]" that contains all the addresses of the list        *
 *            addresses, for example "str_tag[5]" contains the address of    *
 *            "tag_baud". Therefore, "str_tag[n]" is the address of the      *
 *            string list for variable "n". Then, the value of variable "n"  *
 *            is "cfg[n]" and therefore "str_tag[n][ cfg[n] ]" is the C      *
 *            string to be used as tag list, and the second parameter to be  *
 *            given to "myio_setval".                                        *
 *****************************************************************************/

    ret = myio_setval(iods[n],         // Push the pattern that corresponds
                str_tag[n][cfg[n]])  ; // to "cfg[n] = 0" or "cfg[n] = 1"

    return ret                     ; // No error
  }
```

```
/*  Procedure sint ------------------------------------------------------------

    Purpose : UI treatment - Converts the decimal string from "bufstr" to a
              binary integer value.
*/

static int sint(int n)
  {
    if (lgchr == 0)                   // If the buffer is empty we
      bufchr[lgchr++] = '0'          ; // put a '0' string (one byte)

    bufchr[lgchr] = 0                ; // Add the nul end of C string

    hsscanf(bufchr,"%d",&cfg[n])     ; // Convert decimal string to integer

    return 0                         ; // No error
  }


/*  Procedure rasy ------------------------------------------------------------

    Purpose : UI treatment - Start or stops the continuous sending on data
              using ASY1/ASY2. We send a SND_START or SND_STOP event according
              to the value of "cfg[ VASY1/VASY2 ]". We use logical way
              VLASY1/VLASY2 of the AUT_USND FSM. The "n" input parameter value
              is VASY1 or VASY2.
*/

static int rasy(int n)
  {
    int                vl            ; // AUT_USND logical way
    unsigned char     *snd           ; // Buffer to be sent

/*************************************************************************
 * Step 1 : The value of "n" is VASY1 or VASY2. We select the "vl" logical   *
 * ------   way of AUT_USND and the iod to be used, we use VLASY1 with ASY1  *
 *          ("n" is VASY1) or VLASY2 with ASY2 (n is VASY2).                 *
 *************************************************************************/

    vl  = (n == VASY1) ? VLASY1 :      // Logical way of AUT_UNSD is VLASY1
                         VLASY2      ; // or VLASY2

    snd = buf_snd[(n == VASY1) ? 1:2]; // Buffer is "buf_snd[1]" with ASY1
                                       // and "buf_snd[2]" with ASY2

/*************************************************************************
 * Step 2 : According the value of "cfg[n]", send a SND_START event or a     *
 * ------   SND_STOP event to the AUT_USND FSM. Then wait (unschedule) for   *
 *          the response event of AUT_USND (R_SND_START or R_SND_STOP). We   *
 *          use "req_snd_start" or "req_snd_stop". Apart from the code event *
 *          SND_START and the logical way number "vl, the start event also   *
 *          conveys 3 parameters that are:                                   *
 *          - The IOD to be used for transmission, here "iods[n]".           *
 *          - The address of the buffer to be cyclically send, here          *
 *            "buf_snd[1].                                                   *
 *          - The count of bytes in the buffer, here 256 bytes.              *
 *************************************************************************/

    if ( cfg[n] )                      // If new state is 1, send a
      req_snd_start(vl      ,          // SND_START event to LW 0/1 of AUT_USND
                    iods[n] ,          // Iod for the ASY1/ASY2
                    snd     ,          // Data buffer for the ASY1/ASY2
                    256      )       ; // Number of bytes in buffer
    else                               // Else, new state is 0, send a
      req_snd_stop(vl)                ; // SND_STOP event to LW 0/1 of AUT_USND

    return 0                         ; // No error
```

```
  }


/*  Procedure rdac ----------------------------------------------------------

    Purpose : UI treatment - Start or stops the continuous sending on data
              using DAC. We send a SND_START or SND_STOP event according to
              the value of "cfg[VDAC]". We use logical way VLDAC of the
              AUT_USND FSM.
*/

static int rdac(int n)
  {
/*****************************************************************************
 * Step 1 : According the value of "cfg[VDAC]", send a SND_START event or a  *
 * ------   SND_STOP event to the AUT_USND FSM. Then wait (unschedule) for   *
 *          the response event of AUT_USND (R_SND_START or R_SND_STOP). We   *
 *          use "req_snd_start" or "req_snd_stop". Apart from the code event *
 *          SND_START and the logical way number VLDAC, the start event      *
 *          also conveys 3 parameters that are:                              *
 *          - The IOD to be used for transmission, here "dac_iod".           *
 *          - The address of the buffer to be cyclically send, here          *
 *            "buf_aud".                                                     *
 *          - The count of bytes in the buffer, here 200 bytes.              *
 *****************************************************************************/

    if ( cfg[VDAC] )                    // If new state is 1, send a
      req_snd_start(VLDAC  ,            // SND_START event to LW 2 of AUT_USND
                    dac_iod,            // Iod for the DAC
                    buf_aud,            // Data buffer for the DAC
                    200     )         ; // Number of bytes in buffer
    else                                // Else, new state is 0, send a
      req_snd_stop(VLDAC)               ; // SND_STOP event to LW 2 of AUT_USND

    return 0                            ; // No error
  }


/*  Procedure rrcu ----------------------------------------------------------

    Purpose : UI treatment - Start or stops the period sending of one RCU
              message. We send a SND_START or SND_STOP event according to
              the value of "cfg[VUHFRCU]". We use logical way VLSNDRCU of the
              AUT_USND FSM.
*/

static int rrcu(int n)
  {
    if ( cfg[n] )                       // If new state is 1, send a
      req_snd_start(VLSNDRCU   ,        // SND_START event to LW 3 of AUT_USND
                    iods[n]    ,        // Iod no for the RCU
                    buf_rcu    ,        // Data buffer for the RCU transmitter
                    0x00030004  )     ; // 4 symbols in the buffer
                                        // 3 consecutive repeats
    else                                // Else, new state is 0, send a
      req_snd_stop(VLSNDRCU)            ; // SND_STOP event to LW 3 of AUT_USND

    return 0                            ; // No error
  }


/*  Procedure ewreq ---------------------------------------------------------

    Purpose : UI treatment - Writes in EEPROM (offset specified). We call the
              non blocking procedure i2c_send which writes in the EEPROM.
    Note    : The EEPROM AT24C08A is used.
*/
```

```
static int ewreq(int n)
  {
    unsigned char *ptr              ; // Pointer
    unsigned char i                 ; // Index

/*************************************************************************
 * Step 1 : Fill the frame to be transmitted:                            *
 * ------   - Message length (without address I2C)                       *
 *          - Address I2C of the EEPROM                                  *
 *          - Offset in EEPROM (ex: 0x0322)                              *
 *          - Data to write                                              *
 *************************************************************************/

    ptr    = i2c_tx_frame           ; // Init of the pointer
    *ptr++ = n + 2                  ; // Message length (without address I2C)
    *ptr++ = I2C_ADDR_EEP_W         ; // Address I2C of the EEPROM
    *ptr++ = 0x03                   ; // Offset in EEPROM (0x0322)
    *ptr++ = 0x22                   ; //
    for (i = 1; i <= n; i++)          // Data to write
      *ptr++ = n + i                ; //
    *ptr   = 0x00                   ; // Next message length = 0: current
                                      // block = last block to be transmitted

/*************************************************************************
 * Step 2 : Deposit an event to the automaton I2C. This event contains the   *
 * ------   frame to be transmitted.                                     *
 *************************************************************************/

    i2c_send(I2C_BUS_1 | 0x1000,      // Bus I2C number = 1
             I2C_ADDR_EEP_W    ,      // Address I2C of the EEPROM
             i2c_tx_frame      ,      // Frame to be transmitted
             0                 ,      // Parameter unused
             -1                ,      // Not called from an automaton
             -1                ,      // EVTS_I2C_END not requested
             0x01              )    ; // Flags:
                                      // - b2 = 0: see b0
                                      // - b1 = 0: block length on 1 byte
                                      // - b0 = 1: "stop" generated only at
                                      //           the last block end

    return 0                        ; // No error
  }


/*  Procedure eroreq ----------------------------------------------------------

    Purpose : UI treatment - Reads data in EEPROM (offset specified). We call
              the non blocking procedure i2c_io which reads data in the EEPROM.
    Note    : The EEPROM AT24C08A is used.
*/

static int eroreq(int n)
  {
    I2c_io        *ptr              ; // Pointer on struct describing a frame
                                      // to be transmitted

/*************************************************************************
 * Step 1 : Fill the request:                                            *
 * ------   - 1st frame (EEPROM offset):                                 *
 *            - Writing address I2C of the EEPROM                        *
 *            - Flags (b0: "start" generation, b1: "stop" generation, b2:    *
 *              received byte acknowledgement)                           *
 *            - Offset in EEPROM (ex: 0x0322)                            *
 *            - Address of buffer and number of bytes to be transmitted      *
 *          - 2nd frame (wait request):                                  *
 *            - Code for wait request                                    *
 *            - Number of milliseconds to wait                           *
 *          - 3rd frame (reading):                                       *
```

```
 *            - Reading address I2C of the EEPROM                           *
 *            - Flags (unused when reading)                                 *
 *            - Reception buffer address and number of bytes to be read     *
 ******************************************************************************/

    ptr        = i2c_iolist          ; // Init of the pointer
    ptr->io    = I2C_ADDR_EEP_W      ; // Writing address I2C of the EEPROM
    ptr->flags = 0x03                ; // b0 = 1: generate "start"
                                       // b1 = 1: generate "stop"
                                       // b2 = 0: acknowledge received bytes
                                       //         (NA for writing)
    i2c_tx[0]  = 0x03                ; // Offset in EEPROM (b8-15)
    i2c_tx[1]  = 0x22                ; // Offset in EEPROM (b0-7)
    ptr->adr   = i2c_tx              ; // Address of buffer to be transmitted
    ptr->lg    = 2                   ; // Number of bytes to be transmitted

    ptr++                            ; // Increment pointer (next frame)
    ptr->io    = 0xFF                ; // Wait request
    ptr->lg    = 5                   ; // Wait 5ms

    ptr++                            ; // Increment pointer (next frame)
    ptr->io    = I2C_ADDR_EEP_R      ; // Reading address I2C of the EEPROM
    ptr->flags = 0                   ; // Flags (unused when reading: "start"
                                       // & "stop" always generated and
                                       // acknowledgement managed by dependent
                                       // hardware part of the I2C)
    ptr->adr   = i2c_rx              ; // Reception buffer address
    ptr->lg    = n                   ; // Number of bytes to be read

/******************************************************************************
 * Step 2 : Deposit an event to the automaton I2C. This event contains the    *
 * ------   frame to be transmitted.                                          *
 ******************************************************************************/

    i2c_io(I2C_BUS_1           ,       // Bus I2C number = 1
           i2c_iolist          ,       // Buffer containing struct describing
                                       // frames to be transmitted
           ptr - i2c_iolist + 1,       // Number of struct describing frames
           -1                  ,       // Not called from an automaton
           -1                  )     ; // EVTS_I2C_END not requested

    return 0                         ; // No error
  }


/*  Procedure erreq ------------------------------------------------------------

    Purpose : UI treatment - Reads data in EEPROM (next bytes). We call the
              non blocking procedure i2c_io which reads data in the EEPROM.
    Note    : The EEPROM AT24C08A is used.
*/

static int erreq(int n)
  {
    I2c_io        *ptr               ; // Pointer on struct describing a frame
                                       // to be transmitted

/******************************************************************************
 * Step 1 : Fill the request:                                                 *
 * ------   - Reading address I2C of the EEPROM                               *
 *          - Flags (unused when reading)                                     *
 *          - Reception buffer address and number of bytes to be read         *
 ******************************************************************************/

    ptr        = i2c_iolist          ; // Init of the pointer
    ptr->io    = I2C_ADDR_EEP_R      ; // Reading address I2C of the EEPROM
    ptr->flags = 0                   ; // Flags (unused when reading)
    ptr->adr   = i2c_rx              ; // Reception buffer address
```

```
    ptr->lg    = n                  ; // Number of bytes to be read

/*****************************************************************************
 * Step 2 : Deposit an event to the automaton I2C. This event contains the   *
 * ------   frame to be transmitted.                                         *
 *****************************************************************************/

    i2c_io(I2C_BUS_1 ,                // Bus I2C number = 1
           i2c_iolist,                // Buffer containing struct describing
                                      // frames to be transmitted
           1         ,                // Number of struct describing frames
           -1        ,                // Not called from an automaton
           -1        )              ; // EVTS_I2C_END not requested

    return 0                         ; // No error
  }


/*  Procedure ldreq ----------------------------------------------------------

    Purpose : UI treatment - Display text on screen LCD.
    Note    : The screen LCD160CR is used.
*/

static int ldreq(int n)
  {
    I2c_io        *pio               ; // Pointer on struct describing a frame
                                      // to be transmitted
    Lcd_text      *ptxt              ; // Pointer on struct describing text to
                                      // be displayed

/*****************************************************************************
 * Step 1 : Initialize pointers                                              *
 * ------                                                                    *
 *****************************************************************************/

    pio  = i2c_iolist                ; // Init of the pointer on struct
                                      // describing a frame to be transmitted
    ptxt = &i2c_text[n]              ; // Init of the pointer on struct
                                      // describing text to be displayed

/*****************************************************************************
 * Step 2 : Fill the request:                                                *
 * ------   - 1st frame (set position):                                      *
 *            - Writing address I2C of the screen LCD                        *
 *            - Flags (b0: "start" generation, b1: "stop" generation, b2:    *
 *              received byte acknowledgement)                               *
 *            - Command "set screen position"                                *
 *            - Horizontal and vertical coordinates                          *
 *            - Address of buffer and number of bytes to be transmitted      *
 *          - 2nd frame (set font):                                          *
 *            - Writing address I2C of the screen LCD                        *
 *            - Flags (b0: "start" generation, b1: "stop" generation, b2:    *
 *              received byte acknowledgement)                               *
 *            - Command "set font"                                           *
 *            - Parameters:                                                  *
 *              0b00ss ssss STff hhvv                                        *
 *                  |       |||  | vertical bold offset                     *
 *                  |       |||  horizontal bold offset                     *
 *                  |       ||font                                          *
 *                  |       |transparency flag (???)                       *
 *                  |       soft scroll flag (???)                         *
 *                 pixel replication (s+1) (???)                           *
 *          - 3rd frame (set colors):                                        *
 *            - Writing address I2C of the screen LCD                        *
 *            - Flags (b0: "start" generation, b1: "stop" generation, b2:    *
 *              received byte acknowledgement)                               *
 *            - Command "set colors"                                         *
```

```
*            - Foreground and background colors                              *
*          - 4th frame (transmit text to be displayed on screen):            *
*            - Writing address I2C of the screen LCD                         *
*            - Flags (b0: "start" generation, b1: "stop" generation, b2:     *
*              received byte acknowledgement)                                *
*            - Address of buffer and number of bytes to be transmitted       *
*********************************************************************************/

   pio        = i2c_iolist          ; // Init of the pointer
   pio->io    = I2C_ADDR_SCR_LCD_W  ; // Writing address I2C of screen LCD
   pio->flags = 0x03                ; // b0 = 1: generate "start"
                                      // b1 = 1: generate "stop"
                                      // b2 = 0: acknowledge received bytes
                                      //         (NA for writing)
   i2c_tx[0]  = 0x02                ; // Mode command
   i2c_tx[1]  = 'X'                 ; // Set position
   i2c_tx[2]  = ptxt->x             ; // Horizontal coordinate
   i2c_tx[3]  = ptxt->y             ; // Vertical coordinate
   pio->adr   = i2c_tx              ; // Address of buffer to be transmitted
   pio->lg    = 4                   ; // Number of bytes to be transmitted

   pio++                            ; // Increment pointer (next frame)
   pio->io    = I2C_ADDR_SCR_LCD_W  ; // Writing address I2C of screen LCD
   pio->flags = 0x03                ; // b0 = 1: generate "start"
                                      // b1 = 1: generate "stop"
                                      // b2 = 0: acknowledge received bytes
                                      //         (NA for writing)
   i2c_tx[4]  = 0x02                ; // Mode command
   i2c_tx[5]  = 'F'                 ; // Set font
   i2c_tx[6]  = 0x00                ; // Parameters (see header)
   i2c_tx[7]  = 0x01                ; //
   pio->adr   = &i2c_tx[4]          ; // Address of buffer to be transmitted
   pio->lg    = 4                   ; // Number of bytes to be transmitted

   pio++                            ; // Increment pointer (next frame)
   pio->io    = I2C_ADDR_SCR_LCD_W  ; // Writing address I2C of screen LCD
   pio->flags = 0x03                ; // b0 = 1: generate "start"
                                      // b1 = 1: generate "stop"
                                      // b2 = 0: acknowledge received bytes
                                      //         (NA for writing)
   i2c_tx[8]  = 0x02                ; // Mode command
   i2c_tx[9]  = 'c'                 ; // Set colors
   memcpy(&i2c_tx[10], &ptxt->fc, 2); // Foreground color
   memcpy(&i2c_tx[12], &ptxt->bc, 2); // Background color
   pio->adr   = &i2c_tx[8]          ; // Address of buffer to be transmitted
   pio->lg    = 6                   ; // Number of bytes to be transmitted

   pio++                            ; // Increment pointer (next frame)
   pio->io    = I2C_ADDR_SCR_LCD_W  ; // Writing address I2C of screen LCD
   pio->flags = 0x03                ; // b0 = 1: generate "start"
                                      // b1 = 1: generate "stop"
                                      // b2 = 0: acknowledge received bytes
                                      //         (NA for writing)
   strcpy((char*)&i2c_tx[14],         // Copy string to display
          ptxt->txt        )        ; //
   pio->adr   = &i2c_tx[14]         ; // Address of buffer to be transmitted
   pio->lg    = sizeof(ptxt->txt) + 1;// Number of bytes to be transmitted

/*********************************************************************************
 * Step 3 : Deposit an event to the automaton I2C. This event contains the   *
 * ------   frame to be transmitted.                                         *
 *********************************************************************************/

   i2c_io(I2C_BUS_1           ,       // Bus I2C number = 1
          i2c_iolist          ,       // Buffer containing struct describing
                                      // frames to be transmitted
          pio - i2c_iolist + 1,       // Number of struct describing frames
          -1                  ,       // Not called from an automaton
```

```
           -1                  )     ; // EVTS_I2C_END not requested

    return 0                         ; // No error
  }


/*  Procedure lcreq ----------------------------------------------------------

    Purpose : UI treatment - Clear screen LCD.
    Note    : The screen LCD160CR is used.
*/

static int lcreq(int n)
  {
    I2c_io        *ptr               ; // Pointer on struct describing a frame
                                       // to be transmitted

/*************************************************************************
 * Step 1 : Fill the request:                                            *
 * ------   - Writing address I2C of the screen LCD                      *
 *          - Flags (b0: "start" generation, b1: "stop" generation, b2:  *
 *            received byte acknowledgement)                             *
 *          - Command "clear screen LCD"                                 *
 *          - Address of buffer and number of bytes to be transmitted    *
 *************************************************************************/

    ptr        = i2c_iolist          ; // Init of the pointer
    ptr->io    = I2C_ADDR_SCR_LCD_W  ; // Writing address I2C of screen LCD
    ptr->flags = 0x03                ; // b0 = 1: generate "start"
                                       // b1 = 1: generate "stop"
                                       // b2 = 0: acknowledge received bytes
                                       //         (NA for writing)
    i2c_tx[0]  = 0x02                ; // Mode command
    i2c_tx[1]  = 'E'                 ; // Clear screen LCD
    ptr->adr   = i2c_tx              ; // Address of buffer to be transmitted
    ptr->lg    = 2                   ; // Number of bytes to be transmitted

/*************************************************************************
 * Step 2 : Deposit an event to the automaton I2C. This event contains the   *
 * ------   frame to be transmitted.                                     *
 *************************************************************************/

    i2c_io(I2C_BUS_1 ,               // Bus I2C number = 1
           i2c_iolist,               // Buffer containing struct describing
                                     // frames to be transmitted
           1         ,               // Number of struct describing frames
           -1        ,               // Not called from an automaton
           -1        )               ; // EVTS_I2C_END not requested

    return 0                         ; // No error
  }


/*  Procedure ltcreq ---------------------------------------------------------

    Purpose : UI treatment - Get touch coordinates.
    Note    : The screen LCD160CR is used.
*/

static int ltcreq(int n)
  {
    I2c_io        *ptr               ; // Pointer on struct describing a frame
                                       // to be transmitted

/*************************************************************************
 * Step 1 : Fill the request:                                            *
 * ------   - 1st frame (transmit command):                              *
 *            - Writing address I2C of the screen LCD                    *
```

```
 *            - Flags (b0: "start" generation, b1: "stop" generation, b2:    *
 *              received byte acknowledgement)                               *
 *            - Command "get touch coordinates"                              *
 *            - Address of buffer and number of bytes to be transmitted      *
 *          - 2nd frame (wait request):                                      *
 *            - Code for wait request                                        *
 *            - Number of milliseconds to wait                               *
 *          - 3rd frame (position reading):                                  *
 *            - Reading address I2C of the screen LCD                        *
 *            - Flags (unused when reading)                                  *
 *            - Reception buffer address and number of bytes to be read      *
 *            - Received bytes:                                              *
 *              - 1st byte = 0x03: ???                                       *
 *              - 2nd byte: b7 = 0: screen not currently touched             *
 *                          1: screen currently touched                      *
 *              - 3rd byte: vertical coordinate*                             *
 *              - 4th byte: horizontal coordinate*                           *
 *                *: "Global screen orientation applies also to touch        *
 *                   coordinates."                                           *
 ******************************************************************************/

   ptr        = i2c_iolist          ; // Init of the pointer
   ptr->io    = I2C_ADDR_SCR_LCD_W  ; // Writing address I2C of screen LCD
   ptr->flags = 0x03                ; // b0 = 1: generate "start"
                                      // b1 = 1: generate "stop"
                                      // b2 = 0: acknowledge received bytes
                                      //         (NA for writing)
   i2c_tx[0]  = 0x02                ; // Mode command
   i2c_tx[1]  = 'T'                 ; // Get touch coordinates
   ptr->adr   = i2c_tx              ; // Address of buffer to be transmitted
   ptr->lg    = 2                   ; // Number of bytes to be transmitted

   ptr++                            ; // Increment pointer (next frame)
   ptr->io    = 0xFF                ; // Wait request
   ptr->lg    = 1                   ; // Wait 1ms

   ptr++                            ; // Increment pointer (next frame)
   ptr->io    = I2C_ADDR_SCR_LCD_R  ; // Reading address I2C of screen LCD
   ptr->flags = 0                   ; // Flags (unused when reading: "start"
                                      // & "stop" always generated and
                                      // acknowledgement managed by dependent
                                      // hardware part of the I2C)
   ptr->adr   = i2c_rx              ; // Reception buffer address
   ptr->lg    = 4                   ; // Number of bytes to be read

/******************************************************************************
 * Step 2 : Deposit an event to the automaton I2C. This event contains the    *
 * ------   frame to be transmitted.                                          *
 *          Note: The automaton number and logical channel number are         *
 *                specified as the event EVTS_I2C_END will be deposit when    *
 *                reading is ended.                                           *
 ******************************************************************************/

   i2c_io(I2C_BUS_1           ,       // Bus I2C number = 1
          i2c_iolist          ,       // Buffer containing struct describing
                                      // frames to be transmitted
          ptr - i2c_iolist + 1,       // Number of struct describing frames
          AUT_TASK            ,       // Automaton number
          myapp_taskid && 0xFF)     ; // Logical channel number

   return 0                         ; // No error
  }


/*  Procedure sapt ------------------------------------------------------------

    Purpose : UI treatment - Set a new ASY1/2 transmit pattern
*/
```

```
static int sapt(int n)
  {
    fill256(buf_snd[n == VASY1PT ? 1 : 2],cfg[n]) ;

    return 0                          ; // No error
  }


/*  Procedure sdpt ----------------------------------------------------------

    Purpose : UI treatment - Set a new DAC transmit pattern
*/

static int sdpt(int n)
  {

    return 0                          ; // No error
  }


/*  Procedure srcu ----------------------------------------------------------

    Purpose : UI treatment - Set a new input according to VINPUT and a new RCU
              according to VNUMRCU.
*/

static int srcu(int n)
  {
    INT           ret                 ; // Error code

    set_rcu(rcurcv_id              ,   // id     = the indicated receiver
            cfg[VINPUT]            ,   // swpos  = the selected input (IR/UHF)
            str_rcuna[cfg[VNUMRCU]],   // name   = the selected RCU name
            &ret                    ); // ret    = error code

    return ret                        ; // No error
  }


/*  Procedure scap ----------------------------------------------------------

    Purpose : UI treatment - Starts a RCU symbols capture.
*/

static int scap(int n)
  {
    INT           ret                 ; // Error code

    capture_rcu(rcurcv_id    ,          // id     = the indicated receiver
                cfg[VONMIN]  ,          // otmin  = start on time min duration
                cfg[VONMAX]  ,          // otmax  = start on time max duration
                cfg[VSTMIN]  ,          // stmin  = start symb time min duration
                cfg[VSTMAX]  ,          // stmax  = start symb time max duration
                &ret          )     ; // ret    = error code

    return ret                        ; // No error
  }


/*  Procedure sret ----------------------------------------------------------

    Purpose : UI treatment - Affects parameter to "condret".
*/

static int sret(int n)
  {
    condret = n                       ; // Affects parameter to "condret"
```

```
    return 0                        ; // No error
  }


/*  Procedure scon ----------------------------------------------------------

    Purpose : UI treatment - Affects "condret" to "condcode".
*/

static int scon(int n)
  {
    condcode = condret              ; // Affects "condret" to "condcode"
    return 0                        ; // No error
  }


/*  Procedure rest ----------------------------------------------------------

    Purpose : UI treatment - Set to 1 the "flag_rest" global variable. The
              UI main event loop (step 6 of "myproc_task") checks this flag
              a each loop end.

*/

static int rest(int n)
  {
    flag_rest = 1                   ; // Ask for an end of loop
    return 0                        ; // No error
  }


/*  Procedure ev_opt --------------------------------------------------------

    Purpose : Convert the EV_ code event to a OPT_ menu event.
*/

static int ev_opt(int ev)
  {
    int             opt             ; // Return code

    switch ( ev )
      {
        case EV_ASY0_RCV            : // Keyboard console data received
          opt = ev_asy0_rcv()       ; // Determine option selected
          break                     ;

        case EV_ASY1_RCV            : // Data received on ASY\DEV1
          opt = ev_asy12_rcv(1)     ; // Determine option selected
          break                     ;

        case EV_ASY2_RCV            : // Data received on ASY\DEV2
          opt = ev_asy12_rcv(2)     ; // Determine option selected
          break                     ;

        case EV_ASY0_SND            : // Data sent on ASY\DEV0
          opt = ev_asy0_snd()       ; // Determine option selected
          break                     ;

        case EV_ASY1_SND            : // Data sent on ASY\DEV1
          opt = ev_asy12_snd(1)     ; // Determine option selected
          break                     ;

        case EV_ASY2_SND            : // Data sent on ASY\DEV2
          opt = ev_asy12_snd(2)     ; // Determine option selected
          break                     ;

        case EV_GPIO0_IN            : // GPIO input pin change button 1
          opt = ev_gpio012_in(0)    ; // Determine option selected
```

```
          break                       ;

        case EV_GPIO1_IN              : // GPIO input pin change button 2
          opt = ev_gpio012_in(1)      ; // Determine option selected
          break                       ;

        case EV_GPIO2_IN              : // GPIO input pin change button 3
          opt = ev_gpio012_in(2)      ; // Determine option selected
          break                       ;

        case EV_I2C_END               : // I2C job ended
          opt = ev_i2c_end()          ; // Determine option selected
          break                       ;

        case EV_KBD_RCV               : // Keyboard event received from RCU
          opt = ev_kbd_rcv()          ; // Determine option selected
          break                       ;

        case EV_MSEC                  : // 1000 sec have been elapsed
          opt = ev_msec()             ; // Determine option selected
          break                       ;
      }

    return opt                        ;
  }


/*  Procedure ev_asy0_rcv ----------------------------------------------------

    Purpose : Parses the RESP_READ event for DEV_ASY0 and then selects the
              treatment when EV_ASY0_RCV event is received. Also, as a receive
              token has been used, give to the ASY driver a new receive token.
*/

static int ev_asy0_rcv(void)
  {
    unsigned char  *buf               ; // Buffer address
    char            c                 ; // First character of buffer
    int             ret               ; // Return code
    int             err               ; // Error code

/*************************************************************************
 * Step 1 : Update the number of tokens. We just receive one RESP_READ       *
 * ------   event from ASY device DEV0, so we decrement "tok_rcv[0]" counter.*
 *          We also initialize "ret" to ignore the event.                    *
 *************************************************************************/

    tok_rcv[0] --                     ; // One token less.
    ret = OPT_IGNO                    ; // This character will be ignored


/*************************************************************************
 * Step 2 : Get the data buffer address. We have just received a READ_RESP   *
 * ------   event. A copy is available in the "task_evt" global variable     *
 *          (type is S_"evt"). The buffer address is contained in the        *
 *          "adresse" field of the "task_evt" event structure. If no buffer  *
 *          has been received, we jump to step 6 to give a new reveive       *
 *          token.                                                           *
 *************************************************************************/

    buf = task_evt.adresse            ; // Read buffer address from event
    if (buf EQ HNULL)                   // If no buffer received,
      goto step6                      ; //   go to step 6.


/*************************************************************************
 * Step 3 : Check for errors. Field "length" of "task_evt" global variable   *
 * ------   is an error code if negative. In case of error, we jump to step  *
```

```
 *          5 to free the received buffer.                                  *
 ****************************************************************************/

    err = task_evt.longueur          ; // Read error code from event
    if (err < 0)                       // If an error occured,
      goto step5                     ; //   go to step 5.


/****************************************************************************
 * Step 4 : Parse the content of the received buffer. Here we implement a    *
 * ------   very naive parsing. We extract the first received character and  *
 *          we just check it is an ASCII code:                              *
 *          - Codes  '0' to '9' are converted to OPT_0 to OPT_9.            *
 *          - Codes  'A' to 'D' are converted to OPT_BUT1 to OPT_BUT4.      *
 *          - Codes  'a' to 'd' are converted to OPT_BUT1 to OPT_BUT4.      *
 *          - Code   '\r'       is  converted to OPT_ENTER.                 *
 *          - Code   8          is  converted to OPT_BACK.                  *
 *          - With other values, we return OPT_INVALID.                     *
 ****************************************************************************/

    c   = (char) buf[0]              ; // Extract the first byte of data

    if      (c >= '0' && c <= '9')     // If byte is character '0' to '9'
      ret = OPT_0    + (c - '0')     ; // then return OPT_<c>

    else if (c >= 'A' && c <= 'D')     // If byte is character 'A' to 'D'
      ret = OPT_BUT1 + (c - 'A')     ; // then return OPT_BUT1/2/3/4

    else if (c >= 'a' && c <= 'd')     // If byte is character 'a' to 'd'
      ret = OPT_BUT1 + (c - 'a')     ; // then return OPT_BUT1/2/3/4

    else if (c == '\r'           )     // If byte is character '\r'
      ret = OPT_ENTER                ; // then return OPT_ENTER

    else if (c == 8              )     // If byte is character BACKSPACE
      ret = OPT_BACK                 ; // then return OPT_ENTER

    else if (c == 27             )     // If byte is character ESCAPE
      ret = OPT_ESC                  ; // then return OPT_ESC

    else
      ret = OPT_INVALID              ; // This character will be ignored


/****************************************************************************
 * Step 5 : We have finished to use the buffer data, we have now to free     *
 * ------   that buffer. If we forget that, the STM32 will quickly get out   *
 *          of memory (a STM32re has only 80 KB of RAM).                    *
 ****************************************************************************/

    step5                            : // Step 5 label
    free_buf(buf, &err)              ; // Free the buffer.


/****************************************************************************
 * Step 6 : Give a new receive token to the ASY driver for device DEV0.      *
 * ------                                                                    *
 ****************************************************************************/

    step6                            : // Step 6 label
    myio_givetok(asy_iod0   ,          // I/O descriptor
                 1          ,          // Number of receive token
                 1          ,          // Size of receive buffer
                 &tok_rcv[0]  )      ; // Counter of given tokens

    return ret                       ;
  }
```

```
/*  Procedure ev_asy0_snd ----------------------------------------------------

    Purpose : Select the treatment when EV_ASY0_SND event is received.
*/

static int ev_asy0_snd(void)
  {
    return OPT_IGNO                 ; // Ignore these events
  }


/*  Procedure ev_asy12_rcv ---------------------------------------------------

    Purpose : Select the treatment when EV_ASY1_RCV/EV_ASY2_RCV  event is
              received. The "n" input parameter value is 1 for ASY1 and 2 for
              ASY2.
*/

static int ev_asy12_rcv(int n)
  {
    unsigned char  *buf             ; // Buffer address
    int             len             ; // Data length
    int             vspeed          ; // VPEED1 or VSPEED2
    int             vasy            ; // VASY1  or VASY2
    int             err             ; // Return code

/*****************************************************************************
 * Step 1 : Update the number of tokens. We just receive one RESP_READ       *
 * ------   event from ASY device DEV1/DEV2, so we decrement "tok_rcv[1]" or *
 *          "tok_rcv[2]" counter.                                            *
 *****************************************************************************/

    tok_rcv[n] --                   ; // One token less.


/*****************************************************************************
 * Step 2 : Get the data buffer address. We have just received a READ_RESP   *
 * ------   event. A copy is available in the "task_evt" global variable     *
 *          (type is "S_evt"). The buffer address is contained in the        *
 *          "adresse" field of the "task_evt" event structure. Length of     *
 *          received data is available in the "longueur" field.              *
 *****************************************************************************/

    buf = task_evt.adresse          ; // Read buffer address from event
    len = task_evt.longueur         ; // Read data length from event


/*****************************************************************************
 * Step 3 : If we are in fast mode, we give tokens to ASY device DEV1 or     *
 * ------   DEV2, so that it has always 2 tokens available for reception. In *
 *          slow mode, 1 token will be given when "ev_msec" is called in at  *
 *          most 1 second from now.                                          *
 *****************************************************************************/

    vspeed = (n == 1) ? VSPEED1 :
                        VSPEED2     ;

    vasy   = (n == 1) ? VASY1   :
                        VASY2       ;

    if ( cfg[vspeed] EQ CFG_FAST )     // If in fast mode,
      while ( tok_rcv[n] LT 2 )        // give at most 2 tokens.
        myio_givetok(iods[vasy] ,      // I/O descriptor
                     1          ,      // Number of receive token
                     LGRCV12    ,      // Size of receive buffer
                     &tok_rcv[n]  )  ; // Counter of given tokens
```

```
/*****************************************************************************
 * Step 4 : If no buffer has been received, exit the procedure. If length is *
 * ------   negative, an error occured: free the buffer and exit.            *
 *****************************************************************************/

    if (buf EQ HNULL)                  // If no buffer received,
      goto end                       ; //   exit the procedure.

    if (len < 0)                       // If an error occured, received
      BEGIN                            // data are incorrect.
        free_buf(buf, &err)          ; // Free the buffer.
        goto end                     ; // Exit from the procedure.
      END_IF                           //


/*****************************************************************************
 * Step 5 : Test state of ASY1. If currently sending, free the telecom       *
 * ------   buffer and exit. If we forget to free the buffer, the STM32 will *
 *          quickly get out of memory (a STM32re has only 80 KB of RAM).     *
 *****************************************************************************/

    if (cfg[vasy] EQ 1)                // If broadcasting enabled,
      BEGIN                            // we cannot send.
        free_buf(buf, &err)          ; // Free the buffer.
        goto end                     ; // Exit from the procedure.
      END_IF                           //


/*****************************************************************************
 * Step 6 : Output is available, we send back the received data as an "echo".*
 * ------   We do not unschedule.                                            *
 *****************************************************************************/

    myio_send   (iods[vasy] ,          // I/O descriptor
                 buf        ,          // Address of buffer
                 len        ,          // Number of bytes to send
                 &cnt_snd[n] )       ; // Number of sending messages

    end                              : // Early exit

    return OPT_IGNO                  ; // Ignore these events
  }


/*  Procedure ev_asy12_snd ----------------------------------------------------

    Purpose : Select the treatment when EV_ASY1_SND/EV_ASY2_SND event is
              received. The "n" input parameter is 1 for ASY1 and 2 for ASY2.
*/

static int ev_asy12_snd(int n)
  {

    unsigned char  *snd              ; // Buffer address
    int             vasy             ; // VASY1 if AYS1, VASY2 if ASY2
    int             ret              ; // Return code

/*****************************************************************************
 * Step 1 : One message has been sent, update the number of messages given   *
 * ------   to ASY driver.                                                   *
 *****************************************************************************/

    cnt_snd[n] --                    ; // One less message in driver's queue


/*****************************************************************************
 * Step 2 : If sent buffer is not the static "buf_snd[1/2]" buffer, it is an *
```

```
 * ------   "echo" buffer sent in "ev_asy12_rcv". We need to free it.        *
 ****************************************************************************/

    snd = task_evt.adresse           ; // Get sent buffer address

    if ( snd NE buf_snd[n] )           // If an "echo" buffer has been sent,
      free_buf(snd, &ret)            ; // free it.


/****************************************************************************
 * Step 2 : If stop is requested, there is nothing more to do.              *
 * ------                                                                   *
 ****************************************************************************/

    vasy = (n == 1) ? VASY1 : VASY2  ; // Associated state variable number

    if (cfg[vasy] == 0)                // If stop has been requested,
      goto end                       ; // exit from the procedure.


/****************************************************************************
 * Step 3 : When running, we want to have at least two messages ready to be  *
 * ------   sent in ASY driver so that there is no time lost between two     *
 *          messages. Variable "msg_snd[n]" holds the number of messages     *
 *          given to ASY driver. This variable is incremented each time a    *
 *          request is sent to ASY driver and decremented each time a        *
 *          response is received from ASY driver (Step 1).                   *
 ****************************************************************************/

    while (cnt_snd[n] LT 2)            // Until we have sent enough messages,
      myio_send   (iods[vasy] ,        // I/O descriptor
                   buf_snd[n] ,        // Address of buffer
                   256      ,          // Number of bytes to send
                   &cnt_snd[n])      ; // Number of sending messages

    end                              : // Early exit
    return OPT_IGNO                  ; // Ignore these events
  }


/*  Procedure ev_gpio012_in ----------------------------------------------------

    Purpose : Parse the received EV_GPIO0/1/2_IN event and select the treatment
              to be executed.
*/

static int ev_gpio012_in(int n)
  {

/****************************************************************************
 * Step 1 : The event we received is an IND_REPORT. Its "res2" field is the  *
 * ------   tag identifier and its "longueur" field the new tag value.       *
 *          In GPIO case, we only expect IND_REPORT events on tag INPUT      *
 *          indicating the button has been pressed or released. When INPUT   *
 *          is LOW (0), the button has been pressed and this corresponds to  *
 *          option BUT1. When INPUT is high (1), the button has been         *
 *          released and we want to ignore.                                  *
 ****************************************************************************/

    if (task_evt.longueur EQ 0)        // If INPUT is LOW,
      return OPT_BUT1 + n            ; //   button 1 has been pressed.
    else                               // Otherwise,
      return OPT_IGNO                ; //   it has been released
  }


/*  Procedure ev_i2c_end -------------------------------------------------------
```

```
    Purpose : Display the touch coordinates.
*/

static int ev_i2c_end(void)
  {

/*****************************************************************************
 * Step 1 : The received event is EVTS_I2C_END.                              *
 * ------   Return the option IEC END                                        *
 *****************************************************************************/

    return OPT_I2C_END              ; // Return option IEC END
  }


/*  Procedure ev_kbd_rcv ----------------------------------------------------

    Purpose : Select the treatment when EV_KBD_RCV event is received. The event
              that we received is stored in "task_evt", from which we can get
              the original event code (DRV_KEY_PRESS or DRV_KEY_RELEASE) and
              the keycode (task_evt.reserve).
*/

static int ev_kbd_rcv(void)
  {

/*****************************************************************************
 * Step 1 : The KEYBOARD handler sends us two different event codes that are *
 * ------   DRV_KEY_PRESS when a key is pressed and DRV_KEY_RELEASE when the *
 *          key is reselased. Here we chose to use the DRV_KEY_PRESS and to  *
 *          ignore all the DRV_KEY_REALEASE. So we test if the event code is *
 *          a DRV_KEY_RELEASE and we exit immedialtely in that case.         *
 *****************************************************************************/

    if (task_evt.code                 // We just ignore all the "release key"
          EQ DRV_KEY_RELEASE)         // events. We return the OPT_IGNO
      return OPT_IGNO               ; // code.

/*****************************************************************************
 * Step 2 : Here we have an event code DRV_KEY_PRESS. The "reserve" field is *
 * ------   the key-code. The codes are KEY_0 to KEY_9 for RCU digit keys.    *
 *          For RCU colored keys RED, GREEN, YELLOW and BLUE key codes are   *
 *          respectively KEY_F1, KEY_F2, KEY_F3 and KEY_F4.                  *
 *****************************************************************************/

    switch (task_evt.reserve)
      {
        case KEY_0                  : // Digit keys, 0 to 9
          return OPT_0              ;
        case KEY_1                  :
          return OPT_1              ;
        case KEY_2                  :
          return OPT_2              ;
        case KEY_3                  :
          return OPT_3              ;
        case KEY_4                  :
          return OPT_4              ;
        case KEY_5                  :
          return OPT_5              ;
        case KEY_6                  :
          return OPT_6              ;
        case KEY_7                  :
          return OPT_7              ;
        case KEY_8                  :
          return OPT_8              ;
        case KEY_9                  :
          return OPT_9              ;
```

```
        case KEY_ENTER              : // OK key
          return OPT_ENTER          ;
        case KEY_BACK               : // BACK key
          return OPT_BACK           ;
        case KEY_EXIT               : // EXIT key
          return OPT_ESC            ;

        case KEY_F1                 : // RED key
          return OPT_BUT1           ;
        case KEY_F2                 : // GREEN KEY
          return OPT_BUT2           ;
        case KEY_F3                 : // YELLOW key
          return OPT_BUT3           ;
        case KEY_F4                 : // BLUE key
          return OPT_BUT4           ;

        case KEY_CAPTURE            : // Pseudo key for CAPTURE
          return OPT_CAPT           ;

        default                     :
          return OPT_INVALID        ;
      }
  }


/*  Procedure ev_msec ----------------------------------------------------------

    Purpose : Select the option when EV_MSEC event is received.
*/

static int ev_msec(void)
  {

/*************************************************************************
 * Step 1 : If we are in slow mode, we give tokens to ASY device DEV1, so    *
 * ------   that it has at least 1 token available for reception. In fast    *
 *          mode, at most 2 tokens are given by "ev_asy12_rcv" as soon as a  *
 *          RESP_READ event is received.                                     *
 *************************************************************************/

    if ( cfg[VSPEED1] EQ CFG_SLOW )     // If in slow mode,
      while ( tok_rcv[1] LT 1 )         // give at most 1 token.
        myio_givetok(asy_iod1 ,         // I/O descriptor
                     1        ,         // Number of receive tokens
                     LGRCV12  ,         // Size of receive buffer
                     &tok_rcv[1]  )  ; // Counter of given tokens

/*************************************************************************
 * Step 2 : Same thing as above but with ASY2.                               *
 * ------                                                                    *
 *************************************************************************/

    if ( cfg[VSPEED2] EQ CFG_SLOW )     // If in slow mode,
      while ( tok_rcv[2] LT 1 )         // give at most 1 token.
        myio_givetok(asy_iod2 ,         // I/O descriptor
                     1        ,         // Number of receive tokens
                     LGRCV12  ,         // Size of receive buffer
                     &tok_rcv[2]  )  ; // Counter of given tokens


/*************************************************************************
 * Step 3 : Option selected is always the system option OPT_TIME used to     *
 * ------   update time display.                                             *
 *************************************************************************/

    return OPT_TIME                 ; // Return time refresh option
  }
```

```
/*  Procedure wait_myevents ----------------------------------------------------

    Purpose : Unschedule until any of my events is received or "msec" seconds
              have been elapsed. A value of 0 for "msec" means no maximum
              time limit. According to the received event, the return value
              of this procedure is as follows:

              Code             Reserve        Return value
              ---------------  --------       ---------------
              RESP_READ        asy_iod0   ->  EV_ASY0_RCV   0
              RESP_READ        asy_iod1   ->  EV_ASY1_RCV   1
              RESP_READ        asy_iod2   ->  EV_ASY2_RCV   2
              RESP_WRITE       asy_iod0   ->  EV_ASY0_SND   3
              RESP_WRITE       asy_iod1   ->  EV_ASY1_SND   4
              RESP_WRITE       asy_iod2   ->  EV_ASY2_SND   5
              IND_REPORT       gpio_iod1  ->  EV_GPIO0_IN   10
              IND_REPORT       gpio_iod2  ->  EV_GPIO1_IN   11
              IND_REPORT       gpio_iod3  ->  EV_GPIO2_IN   12
              EVTS_I2C_END     i2c1       ->  EV_I2C_END    20
              DRV_KEY_PRESS    any        ->  EV_KBD_RCV    30
              DRV_KEY_RELEASE  any        ->  EV_KBD_RCV    30
              TICK             any        ->  EV_MSEC       40

              The "ret" value maybe -1 if we receive something else
*/


static int wait_myevents(void)
  {
    int             waitlist[7][3]   ; // Parameter of "waitevt_task"
    int             res              ; // Field "reserve" of task_evt.reserve
    int             ret              ; // Return code

/*****************************************************************************
 * Step 1 : Build the list of our awaited events and then unschedule until   *
 * ------   one of those is received. The events we are waiting for are:     *
 *          - The RESP_READ. Those are coming from the ASY driver when data  *
 *            is received on serial controller DEV0, DEV1 or DEV2. So the    *
 *            awaited code event is RESP_READ and the awaited "reserve"      *
 *            field value may be "asy_iod0", "asy_iod1" or "asy_iod2". We    *
 *            choose to use -1 (any value) for "reserve".                    *
 *          - The RESP_WRITE. Those are coming from the ASY driver when send *
 *            request has been completed (the last byte has been sent). The  *
 *            code event is RESP_READ and the "reserve" field value may be   *
 *            "asy_iod0", "asy_iod1" or "asy_iod2". We choose to use -1 (any *
 *            value).                                                        *
 *          - The IND_REPORT. Those are coming from the GPIO driver when we  *
 *            have a change state of any input pin. The "reserve" field may  *
 *            be "gpio_iod1" (RED led) "gpio_iod2" (GREEN led) or            *
 *            "gpio_iod3" (YELLOW led).                                      *
 *          - The DRV_KEY_PRESS (RCU key has been pressed) or the            *
 *            DRV_KEY_RELEASE (RCU key has been released).                   *
 *          - The EVTS_I2C_END. Those are coming from the I2C driver when    *
 *            data is received on the I2C (today only when the touch         *
 *            coordinates have been received).                               *
 *          - The TICK event that is received every second. This one is the  *
 *            result of the call to the "set_tto" procedure.                 *
 *          Then we call the "waitevt_task" procedure that will unschedule   *
 *          us until one of the awaited event will be received.              *
 *****************************************************************************/

    waitlist[0][0]  = WAIT_CODERES   ; // All events with
    waitlist[0][1]  = RESP_READ      ; // an event code RESP_READ
    waitlist[0][2]  = -1             ; // whatever the value of "reserve" is

    waitlist[1][0]  = WAIT_CODERES   ; // All events with
    waitlist[1][1]  = RESP_WRITE     ; // an event code RESP_WRITE
```

```
    waitlist[1][2]  = -1            ; // whatever the value of "reserve" is

    waitlist[2][0]  = WAIT_CODERES  ; // All events with
    waitlist[2][1]  = IND_REPORT    ; // an event code IND_REPORT
    waitlist[2][2]  = -1            ; // whatever the value of "reserve" is

    waitlist[3][0]  = WAIT_CODERES  ; // All events with
    waitlist[3][1]  = DRV_KEY_PRESS ; // an event code DRV_KEY_PRESS
    waitlist[3][2]  = -1            ; // whatever the value of "reserve" is

    waitlist[4][0]  = WAIT_CODERES  ; // All events with
    waitlist[4][1]  = DRV_KEY_RELEASE; // an event code DRV_KEY_RELEASE
    waitlist[4][2]  = -1            ; // whatever the value of "reserve" is

    waitlist[5][0]  = WAIT_CODERES  ; // All events with
    waitlist[5][1]  = EVTS_I2C_END  ; // an event code EVTS_I2C_END
    waitlist[5][2]  = -1            ; // whatever the value of "reserve" is

    waitlist[6][0]  = WAIT_CODERES  ; // All events with
    waitlist[6][1]  = TICK          ; // an event code TICK
    waitlist[6][2]  = 0             ; // with "reserve" equal to request

    waitevt_task(waitlist ,           // Address of waiting list
                 7        ,           // Size of "waitlist[]"
                 0        ,           // Maximum waiting time = no
                 0        ,           // Do not purge previous events
                 &ret       )       ; // Return code


/*****************************************************************************
 * Step 2 : Here we are scheduled again. The VMK has written into its global *
 * ------   variable "task_evt" a copy of the event that has scheduled us    *
 *          again. As a code event value, we can have RESP_READ, RESP_WRITE, *
 *          IND_REPORT, EVTS_I2C_END, DRV_KEY_PRESS and DRV_KEY_RELEASE but  *
 *          also the VMK generated event TICK (every second). According to   *
 *          the value of "task_evt.code" and "task_evt.reserve" we compute   *
 *          the return value "ret". If we receive something we are not       *
 *          expecting, we will return a default value of -1.                 *
 *****************************************************************************/

    res = task_evt.reserve          ; // Extract the "reserve" field from the
    ret = -1                        ; // received event.

    switch ( task_evt.code )
      {
        case RESP_READ              : // For a RESP_READ, the "reserve"
          if      (res EQ asy_iod0)   // field is the "iod" value. So
            ret = EV_ASY0_RCV       ; // we compare it to our IOD's,
          else if (res EQ asy_iod1)   // that are "asy_iod0" for ASY\DEV0
            ret = EV_ASY1_RCV       ; // "asy_iod1" for ASY\DEV1 and then
          else if (res EQ asy_iod2)   // "asy_iod2" for ASY\DEV2
            ret = EV_ASY2_RCV       ; //
          break                     ; //

        case RESP_WRITE             : // For a RESP_WRITE, the "reserve"
          if      (res EQ asy_iod0)   // field is the "iod" value. So
            ret = EV_ASY0_SND       ; // we compare it to our IOD's,
          else if (res EQ asy_iod1)   // that are "asy_iod0" for ASY\DEV0,
            ret = EV_ASY1_SND       ; // "asy_iod1" for ASY\DEV1 and
          else if (res EQ asy_iod2)   // then "asy_iod2" for ASY\DEV2
            ret = EV_ASY2_SND       ; //
          break                     ; //

        case IND_REPORT             : // For an IND_REPORT, the "reserve" field
          if      (res EQ gpio_iod1)  // value is the "iod". So we compare
            ret = EV_GPIO0_IN       ; // it to all the IOD's that may send us
          else if (res EQ gpio_iod2)  // an IND_REPORT, so here the 3 buttons,
            ret = EV_GPIO1_IN       ; // so the values may be "gpio_iod1",
```

```
          else if (res EQ gpio_iod3)   // "gpio_iod2" or "gpio_iod3" and not
            ret = EV_GPIO2_IN        ; // more
          break                      ; //

        case EVTS_I2C_END            : // For an EVTS_I2C_END, the "reserve"
          ret = EV_I2C_END           ; // field is a copy of the one received
          break                      ; // and we don't care.

        case TICK                    : // For a TICK, the "reserve"
          ret = EV_MSEC              ; // field is a copy of the one received
          break                      ; // and we don't care.

        case DRV_KEY_PRESS           : // For a DRV_KEY_PRESS, the "reserve"
          ret = EV_KBD_RCV           ; // field is the code of the key that
          break                      ; // has been pressed.

        case DRV_KEY_RELEASE         : // For a DRV_KEY_RELEASE, the "reserve"
          ret = EV_KBD_RCV           ; // field is the code of the key that
          break                      ; // has been released.

        default                      : // This should never occur if there
          break                      ; // is no bug.
      }

    return ret                       ;
   }


/*  Procedure fill256 ----------------------------------------------------------

    Purpose : Fill a buffer with a 256 bytes pattern.
*/

static void fill256(unsigned char *buf, int pat)
  {
    int             i                ; // Loop counter

    switch ( pat )
      {
        case 0                       : // 256 bytes 0x00 to 0xFF
          for(i = 0; i < 256; i++)
            *buf++ = i               ;
          break                      ;

        case 1                       : // 256 bytes 0xFF to 0x00
          for(i = 0; i < 256; i++)
            *buf++ = 255 - i         ;
          break                      ;

        case 2                       : // 256 bytes 0x00
          Memset(buf,0x00,256)       ;
          break                      ;

        case 3                       : // 256 bytes 0xAA
          Memset(buf,0xAA,256)       ;
          break                      ;

        case 4                       : // 256 bytes 0xFF
          Memset(buf,0xFF,256)       ;
          break                      ;
      }
  }
```