```
/*
**  Myapp.c - Sample code for Nucleo STM32L152RE =============== 08/10/18 =  **
**                                                                           **
**  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 nterface, **
**  including remote control unit RCU) and a DAC (Digital to Analog          **
**  converter) driver. Those five drivers are run by the VMIO sub-OS         **
**                                                                           **
**  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 **
**  KBDITF driver has one single logical way for all "keyboard" events,      **
**  including RCU. The DAC driver has one logical way per DAC device.        **
**                                                                           **
**  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 "dvmos.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 "vmos" and the address of the procedure "init_vmk_fsm". The second  **
**  entry contains the name "app" and the address of procedure "init_myapp"  **
**  At the very end of its initialization, the VMK sub-OS starts the         **
**  tasks groups, it calls the group start procedure, so in our case the     **
**  "init_vmk_fsm" procedure is called and the the "init_myapp" is called    **
**  Those call are done using the VMK stack, its address and size are        **
**  definied uin the link command file :                                     **
**  RAM_VMK_STACK        : ORIGIN = 0x2000C800 , LENGTH = 2K                 **
**                                                                           **
**  The first task group "vmos" contains one and only one task which task_id **
**  is 0. This task is systematically created at VMK initialisation. The     **
**  address and size are defined in the command link file:                   **
**  RAM_VMOS_STACK       : ORIGIN = 0x2000D000 , LENGTH = 2K                 **
**  The C treatments of all the VMOS fsm's are executed in task 0, so they   **
**  use the task 0 stack. With this simple demp 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 "app" contains all the user tasks, the taskid'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_myapp" 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 "mytask_proc" 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    asy_modd | ASY\DEV0   asy_iod0 |           | USB cable (*)   |  **
**  |                 | ASY\DEV1   asy_iod1 |           | ASY1_RX ASY1_TX |  **
**  +-----------------+---------------------+-----------+-----------------+  **
**  | GPIO  gpio_modd | GPIO\DEV0  gpio_iod | gpio_iod0 | Switch 1        |  **
**  |                 |                     | gpio_iod1 | Switch 2        |  **
**  |                 |                     | gpio_iod2 | Switch 3        |  **
**  +-----------------+---------------------+-----------+-----------------+  **
**  | LED    led_modd | LED\DEV0    led_iod | led_iod0  | RED Led         |  **
**  |                 |                     | led_iod1  | GREEN Led       |  **
**  |                 |                     | led_iod2  | YELLOW Led      |  **
**  +-----------------+---------------------+-----------+-----------------+  **
**  | KBDITF kbd_modd | KDBIDF\DEV0 kbd_iod |           | Infrared Rcv    |  **
**  +-----------------+---------------------+-----------+-----------------+  **
**  | DAC    dac_modd | 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_iod1, asy_iod1, 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-cahnnel (gpio_iod0, gpio_iod1, **
**  gpio_iod2 for switch buttons and led_iod0, led_iod1, led_iod2 for Leds)  **
**                                                                           **
**  =======================================================================  **
  Menage dans ev_asy1_snd
*/


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

#include <stdtyp.l>
#include <moteur.h>                    /* vmk.c and vmio.c interface         */
#include <drv.h>                       /* drv.c interface                    */
#include <automaton.h>                 /* Automaton definitions.             */
#include <rdrv.h>                      /* drv.c tags interface               */
#include <memoire.h>                   /* mem_buf.c interface                */
#include <hypstring.h>                 /* memcpy strcy .. interface          */
#include <text.h>                      /* Prototype hsprintf().              */
```

```
#include <keycode.h>                   /* Key codes from RCU                 */
#include <telecom.h>                   /* Constants and definitions for tags.*/
#include <engine.h>                    /* Constants for automatong engine.   */
#include <keym.txt>                    /* Applicative key codes              */
#include <systemm.txt>                 /* WAIT_CODERES                       */
#include <vmkm.txt>                    /* TSK_INIT_DATA TASK_INIT_BSS        */


/* Internal data types of the module ........................................*/

static int     usnd_start       (int                                        ) ;
static int     usnd_stop        (int                                        ) ;
static int     usnd_resp        (int                                        ) ;
static int     usnd_end         (int                                        ) ;
static void    req_snd_start    (int, unsigned int, unsigned char*, int     ) ;
static void    req_snd_stop     (int                                        ) ;

static int     mytask_proc      (void*                                      ) ;
static int     pg_main          (int                                        ) ;
static int     pg_asy_opts      (int                                        ) ;
static int     pg_asy_baud      (int                                        ) ;
static int     pg_asy_bits      (int                                        ) ;
static int     pg_asy_stop      (int                                        ) ;
static int     pg_asy_parity    (int                                        ) ;
static int     pg_asy_flowctl   (int                                        ) ;
static int     pg_asy_flushcnd  (int                                        ) ;
static int     pg_asy_tempo     (int                                        ) ;
static int     myfree_proc      (void                                       ) ;

static void    open_asy         (void                                       ) ;
static void    close_asy        (void                                       ) ;
static void    give_asy_token   (int,int,int,int*                           ) ;
static void    write_asy        (int,unsigned char*,int                     ) ;
static void    set_asy          (int,const char*                            ) ;
static void    send_asy_msg     (int,unsigned char*,int,int*                ) ;
static void    open_gpio        (void                                       ) ;
static void    close_gpio       (void                                       ) ;
static void    open_led         (void                                       ) ;
static void    close_led        (void                                       ) ;
static void    set_led_state    (int, const char *                          ) ;
static void    open_kbd         (void                                       ) ;
static void    close_kbd        (void                                       ) ;
static void    open_dac         (void                                       ) ;
static void    close_dac        (void                                       ) ;

static void    print_menu       (void                                       ) ;
static void    print_time       (void                                       ) ;
static void    print_status     (void                                       ) ;
static void    print_asyconfig  (void                                       ) ;
static void    print_cmd        (int                                        ) ;

static int     ev_asy0_rcv      (void                                       ) ;
static int     ev_asy0_snd      (void                                       ) ;
static int     ev_asy1_rcv      (void                                       ) ;
static int     ev_asy1_snd      (void                                       ) ;
static int     ev_gpio0_in      (void                                       ) ;
static int     ev_gpio1_in      (void                                       ) ;
static int     ev_gpio2_in      (void                                       ) ;
static int     ev_kbd_rcv       (void                                       ) ;
static int     ev_msec          (void                                       ) ;

static void    wait_coderes     (int,int                                    ) ;
static int     wait_myevents    (void                                       ) ;


/* Internal defines of this module ------------------------------------------*/

#define  AUT_USND                  20  // Automaton number
```

```
#define  VLASY                      0  // Logical way for serial
#define  VLDAC                      1  // Logical way for DAC
#define  VLNB                       2  // Number of Logical ways of AUT_USND

#define  TICK                   10000  // Code for tick event

#define  PAGE_MAIN                  0  // Main menu
#define  PAGE_ASY_OPTS              1  // Serial options list menu
#define  PAGE_ASY_BAUD              2  // Baudrate config menu
#define  PAGE_ASY_BITS              3  // Number of bits config menu
#define  PAGE_ASY_STOP              4  // Stop bits config menu
#define  PAGE_ASY_PARITY            5  // Parity config
#define  PAGE_ASY_FLOWCTL           6  // Flow control config
#define  PAGE_ASY_FLUSHCND          7  // Flush condition config
#define  PAGE_ASY_TEMPO             8  // Temporization config
#define  NB_PAGES                   9  // Number of menu pages

#define  EV_ASY0_RCV                0  // Incoming data on ASY\DEV0
#define  EV_ASY1_RCV                1  // Incoming data on ASY\DEV1
#define  EV_ASY0_SND                2  // Sent data on ASY\DEV0
#define  EV_ASY1_SND                3  // Sent data on ASY\DEV1
#define  EV_GPIO0_IN               10  // GPIO input pin state change
#define  EV_GPIO1_IN               11  // GPIO input pin state change
#define  EV_GPIO2_IN               12  // GPIO input pin state change
#define  EV_KBD_RCV                20  // Key press or release event
#define  EV_MSEC                   30  // "msec" milliseconds

#define  OPT_0                      0  // Menu option 0
#define  OPT_1                      1  // Menu option 1
#define  OPT_2                      2  // Menu option 2
#define  OPT_3                      3  // Menu option 3
#define  OPT_4                      4  // Menu option 4
#define  OPT_5                      5  // Menu option 5
#define  OPT_6                      6  // Menu option 6
#define  OPT_7                      7  // Menu option 7
#define  OPT_8                      8  // Menu option 8
#define  OPT_9                      9  // Menu option 9
#define  OPT_BUT1                  10  // Button 1
#define  OPT_BUT2                  11  // Button 2
#define  OPT_BUT3                  12  // Button 3
#define  OPT_TIME                 100  // Hidden option (timeout)
#define  OPT_INVALID               -1  // Invalid menu option
#define  OPT_IGNORE                -2  // Simply ignore

#define  TRT_IGNORE                -1  // Do nothing

#define  TRT_TOGGLE_RED             0  // Start/Stop pattern on RED
#define  TRT_TOGGLE_RED_UPD         1  // Start/Stop pattern on RED, print
#define  TRT_TOGGLE_GREEN           2  // Start/Stop pattern on GREEN
#define  TRT_TOGGLE_GREEN_UPD       3  // Start/Stop pattern on GREEN, print
#define  TRT_TOGGLE_YELLOW          4  // Start/Stop pattern on YELLOW
#define  TRT_TOGGLE_YELLOW_UPD      5  // Start/Stop pattern on YELLOW, print
#define  TRT_TOGGLE_SERIAL_UPD      6  // Start/Stop message on serial
#define  TRT_TOGGLE_DAC_UPD         7  // Start/Stop stream on DAC

#define  TRT_RESTART               10  // Warm restart
#define  TRT_PRINT_TIME            11  // Refresh time

#define  TRT_PAGE_MAIN             20  // Display main page
#define  TRT_PAGE_ASY_OPTS         21  // Display serial line options list
#define  TRT_PAGE_ASY_BAUD         22  // Display baudrate config page
#define  TRT_PAGE_ASY_BITS         23  // Display nb of bits config page
#define  TRT_PAGE_ASY_STOP         24  // Display stop bits config page
#define  TRT_PAGE_ASY_PARITY       25  // Display parity config page
#define  TRT_PAGE_ASY_FLOWCTL      26  // Display flow control config page
#define  TRT_PAGE_ASY_FLUSHCND     27  // Display flush condition config page
#define  TRT_PAGE_ASY_TEMPO        28  // Display temporization config page
```

```
#define  TRT_BAUD_9600             30  // Set BAUD=9600
#define  TRT_BAUD_38400            31  // Set BAUD=38400
#define  TRT_BAUD_57600            32  // Set BAUD=57600
#define  TRT_BAUD_19200            33  // Set BAUD=19200
#define  TRT_BAUD_115200           34  // Set BAUD=115200
#define  TRT_6BITS                 35  // Set NBBITS=6
#define  TRT_7BITS                 36  // Set NBBITS=7
#define  TRT_8BITS                 37  // Set NBBITS=8
#define  TRT_1STOP                 38  // Set STOPBIT=1
#define  TRT_2STOP                 39  // Set STOPBIT=2
#define  TRT_NOPARITY              40  // Set PARITY=NONE
#define  TRT_EVENPARITY            41  // Set PARITY=EVEN
#define  TRT_ODDPARITY             42  // Set PARITY=ODD
#define  TRT_TOGGLE_ASYDMA         43  // Change DMA (BUFSIZE=0/64)
#define  TRT_TOGGLE_ASYSPEED       44  // Change speed mode (normal/slow)
#define  TRT_NOFLOWCTL             45  // Flow control: none
#define  TRT_RTSCTS                46  // Flow control: RTS/CTS
#define  TRT_XONXOFF               47  // Flow control: XON/XOFF
#define  TRT_FULLFLUSH             48  // Timing config: none
#define  TRT_TEMPOCHAR             49  // Timing config: CHARACTER
#define  TRT_TEMPOFRAME            50  // Timing config: FRAME
#define  TRT_TEMPO_1MS             51  // Timing: 1 ms
#define  TRT_TEMPO_5MS             52  // Timing: 5 ms
#define  TRT_TEMPO_10MS            53  // Timing: 10 ms
#define  TRT_TEMPO_50MS            54  // Timing: 50 ms
#define  TRT_TEMPO_100MS           55  // Timing: 100 ms
#define  TRT_TEMPO_150MS           56  // Timing: 150 ms
#define  TRT_TEMPO_200MS           57  // Timing: 200 ms
#define  TRT_TEMPO_500MS           58  // Timing: 500 ms
#define  TRT_TEMPO_1000MS          59  // Timing: 1000 ms

#define  CFG_9600                   0  // Baudrate: 9600 bauds
#define  CFG_19200                  1  // Baudrate: 19200 bauds
#define  CFG_38400                  2  // Baudrate: 38400 bauds
#define  CFG_57600                  3  // Baudrate: 57600 bauds
#define  CFG_115200                 4  // Baudrate: 115200 bauds

#define  CFG_6BITS                  0  // Nb bits: 6 bits/sym
#define  CFG_7BITS                  1  // Nb bits: 7 bits/sym
#define  CFG_8BITS                  2  // Nb bits: 8 bits/sym

#define  CFG_1STOP                  0  // Stop bits: 1 stop bit
#define  CFG_2STOP                  1  // Stop bits: 2 stop bits

#define  CFG_NOPARITY               0  // Parity: none
#define  CFG_EVENPARITY             1  // Parity: even
#define  CFG_ODDPARITY              2  // Parity: odd

#define  CFG_DMAOFF                 0  // DMA: OFF
#define  CFG_DMAON                  1  // DMA: ON

#define  CFG_NOFLOWCTL              0  // Flow control: none
#define  CFG_RTSCTS                 1  // Flow control: RTS/CTS
#define  CFG_XONXOFF                2  // Flow control: XON/XOFF

#define  CFG_FULLFLUSH              0  // Timing config: none
#define  CFG_TEMPOCHAR              1  // Timing config: CHARACTER
#define  CFG_TEMPOFRAME             2  // Timing config: FRAME

#define  CFG_1MS                    0  // Timing: 1 ms
#define  CFG_5MS                    1  // Timing: 5 ms
#define  CFG_10MS                   2  // Timing: 10 ms
#define  CFG_50MS                   3  // Timing: 50 ms
#define  CFG_100MS                  4  // Timing: 100 ms
#define  CFG_150MS                  5  // Timing: 150 ms
#define  CFG_200MS                  6  // Timing: 200 ms
#define  CFG_500MS                  7  // Timing: 500 ms
#define  CFG_1000MS                 8  // Timing: 1000 ms
```

```
#define  CFG_FAST                   0  // Speed: normal
#define  CFG_SLOW                   1  // Speed: slow


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

typedef struct                         // AUT_USND Context structure
  {                                    // -----------------------------------
    unsigned short  emet             ; // Sender of SND_START/SND_STOP
    unsigned short  evl              ; // "
    unsigned short  nfa              ; // Internal Queue for response

    unsigned int    iod              ; // IOD for drv_write
    unsigned char  *buf              ; // Buffer address
    int             buflg            ; // Number of bytes in buffer

    int             nbreq            ; // Nomber of ongoing REQ_WRITE's
  }                 Usnd_ctx         ; // -----------------------------------

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                 mystack[512]     ; // The stack of our task (2kB)
int                 myapp_taskid     ; // Our task id
int                 myapp_memuid     ; // Our memory user id

int                 idto             ; // Timer identifier

int                 asy_modd         ; // MODD of the ASY driver
int                 asy_iod0         ; // IOD of devive ASY\DEV0
int                 asy_iod1         ; // IOD of device ASY\DEV1
int                 asy_rteid        ; // Route ID for ASY IND_REPORT evts

unsigned char       *buf_snd0        ; // Address of DEV0 transmit buffer
unsigned char       *buf_rcv0        ; // Address of DEV0 received buffer
int                 tok_rcv0 = 0     ; // Number of receive tokens for DEV0

unsigned char       *buf_snd1        ; // Address of DEV1 transmit buffer
unsigned char       *buf_rcv1        ; // Address of DEV1 received buffer
int                 tok_rcv1 = 0     ; // Number of receive tokens for DEV1
int                 msg_snd1 = 0     ; // Number of sending messages for DEV1

int                 gpio_modd        ; // MODD of the GPIO driver
int                 gpio_iod         ; // IOD of device GPIO\DEV0
int                 gpio_iod0        ; // IOD of subchannel 0 of GPIO\DEV0
int                 gpio_iod1        ; // IOD of subchannel 1 of GPIO\DEV0
int                 gpio_iod2        ; // IOD of subchannel 2 of GPIO\DEV0
int                 gpio_rteid       ; // Route ID for GPIO IND_REPORT evts

int                 led_modd         ; // MODD of the LED driver
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_modd         ; // MODD of the KBDITF driver
int                 kbd_iod          ; // IOD of device KBDITF\DEV0
int                 kbd_rteid        ; // Route ID for DRV_KEY_PRESS/RELEASE

int                 dac_modd         ; // MODD of the DAC driver
int                 dac_iod          ; // IOD of device DAC\DEV0
```

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

unsigned int        page             ; // Current page
unsigned char       state [5]        ; // State of menu variables
unsigned char       config[9]        ; // Current serial configuration


/* 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
#define STARTED        1               // Sending is running
#define STOPPING       2               // SND_STOP has been received
#define STATENB        3               // 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 transmissions are 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 *
 *                                                                           *
 * We have 3 possible states. For each state, we describe the events that    *
 * may be received.                                                          *
 *                                                                           *
 * - 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 repeatidly send 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 "drv_write".   *
 *                           Those calls are non blocking since the device   *
 *                           has been allocated (drv_alloc_device) using the *
 *                           NBLOCKING_IO option bit. So the device device   *
 *                           immediately (0 latency) two REQ_WRITE event     *
 *                           requests and transmition corrresponding 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.                                          *
 *                           ----------------------------------------------- *
 *             . SND_STOP    This event is a request to stop sending. It     *
 *                           never should be received in the present state   *
 *                           since we did not yet received the SND_START.    *
 *                                                                           *
 * - STARTED   The same data buffer is sent again and again. Each time the   *
 *             VMIO driver send us a RESP_WRITE, we issue a new "drv_write", *
 *             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 respnse 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 "drv_write" (the just started       *
 *                           message transmission duration). The "usnd_resp" *
 *                           treatment is called, it issues a call to the    *
 *                           "drv_write" function. A REQ_WRITE is (0 latency)*
 *                           written in the REQ_WRITE driver's internal      *
 *                           queue for write requests. 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    *
 *                           never should be received in the present state   *
 *                           but only from the STOPPED state.                *
 *                           ----------------------------------------------- *
 *             . SND_STOP    This event is a request to stop sending. It     *
 *                           never should 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" valeurs 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.                                         *
 *****************************************************************************/

/* STOPPED (0) State : Waiting for the SND_START  event .....................*/
#define L_STOPPED  2

SND_START   , usnd_start  , 0     , STARTED , // Request to start transmitting
SND_STOP    , trien       , 0     , STOPPED , // Incorrect request here

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

SND_START   , trien       , 0     , STARTED , // Incorrect request hete
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 hete
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

  }                                  ;


static USHORT const lim_usnd[] =
  {
    0            , L_STOPPED    , L_STARTED , L_STOPPING
  }                                  ;



/* Static data for the user task --------------------------------------------*/

static const char *menu [NB_PAGES] = {

// Main page
"\x1B[2J" // Clear screen
"\x1B[H"  // Move to top left corner
"  +==============================================================+\n\r"
"  |                                                              |\n\r"
"  |   STM32 L152RE Nucleo Myapp                       XX:XX:XX   |\n\r"
"  |                                                              |\n\r"
"  |   Main menu                                                  |\n\r"
"  |                                                              |\n\r"
"  |   +---------------------------------------+  +-----------+   |\n\r"
"  |   | 1  Start/Stop pattern on RED          |  |  -------  |   |\n\r"
"  |   | 2  Start/Stop pattern on GREEN        |  |  -------  |   |\n\r"
"  |   | 3  Start/Stop pattern on YELLOW       |  |  -------  |   |\n\r"
"  |   | 4  Start/Stop serial line ASY1        |  |  -------  |   |\n\r"
"  |   | 5  Start/Stop audio DAC               |  |  -------  |   |\n\r"
"  |   |                                       |  +-----------+   |\n\r"
"  |   | 6  Configure serial line              |                  |\n\r"
"  |   |                                       |                  |\n\r"
"  |   |                                       |                  |\n\r"
"  |   |                                       |                  |\n\r"
"  |   | 0  Restart                            |                  |\n\r"
"  |   +---------------------------------------+                  |\n\r"
"  |                                                              |\n\r"
"  |   Enter command :                                            |\n\r"
"  |                                                              |\n\r"
"  +==============================================================+\n\r"
"\x1B[21;23H"                                                           ,

// Serial line options list
```

```
"\x1B[H"  // Move to top left corner
"  +==============================================================+\n\r"
"  |                                                              |\n\r"
"  |   STM32 L152RE Nucleo Myapp                       XX:XX:XX   |\n\r"
"  |                                                              |\n\r"
"  |   Configure serial line ASY1                                 |\n\r"
"  |                                                              |\n\r"
"  |   +---------------------------------------+  +-----------+   |\n\r"
"  |   | 1  Set baudrate                       |  |  -------  |   |\n\r"
"  |   | 2  Set number of bits per symbol      |  |  -------  |   |\n\r"
"  |   | 3  Set number of stop bits            |  |  -------  |   |\n\r"
"  |   | 4  Set parity bit                     |  |  -------  |   |\n\r"
"  |   | 5  Enable/Disable DMA                 |  |  -------  |   |\n\r"
"  |   | 6  Set flow control                   |  |  -------  |   |\n\r"
"  |   | 7  Set flush mode                     |  |  -------  |   |\n\r"
"  |   | 8  Set flush temporization            |  |  -------  |   |\n\r"
"  |   | 9  Toggle reception speed             |  |  -------  |   |\n\r"
"  |   |                                       |  +-----------+   |\n\r"
"  |   | 0  Back to main menu                  |                  |\n\r"
"  |   +---------------------------------------+                  |\n\r"
"  |                                                              |\n\r"
"  |   Enter command :                                            |\n\r"
"  |                                                              |\n\r"
"  +==============================================================+\n\r"
"\x1B[21;23H"                                                            ,

// Serial baudrate configuration
"\x1B[H"  // Move to top left corner
"  +==============================================================+\n\r"
"  |                                                              |\n\r"
"  |   STM32 L152RE Nucleo Myapp                       XX:XX:XX   |\n\r"
"  |                                                              |\n\r"
"  |   Select baudrate                                            |\n\r"
"  |                                                              |\n\r"
"  |   +---------------------------------------+                  |\n\r"
"  |   | 1  9600 bauds                         |                  |\n\r"
"  |   | 2  19200 bauds                        |                  |\n\r"
"  |   | 3  38400 bauds                        |                  |\n\r"
"  |   | 4  57600 bauds                        |                  |\n\r"
"  |   | 5  115200 bauds                       |                  |\n\r"
"  |   |                                       |                  |\n\r"
"  |   |                                       |                  |\n\r"
"  |   |                                       |                  |\n\r"
"  |   |                                       |                  |\n\r"
"  |   |                                       |                  |\n\r"
"  |   | 0  Back to serial line options        |                  |\n\r"
"  |   +---------------------------------------+                  |\n\r"
"  |                                                              |\n\r"
"  |   Enter command :                                            |\n\r"
"  |                                                              |\n\r"
"  +==============================================================+\n\r"
"\x1B[21;23H"                                                            ,

// Serial line number of bits per symbol configuration
"\x1B[H"  // Move to top left corner
"  +==============================================================+\n\r"
"  |                                                              |\n\r"
"  |   STM32 L152RE Nucleo Myapp                       XX:XX:XX   |\n\r"
"  |                                                              |\n\r"
"  |   Select number of bits per symbol                           |\n\r"
"  |                                                              |\n\r"
"  |   +---------------------------------------+                  |\n\r"
"  |   | 1  6 bits per symbol                  |                  |\n\r"
"  |   | 2  7 bits per symbol                  |                  |\n\r"
"  |   | 3  8 bits per symbol                  |                  |\n\r"
"  |   |                                       |                  |\n\r"
"  |   |                                       |                  |\n\r"
"  |   |                                       |                  |\n\r"
"  |   |                                        |                  |\n\r"
"  |   |                                        |                  |\n\r"
"  |   |                                        |                  |\n\r"
"  |   |                                        |                  |\n\r"
"  |   | 0  Back to list of options             |                  |\n\r"
"  |   +----------------------------------------+                  |\n\r"
"  |                                                               |\n\r"
"  |   Enter command :                                             |\n\r"
"  |                                                               |\n\r"
"  +===============================================================+\n\r"
"\x1B[21;23H"                                                            ,

// Serial line stop bits configuration
"\x1B[H"  // Move to top left corner
"  +===============================================================+\n\r"
"  |                                                               |\n\r"
"  |   STM32 L152RE Nucleo Myapp                       XX:XX:XX    |\n\r"
"  |                                                               |\n\r"
"  |   Select number of stop bits                                  |\n\r"
"  |                                                               |\n\r"
"  |   +----------------------------------------+                  |\n\r"
"  |   | 1  1 stop bit                          |                  |\n\r"
"  |   | 2  2 stop bits                         |                  |\n\r"
"  |   |                                        |                  |\n\r"
"  |   |                                        |                  |\n\r"
"  |   |                                        |                  |\n\r"
"  |   |                                        |                  |\n\r"
"  |   |                                        |                  |\n\r"
"  |   |                                        |                  |\n\r"
"  |   |                                        |                  |\n\r"
"  |   |                                        |                  |\n\r"
"  |   | 0  Back to list of options             |                  |\n\r"
"  |   +----------------------------------------+                  |\n\r"
"  |                                                               |\n\r"
"  |   Enter command :                                             |\n\r"
"  |                                                               |\n\r"
"  +===============================================================+\n\r"
"\x1B[21;23H"                                                            ,

// Serial line parity configuration
"\x1B[H"  // Move to top left corner
"  +===============================================================+\n\r"
"  |                                                               |\n\r"
"  |   STM32 L152RE Nucleo Myapp                       XX:XX:XX    |\n\r"
"  |                                                               |\n\r"
"  |   Select parity                                               |\n\r"
"  |                                                               |\n\r"
"  |   +----------------------------------------+                  |\n\r"
"  |   | 1  No parity                           |                  |\n\r"
"  |   | 2  Even parity                         |                  |\n\r"
"  |   | 3  Odd parity                          |                  |\n\r"
"  |   |                                        |                  |\n\r"
"  |   |                                        |                  |\n\r"
"  |   |                                        |                  |\n\r"
"  |   |                                        |                  |\n\r"
"  |   |                                        |                  |\n\r"
"  |   |                                        |                  |\n\r"
"  |   |                                        |                  |\n\r"
"  |   | 0  Back to list of options             |                  |\n\r"
"  |   +----------------------------------------+                  |\n\r"
"  |                                                               |\n\r"
"  |   Enter command :                                             |\n\r"
"  |                                                               |\n\r"
"  +===============================================================+\n\r"
"\x1B[21;23H"                                                            ,

// Serial line flow control configuration
"\x1B[H"  // Move to top left corner
```

```
"  +==============================================================+\n\r"
"  |                                                              |\n\r"
"  |   STM32 L152RE Nucleo Myapp                       XX:XX:XX   |\n\r"
"  |                                                              |\n\r"
"  |   Select flow control mode                                   |\n\r"
"  |                                                              |\n\r"
"  |   +---------------------------------------+                  |\n\r"
"  |   | 1  No flow control                    |                  |\n\r"
"  |   | 2  Flow control with RTS/CTS          |                  |\n\r"
"  |   | 3  Flow control with XON/XOFF         |                  |\n\r"
"  |   |                                       |                  |\n\r"
"  |   |                                       |                  |\n\r"
"  |   |                                       |                  |\n\r"
"  |   |                                       |                  |\n\r"
"  |   |                                       |                  |\n\r"
"  |   |                                       |                  |\n\r"
"  |   |                                       |                  |\n\r"
"  |   | 0  Back to list of options            |                  |\n\r"
"  |   +---------------------------------------+                  |\n\r"
"  |                                                              |\n\r"
"  |   Enter command :                                            |\n\r"
"  |                                                              |\n\r"
"  +==============================================================+\n\r"
"\x1B[21;23H"                                                           ,

// Serial line reception flush configuration
"\x1B[H"  // Move to top left corner
"  +==============================================================+\n\r"
"  |                                                              |\n\r"
"  |   STM32 L152RE Nucleo Myapp                       XX:XX:XX   |\n\r"
"  |                                                              |\n\r"
"  |   Select reception flush mode                                |\n\r"
"  |                                                              |\n\r"
"  |   +---------------------------------------+                  |\n\r"
"  |   | 1  When buffer is full only           |                  |\n\r"
"  |   | 2  Character temporization            |                  |\n\r"
"  |   | 3  Frame temporization                |                  |\n\r"
"  |   |                                       |                  |\n\r"
"  |   |                                       |                  |\n\r"
"  |   |                                       |                  |\n\r"
"  |   |                                       |                  |\n\r"
"  |   |                                       |                  |\n\r"
"  |   |                                       |                  |\n\r"
"  |   |                                       |                  |\n\r"
"  |   | 0  Back to list of options            |                  |\n\r"
"  |   +---------------------------------------+                  |\n\r"
"  |                                                              |\n\r"
"  |   Enter command :                                            |\n\r"
"  |                                                              |\n\r"
"  +==============================================================+\n\r"
"\x1B[21;23H"                                                           ,

// Serial line reception timing configuration
"\x1B[H"  // Move to top left corner
"  +==============================================================+\n\r"
"  |                                                              |\n\r"
"  |   STM32 L152RE Nucleo Myapp                       XX:XX:XX   |\n\r"
"  |                                                              |\n\r"
"  |   Select reception temporization duration                    |\n\r"
"  |                                                              |\n\r"
"  |   +---------------------------------------+                  |\n\r"
"  |   | 1  1 ms                               |                  |\n\r"
"  |   | 2  5 ms                               |                  |\n\r"
"  |   | 3  10 ms                              |                  |\n\r"
"  |   | 4  50 ms                              |                  |\n\r"
"  |   | 5  100 ms                             |                  |\n\r"
"  |   | 6  150 ms                             |                  |\n\r"
"  |   | 7  200 ms                             |                  |\n\r"
"  |   | 8  500 ms                             |                  |\n\r"
"  |   | 9  1000 ms                            |                  |\n\r"
"  |   |                                       |                  |\n\r"
"  |   | 0  Back to list of options            |                  |\n\r"
"  |   +---------------------------------------+                  |\n\r"
"  |                                                              |\n\r"
"  |   Enter command :                                            |\n\r"
"  |                                                              |\n\r"
"  +==============================================================+\n\r"
"\x1B[21;23H"                                                            ,

} ;
static const char loc_save[]         = "\x1Bs"                              ;
static const char loc_restore[]      = "\x1Bu"                              ;
static const char loc_time[]         = "\x1B[3;55H"                         ;
static const char *loc_cmd[NB_PAGES] = { "\x1B[21;23H", "\x1B[21;23H",
                                         "\x1B[21;23H", "\x1B[21;23H",
                                         "\x1B[21;23H", "\x1B[21;23H",
                                         "\x1B[21;23H", "\x1B[21;23H",
                                         "\x1B[21;23H"                 }    ;
static const char *loc_sta[5]        = { "\x1B[8;53H" , "\x1B[9;53H" ,
                                         "\x1B[10;53H", "\x1B[11;53H",
                                         "\x1B[12;53H"                 }    ;
static const char *loc_cfg[9]        = { "\x1B[8;53H" , "\x1B[9;53H" ,
                                         "\x1B[10;53H", "\x1B[11;53H",
                                         "\x1B[12;53H", "\x1B[13;53H",
                                         "\x1B[14;53H", "\x1B[15;53H",
                                         "\x1B[16;53H"                 }    ;

static const char *str_sta []        = { "STOPPED", "STARTED"             } ;

static const char *str_cfg0[]        = { "   9600", "  19200", "  38400",
                                         "  57600", " 115200",            } ;
static const char *str_cfg1[]        = { " 6 BITS", " 7 BITS", " 8 BITS"  } ;
static const char *str_cfg2[]        = { " 1 STOP", " 2 STOP"             } ;
static const char *str_cfg3[]        = { " NO PAR", "   EVEN", "    ODD"  } ;
static const char *str_cfg4[]        = { "DMA OFF", " DMA ON"             } ;

static const char *str_cfg5[]        = { "NO FCTL", "RTS/CTS", "XON/OFF"  } ;
static const char *str_cfg6[]        = { "   FULL", "   CHAR", "  FRAME"  } ;
static const char *str_cfg7[]        = { "   1 MS", "   5 MS", "  10 MS",
                                         "  50 MS", " 100 MS", " 150 MS",
                                         " 200 MS", " 500 MS", "1000 MS"  } ;

static const char *str_cfg8[]        = { " NORMAL", "   SLOW"             } ;
static const char **str_cfg[9]       = {  str_cfg0,  str_cfg1,  str_cfg2,
                                          str_cfg3,  str_cfg4,  str_cfg5,
                                          str_cfg6,  str_cfg7,  str_cfg8  } ;



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 *asy_tagbauds [] = { "BAUDS=9600"  , "BAUDS=19200" ,
                                       "BAUDS=38400" , "BAUDS=57600" ,
                                       "BAUDS=115200"                     } ;
static const char *asy_tagnbbits[] = { "NBBITS=6"    , "NBBITS=7"    ,
                                       "NBBITS=8"                         } ;
static const char *asy_tagstop  [] = { "STOPBIT=1"   , "STOPBIT=2"        } ;
static const char *asy_tagparity[] = { "PARITY=NONE" , "PARITY=EVEN" ,
                                       "PARITY=ODD"                       } ;
static const char *asy_tagdma   [] = { "BUFSIZE=64"  , "BUFSIZE=0"        } ;
static const char *asy_tagflow  [] = { "FLOWCTL=NONE", "FLOWCTL=RTS/CTS",
                                       "FLOWCTL=XON/XOFF"                 } ;
static const char *asy_tagtmode [] = { "TMODE=NONE"  , "TMODE=CHARACTER",
                                       "TMODE=FRAME"                     } ;
static const char *asy_tagtempo [] = { "TEMPO=1"     , "TEMPO=5"        ,
                                       "TEMPO=10"    , "TEMPO=50"       ,
                                       "TEMPO=100"   , "TEMPO=150"      ,
                                       "TEMPO=200"   , "TEMPO=500"      ,
                                       "TEMPO=1000"                      } ;


static const char *gpio_taglist = "FUNC=GPIO\nOUTPUT=OFF\nWEAKPULL=UP\n"
                                  "EVENTS=BOTH\nDEBOUNCE=50"                  ;

static const char *led_tagpop   = "CMD=POPALL"                                ;
static const char *led_tagon0   = "PATTERN=ON:5,OFF:5\nCMD=PUSH"              ;
static const char *led_tagon1   = "PATTERN=ON:2,OFF:8\nCMD=PUSH"              ;
static const char *led_tagon2   = "PATTERN=ON:8,OFF:2\nCMD=PUSH"              ;
static const char *led_tagoff   = "PATTERN=OFF:0\nCMD=PUSH"                   ;

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

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

init_vmk_fsm        VLK Finite State Machine's creation function
init_myapp          VMK Task creation function

usnd_start          SND_START event treatment
usnd_stop           SND_STOP  event treatment
usnd_resp           RESP_WRITE event treatment
usnd_end            C_END condition treatment
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

mytask_proc         Entry point for create_task - Main event loop
pg_main             Action selection for main menu mage
pg_asy_opt          Action selection for serial line option page
pg_asy_baud         Action selection for serial baud rate selection page
pg_asy_bits         Action selection for serial bits/char selection page
pg_asy_stop         Action selection for serial stop bits selection page
pg_asy_parity       Action selection for serial parity bits selection page
myfree_proc         Termination function for create_task

open_asy            Open of two serial ports
close_asy           Close of two serial ports
give_asy_token      Give one or more receive token to ASY driver
write_asy           Blocking transmission on one serial line
set_asy             Set a parameter of one serial line
send_asy_msg        Non blocking sending of one serial message
open_gpio           Open the GPIO driver
close_gpio          Close the GPIO driver
open_led            Open the LED driver
close_led           Close the LED driver
set_led_state       Defines a new blinking pattern for one LED
open_kbd            Open the KBDITF driver
close_kbd           Close the KBDITF driver
open_dac            Open the DAC driver
close_dac           Close the DAC driver

print_menu          Print the menu on ASY\DEV0
print_time          Print the time in seconds
print_status        Print the 9 status strings
print_asyconfig     Print the 9 config parameters of serial line 1
print_cmd           Print the last selected option

ev_asy0_rcv         EV_ASY0_RCV  has been received - Determine command
ev_asy0_snd         EV_ASY0_SND  has been received - Determine command
ev_asy1_rcv         EV_ASY1_RCV  has been received - Determine command
ev_asy1_snd         EV_ASY1_SND  has been received - Determine command
ev_gpio0_in         EV_GPIO0_IN  has been received - Determine command
ev_gpio1_in         EV_GPIO1_IN  has been received - Determine command
```

```
ev_gpio2_in         EV_GPIO2_IN  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

wait_coderes        Wait one and only one event
wait_myevents       Wait for all my events

*/


#define SIZE_B_E 100
static UINT ind_b_e = 0;
typedef struct
  BEGIN
    UINT            et               ; /* Etiquette unique a chaque appel    */
    UINT            tp               ; /* Temps ecoule depuis le boot (ms)   */
    UINT            ev               ; /* Data 1 (ev)                        */
    UINT            op               ; /* Data 2 (opt)                       */
    UINT            pa               ; /* Data 3 (page)                      */
    UINT            tr               ; /* Data 74 (trt)                       */
  END               Elt_e            ;
Elt_e buf_e[SIZE_B_E];
static void  log_ev(UINT entete, UINT ev, UINT op, UINT pa, UINT tr)
  BEGIN                                /* Debut de cette procedure.          */
    if (!(entete & 0x01))
      BEGIN
        extern UINT temps               ;
        buf_e[ind_b_e].et = entete;
        buf_e[ind_b_e].tp = temps;
        buf_e[ind_b_e].ev = ev;
        buf_e[ind_b_e].op = op;
        buf_e[ind_b_e].pa = pa;
        buf_e[ind_b_e].tr = tr;
        ind_b_e++;
        if (ind_b_e >= SIZE_B_E)
          ind_b_e = 0;
        buf_e[ind_b_e].et = 1111111111;
      END_IF
  END_PROC                             /* Fin de la procedure.               */

/*  Procedure init_vmk_fsm ----------------------------------------------------

    Purpose : Creates one VMK Finite State Machine (automaton), then send
              one initialization event to it.
*/

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

/*****************************************************************************
 * Step 1 : Declare our FSM (automaton) to the VMK
 * ------
 *****************************************************************************/

    iniaut(AUT_USND   ,                // Automaton number
           trans_usnd ,                // Transition table address
           lim_usnd   ,                // Firt transition number 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 init_myapp --------------------------------------------------

    Purpose : Allocates a task context and an associated Logical Way of
              the AUT_TASK automaton, then requests the start of the task.
*/

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

/**********************************************************************
 * Step 1 : The "create_task" procedure initializes all the data that is     *
 * ------   necessary for a new task, it allocates a task context and a      *
 *          logical way of the AUT_TASK automaton. This LW will be used  to  *
 *          receive all the events for that task. The returned identifier    *
 *          "taskid" is in fact equal to 0x12FFFF00 | LW, where LW is a      *
 *          Logical way number of AUT_TASK, and also an index in the context *
 *          table "task_ctx[]". The input parameters of "create_task" are    *
 *          stored into "task_ctx[LW]" :                                     *
 *          - The task entry point is always a C procedure and here our      *
 *            procedure is "mytask_proc".                                    *
 *          - In the event that the task entry point procedure returns, the  *
 *            VMK will call a "clean-up" procedure we have to provide. Here  *
 *            our termination procedure is "myfree_proc".                    *
 *          - The stack address ("mystack") and its size in bytes.           *
 *          - The task Priority and the task initialization flags            *
 ***********************************************************************/

    create_task("app"           ,      // Taskgroup name
                mytask_proc     ,      // Task entry point
                myfree_proc     ,      // Tasm termination call-back
                mystack         ,      // Stack address
                sizeof(mystack) ,      // Stack size in bytes
                PRIO_TSK_MIN    ,      // Task Priority
                HNULL           ,      // Parameter for "mytask_proc"
                TASK_INIT_BSS  +       // Init Flags
                TASK_INIT_DATA +       //
                TASK_INIT_CODE  ,      //
                &myapp_taskid     )  ; // Task_id = 0x12FFFF00 + LC


/**********************************************************************
 * Step 2 : Here the new task has been created but this does not implies it  *
 * ------   starts running. A call to "start_task" has to be issued. This    *
 *          procedure sends an event (RUN_TASK) that is a request to start   *
 *          task execution. This event is sent to the "myapp_task_id & 0xFF" *
 *          logical way of automaton AUT_TASK.                               *
 ***********************************************************************/

    start_task(myapp_taskid,           // Send an event to VMK in order
               &ret         )       ; // to request the task start


/**********************************************************************
 * Step 3 : In order to be able to use the "alloc_buf" procedure one needs   *
 * ------   a user identifier from the allocation subsystem. We just have    *
 *          to make a call to "alloc_memuid", we get the identifier in the   *
 *          "myapp_memuid" global variable.                                  *
 ***********************************************************************/

    alloc_memuid(&myapp_memuid,        // Allocates a user id for alloc_buf
                 &ret         )     ; // and other allocation procedures
```

```
/*****************************************************************************
 * Step 4 : Initialize the driver/protocol/services interface. Once done,   *
 * -------  the VMK "app" can use the driver/protocole/services interface   *
 *          through procedures named "drv_***". Those procedures are        *
 *          sending REQ_xyz events to the VMIO automatons.                  *
 ****************************************************************************/

    init_telecom()                   ; // Initialize driver/protocol/services
                                       // API

    return 0                         ; // Exit without any error
  }


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

    Purpose : SND_START event treatment. We store the received event parameters
              in the context and we call "drv_write" 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 paramers 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 "voieelog" 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 adress of the "voielog"   *
 *            entry of the "usnd_ctx[]" table of context structures          *
 *          - All the events are containg 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 repeatidly sent. We store it in "ctx->buf"      *
 *            . The "longueur" field is the number of data bytes that are    *
 *              in the buffer, we store that count in "ctx->buflg".          *
 *          - In the next step we will do two writes, two calls to the       *
 *            "drv_write" procedure. We set "ctx->nbreq" to 2, this value    *
 *            is the count of RESP_WRITE event we are waiting for.           *
 *****************************************************************************/

    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

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


/***************************************************************************
 * Step 2 : We issue two write request, two calls to the "drv_write"        *
 * ------   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 "drv_write" 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 :         *
 *          - The first call to "drv_write" send 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 execution  *
 *            the code of the "drv_write" procedure.                         *
 *          - So when the first call to "drv_write" 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 "drv_write" 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 "drv_write" *
 *            sends a second REQ_WRITE that is written in the VMIO event     *
 *            input queue. This event writing immediately interrupts the     *
 *            code of "drv_write" code, the VMIO statck is installed and the *
 *            driver's REQ_WRITE handling treament 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 send.                                         *
 *         So just after we exit from the second call to "drv_write", 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.                    *
 ***************************************************************************/

   for(i = 0; i LT 2; i++)            // Send 2 write requests to the VMIO
     drv_write(ctx->iod   ,           // IOD of device
               ctx->buf   ,           // Buffer Address
               0          ,           // Buffer Offset
               ctx->buflg ,           // Number of bytes to be sent
               0          ,           // Flags
               1          ,           // Reqid
               &ret        )        ; // Return code
```

```
/******************************************************************************
 * Step 3 : 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             *
 *****************************************************************************/

    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, as set the "voielog" global variable 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 "drv_write" 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 the the "voielog"  *
 *            global variable the value of 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 "drv_write" 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 "drv_write"            *
 *          - The "drv_write" 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 "drv_write" code, the VMIO    *
 *            stack is installed and the driver's REQ_WRITE handling         *
 *            treament is immediately executed. But currently we already a   *
 *            frame beeing 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 execution of the        *
 *            "drv_write" code and then we return from "drv_write".          *
 *          So after returning from "drv_write", 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   *
 ******************************************************************************/

    drv_write(ctx->iod   ,             // IOD of device
              ctx->buf   ,             // Buffer Address
              0          ,             // Buffer Offset
              ctx->buflg ,             // Number of bytes to be sent
              0          ,             // Flags
              1          ,             // Reqid
              &ret        )          ; // Return code

    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" treament 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              ;

/******************************************************************************
 * 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 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.
*/

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

/******************************************************************************
 * Step 1 : Initialize the "evt" structure fields :                           *
 * ------
 ******************************************************************************/

    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               ; // adress  = telecom buffer address
    evt.longueur = lg                ; // lg      = Number of bytes in buffer


/******************************************************************************
 * Step 2 : Write in VMK internal waiting queue the event for the FSM         *
 * ------
 ******************************************************************************/

    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
 * ------
 ******************************************************************************/

    wait_coderes(R_SND_START,0)      ;

  }


/*  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              ; // Event structure

/******************************************************************************
 * Step 1 : Initialize the "evt" structure fields :                           *
 * ------
 ******************************************************************************/

    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         *
 * ------
 ******************************************************************************/

    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
 * ------
 ******************************************************************************/

    wait_coderes(R_SND_STOP,0)       ;

  }


/*  Procedure mytask_proc ------------------------------------------------------

    Purpose : This is our task main loop.
*/

static INT mytask_proc(void *param)
  {
    int             ev               ; // Our event
    int             opt              ; // Option selected by user
    int             trt              ; // Treatment to do

/******************************************************************************
 * 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.                            *
 ******************************************************************************/

    start                            :

    open_asy()                       ; // Open the two 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

    give_asy_token(asy_iod0  ,         // I/O descriptor
                   2         ,         // Number of receive token
                   1         ,         // Size of receive buffer
                   &tok_rcv0   )      ; // Counter of given tokens

    give_asy_token(asy_iod1 ,          // I/O descriptor
                   2        ,          // Number of receive token
                   8        ,          // Size of receive buffer
                   &tok_rcv1  )      ; // Counter of given tokens

/*****************************************************************************
 * Step 2 : We initialize printing state for menu, initial configuration of  *
 * ------   serial line and we print the menu through ASY\DEV0.             *
 *****************************************************************************/

    page = PAGE_MAIN                 ; // Select first page, main menu

    state[0]  =                        // Initiate states displayed
    state[1]  =                        // by the menu at start-up
    state[2]  =                        //
    state[3]  =                        //
    state[4]  = 0                    ; //

    config[0] = CFG_9600             ; // Baudrate: 9600 bauds
    config[1] = CFG_8BITS            ; // Nb of bits/sym: 8 bits/sym
    config[2] = CFG_1STOP            ; // Nb of stop bits: 1 stop bit
    config[3] = CFG_NOPARITY         ; // Parity: none
    config[4] = CFG_DMAON            ; // DMA: ON
    config[5] = CFG_NOFLOWCTL        ; // Flow control: none
    config[6] = CFG_FULLFLUSH        ; // Timing config: none
    config[7] = CFG_1MS              ; // Timing: 1 ms
    config[8] = CFG_FAST             ; // Speed: normal

    print_menu()                     ; // Print menu on serial line DEV0
    print_status()                   ; // Update status
    print_time()                     ; // Update time


/*****************************************************************************
 * Step 3 : We start a timer that will send us an event every 1 second so    *
 * ------   that we can update the time.                                     *
 *****************************************************************************/

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


/*****************************************************************************
 * Step 4 : Here is our main event loop. For each loop, we do as follows :   *
 * -------  - First we wait for an event.                                    *
 *          - Then we call a procedure that will "parse" the event and       *
 *            convert it into a chosen option OPT_*.                         *
 *          - Then we call a procedure that will decide what to do of the    *
 *            chosen option, depending on the current page.                  *
 *          - The returned value TRT_* of the procedure will be then used to *
 *            do what as to be done depending on the current page.           *
 *****************************************************************************/

    wait_ev                          :

    ev = wait_myevents()             ; // Unschedule until an event is
                                       // received
    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_asy1_rcv()        ; // 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_asy1_snd()        ; // Determine option selected
          break                      ;

        case EV_GPIO0_IN             : // GPIO input pin change button 1
          opt = ev_gpio0_in()        ; // Determine option selected
          break                      ;

        case EV_GPIO1_IN             : // GPIO input pin change button 2
          opt = ev_gpio1_in()        ; // Determine option selected
          break                      ;

        case EV_GPIO2_IN             : // GPIO input pin change button 3
          opt = ev_gpio2_in()        ; // 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                      ;
      }

    switch ( page )
      {
        case PAGE_MAIN               : // Main menu page
          trt = pg_main      (opt)   ; // Determine treatment for page
          break                      ;

        case PAGE_ASY_OPTS           : // List of serial configurable options
          trt = pg_asy_opts  (opt)   ; // Determine treatment for page
          break                      ;

        case PAGE_ASY_BAUD           : // Baudrate configuration
          trt = pg_asy_baud  (opt)   ; // Determine treatment for page
          break                      ;

        case PAGE_ASY_BITS           : // Number of bits configuration
          trt = pg_asy_bits  (opt)   ; // Determine treatment for page
          break                      ;

        case PAGE_ASY_STOP           : // Stop bits configuration
          trt = pg_asy_stop  (opt)   ; // Determine treatment for page
          break                      ;

        case PAGE_ASY_PARITY         : // Parity configuration
          trt = pg_asy_parity(opt)   ; // Determine treatment for page
          break                      ;

        case PAGE_ASY_FLOWCTL        : // Flow control configuration
          trt = pg_asy_flowctl(opt)  ; // Determine treatment for page
          break                      ;

        case PAGE_ASY_FLUSHCND       : // Flush condition configuration
          trt = pg_asy_flushcnd(opt) ; // Determine treatment for page
          break                      ;
```

```
        case PAGE_ASY_TEMPO          : // Temporization configuration
          trt = pg_asy_tempo(opt)    ; // Determine treatment for page
          break                      ;
      }

log_ev(10, ev, opt, page, trt);
    switch ( trt )                     // Execute the selected treatment
      {
        case TRT_TOGGLE_RED_UPD      : // Start/Stop pattern on RED, print
          state[0] ^= 1              ; // Change menu state
          set_led_state(led_iod0 ,     // IOD for LED0 subchannel
                         state[0]   ?  // Taglist = according state,
                         led_tagon0 :  // if 1, blinking pattern
                         led_tagoff ); // if 0, led is off
          print_status()             ; //   refresh menu statuses
          break                      ;

        case TRT_TOGGLE_RED          : // Start/Stop pattern on RED
          state[0] ^= 1              ; // Change menu state
          set_led_state(led_iod0 ,     // IOD for LED0 subchannel
                         state[0]   ?  // Taglist = according state,
                         led_tagon0 :  // if 1, blinking pattern
                         led_tagoff ); // if 0, led is off
          break                      ;

        case TRT_TOGGLE_GREEN_UPD    : // Start/Stop pattern on GREEN, print
          state[1] ^= 1              ; // Change menu state
          set_led_state(led_iod1 ,     // IOD for LED1 subchannel
                         state[1]   ?  // Taglist = according state,
                         led_tagon1 :  // if 1, blinking pattern
                         led_tagoff ); // if 0, led is off
          print_status()             ; // Refresh menu statuses
          break                      ;

        case TRT_TOGGLE_GREEN        : // Start/Stop pattern on GREEN
          state[1] ^= 1              ; // Change menu state
          set_led_state(led_iod1 ,     // IOD for LED1 subchannel
                         state[1]   ?  // Taglist = according state,
                         led_tagon1 :  // if 1, blinking pattern
                         led_tagoff ); // if 0, led is off
          break                      ;

        case TRT_TOGGLE_YELLOW_UPD   : // Start/Stop pattern on YELLOW, print
          state[2] ^= 1              ; // Change menu state
          set_led_state(led_iod2 ,     // IOD for LED1 subchannel
                         state[2]   ?  // Taglist = according state,
                         led_tagon2 :  // if 1, blinking pattern
                         led_tagoff ); // if 0, led is off
          print_status()             ; // Refresh menu statuses
          break                      ;

        case TRT_TOGGLE_YELLOW       : // Start/Stop pattern on YELLOW
          state[2] ^= 1              ; // Change menu state
          set_led_state(led_iod2 ,     // IOD for LED1 subchannel
                         state[2]   ?  // Taglist = according state,
                         led_tagon2 :  // if 1, blinking pattern
                         led_tagoff ); // if 0, led is off
          break                      ;

        case TRT_TOGGLE_SERIAL_UPD   : // Start/Stop message on serial
          state[3] ^= 1              ; // Change menu state
          if ( state[3] )              // If new state is 1, send a
            req_snd_start(VLASY   ,    // SND_START event to LW 0 of AUT_USND
                          asy_iod1,    // Iod for the ASY1
                          buf_snd1,    // Data buffer for the ASY1
                          256      ) ; // Number of bytes in buffer
          else                         // Else, new state is 0, send a
            req_snd_stop(VLASY)      ; // SND_STOP event to LW 0 of AUT_USND
```

```
  print_status()             ; // Refresh menu statuses
  break                      ;

case TRT_TOGGLE_DAC_UPD      : // Start/Stop stream on DAC
  state[4] ^= 1              ; // Change menu state
  if ( state[4] )              // If new state is 1, send a
    req_snd_start(VLDAC ,      // SND_START event to LW 1 of AUT_USND
                  dac_iod,     // Iod for the DAC
                  buf_aud,     // Data buffet 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 1 of AUT_USND
  print_status()             ; // Refresh menu statuses
  break                      ;

case TRT_RESTART             : // Warm_restart
  goto restart               ;

case TRT_PRINT_TIME          : // Display time in seconds
  print_time()               ; // Display hour, minute, seconds
  break                      ;

case TRT_PAGE_MAIN           : // Display main page
  page = PAGE_MAIN           ; // Select page
  print_menu()               ; // Display page
  print_time()               ; // Update time
  print_status()             ; // Update status
  break                      ;

page_asy_opts                :
case TRT_PAGE_ASY_OPTS       : // Display serial line options list
  page = PAGE_ASY_OPTS       ; // Select page
  print_menu()               ; // Display page
  print_time()               ; // Update time
  print_asyconfig()          ; // Update config
  break                      ;

case TRT_PAGE_ASY_BAUD       : // Display baudrate config page
  page = PAGE_ASY_BAUD       ; // Select page
  print_menu()               ; // Display page
  print_time()               ; // Update time
  break                      ;

case TRT_PAGE_ASY_BITS       : // Display nb of bits config page
  page = PAGE_ASY_BITS       ; // Select page
  print_menu()               ; // Display page
  print_time()               ; // Update time
  break                      ;

case TRT_PAGE_ASY_STOP       : // Display stop bits config page
  page = PAGE_ASY_STOP       ; // Select page
  print_menu()               ; // Display page
  print_time()               ; // Update time
  break                      ;

case TRT_PAGE_ASY_PARITY     : // Display parity config page
  page = PAGE_ASY_PARITY     ; // Select page
  print_menu()               ; // Display page
  print_time()               ; // Update time
  break                      ;

case TRT_PAGE_ASY_FLOWCTL    : // Display flow control config page
  page = PAGE_ASY_FLOWCTL    ; // Select page
  print_menu()               ; // Display page
  print_time()               ; // Update time
  break                      ;

case TRT_PAGE_ASY_FLUSHCND   : // Display flush condition config page
```

```
  page = PAGE_ASY_FLUSHCND   ; // Select page
  print_menu()               ; // Display page
  print_time()               ; // Update time
  break                      ;

case TRT_PAGE_ASY_TEMPO      : // Display temporization config page
  page = PAGE_ASY_TEMPO      ; // Select page
  print_menu()               ; // Display page
  print_time()               ; // Update time
  break                      ;

case TRT_BAUD_9600           : // Set BAUD=9600
  config[0] = CFG_9600       ; // Select config
  set_asy(asy_iod1           , // IOD for ASY1 subchannel
    asy_tagbauds[config[0]]) ; // Configuration tag list
  goto page_asy_opts         ; // Back to list of options
  break                      ;

case TRT_BAUD_19200          : // Set BAUD=19200
  config[0] = CFG_19200      ; // Select config
  set_asy(asy_iod1           , // IOD for ASY1 subchannel
    asy_tagbauds[config[0]]) ; // Configuration tag list
  goto page_asy_opts         ; // Back to list of options
  break                      ;

case TRT_BAUD_38400          : // Set BAUD=38400
  config[0] = CFG_38400      ; // Select config
  set_asy(asy_iod1           , // IOD for ASY1 subchannel
    asy_tagbauds[config[0]]) ; // Configuration tag list
  goto page_asy_opts         ; // Back to list of options
  break                      ;

case TRT_BAUD_57600          : // Set BAUD=57600
  config[0] = CFG_57600      ; // Select config
  set_asy(asy_iod1           , // IOD for ASY1 subchannel
    asy_tagbauds[config[0]]) ; // Configuration tag list
  goto page_asy_opts         ; // Back to list of options
  break                      ;

case TRT_BAUD_115200         : // Set BAUD=115200
  config[0] = CFG_115200     ; // Select config
  set_asy(asy_iod1           , // IOD for ASY1 subchannel
    asy_tagbauds[config[0]]) ; // Configuration tag list
  goto page_asy_opts         ; // Back to list of options
  break                      ;

case TRT_6BITS               : // Set NBBITS=6
  config[1] = CFG_6BITS      ; // Select config
  set_asy(asy_iod1           , // IOD for ASY1 subchannel
    asy_tagnbbits[config[1]]); // Configuration tag list
  goto page_asy_opts         ; // Back to list of options
  break                      ;

case TRT_7BITS               : // Set NBBITS=7
  config[1] = CFG_7BITS      ; // Select config
  set_asy(asy_iod1           , // IOD for ASY1 subchannel
    asy_tagnbbits[config[1]]); // Configuration tag list
  goto page_asy_opts         ; // Back to list of options
  break                      ;

case TRT_8BITS               : // Set NBBITS=8
  config[1] = CFG_8BITS      ; // Select config
  set_asy(asy_iod1           , // IOD for ASY1 subchannel
    asy_tagnbbits[config[1]]); // Configuration tag list
  goto page_asy_opts         ; // Back to list of options
  break                      ;

case TRT_1STOP               : // Set STOPBIT=1
```

```
  config[2] = CFG_1STOP      ; // Select config
  set_asy(asy_iod1           , // IOD for ASY1 subchannel
    asy_tagstop[config[2]] ) ; // Configuration tag list
  goto page_asy_opts         ; // Back to list of options
  break                      ;

case TRT_2STOP               : // Set STOPBIT=2
  config[2] = CFG_2STOP      ; // Select config
  set_asy(asy_iod1           , // IOD for ASY1 subchannel
    asy_tagstop[config[2]] ) ; // Configuration tag list
  goto page_asy_opts         ; // Back to list of options
  break                      ;

case TRT_NOPARITY            : // Set PARITY=NONE
  config[3] = CFG_NOPARITY   ; // Select config
  set_asy(asy_iod1           , // IOD for ASY1 subchannel
    asy_tagparity[config[3]]); // Configuration tag list
  goto page_asy_opts         ; // Back to list of options
  break                      ;

case TRT_EVENPARITY          : // Set PARITY=EVEN
  config[3] = CFG_EVENPARITY ; // Select config
  set_asy(asy_iod1           , // IOD for ASY1 subchannel
    asy_tagparity[config[3]]); // Configuration tag list
  goto page_asy_opts         ; // Back to list of options
  break                      ;

case TRT_ODDPARITY           : // Set PARITY=ODD
  config[3] = CFG_ODDPARITY  ; // Select config
  set_asy(asy_iod1           , // IOD for ASY1 subchannel
    asy_tagparity[config[3]]); // Configuration tag list
  goto page_asy_opts         ; // Back to list of options
  break                      ;

case TRT_TOGGLE_ASYDMA       : // Enable/disable DMA on ASY
  config[4] ^= 1             ; // Change config
  set_asy(asy_iod1           , // IOD for ASY1 subchannel
    asy_tagdma[config[4]]  ) ; // Configuration tag list
  print_asyconfig()          ; // Update config
  break                      ;

case TRT_NOFLOWCTL           : // Flow control: none
  config[5] = CFG_NOFLOWCTL  ; // Select config
  set_asy(asy_iod1           , // IOD for ASY1 subchannel
    asy_tagflow[config[5]] ) ; // Configuration tag list
  goto page_asy_opts         ; // Back to list of options
  break                      ;

case TRT_RTSCTS              : // Flow control: RTS/CTS
  config[5] = CFG_RTSCTS     ; // Select config
  set_asy(asy_iod1           , // IOD for ASY1 subchannel
    asy_tagflow[config[5]] ) ; // Configuration tag list
  goto page_asy_opts         ; // Back to list of options
  break                      ;

case TRT_XONXOFF             : // Flow control: XON/XOFF
  config[5] = CFG_XONXOFF    ; // Select config
  set_asy(asy_iod1           , // IOD for ASY1 subchannel
    asy_tagflow[config[5]] ) ; // Configuration tag list
  goto page_asy_opts         ; // Back to list of options
  break                      ;

case TRT_FULLFLUSH           : // Timing config: none
  config[6] = CFG_FULLFLUSH  ; // Select config
  set_asy(asy_iod1           , // IOD for ASY1 subchannel
    asy_tagtmode[config[6]]) ; // Configuration tag list
  goto page_asy_opts         ; // Back to list of options
  break                      ;
```

```
case TRT_TEMPOCHAR           : // Timing config: CHARACTER
  config[6] = CFG_TEMPOCHAR  ; // Select config
  set_asy(asy_iod1           , // IOD for ASY1 subchannel
    asy_tagtmode[config[6]]) ; // Configuration tag list
  goto page_asy_opts         ; // Back to list of options
  break                      ;

case TRT_TEMPOFRAME          : // Timing config: FRAME
  config[6] = CFG_TEMPOFRAME ; // Select config
  set_asy(asy_iod1           , // IOD for ASY1 subchannel
    asy_tagtmode[config[6]]) ; // Configuration tag list
  goto page_asy_opts         ; // Back to list of options
  break                      ;

case TRT_TEMPO_1MS           : // Timing: 1 ms
  config[7] = CFG_1MS        ; // Select config
  set_asy(asy_iod1           , // IOD for ASY1 subchannel
    asy_tagtempo[config[7]]) ; // Configuration tag list
  goto page_asy_opts         ; // Back to list of options
  break                      ;

case TRT_TEMPO_5MS           : // Timing: 5 ms
  config[7] = CFG_5MS        ; // Select config
  set_asy(asy_iod1           , // IOD for ASY1 subchannel
    asy_tagtempo[config[7]]) ; // Configuration tag list
  goto page_asy_opts         ; // Back to list of options
  break                      ;

case TRT_TEMPO_10MS          : // Timing: 10 ms
  config[7] = CFG_10MS       ; // Select config
  set_asy(asy_iod1           , // IOD for ASY1 subchannel
    asy_tagtempo[config[7]]) ; // Configuration tag list
  goto page_asy_opts         ; // Back to list of options
  break                      ;

case TRT_TEMPO_50MS          : // Timing: 50 ms
  config[7] = CFG_50MS       ; // Select config
  set_asy(asy_iod1           , // IOD for ASY1 subchannel
    asy_tagtempo[config[7]]) ; // Configuration tag list
  goto page_asy_opts         ; // Back to list of options
  break                      ;

case TRT_TEMPO_100MS         : // Timing: 100 ms
  config[7] = CFG_100MS      ; // Select config
  set_asy(asy_iod1           , // IOD for ASY1 subchannel
    asy_tagtempo[config[7]]) ; // Configuration tag list
  goto page_asy_opts         ; // Back to list of options
  break                      ;

case TRT_TEMPO_150MS         : // Timing: 150 ms
  config[7] = CFG_150MS      ; // Select config
  set_asy(asy_iod1           , // IOD for ASY1 subchannel
    asy_tagtempo[config[7]]) ; // Configuration tag list
  goto page_asy_opts         ; // Back to list of options
  break                      ;

case TRT_TEMPO_200MS         : // Timing: 200 ms
  config[7] = CFG_200MS      ; // Select config
  set_asy(asy_iod1           , // IOD for ASY1 subchannel
    asy_tagtempo[config[7]]) ; // Configuration tag list
  goto page_asy_opts         ; // Back to list of options
  break                      ;

case TRT_TEMPO_500MS         : // Timing: 500 ms
  config[7] = CFG_500MS      ; // Select config
  set_asy(asy_iod1           , // IOD for ASY1 subchannel
    asy_tagtempo[config[7]]) ; // Configuration tag list
```

```
          goto page_asy_opts         ; // Back to list of options
          break                      ;

        case TRT_TEMPO_1000MS        : // Timing: 1000 ms
          config[7] = CFG_1000MS     ; // Select config
          set_asy(asy_iod1           , // IOD for ASY1 subchannel
            asy_tagtempo[config[7]]) ; // Configuration tag list
          goto page_asy_opts         ; // Back to list of options
          break                      ;

        case TRT_TOGGLE_ASYSPEED     : // Toggle DMA speed mode
          config[8] ^= 1             ; // Change config
          print_asyconfig()          ; // Update config
          break                      ;

        default                      : // Anything else we want to ignore
          break                      ;

      }

    goto wait_ev                     ; // Wait for the next event


/*****************************************************************************
 * Step 5 : Close the five drivers, ASY, GPIO, LED, DRVITF and DAC          *
 * ------                                                                    *
 *****************************************************************************/

    restart                          : // Warm restart

    clear_tto(idto)                  ; // Stop the timer
    if ( state[4] )                    // If DAC is sending data,
      req_snd_stop(VLDAC)            ; // send a SND_STOP event to AUT_USND

    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 pg_main ---------------------------------------------------------

    Purpose : This procedure decides what to do depending on user option
              on main menu page.
*/

static INT pg_main(int opt)
  {
    int             trt              ; // Selected treatment

    switch ( opt )                     // Convert option to treatment
      {
        case OPT_1                   : // Choice: option 1
        case OPT_BUT1                : // Choice: button 1
          print_cmd('1')             ; // Echo selected option
          trt = TRT_TOGGLE_RED_UPD   ; // Start/Stop pattern on RED
          break                      ; // and show new status

        case OPT_2                   : // Choice: option 2
        case OPT_BUT2                : // Choice: button 2
          print_cmd('2')             ; // Echo selected option
          trt = TRT_TOGGLE_GREEN_UPD ; // Start/Stop pattern on GREEN
          break                      ; // and show new status

        case OPT_3                   : // Choice: option 3
        case OPT_BUT3                : // Choice: button 3
          print_cmd('3')             ; // Echo selected option
          trt = TRT_TOGGLE_YELLOW_UPD; // Start/Stop pattern on YELLOW
          break                      ; // and show new status

        case OPT_4                   : // Choice: option 4
          print_cmd('4')             ; // Echo selected option
          trt = TRT_TOGGLE_SERIAL_UPD; // Start/Stop serial line
          break                      ; // and show new status

        case OPT_5                   : // Choice: option 5
          print_cmd('5')             ; // Echo selected option
          trt = TRT_TOGGLE_DAC_UPD   ; // Select serial config page
          break                      ;

        case OPT_6                   : // Choice: option 6
          print_cmd('6')             ; // Echo selected option
          trt = TRT_PAGE_ASY_OPTS    ; // Select serial config page
          break                      ;

        case OPT_0                   : // Choice: option 0
          print_cmd('0')             ; // Echo selected option
          trt = TRT_RESTART          ; // Warm_restart
          break                      ;

        case OPT_TIME                : // Choice: hidden option timeout
          trt = TRT_PRINT_TIME       ; // Display time in seconds
          break                      ;

        case OPT_IGNORE              : // Choice: none
          trt = TRT_IGNORE           ; // Ignore
          break                      ;

        case OPT_INVALID             : // Anything else is invalid.
        default                      : // Anything else is invalid.
          print_cmd('?')             ; // Print a question mark.
          trt = TRT_IGNORE           ; // Invalid option
      }

    return trt                       ; // Return chose treatment
  }


/*  Procedure pg_asy_opts -----------------------------------------------------

    Purpose : This procedure decides what to do depending on user option
              on serial line options list menu page.
*/

static INT pg_asy_opts(int opt)
  {
    int             trt              ; // Selected treatment

    switch ( opt )                     // Convert option to treatment
      {
        case OPT_1                   : // Choice: option 1
          print_cmd('1')             ; // Echo selected option
          trt = TRT_PAGE_ASY_BAUD    ; // Select baudrate config page
          break                      ;
        case OPT_2                   : // Choice: option 2
          print_cmd('2')             ; // Echo selected option
          trt = TRT_PAGE_ASY_BITS    ; // Select nb bits config page
          break                      ;
        case OPT_3                   : // Choice: option 3
          print_cmd('3')             ; // Echo selected option
          trt = TRT_PAGE_ASY_STOP    ; // Select stop bits config page
          break                      ;
        case OPT_4                   : // Choice: option 4
          print_cmd('4')             ; // Echo selected option
          trt = TRT_PAGE_ASY_PARITY  ; // Select parity config page
          break                      ;
        case OPT_5                   : // Choice: option 5
          print_cmd('5')             ; // Echo selected option
          trt = TRT_TOGGLE_ASYDMA    ; // Change DMA setting for ASY1
          break                      ;
        case OPT_6                   : // Choice: option 6
          print_cmd('6')             ; // Echo selected option
          trt = TRT_PAGE_ASY_FLOWCTL ; // Select flow control config page
          break                      ;
        case OPT_7                   : // Choice: option 7
          print_cmd('7')             ; // Echo selected option
          trt = TRT_PAGE_ASY_FLUSHCND; // Select flush condition config page
          break                      ;
        case OPT_8                   : // Choice: option 8
          print_cmd('8')             ; // Echo selected option
          trt = TRT_PAGE_ASY_TEMPO   ; // Select temporization config page
          break                      ;
        case OPT_9                   : // Choice: option 9
          print_cmd('9')             ; // Echo selected option
          trt = TRT_TOGGLE_ASYSPEED  ; // Change speed mode for ASY1
          break                      ;
        case OPT_0                   : // Choice: option 0
          print_cmd('0')             ; // Echo selected option
          trt = TRT_PAGE_MAIN        ; // Select main menu page
          break                      ;

        case OPT_BUT1                : // Choice: button 1
          trt = TRT_TOGGLE_RED       ; // Start/Stop pattern on RED
          break                      ; // without updating status
        case OPT_BUT2                : // Choice: button 2
          trt = TRT_TOGGLE_GREEN     ; // Start/Stop pattern on GREEN
          break                      ; // without updating status
        case OPT_BUT3                : // Choice: button 3
          trt = TRT_TOGGLE_YELLOW    ; // Start/Stop pattern on YELLOW
          break                      ; // without updating status
        case OPT_TIME                : // Choice: hidden option timeout
          trt = TRT_PRINT_TIME       ; // Display time in seconds
          break                      ;

        case OPT_IGNORE              : // Choice: none
          trt = TRT_IGNORE           ; // Ignore
          break                      ;

        case OPT_INVALID             : // Anything else is invalid.
        default                      : // Anything else is invalid.
          print_cmd('?')             ; // Print a question mark.
          trt = TRT_IGNORE           ; // Invalid option
      }

    return trt                       ; // Return chose treatment
  }


/*  Procedure pg_asy_baud -----------------------------------------------------

    Purpose : This procedure decides what to do depending on user option
              on serial line baudrate configuration menu page.
*/

static INT pg_asy_baud(int opt)
  {
    int             trt              ; // Selected treatment
    switch ( opt )                     // Convert option to treatment
      {
        case OPT_1                   : // Choice: option 1
          print_cmd('1')             ; // Echo selected option
          trt = TRT_BAUD_9600        ; // Configure 9600 bauds
          break                      ;
        case OPT_2                   : // Choice: option 2
          print_cmd('2')             ; // Echo selected option
          trt = TRT_BAUD_19200       ; // Configure 19200 bauds
          break                      ;
        case OPT_3                   : // Choice: option 3
          print_cmd('3')             ; // Echo selected option
          trt = TRT_BAUD_38400       ; // Configure 38400 bauds
          break                      ;
        case OPT_4                   : // Choice: option 4
          print_cmd('4')             ; // Echo selected option
          trt = TRT_BAUD_57600       ; // Configure 57600 bauds
          break                      ;
        case OPT_5                   : // Choice: option 4
          print_cmd('5')             ; // Echo selected option
          trt = TRT_BAUD_115200      ; // Configure 115200 bauds
          break                      ;
        case OPT_0                   : // Choice: option 0
          print_cmd('0')             ; // Echo selected option
          trt = TRT_PAGE_ASY_OPTS    ; // Select options list menu page
          break                      ;

        case OPT_BUT1                : // Choice: button 1
          trt = TRT_TOGGLE_RED       ; // Start/Stop pattern on RED
          break                      ; // without updating status
        case OPT_BUT2                : // Choice: button 2
          trt = TRT_TOGGLE_GREEN     ; // Start/Stop pattern on GREEN
          break                      ; // without updating status
        case OPT_BUT3                : // Choice: button 3
          trt = TRT_TOGGLE_YELLOW    ; // Start/Stop pattern on YELLOW
          break                      ; // without updating status
        case OPT_TIME                : // Choice: hidden option timeout
          trt = TRT_PRINT_TIME       ; // Display time in seconds
          break                      ;

        case OPT_IGNORE              : // Choice: none
          trt = TRT_IGNORE           ; // Ignore
          break                      ;

        case OPT_INVALID             : // Anything else is invalid.
        default                      : // Anything else is invalid.
          print_cmd('?')             ; // Print a question mark.
          trt = TRT_IGNORE           ; // Invalid option
      }

    return trt                       ; // Return chose treatment
  }


/*  Procedure pg_asy_bits -----------------------------------------------------

    Purpose : This procedure decides what to do depending on user option
              on serial line number of bits configuration menu page.
*/

static INT pg_asy_bits(int opt)
  {
    int             trt              ; // Selected treatment

    switch ( opt )                     // Convert option to treatment
      {
        case OPT_1                   : // Choice: option 1
          print_cmd('1')             ; // Echo selected option
          trt = TRT_6BITS            ; // Configure 6bits/sym
          break                      ;
        case OPT_2                   : // Choice: option 2
          print_cmd('2')             ; // Echo selected option
          trt = TRT_7BITS            ; // Configure 7bits/sym
          break                      ;
        case OPT_3                   : // Choice: option 3
          print_cmd('3')             ; // Echo selected option
          trt = TRT_8BITS            ; // Configure 8bits/sym
          break                      ;
        case OPT_0                   : // Choice: option 0
          print_cmd('0')             ; // Echo selected option
          trt = TRT_PAGE_ASY_OPTS    ; // Select options list menu page
          break                      ;

        case OPT_BUT1                : // Choice: button 1
          trt = TRT_TOGGLE_RED       ; // Start/Stop pattern on RED
          break                      ; // without updating status
        case OPT_BUT2                : // Choice: button 2
          trt = TRT_TOGGLE_GREEN     ; // Start/Stop pattern on GREEN
          break                      ; // without updating status
        case OPT_BUT3                : // Choice: button 3
          trt = TRT_TOGGLE_YELLOW    ; // Start/Stop pattern on YELLOW
          break                      ; // without updating status
        case OPT_TIME                : // Choice: hidden option timeout
          trt = TRT_PRINT_TIME       ; // Display time in seconds
          break                      ;

        case OPT_IGNORE              : // Choice: none
          trt = TRT_IGNORE           ; // Ignore
          break                      ;

        case OPT_INVALID             : // Anything else is invalid.
        default                      : // Anything else is invalid.
          print_cmd('?')             ; // Print a question mark.
          trt = TRT_IGNORE           ; // Invalid option
      }

    return trt                       ; // Return chose treatment
  }


/*  Procedure pg_asy_stop -----------------------------------------------------

    Purpose : This procedure decides what to do depending on user option
              on serial line stop bits configuration menu page.
*/

static INT pg_asy_stop(int opt)
  {
    int             trt              ; // Selected treatment

    switch ( opt )                     // Convert option to treatment
      {
        case OPT_1                   : // Choice: option 1
          print_cmd('1')             ; // Echo selected option
          trt = TRT_1STOP            ; // Configure 1 stop bit
          break                      ;
        case OPT_2                   : // Choice: option 2
          print_cmd('2')             ; // Echo selected option
          trt = TRT_2STOP            ; // Configure 2 stop bit
          break                      ;
        case OPT_0                   : // Choice: option 0
          print_cmd('0')             ; // Echo selected option
          trt = TRT_PAGE_ASY_OPTS    ; // Select options list menu page
          break                      ;

        case OPT_BUT1                : // Choice: button 1
```

```
          trt = TRT_TOGGLE_RED       ; // Start/Stop pattern on RED
          break                      ; // without updating status
        case OPT_BUT2                : // Choice: button 2
          trt = TRT_TOGGLE_GREEN     ; // Start/Stop pattern on GREEN
          break                      ; // without updating status
        case OPT_BUT3                : // Choice: button 3
          trt = TRT_TOGGLE_YELLOW    ; // Start/Stop pattern on YELLOW
          break                      ; // without updating status
        case OPT_TIME                : // Choice: hidden option timeout
          trt = TRT_PRINT_TIME       ; // Display time in seconds
          break                      ;

        case OPT_IGNORE              : // Choice: none
          trt = TRT_IGNORE           ; // Ignore
          break                      ;

        case OPT_INVALID             : // Anything else is invalid.
        default                      : // Anything else is invalid.
          print_cmd('?')             ; // Print a question mark.
          trt = TRT_IGNORE           ; // Invalid option
      }

    return trt                       ; // Return chose treatment
  }


/*  Procedure pg_asy_parity ---------------------------------------------------

    Purpose : This procedure decides what to do depending on user option
              on serial line parity configuration menu page.
*/

static INT pg_asy_parity(int opt)
  {
    int             trt              ; // Selected treatment

    switch ( opt )                     // Convert option to treatment
      {
        case OPT_1                   : // Choice: option 1
          print_cmd('1')             ; // Echo selected option
          trt = TRT_NOPARITY         ; // Configure no parity
          break                      ;
        case OPT_2                   : // Choice: option 2
          print_cmd('2')             ; // Echo selected option
          trt = TRT_EVENPARITY       ; // Configure no parity
          break                      ;
        case OPT_3                   : // Choice: option 3
          print_cmd('3')             ; // Echo selected option
          trt = TRT_ODDPARITY        ; // Configure no parity
          break                      ;
        case OPT_0                   : // Choice: option 0
          print_cmd('0')             ; // Echo selected option
          trt = TRT_PAGE_ASY_OPTS    ; // Select options list menu page
          break                      ;

        case OPT_BUT1                : // Choice: button 1
          trt = TRT_TOGGLE_RED       ; // Start/Stop pattern on RED
          break                      ; // without updating status
        case OPT_BUT2                : // Choice: button 2
          trt = TRT_TOGGLE_GREEN     ; // Start/Stop pattern on GREEN
          break                      ; // without updating status
        case OPT_BUT3                : // Choice: button 3
          trt = TRT_TOGGLE_YELLOW    ; // Start/Stop pattern on YELLOW
          break                      ; // without updating status
        case OPT_TIME                : // Choice: hidden option timeout
          trt = TRT_PRINT_TIME       ; // Display time in seconds
          break                      ;
        case OPT_IGNORE              : // Choice: none
          trt = TRT_IGNORE           ; // Ignore
          break                      ;

        case OPT_INVALID             : // Anything else is invalid.
        default                      : // Anything else is invalid.
          print_cmd('?')             ; // Print a question mark.
          trt = TRT_IGNORE           ; // Invalid option
      }

    return trt                       ; // Return chose treatment
  }


/*  Procedure pg_asy_flowctl --------------------------------------------------

    Purpose : This procedure decides what to do depending on user option
              on serial line flow control configuration menu page.
*/

static INT pg_asy_flowctl(int opt)
  {
    int             trt              ; // Selected treatment

    switch ( opt )                     // Convert option to treatment
      {
        case OPT_1                   : // Choice: option 1
          print_cmd('1')             ; // Echo selected option
          trt = TRT_NOFLOWCTL        ; // Configure no flow control
          break                      ;
        case OPT_2                   : // Choice: option 2
          print_cmd('2')             ; // Echo selected option
          trt = TRT_RTSCTS           ; // Configure control using RTS/CTS
          break                      ;
        case OPT_3                   : // Choice: option 3
          print_cmd('3')             ; // Echo selected option
          trt = TRT_XONXOFF          ; // Configure control using XON/XOFF
          break                      ;
        case OPT_0                   : // Choice: option 0
          print_cmd('0')             ; // Echo selected option
          trt = TRT_PAGE_ASY_OPTS    ; // Select options list menu page
          break                      ;

        case OPT_BUT1                : // Choice: button 1
          trt = TRT_TOGGLE_RED       ; // Start/Stop pattern on RED
          break                      ; // without updating status
        case OPT_BUT2                : // Choice: button 2
          trt = TRT_TOGGLE_GREEN     ; // Start/Stop pattern on GREEN
          break                      ; // without updating status
        case OPT_BUT3                : // Choice: button 3
          trt = TRT_TOGGLE_YELLOW    ; // Start/Stop pattern on YELLOW
          break                      ; // without updating status
        case OPT_TIME                : // Choice: hidden option timeout
          trt = TRT_PRINT_TIME       ; // Display time in seconds
          break                      ;

        case OPT_IGNORE              : // Choice: none
          trt = TRT_IGNORE           ; // Ignore
          break                      ;

        case OPT_INVALID             : // Anything else is invalid.
        default                      : // Anything else is invalid.
          print_cmd('?')             ; // Print a question mark.
          trt = TRT_IGNORE           ; // Invalid option
      }

    return trt                       ; // Return chose treatment
  }
```

```
/*  Procedure pg_asy_flushcnd -------------------------------------------------

    Purpose : This procedure decides what to do depending on user option
              on serial line flush condition configuration menu page.
*/

static INT pg_asy_flushcnd(int opt)
  {
    int             trt               ; // Selected treatment

    switch ( opt )                      // Convert option to treatment
      {
        case OPT_1                    : // Choice: option 1
          print_cmd('1')              ; // Echo selected option
          trt = TRT_FULLFLUSH         ; // Configure flush when full
          break                       ;
        case OPT_2                    : // Choice: option 2
          print_cmd('2')              ; // Echo selected option
          trt = TRT_TEMPOCHAR         ; // Configure flush on char tempo
          break                       ;
        case OPT_3                    : // Choice: option 3
          print_cmd('3')              ; // Echo selected option
          trt = TRT_TEMPOFRAME        ; // Configure flush on frame tempo
          break                       ;
        case OPT_0                    : // Choice: option 0
          print_cmd('0')              ; // Echo selected option
          trt = TRT_PAGE_ASY_OPTS     ; // Select options list menu page
          break                       ;

        case OPT_BUT1                 : // Choice: button 1
          trt = TRT_TOGGLE_RED        ; // Start/Stop pattern on RED
          break                       ; // without updating status
        case OPT_BUT2                 : // Choice: button 2
          trt = TRT_TOGGLE_GREEN      ; // Start/Stop pattern on GREEN
          break                       ; // without updating status
        case OPT_BUT3                 : // Choice: button 3
          trt = TRT_TOGGLE_YELLOW     ; // Start/Stop pattern on YELLOW
          break                       ; // without updating status
        case OPT_TIME                 : // Choice: hidden option timeout
          trt = TRT_PRINT_TIME        ; // Display time in seconds
          break                       ;

        case OPT_IGNORE               : // Choice: none
          trt = TRT_IGNORE            ; // Ignore
          break                       ;

        case OPT_INVALID              : // Anything else is invalid.
        default                       : // Anything else is invalid.
          print_cmd('?')              ; // Print a question mark.
          trt = TRT_IGNORE            ; // Invalid option
      }

    return trt                        ; // Return chose treatment
  }


/*  Procedure pg_asy_tempo ----------------------------------------------------

    Purpose : This procedure decides what to do depending on user option
              on serial line flush temporization configuration menu page.
*/

static INT pg_asy_tempo(int opt)
  {
    int             trt               ; // Selected treatment
    switch ( opt )                     // Convert option to treatment
      {
        case OPT_1                   : // Choice: option 1
          print_cmd('1')             ; // Echo selected option
          trt = TRT_TEMPO_1MS        ; // Configure tempo 1ms
          break                      ;
        case OPT_2                   : // Choice: option 2
          print_cmd('2')             ; // Echo selected option
          trt = TRT_TEMPO_5MS        ; // Configure tempo 5ms
          break                      ;
        case OPT_3                   : // Choice: option 3
          print_cmd('3')             ; // Echo selected option
          trt = TRT_TEMPO_10MS       ; // Configure tempo 10ms
          break                      ;
        case OPT_4                   : // Choice: option 4
          print_cmd('4')             ; // Echo selected option
          trt = TRT_TEMPO_50MS       ; // Configure tempo 50ms
          break                      ;
        case OPT_5                   : // Choice: option 5
          print_cmd('5')             ; // Echo selected option
          trt = TRT_TEMPO_100MS      ; // Configure tempo 100ms
          break                      ;
        case OPT_6                   : // Choice: option 6
          print_cmd('6')             ; // Echo selected option
          trt = TRT_TEMPO_150MS      ; // Configure tempo 150ms
          break                      ;
        case OPT_7                   : // Choice: option 7
          print_cmd('7')             ; // Echo selected option
          trt = TRT_TEMPO_200MS      ; // Configure tempo 200ms
          break                      ;
        case OPT_8                   : // Choice: option 8
          print_cmd('8')             ; // Echo selected option
          trt = TRT_TEMPO_500MS      ; // Configure tempo 500ms
          break                      ;
        case OPT_9                   : // Choice: option 9
          print_cmd('9')             ; // Echo selected option
          trt = TRT_TEMPO_1000MS     ; // Configure tempo 1000ms
          break                      ;
        case OPT_0                   : // Choice: option 0
          print_cmd('0')             ; // Echo selected option
          trt = TRT_PAGE_ASY_OPTS    ; // Select options list menu page
          break                      ;

        case OPT_BUT1                : // Choice: button 1
          trt = TRT_TOGGLE_RED       ; // Start/Stop pattern on RED
          break                      ; // without updating status
        case OPT_BUT2                : // Choice: button 2
          trt = TRT_TOGGLE_GREEN     ; // Start/Stop pattern on GREEN
          break                      ; // without updating status
        case OPT_BUT3                : // Choice: button 3
          trt = TRT_TOGGLE_YELLOW    ; // Start/Stop pattern on YELLOW
          break                      ; // without updating status
        case OPT_TIME                : // Choice: hidden option timeout
          trt = TRT_PRINT_TIME       ; // Display time in seconds
          break                      ;

        case OPT_IGNORE              : // Choice: none
          trt = TRT_IGNORE           ; // Ignore
          break                      ;

        case OPT_INVALID             : // Anything else is invalid.
        default                      : // Anything else is invalid.
          print_cmd('?')             ; // Print a question mark.
          trt = TRT_IGNORE           ; // Invalid option
      }

    return trt                       ; // Return chose treatment
  }
```

```
/*  Procedure myfree_proc ------------------------------------------------------

    Purpose : This is our task "clean-up" procedure
*/

static int myfree_proc(void)
  {
    int             ret              ; // Procedurs return code

/******************************************************************************
 * Step 1 : We free the memory user identifier that was allocated by the      *
 * ------   "init_myapp" procedure.                                           *
 ******************************************************************************/

    free_memuid(myapp_memuid,          // Frees a user id for alloc_buf
                0x00000007  ,          // mask = buffers/links/huge
                &ret          )      ; //
    return 0                         ;
  }


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

    Purpose : Initialize the code of the ASY driver and then opens the two
              serial port that are available. The module identifier of the
              ASY driver is stored in "asy_modd" whilst the two opened devices
              descriptors (Input/Output Descriptors") are kept in "asy_iod0"
              and "asy_iod1" 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
    int             min_iod          ; // MIN(asy_iod0,asy_iod1)
    int             max_iod          ; // MAX(asy_iod0,asy_iod1)
    int             i                ; // Loop index to fill send buffer

/******************************************************************************
 * Step 1 : We have to initialize the piece of code that is the ASY driver   *
 * ------   before allocating devices that its manages. We have to use       *
 *          "drv_init_driver" that sends an REQ_INIT to the driver. The      *
 *          driver is not located here as we are in the Application container*
 *          but inside the IO container. We the REQ_INIT event request will  *
 *          have been processed by the AUT_ASY VMIO automaton, it will send  *
 *          us back a RESP_INIT response event. Here we decide to be         *
 *          unscheduled until the RESP_INIT event is received                *
 ******************************************************************************/

    drv_init_driver("ASY"     ,        // VMIO driver name
                    0         ,        // Node number = local (0)
                    1         ,        // reqid value
                    &asy_modd ,        // Module identifier
                    &ret       )     ; // Error code


/******************************************************************************
 * Step 2 : Now that VMIO ASY driver is ready, we can now issue REQ_ALLOC    *
 * ------   requests to it, in order to allocate the two available USARTS    *
 *          We use "drv_alloc_device" that sends a REQ_ALLOC request event   *
 *          to the VMIO AUT_ASY automaton, we we ask to be unscheduled       *
 *          until the RESP_ALLOC response event is received from the IO      *
 *          container                                                        *
 ******************************************************************************/
```

```
    drv_alloc_device(asy_modd    ,     // Module Descriptor Id
                     "ASY\\DEV0" ,     // Device name = DEV0
                     ""          ,     // Tagini  = empty
                     ""          ,     // Taglist = empty
                     NBLOCKING_IO,     // Flags
                     2           ,     // Reqid
                     0           ,     // Route ID   = empty
                     0           ,     // Route ID 2 = empty
                     &asy_iod0   ,     // Returned I/O descriptor
                     &ret         )  ; // Return code

    drv_alloc_device(asy_modd    ,     // Module Descriptor Id
                     "ASY\\DEV1" ,     // Device name = DEV1
                     ""          ,     // Tagini  = empty
                     ""          ,     // Taglist = empty
                     NBLOCKING_IO,     // Flags
                     3           ,     // Reqid
                     0           ,     // Route ID   = empty
                     0           ,     // Route ID 2 = empty
                     &asy_iod1   ,     // Returned I/O descriptor
                     &ret         )  ; // Return code


/****************************************************************************
 * Step 3 : Now we have allocated both serial ports, we have a special thing *
 * ------   to do in order to receive unsollicited events IND_REPORT. For a  *
 *          serial port, when a state change of the CTS control signal will  *
 *          occur, then the ASY driver will issue a IND_REPORT. This event   *
 *          will be written into the external VMK waiting queue. Then the    *
 *          VMK will copy that event in each task waiting queue that has     *
 *          created a route that matches with this event. If there is no     *
 *          matching route, the event will not be copied in any task event   *
 *          wiating queue. But here we want to receive IND_REPORT events     *
 *          are comming from both "asy_iod0" and "asy_iod1". We cas create   *
 *          a unique route that will "catch" the IND_REPORTs events with a   *
 *          "reserve" value between MIN(asy_iod0) and MAX(asy_iod1)          *
 ****************************************************************************/

    min_iod = asy_iod0 < asy_iod1 ?    // Compute minimum value for "reserve"
              asy_iod0            :    // field of incoming events
              asy_iod1               ; //

    max_iod = asy_iod0 > asy_iod1 ?    // Compute maximum value for "reserve"
              asy_iod0            :    // field of incoming events
              asy_iod1               ; //

    add_uroute(IND_REPORT ,            // Min value for "code"
               IND_REPORT ,            // Max value for "code"
               min_iod    ,            // Min value for "reserve"
               max_iod    ,            // Max value for "reserve"
               0          ,            // Flags
               &asy_rteid  )         ; // Route identifier


/****************************************************************************
 * Step 4 : Here we initialize the hardware of the two serial ports. To do   *
 * ------   this, we have to call "device_open_device. This procedures sends *
 *          a REQ_OPEN event to the AUT_ASY automaton that is located inside *
 *          the IO container. Then we ask to be unscheduled until the        *
 *          VMIO AUT_ASY automaton sends us the RESP_OPEN response event     *
 ****************************************************************************/

    drv_open_device(asy_iod0    ,      // I/O descriptor
                    ""          ,      // Taglist = empty
                    4           ,      // Reqid
                    &ret          )  ; // Return code

    drv_open_device(asy_iod1    ,      // I/O descriptor
                    ""          ,      // Taglist = empty
                    5           ,      // Reqid
                    &ret          )  ; // Return code


/*****************************************************************************
 * Step 5 : Set line parameters for the two serial line controllers. We have *
 * ------   to call the "drv_setval" procedure. This procedure sends a       *
 *          REQ_SETVAL request event the the AUT_ASY VMIO automaton. Then    *
 *          we unschedule until the RESP_SETVAL response event is received   *
 *****************************************************************************/

    drv_setval(asy_iod0            ,   // I/O descriptor for DEV0
               (char*)asy_taglist0 ,   // taglist
               6                   ,   // Reqid
               &ret                 ); // Return code

    drv_setval(asy_iod1            ,   // I/O descriptor for DEV1
               (char*)asy_taglist1 ,   // taglist
               7                   ,   // Reqid
               &ret                 ); // Return code


/*****************************************************************************
 * Step 6 : Allocates memory buffers for transmit, one for DEV0 and another  *
 * -------  one for DEV1. We store the buffers addresses in "buf_snd0" and   *
 *          "buf-snd1". 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 allowaed receive tokens and 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_snd0    ,           // Address of allocated buffer
              &ret          )        ; // Return code

    alloc_buf(myapp_memuid ,           // Memory user id
              256          ,           // Size in bytes of requested buffer
              0            ,           // Flags
              &buf_snd1    ,           // Address of allocated buffer
              &ret          )        ; // Return code


/*****************************************************************************
 * Step 7 : We will send a continuous byte stream through ASY\DEV1 when      *
 * ------   enabled, to test throughput. We initialize the send buffer once  *
 *          for all with bytes from 0xFF to 0x00.                            *
 *****************************************************************************/

    for (i = 0 ; i < 256  ; i++)       // Fill up the send buffer with
      buf_snd1[i] = ~i & 0xFF        ; // bytes from 0xFF to 0x00.

  }


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

    Purpose : This procedure closes the two USART, then frees those two
              devices, terminates the ASY module, delete the route that has
              been created for the IND_REPORT events, and finally frees the 2
              telecom buffers that were allocated by "open_asy".
*/

static void close_asy(void)
```

```
  {

    int             list[2]          ; // Parameter for "del_uroute"
    int             ret              ; // Return code

/*****************************************************************************
 * Step 1 : Close the two serial ports controller devices. We have to call   *
 * ------   "drv_close_device" for DEV0 (so asy_iod0) and DEV1 (so asy_iod1) *
 *          This procedure sends a REQ_CLOSE request event to the VMIO       *
 *          AUT_ASY automaton and here we unschedule until the corresponding *
 *          RESP_CLOSE, sent by AUT_ASY, is received                         *
 *****************************************************************************/

    drv_close_device(asy_iod0 ,        // IOD of the device to be closed
                     1        ,        // Reqid
                     &ret      )     ; // Return code

    drv_close_device(asy_iod1 ,        // IOD of the device to be closed
                     2        ,        // Reqid
                     &ret      )     ; // Return code


/*****************************************************************************
 * Step 2 : Free the two serial port controller devices. We have to send to  *
 * ------   AUT_ASY a REQ_FREE request event for each. We have to call the   *
 *          "drv_free" procedure to do that. Then, we unschedule until the   *
 *          RESP_FREE response event is received.                            *
 *****************************************************************************/

    drv_free(asy_iod0 ,                // IOD of the device to be freed
             3        ,                // reqid
             &ret       )            ; // Return code

    drv_free(asy_iod1 ,                // IOD of the device to be freed
             4        ,                // reqid
             &ret       )            ; // Return code


/*****************************************************************************
 * Step 3 : Terminates the ASY driver. We have to send to the AUT_ASY VMIO   *
 * -------  automaton a REQ_END request event. This is done with a call to   *
 *          the "drv_end" procedure. The AUT_ASY VMIO automaton receives     *
 *          this event and then sends back a RESP_END respense event. Here,  *
 *          we unschedule until the RESP_END response event is received      *
 *****************************************************************************/

    drv_end(asy_modd ,                 // MODD of the module to be ended
            5        ,                 // Reqid
            &ret       )             ; // Return code


/*****************************************************************************
 * Step 4 : Delete the route we were using for IND_REPORT events. We have    *
 * -------  to call the "del_uroute" procedure. As this procedure takes as   *
 *          in put an array containing a list of route identifiers, we use   *
 *          here an array of one entry in which we store "asy_rteid" before  *
 *          calling "del_uroute"                                             *
 *          ---------------------------------------------------------------- *
 *          We could directly call "del_route" with the address of the       *
 *          "asy_rteid" global variable because there is no difference       *
 *          between a variable adress and an array address                   *
 *****************************************************************************/

    list[0] = asy_rteid              ; // Store "asy_rteid" in a one entry
                                       // array
    del_uroute(list ,                  // We call "del_uroute" with a list
               1     )               ; // of route identifers tant only
                                       // contains one element.
```

```
/******************************************************************************
 * Step 5 : Free the 2 memory buffers that were allocated by "open_asy".      *
 * ------   For each of them, we just call the "free_buffer" procedure        *
 ******************************************************************************/

    free_buf(buf_snd0, &ret )        ; // Free "buf_snd0"
    free_buf(buf_snd1, &ret )        ; // Free "buf_snd1"

  }


/*  Procedure give_asy_token ---------------------------------------------------

    Purpose : Give one or more receive token to the ASY driver
*/

static void give_asy_token(int iod, int nbtok, int size, int *count)
  {
    unsigned char   *adr             ; // Adr of allocated buffer (unused)
    int             of7              ; // Firts byte in buffer (unused)
    int             lg               ; // Number of bytes read (unused)
    int             ret              ; // Return code

/******************************************************************************
 * Step 1 : In order to have data flow control, the application have to       *
 * ------   give to the ASY driver receive tokens. One token is one buffer.   *
 *          The applications gives to the driver a number of tokens "nbtok"   *
 *          and the size "size" of the buffers. The application as to send    *
 *          a REQ_READ request to the VMIO ASY automaton. The buffers are     *
 *          allocated  by the ASY driver (the VMIO AUT_ASY automaton). To     *
 *          send the REQ_READ event, we call "drv_read".                      *
 *          ----------------------------------------------------------------- *
 *          RESP_READ event will be received after data will be received      *
 *          from the asynchronous line. This may take a long time, or may     *
 *          never occure. So we are not going to unschedule right now         *
 ******************************************************************************/

    drv_read(iod       ,               // I/O descriptor for DEV0
             size      ,               // Maximum number of bytes / buffer
             nbtok     ,               // token/buffer number
             0         ,               // Flags
             &adr      ,               // Buffer address (unused)
             &of7      ,               // Offset (unused)
             &lg       ,               // Number of bytes read (unused)
             1         ,               // Reqid
             &ret        )           ; // Return code

    *count += nbtok                  ; // Update token counter

  }


/*  Procedure write_asy  -------------------------------------------------------

    Purpose : Prints (sends) on the serial line one buffer. This is a blocking
              procedure, we are unscheduled until the end of transmission
*/

static void write_asy(int iod, unsigned char *buf, int lg)
  {
    int             ret              ; // Return code

/******************************************************************************
 * Step 1 : Send using serial controller "iod" the "lg" bytes that are        *
 * ------   in telecom buffer ponted by "buf".  We call the "drv_write"       *
 *          procedure that sends a REQ_WRITE request event to the AUT_ASY     *
 *          VMIO automaton.
 *****************************************************************************/

    drv_write(iod       ,              // IOD of serial controller DEV0
              buf       ,              // Buffer Address
              0         ,              // Buffer Offset
              lg        ,              // Number of bytes to be sent
              0         ,              // Flags
              1         ,              // Reqid
              &ret       )           ; // Return code


/*****************************************************************************
 * Step 2 : We unschedule until the RESP_WRITE event is received. This event *
 * ------   is sent by AUY_ASY only when the last byte as been sent on the   *
 *          serial line and not before                                       *
 *****************************************************************************/

    wait_coderes(RESP_WRITE,iod)     ; // Unschedule until the RESP_WRITE
                                       // event is received.
  }


/*  Procedure set_asy ---------------------------------------------------------

    Purpose : Sets a parameter of the serial line
*/

static void set_asy(int iod, const char *param)
  {
    int             ret              ; // Return code

/*****************************************************************************
 * Step 1 : We simply send the parameter to the ASY driver, so we call the   *
 * ------   "drv_setval" procedure that sends the REQ_SETVAL event to the    *
 *          VMIO AUT_ASY automaton. The procedure unschedules the task until *
 *          the event RESP_SETVAL is received.                               *
 *****************************************************************************/

    drv_setval(iod                ,    // IOD descriptor of ASY device
               (char*) param      ,    // Taglist as received
               0                  ,    // Reqid
               &ret                ) ; // Return code

  }



/*  Procedure send_asy_msg ----------------------------------------------------

    Purpose : Give one message to the ASY driver. This procedure is
              non-blocking.
*/

static void send_asy_msg(int iod, unsigned char *buf, int lg, int *count)
  {
    int             ret              ; // Return code

/*****************************************************************************
 * Step 1 : Send using serial controller "iod" the "lg" bytes that are       *
 * ------   in telecom buffer ponted by "buf".  We call the "drv_write"      *
 *          procedure that sends a REQ_WRITE request event to the AUT_ASY    *
 *          VMIO automaton. We want to maximize throughput, so we don't wait *
 *          for data ta have been sent here. We therefore don't unschedule.  *
 *****************************************************************************/

    drv_write(iod       ,              // IOD of serial controller DEV0
              buf       ,              // Buffer Address
              0         ,              // Buffer Offset
              lg        ,              // Number of bytes to be sent
              0         ,              // Flags
              1         ,              // Reqid
              &ret       )           ; // Return code

    *count += 1                      ; // Update number of inflight messages

  }




/*  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 software module descriptor is kept in global variable
              "gpio_modd" whilst the input/output descriptor is kepts in the
              global variable "gpio_iod". Also the route identifier is stored
              in "gpio_rteid" global variable.
*/

static void open_gpio(void)
  {
    int             ret              ; // Return code
    int             min_iod          ; // Minimum GPIO iod.
    int             max_iod          ; // Maximem GPIO iod.

/*****************************************************************************
 * 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_GPIO automaton. To do this, we call the    *
 *          "drv_init_driver" procedure. The AUT_GPIO automaton responds     *
 *          with a RESP_INIT event, we here unschedule until the reception   *
 *          of this response event                                           *
 *****************************************************************************/

    drv_init_driver("GPIO"     ,       // Name of the driver module
                    0          ,       // Node number = local (0)
                    1          ,       // Reqid
                    &gpio_modd ,       // Module identifier MODD
                    &ret        )    ; // Return code


/*****************************************************************************
 * Step 2 : Now that VMIO GPIO driver is ready, we can now issue a REQ_ALLOC *
 * ------   requests to it, in order to allocate the GPIO controller device  *
 *          We call the "drv_alloc_device" that sends a REQ_ALLOC request    *
 *          event to the VMIO AUT_GPIO automaton. Then we we ask to be       *
 *          unscheduled until the RESP_ALLOC response event is received from *
 *          the IO container, from the AUT_GPIO VMIO automaton               *
 *****************************************************************************/

    drv_alloc_device(gpio_modd   ,     // Module Descriptor Id
                     "GPIO\\DEV0",     // Device name = DEV0
                     ""          ,     // Tagini  = empty
                     ""          ,     // Taglist = empty
                     0           ,     // Flags
                     2           ,     // Reqid
                     0           ,     // Route ID   = empty
                     0           ,     // Route ID 2 = empty
                     &gpio_iod   ,     // Returned I/O descriptor
                     &ret         )  ; // Return code


/*****************************************************************************
 * Step 3 : We have allocated the DEV0 device, the GPIO controller device.   *
 * ------   When an input PIO state will change, then we will receive an     *
 *          IND_REPORT event. Unsollicited events are written in the unique  *
 *          VMK external waiting queue, then when they are extracted from    *
 *          this external file a copy of the event is written in all the     *
 *          internal queues (task queues) that have requested to get it. In  *
 *          order to ask for a copy of an unsollicited event, a task has to  *
 *          create a "user route", by calling the "add_uroute" procedure.    *
 *****************************************************************************/

    min_iod = gpio_iod & ~ 0xFF      ; // Compute minimum value for "reserve"
    max_iod = gpio_iod               ; // Compute minimum value for "reserve"

    add_uroute(IND_REPORT ,            // Min value for "code"
               IND_REPORT ,            // Max value for "code"
               min_iod    ,            // Min value for "reserve"
               max_iod    ,            // Max value for "reserve"
               0          ,            // Flags
               &gpio_rteid  )        ; // Route identifier


/******************************************************************************
 * Step 4 : Here we initialize the hardware of the GPIO controller. We have  *
 * ------   to send a REQ_OPEN event request to the VMIO AUT_GPIO automaton. *
 *          This is done calling the "drv_open_device" procedure. When the   *
 *          hardware initialization will be completed, the VMIO AUT_GPIO     *
 *          automaton will send us back a RESP_OPEN response event. Here we  *
 *          ask to be unscheduled until the RESP_OPEN event is received      *
 *****************************************************************************/

    drv_open_device(gpio_iod    ,      // I/O descriptor
                    ""          ,      // Taglist = empty
                    4           ,      // Reqid
                    &ret          )  ; // Return code


/******************************************************************************
 * Step 5 : 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 are respectively connected    *
 *          to GPIO pins PB5, PA1, PB0.                                      *
 *****************************************************************************/

    drv_alloc_subchan(gpio_iod  ,      // I/O descriptor of controller
                      "PIN=PB5" ,      // Tagini  = PIN identifier
                      ""        ,      // Taglist = empty
                      0         ,      // Flags
                      5         ,      // Reqid
                      0         ,      // Route ID   = empty
                      0         ,      // Route ID 2 = empty
                      &gpio_iod0,      // Returned I/O descriptor of PIN
                      &ret        )  ; // Return code

    drv_alloc_subchan(gpio_iod  ,      // I/O descriptor of controller
#ifdef VM_STM32M3
                      "PIN=PA1" ,      // Tagini  = PIN identifier
#endif
#ifdef VM_STM32M4
                      "PIN=PB1" ,      // Tagini  = PIN identifier
#endif
                      ""        ,      // Taglist = empty
                      0         ,      // Flags
                      6         ,      // Reqid
                      0         ,      // Route ID   = empty
                      0         ,      // Route ID 2 = empty
                      &gpio_iod1,      // Returned I/O descriptor of PIN
                      &ret        )  ; // Return code

    drv_alloc_subchan(gpio_iod  ,      // I/O descriptor of controller
                      "PIN=PB0" ,      // Tagini  = PIN identifier
                      ""        ,      // Taglist = empty
                      0         ,      // Flags
                      7         ,      // Reqid
                      0         ,      // Route ID   = empty
                      0         ,      // Route ID 2 = empty
                      &gpio_iod2,      // Returned I/O descriptor of PIN
                      &ret        )  ; // Return code


/*****************************************************************************
 * Step 6 : The pins are allocated. We need to configure the buttons to get  *
 * ------   IND_REPORT events on state change. We call the "drv_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_iod0", "gpio_iod1" and "gpio_iod2" *
 *****************************************************************************/

    drv_setval(gpio_iod0           ,   // I/O descriptor for BUT1
               (char*)gpio_taglist ,   // taglist
               8                   ,   // Reqid
               &ret                 ); // Return code

    drv_setval(gpio_iod1           ,   // I/O descriptor for BUT2
               (char*)gpio_taglist ,   // taglist
               9                   ,   // Reqid
               &ret                 ); // Return code

    drv_setval(gpio_iod2           ,   // I/O descriptor for BUT3
               (char*)gpio_taglist ,   // taglist
               10                  ,   // Reqid
               &ret                 ); // Return code

  }


/*  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)
  {
    int             list[2]          ; // Parameter for "del_uroute"
    int             ret              ; // Return code

/*****************************************************************************
 * Step 1 : Free all used subchannels. We call "drv_free_subchan" for each   *
 * ------   of the 3 buttons which IO descriptor are "gpio_iod0/1/2". 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.      *
 *****************************************************************************/

    drv_free_subchan(gpio_iod0 ,       // IOD of the subchannel to be freed
                     1         ,       // Reqid
                     &ret        )   ; // Return code

    drv_free_subchan(gpio_iod1 ,       // IOD of the subchannel to be freed
                     2         ,       // Reqid
                     &ret        )   ; // Return code
    drv_free_subchan(gpio_iod2 ,       // IOD of the subchannel to be freed
                     3         ,       // Reqid
                     &ret        )   ; // Return code


/*****************************************************************************
 * Step 1 : Close the GPIO controller. We call drv_close_device" for DEV0    *
 * ------   which IO descriptor is "gpio_iod". This procedure sends a        *
 *          REQ_CLOSE request event to the VMIO AUT_GPIO automaton and here  *
 *          we unschedule until the corresponding  RESP_CLOSE, sent by       *
 *          AUT_GPIO, is received                                            *
 *****************************************************************************/

    drv_close_device(gpio_iod ,        // IOD of the device to be closed
                     4        ,        // Reqid
                     &ret      )     ; // Return code


/*****************************************************************************
 * Step 2 : Free the GPIO controller devices. We have to send a REQ_FREE     *
 * ------   request event to AUT_GPIO, we have to call the "drv_free"        *
 *          procedure. Then we unschedule until the RESP_FREE response event *
 *          is received.                                                     *
 *****************************************************************************/

    drv_free(gpio_iod ,                // IOD of the device to be freed
             5        ,                // reqid
             &ret       )            ; // Return code


/*****************************************************************************
 * Step 3 : Terminates the GPIO driver by sending a REQ_END request event    *
 * ------   to the AUT_GPIO automaton. This is done using a call to the      *
 *          "dev_end" procedure                                              *
 *****************************************************************************/

    drv_end(gpio_modd ,                // MODD identifier
            6         ,                // Reqid
            &ret        )            ; // Return code


/*****************************************************************************
 * Step 4 : Delete the route we were using for IND_REPORT events. We have    *
 * -------  to call the "del_uroute" procedure. As this procedure takes as   *
 *          in put an array containing a list of route identifiers, we use   *
 *          here an array of one entr in which we store "gpio_rteid" before  *
 *          calling "del_uroute"                                             *
 *****************************************************************************/

    list[0] = gpio_rteid             ; // Store "asy_rteid" in a one entry
                                       // array
    del_uroute(list ,                  // We call "del_uroute" with a list
               1     )               ; // of route identifers tant only
                                       // contains one element.
  }


/*  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)
  {
    int             ret              ; // Return code
```

```
/*****************************************************************************
 * 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_LED automaton. To do this, we call the     *
 *          "drv_init_driver" procedure. The AUT_LED automaton responds with *
 *          RESP_INIT event, we here unschedule until the reception of this  *
 *          of this response event                                           *
 *****************************************************************************/

    drv_init_driver("LED"      ,       // Name of the driver module
                    0          ,       // Node number = local (0)
                    1          ,       // Reqid
                    &led_modd  ,       // Module identifier MODD
                    &ret        )    ; // Return code


/*****************************************************************************
 * Step 2 : The VMIO LED driver is ready. We are going now to allocate the   *
 * ------   DEV0 LED controller device. We call the "drv_alloc_device"       *
 *          procedure, this procedure sends a REQ_ALLOC event to the VMIO    *
 *          AUT_LED automaton. This automaton will then send us a RESP_ALLOC *
 *          response event. Here we unschedule until this event is received  *
 *****************************************************************************/

    drv_alloc_device(led_modd    ,     // Module Descriptor Id
                     "LED\\DEV0" ,     // Device name = DEV0
                     ""          ,     // Tagini  = empty
                     ""          ,     // Taglist = empty
                     0           ,     // Flags
                     2           ,     // Reqid
                     0           ,     // Route ID   = empty
                     0           ,     // Route ID 2 = empty
                     &led_iod    ,     // Returned I/O descriptor
                     &ret         )  ; // Return code


/*****************************************************************************
 * Step 3 : Here we initialize the hardware of the LED controller. We call   *
 * ------   the "device_open_device" procedure. This procedures sends a      *
 *          REQ_OPEN request event to the AUT_ASY automaton, that is located *
 *          inside the IO container. Then we ask to be unscheduled until the *
 *          AUT_LED send us the RESP_OPEN response event                     *
 *****************************************************************************/

    drv_open_device(led_iod     ,      // I/O descriptor
                    ""          ,      // Taglist = empty
                    3           ,      // Reqid
                    &ret          )  ; // Return code


/*****************************************************************************
 * Step 4 : The contoller is ready. We are going to allocate the LED         *
 * ------   subchannels, one for each LED. We call the "drv_alloc_subchan"   *
 *          procedure, this 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.                                               *
 *****************************************************************************/

    drv_alloc_subchan(                 // Allocate RED led subchannel
              led_iod           ,      // I/O descriptor of controller
              "LEDNAME=RED0"    ,      // Tagini  = LED name
              ""                ,      // Taglist = empty
              0                 ,      // Flags
              4                 ,      // Reqid
              0                 ,      // Route ID   = empty
              0                 ,      // Route ID 2 = empty
              &led_iod0         ,      // Returned I/O descriptor of LED
              &ret                )  ; // Return code

    drv_alloc_subchan(                 // Allocate RED led subchannel
              led_iod           ,      // I/O descriptor of controller
              "LEDNAME=GREEN0"  ,      // Tagini  = LED name
              ""                ,      // Taglist = empty
              0                 ,      // Flags
              5                 ,      // Reqid
              0                 ,      // Route ID   = empty
              0                 ,      // Route ID 2 = empty
              &led_iod1         ,      // Returned I/O descriptor of LED
              &ret                )  ; // Return code

    drv_alloc_subchan(                 // Allocate RED led subchannel
              led_iod           ,      // I/O descriptor of controller
              "LEDNAME=YELLOW0" ,      // Tagini  = LED name
              ""                ,      // Taglist = empty
              0                 ,      // Flags
              6                 ,      // Reqid
              0                 ,      // Route ID   = empty
              0                 ,      // Route ID 2 = empty
              &led_iod2         ,      // Returned I/O descriptor of LED
              &ret                )  ; // Return code


  }


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

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

static void close_led(void)
  {
    int             ret              ; // Return code

/*****************************************************************************
 * Step 1 : Free all used subchannels. We call "drv_free_subchan" for each   *
 * ------   of the 3 LEDs which IO descriptor are "led_iod0/1/2". This       *
 *          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.       *
 *****************************************************************************/

    drv_free_subchan(led_iod0  ,       // IOD of the subchannel to be freed
                     1         ,       // Reqid
                     &ret        )   ; // Return code

    drv_free_subchan(led_iod1  ,       // IOD of the subchannel to be freed
                     2         ,       // Reqid
                     &ret        )   ; // Return code

    drv_free_subchan(led_iod2  ,       // IOD of the subchannel to be freed
                     3         ,       // Reqid
                     &ret        )   ; // Return code


/*****************************************************************************
 * Step 1 : Close the LED controller. We call drv_close_device" for DEV0     *
 * ------   which IO descriptor is "led_iod". This procedure sends a         *
 *          REQ_CLOSE request event to the VMIO AUT_LED automaton and here   *
 *          we unschedule until the corresponding  RESP_CLOSE, sent by       *
 *          AUT_LED, is received                                             *
 *****************************************************************************/

    drv_close_device(led_iod  ,        // IOD of the device to be closed
                  4        ,        // Reqid
                  &ret       )    ; // Return code


/*****************************************************************************
 * Step 2 : Free the LED controller device. We have to send a REQ_FREE       *
 * ------   request event to AUT_LED, to do this we have to call the         *
 *          "drv_free" procedure. Then we unschedule until the RESP_FREE     *
 *          response event is received. This response event will be issued   *
 *          by the VMIO AUT_LED automaton                                    *
 *****************************************************************************/

    drv_free(led_iod  ,                 // IOD of the device to be freed
             5        ,                 // Reqid
             &ret       )             ; // Return code


/*****************************************************************************
 * Step 3 : Terminates the LED driver by sending a REQ_END request event     *
 * ------   to the AUT_LED automaton. This is done using a call to the       *
 *          "dev_end" procedure                                              *
 *****************************************************************************/

    drv_end(led_modd ,                  // MODD identifier
            6        ,                  // Reqid
            &ret       )              ; // Return code


  }


/*  Procedure set_led_state ---------------------------------------------------

    Purpose : Sets a new blinking pattern for one led
*/

static void set_led_state(int iod, const char *pattern)
  {
    int             ret              ; // Return code

/*****************************************************************************
 * Step 1 : The LED drivers has, for each led, a stack of blinking patterns  *
 * ------   Before pushing in the stack a new blinking pattern, we here      *
 *          request to the led driver to empty the blink pattern stack. To   *
 *          do this, we send a REQ_SETVAL "CMD=POPALL" event to the LED      *
 *          driver, so we call the "drv_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  *
 *****************************************************************************/

    drv_setval(iod                ,    // IOD descriptor of LED subchannel
               (char*) led_tagpop ,    // Taglist = "CMD_POPALL"
               0                  ,    // Reqid
               &ret                ) ; // Return code


/*****************************************************************************
 * Step 2 : The blink pattern stack for our LED is now empty. We push a new  *
 * ------   blinking pattern. We call "drv_setval" that sends to the VMIO    *
 *          AUT_LED a REQ_SETVAL that carries the new blinking pattern, that *
 *          heas to be ended by "CMD=PUSH". So we call the "drv_setval"      *
 *          procedure and then we ask to be unscheduled until the RESP_SETVAL*
 *          response event, replied by AUT_LED, has been received.           *
 *****************************************************************************/

    drv_setval(iod             ,       // IOD descriptor of LED subchannel
               (char*) pattern ,       // Taglist = new blinking pattern
               0               ,       // Reqid
               &ret             )    ; // Return code

  }


/*  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 module identifier of the KBDIT driver is stored in "kbd_modd"
              whilst the opened device descriptor (Input/Output Descriptor) is
              kept in "kbd_iod".
*/

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

/*****************************************************************************
 * 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_KBDITF automaton. To do this, we call the  *
 *          "drv_init_driver" procedure. The AUT_KBDITF automaton responds   *
 *          with a RESP_INIT event, we here unschedule until the reception   *
 *          of this response event                                           *
 *****************************************************************************/

    drv_init_driver("KBDITF"  ,        // VMIO driver name
                    0         ,        // Node number = local (0)
                    1         ,        // reqid value
                    &kbd_modd ,        // Module identifier
                    &ret       )     ; // Error code


/*****************************************************************************
 * Step 2 : Now that VMIO KBDITF driver is ready, we can now issue REQ_ALLOC *
 * ------   requests to it, in order to allocate a keyboard device from      *
 *          which we will receive DRV_KEY_PRESS and DVR_KEY_RELEASE events.  *
 *          We use "drv_alloc_device" that sends a REQ_ALLOC request event   *
 *          to the VMIO AUT_KBDITF automaton. Then we ask to be unscheduled  *
 *          until the RESP_ALLOC response event is received from the IO      *
 *          container                                                        *
 *****************************************************************************/

    drv_alloc_device(kbd_modd      ,   // Module Descriptor Id
                     "KBDITF\\DEV0",   // Device name = DEV0
                     ""            ,   // Tagini  = empty
                     ""            ,   // Taglist = empty
                     0             ,   // Flags
                     2             ,   // Reqid
                     0             ,   // Route ID   = empty
                     0             ,   // Route ID 2 = empty
                     &kbd_iod      ,   // Returned I/O descriptor
                     &ret           ); // Return code


/*****************************************************************************
 * Step 3 : We have allocated the DEV0 device. When an RCU key will be       *
 * ------   pressed, we will reveive a DRV_KEY_PRESS or DRV_KEY_RELEASE.     *
 *          Unsollicited events are written in the unique VMK external       *
 *          waiting queue, then when they are extracted from this external   *
 *          queue a copy of the event is written in all the internal queues  *
 *          (task queues) that have requested to get it. In order to ask for *
 *          a copy of an unsollicited event, a task has to create a "user    *
 *          route", by calling the "add_uroute" procedure.                    *
 *****************************************************************************/

    add_uroute(DRV_KEY_PRESS   ,       // Min value for "code"
               DRV_KEY_RELEASE ,       // Max value for "code"
               -1              ,       // No filter for "reserve"
               -1              ,       // No filter for "reserve"
               0               ,       // Flags
               &kbd_rteid        )   ; // Route identifier


/******************************************************************************
 * Step 4 : Here we initialize the keyboard interface. We have to send a      *
 * ------   REQ_OPEN event request to the VMIO AUT_KBDITF automaton. This is  *
 *          done by calling the "drv_open_device" procedure. When done, the   *
 *          VMIO AUT_KBDITF automaton will send us back a RESP_OPEN response  *
 *          event.                                                            *
 ******************************************************************************/

    drv_open_device(kbd_iod     ,      // I/O descriptor
                    ""          ,      // Taglist = empty
                    3           ,      // Reqid
                    &ret          )  ; // Return code


  }


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

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

static void close_kbd(void)
  {

    int             list[2]          ; // Parameter for "del_uroute"
    int             ret              ; // Return code

/******************************************************************************
 * Step 1 : Close the keyboard interface device. We have to call              *
 * ------   "drv_close_device" for DEV0 (so kbd_iod). This procedure sends    *
 *          a REQ_CLOSE request event to the VMIO AUT_KBDITF automaton and    *
 *          wait for the corresponding RESP_CLOSE from VMIO AUT_KBDITF.       *
 ******************************************************************************/

    drv_close_device(kbd_iod  ,        // IOD of the device to be closed
                     1        ,        // Reqid
                     &ret      )     ; // Return code


/******************************************************************************
 * Step 2 : Free the keyboard interface device. We have to call "drv_free"    *
 * ------   for DEV0 (so kbd_iod). This procedure sends a REQ_FREE request    *
 *          event to the VMIO AUT_KBDITF automaton and wait for the           *
 *          corresponding RESP_FREE from VMIO AUT_KBDITF.                     *
 ******************************************************************************/

    drv_free(kbd_iod  ,                // IOD of the device to be freed
             2        ,                // reqid
             &ret       )            ; // Return code


/******************************************************************************
 * Step 3 : Terminates the KBDITF driver. We have to send to the AUT_KBDITF   *
 * -------  VMIO automaton a REQ_END request event. This is done with a call  *
 *          to the "drv_end" procedure. The AUT_KBDITF VMIO automaton         *
 *          receives this event and then sends back a RESP_END response       *
 *          event.                                                            *
 ******************************************************************************/
```

```
    drv_end(kbd_modd ,                 // MODD of the module to be ended
            3        ,                 // Reqid
            &ret       )             ; // Return code


/*****************************************************************************
 * Step 4 : Delete the route we were using for DRV_KEY_PRESS and             *
 * -------  DRV_KEY_RELEASE events. We have to call the "del_uroute"         *
 *          procedure. As this procedure takes as in put an array containing *
 *          a list of route identifiers, we use  here an array of one entry  *
 *          in which we store "kbd_rteid" before calling "del_uroute".       *
 *          -------------------------------------------------------------- *
 *          We could directly call "del_route" with the address of the       *
 *          "kbd_rteid" global variable because there is no difference       *
 *          between a variable adress and an array address                   *
 *****************************************************************************/

    list[0] = kbd_rteid              ; // Store "kbd_rteid" in a one entry
                                       // array
    del_uroute(list ,                  // We call "del_uroute" with a list
               1     )               ; // of route identifers tant only
                                       // contains one element.

  }


/*  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                                              *
 *****************************************************************************/

    drv_init_driver("DAC"     ,        // VMIO driver name
                    0         ,        // Node number = local (0)
                    1         ,        // reqid value
                    &dac_modd ,        // Module identifier
                    &ret       )     ; // Error code


/*****************************************************************************
 * Step 2 : Now that VMIO DAC driver is ready, we can now issue REQ_ALLOC    *
 * ------   requests to it, in order to allocate a DAC from through which   *
 *          we will convert digital data into analog signal. We use          *
 *          "drv_alloc_device" that sends a REQ_ALLOC request event to the   *
 *          VMIO AUT_DAC automaton. Then we ask to be unscheduled until the  *
 *          RESP_ALLOC response event is received from the IO container      *
 *****************************************************************************/

    drv_alloc_device(dac_modd      ,   // Module Descriptor Id
                     "DAC\\DEV0"   ,   // Device name = DEV0
                     ""            ,   // Tagini  = empty
                     ""            ,   // Taglist = empty
                     NBLOCKING_IO  ,   // Flags
                     2             ,   // Reqid
                     0             ,   // Route ID   = empty
                     0             ,   // Route ID 2 = empty
                     &dac_iod      ,   // Returned I/O descriptor
                     &ret           ); // Return code


/*****************************************************************************
 * Step 3 : Here we initialize the hardware of the DAC. We have to send a    *
 * ------   REQ_OPEN event request to the VMIO AUT_DAC automaton. This is    *
 *          done by calling the "drv_open_device" procedure. When done, the  *
 *          VMIO AUT_DAC automaton will send us back a RESP_OPEN response    *
 *          event.                                                           *
 *****************************************************************************/

    drv_open_device(dac_iod     ,      // I/O descriptor
             (char*)dac_taglist ,      // Taglist = "DEPTH=16\nSFREQ=8000"
                    3           ,      // Reqid
                    &ret          )  ; // Return code

/*****************************************************************************
 * Step 4 : Allocate a memory buffer to hold samples to convert.             *
 * ------   Sample freq at 8kHz. 80evt/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 5 : Fill buffer with samples. We build a 400 Hz triangle wave.       *
 * ------   ~400 Hz at 8kHz => 20 samples per cycles, 10up, 10down           *
 *****************************************************************************/

    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 20 samples to fill the     */
             buf_aud      ,            /* 80 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 : Close the DAC device. We have to call "drv_close_device" for     *
 * ------   DEV0 (so dac_iod). This procedure sends a REQ_CLOSE request      *
 *          event to the VMIO AUT_DAC automaton and wait for the             *
 *          corresponding RESP_CLOSE from VMIO AUT_DAC.                      *
 *****************************************************************************/

    drv_close_device(dac_iod  ,        // IOD of the device to be closed
                     1        ,        // Reqid
                     &ret      )     ; // Return code


/*****************************************************************************
 * Step 2 : Free the DAC device. We have to call "drv_free" for DEV0         *
 * ------   (so dac_iod). This procedure sends a REQ_FREE request event to   *
 *          the VMIO AUT_DAC automaton and wait for the corresponding        *
 *          RESP_FREE from VMIO AUT_DAC.                                     *
 *****************************************************************************/

    drv_free(dac_iod  ,                // IOD of the device to be freed
             2        ,                // Reqid
             &ret       )            ; // Return code


/*****************************************************************************
 * Step 3 : Terminates the DAC driver. We have to send to the AUT_DAC VMIO   *
 * -------  automaton a REQ_END request event. This is done with a call to   *
 *          the "drv_end" procedure. The AUT_DAC VMIO automaton receives     *
 *          this event and then sends back a RESP_END response event.        *
 *****************************************************************************/

    drv_end(dac_modd ,                 // MODD of the module to be ended
            3        ,                 // Reqid
            &ret       )             ; // Return code

/*****************************************************************************
 * Step 4 :
 *****************************************************************************/

    free_buf(buf_aud , &ret )        ; // Free "buf_snd0"



  }


/*  Procedure print_menu -------------------------------------------------------

    Purpose : Prints (sends) on the serial line the application menu
*/

static void print_menu(void)
  {
    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 print the menu. They are simply stored into the      *
 *          static array "menu[]" depending on the current page.             *
 *****************************************************************************/

    lg = strlen(menu[page])          ; // Number of characters to be copied
                                       // and then to be sent on the serial
    Memcpy(buf_snd0,menu[page],lg)   ; // Copy into "buf_snd0" the static part
```

```
                                          // of the menu


/*****************************************************************************
 * Step 2 : Send using serial DEV0, so IOD "asy_iod0", the "lg" characters   *
 * ------   that are in the buffer pointed by "buf_snd0". We call the        *
 *          "write_asy" subroutine that sends the buffer and unschedules us  *
 *          until the effective transmission of the last byte.               *
 *****************************************************************************/

    write_asy(asy_iod0  ,              // I/O descriptor for ASY\DEV0
              buf_snd0  ,              // Address of buffer
              lg        )            ; // Number of bytes to be sent
  }


/*  Procedure print_time ----------------------------------------------------

    Purpose : Prints (sends) on the serial line the current time in seconds
*/

static void print_time(void)
  {
    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.                      *
 * ------                                                                    *
 *****************************************************************************/

    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 number of 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 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 string "HH:MM:SS" representing*
 *            elapsed time. Then we increment the "lg" number of bytes.      *
 *          - Copy into the transmit buffer the ANSI escape sequence that    *
 *            will restore 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                                  *
 *****************************************************************************/

    pt = (char*) buf_snd0            ; // 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)             ; // ANSI Escape Sequence
                                       // 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_snd0". We call the         *
 *          "drv_write" procedure that sends a REQ_WRITE request event to     *
 *          the AUT_ASY VMIO automaton. The we unschedule until the           *
 *          RESP_WRITE response event is received                             *
 *****************************************************************************/

    write_asy(asy_iod0  ,              // I/O descriptor for ASY\DEV0
              buf_snd0  ,              // Address of buffer
              lg         )           ; // Number of bytes to be sent
  }


/*  Procedure print_status-----------------------------------------------------

    Purpose : Prints (sends) on the serial line the 4 string status
*/

static void print_status(void)
  {
    int             i                ; // Loop counter
    int             sz               ; // Length of string item
    char            *pt              ; // Write pointer
    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 print the five 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 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 for each of the 5 statuses:                                  *
 *          - 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_sta[5]). Then we increment the    *
 *            "pt" current write address and the "lg" number of bytes         *
 *          - Copy in the transmit buffer the string "STOPPED" or "STARTED"   *
 *            according to the value of "state[i]". Then increment by 7 the   *
 *            write pointer "pt" and the number of bytes "lg"               *
 *          Then we restore 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.            *
 *****************************************************************************/

    pt = (char*) buf_snd0            ; // 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 write pointer
    lg += sizeof(loc_save) - 1       ; // Increment the number of bytes

    for (i = 0 ; i < 5 ; i++)          // Loop on the five menu statuses
      {                                //
        sz = strlen(loc_sta[i])      ; // Copy into the transmit buffer
        strcpy(pt,loc_sta[i])        ; // the ANSI escape sequence
        pt += sz                     ; // Increment the write pointer
        lg += sz                     ; // Increment the number of bytes

        strcpy(pt, str_sta             // Copy in the transmit buffer
                        [state[i]] ) ; // the 7 character strings according
                                       // to state[i] value.
        pt += 7                      ; // Increment the write pointer
        lg += 7                      ; // 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 2 : Send using serial DEV0, so IOD "asy_iod0", the "lg" characters   *
 * ------   that are in the buffer pointed by "buf_snd0". We call the        *
 *          "drv_write" procedure that sends a REQ_WRITE request event to    *
 *          the AUT_ASY VMIO automaton. The we unschedule until the          *
 *          RESP_WRITE response event is received                            *
 *****************************************************************************/

    write_asy(asy_iod0  ,              // I/O descriptor for ASY\DEV0
              buf_snd0  ,              // Address of buffer
              lg        )            ; // Number of bytes to be sent
  }


/*  Procedure print_asyconfig -------------------------------------------------

    Purpose : Prints (sends) on the serial line the 8 config strings.
*/

static void print_asyconfig(void)
  {
    int             i                ; // Loop counter
    int             sz               ; // Length of string item
    char            *pt              ; // Write pointer
    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 print the five configuration items. 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 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.           *
 *           For each of the nine config items:                               *
 *           - Copy into the transmit buffer the ANSI escape sequence that    *
 *             will move the console cursor to the location where the first   *
 *             config item character will be drawn. We have a static array of *
 *             address strings for that (loc_cfg[8]). Then we increment the   *
 *             "pt" current write address and the "lg" number of bytes        *
 *           - Copy in the transmit buffer the string representig the current *
 *             config. The "str_cfg[][]" contains the converted values. Then  *
 *             we increment the "pt" current write address and the "lg"       *
 *             number of bytes.                                               *
 *           Then we restore 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.           *
 *****************************************************************************/

    pt = (char*) buf_snd0            ; // 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

    for (i = 0 ; i < 9 ; i++)          // Loop on the nine config items
      {                                //
        sz = strlen(loc_cfg[i])      ; // Copy into the transmit buffer
        strcpy(pt,loc_cfg[i])        ; // the ANSI escape sequence
        pt += sz                     ; // Increment the write pointer
        lg += sz                     ; // Increment the number of bytes

        strcpy(pt, str_cfg[i]          // Copy in the transmit buffer
                       [config[i]] ) ; // the 7 character strings according
                                       // to config[i] value.
        pt += 7                      ; // Increment the write pointer
        lg += 7                      ; // 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
    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_snd0". We call the        *
 *          "drv_write" procedure that sends a REQ_WRITE request event to    *
 *          the AUT_ASY VMIO automaton. The we unschedule until the          *
 *          RESP_WRITE response event is received                            *
 *****************************************************************************/

    write_asy(asy_iod0  ,              // I/O descriptor for ASY\DEV0
              buf_snd0  ,              // Address of buffer
              lg         )           ; // Number of bytes to be sent
  }


/*  Procedure print_cmd --------------------------------------------------------

    Purpose : Prints (sends) on the serial line the last command received.
*/

static void print_cmd(int c)
  {
    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 echo.             *
 *****************************************************************************/

    strcpy((char*)buf_snd0 ,           // ANSI Escape Sequence
            loc_cmd[page]          ) ; // Move cursor to line, column

    lg = strlen(loc_cmd[page])       ; // Number of bytes in the buffer

    buf_snd0[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_snd0". We call the        *
 *          "drv_write" procedure that sends a REQ_WRITE request event to    *
 *          the AUT_ASY VMIO automaton. The we unschedule until the          *
 *          RESP_WRITE response event is received                            *
 *****************************************************************************/

    write_asy(asy_iod0  ,              // I/O descriptor for ASY\DEV0
              buf_snd0  ,              // Address of buffer
              lg         )           ; // Number of bytes to be sent
  }


/*  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_rcv0" counter.  *
 *          We also initialize "ret" to ignore the event.                    *
 *****************************************************************************/

    tok_rcv0 --                      ; // One token less.
    ret = OPT_IGNORE                 ; // This character will be ignored


/*****************************************************************************
 * Step 2 : Get the data buffer address. We have just received a READ_RESP   *
 * ------   event. A copy of it 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 between '0' and '9'. Then       *
 *          we convert '0' to OPT_0, '1' to OPT_1, ... If the received byte   *
 *          is not one expected 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                               //
      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
    give_asy_token(asy_iod0 ,          // I/O descriptor
                   1        ,          // Number of receive token
                   1        ,          // Size of receive buffer
                   &tok_rcv0  )      ; // 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_IGNORE                ; // Ignore these events
  }
```

```
/*  Procedure ev_asy1_rcv ------------------------------------------------------

    Purpose : Select the treatment when EV_ASY1_RCV  event is received
*/

static int ev_asy1_rcv(void)
  {
    unsigned char  *buf              ; // Buffer address
    int             len              ; // Data length
    int             err              ; // Return code

/******************************************************************************
 * Step 1 : Update the number of tokens. We just receive one RESP_READ        *
 * ------   event from ASY device DEV1, so we decrement "tok_rcv1" counter.   *
 ******************************************************************************/

    tok_rcv1 --                      ; // One token less.


/******************************************************************************
 * Step 2 : Get the data buffer address. We have just received a READ_RESP    *
 * ------   event. A copy of it 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, 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.                                                *
 ******************************************************************************/

    if ( config[8] EQ CFG_FAST )       // If in fast mode,
      while ( tok_rcv1 LT 2 )          // give at most 2 tokens.
        give_asy_token(asy_iod1 ,      // I/O descriptor
                       1        ,      // Number of receive token
                       8        ,      // Size of receive buffer
                       &tok_rcv1  )  ; // 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 (state[3] 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.                                            *
 *****************************************************************************/

    send_asy_msg(asy_iod1 ,            // I/O descriptor
                 buf      ,            // Address of buffer
                 len      ,            // Number of bytes to send
                 &msg_snd1 )         ; // Number of sending messages

    end                              : // Early exit

    return OPT_IGNORE                ; // Ignore these events
  }


/*  Procedure ev_asy1_snd -----------------------------------------------------

    Purpose : Select the treatment when EV_ASY1_SND event is received.
*/

static int ev_asy1_snd(void)
  {

    unsigned char  *buf              ; // Buffer address
    int             ret              ; // Return code

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

    msg_snd1 -= 1                    ; // One less message in queue


/*****************************************************************************
 * Step 2 : If sent buffer is not the static "buf_snd1" buffer, it is an     *
 * ------   "echo" buffer sent in "ev_asy1_rcv". We need to free it.         *
 *****************************************************************************/

    buf = task_evt.adresse           ; // Get sent buffer addres.
    if ( buf NE buf_snd1 )             // If an "echo" buffer has been sent,
      free_buf(buf, &ret)            ; // free it.


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

    if (state[3] EQ 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_snd1" 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.                                        *
 *****************************************************************************/

    while (msg_snd1 LT 2)               // Until we have sent enough messages,
      send_asy_msg(asy_iod1 ,           // I/O descriptor
                   buf_snd1 ,           // Address of buffer
                   256      ,           // Number of bytes to send
                   &msg_snd1 )        ; // Number of sending messages

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



/*  Procedure ev_gpio0_in ------------------------------------------------------

    Purpose : Parse the received EV_GPIO0_IN event and select the treatment
              to be executed.
*/

static int ev_gpio0_in(void)
  {

/*****************************************************************************
 * 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 press or released. When INPUT is  *
 *          LOW (0), the button has been pressed and this correspond 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                 ; //   button 1 has been pressed.
    else                                // Otherwise,
      return OPT_IGNORE               ; //   it has been released

  }


/*  Procedure ev_gpio1_in ------------------------------------------------------

    Purpose : Parse the received EV_GPIO1_IN event and select the treatment
              to be executed.
*/

static int ev_gpio1_in(void)
  {

/*****************************************************************************
 * 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 press or released. When INPUT is  *
 *          LOW (0), the button has been pressed and this correspond to      *
 *          option BUT2. 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_BUT2                 ; //   button 2 has been pressed.
    else                                // Otherwise,
      return OPT_IGNORE               ; //   it has been released

  }
```

```
/*  Procedure ev_gpio2_in -----------------------------------------------------

    Purpose : Parse the received EV_GPIO2_IN event and select the treatment
              to be executed.
*/

static int ev_gpio2_in(void)
  {

/*****************************************************************************
 * 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 press or released. When INPUT is  *
 *          LOW (0), the button has been pressed and this correspond to      *
 *          option BUT3. 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_BUT3                ; //   button 3 has been pressed.
    else                               // Otherwise,
      return OPT_IGNORE              ; //   it has been released

  }




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

    Purpose : Select the treatment when EV_KBD_RCV event is received
*/

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 tu 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_IGNORE
      return OPT_IGNORE              ; // 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 and YELLOW key codes are         *
 *          respectively KEY_F1, KEY_F2 and KEY_F3                           *
 *****************************************************************************/

    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_F1                  : // RED key
          return OPT_BUT1            ;
        case KEY_F2                  : // GREEN KEY
          return OPT_BUT2            ;
        case KEY_F3                  : // YELLOW key
          return OPT_BUT3            ;
        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_asy1_rcv" as soon as a   *
 *          RESP_READ event is received.                                     *
 *****************************************************************************/

    if ( config[8] EQ CFG_SLOW )       // If in slow mode,
      while ( tok_rcv1 LT 1 )          // give at most 1 token.
        give_asy_token(asy_iod1 ,      // I/O descriptor
                       1        ,      // Number of receive token
                       8        ,      // Size of receive buffer
                       &tok_rcv1  )  ; // Counter of given tokens


/*****************************************************************************
 * Step 2 : Option selected is always the system option OPT_TIME used to     *
 * ------   update time display.                                             *
 *****************************************************************************/

    return OPT_TIME                  ; // Return time refresh option

  }


/*  Procedure wait_coderes ----------------------------------------------------

    Purpose : Unschedule until one specific event is received. If non awaited
              events are received meanwhile, they will be kept inside the task
              events waiting queue.
*/

static void wait_coderes(int code, int res)
  {
    int             ret              ; // Return code
    int             waitlist[1][3]   ; // Parameter of "waitevt_task
```

```
/*****************************************************************************
 * Step 1 : We build a list of a single pair (code,reserve). The task will   *
 * ------   be unschedule until an event corresponding to the one expected   *
 *          is received by the task. This wait may be infinite if the event  *
 *          is never received.                                               *
 *****************************************************************************/

    waitlist[0][0]  = WAIT_CODERES   ; // Test "code" and "reserve" fields
    waitlist[0][1]  = code           ; // The awaited value for "code"
    waitlist[0][2]  = res            ; // The awaited value for "reserve"

    waitevt_task(waitlist ,            // Address of waiting list
                 1        ,            // Size of "waitlist[]"
                 0        ,            // No maximum waiting time
                 0        ,            // Do not purge previous events
                 &ret       )        ; // Return code
  }


/*  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. Accordind to the received event, the retourn 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_WRITE       asy_iod0   ->  EV_ASY0_SND   2
              RESP_WRITE       asy_iod1   ->  EV_ASY1_SND   3
              IND_REPORT       gpio_iod0  ->  EV_GPIO0_IN   10
              IND_REPORT       gpio_iod1  ->  EV_GPIO1_IN   11
              IND_REPORT       gpio_iod2  ->  EV_GPIO2_IN   12
              DRV_KEY_PRESS    any        ->  EV_KBD_RCV    20
              DRV_KEY_RELEASE  any        ->  EV_KBD_RCV    20
              TICK             any        ->  EV_MSEC       30

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

static int wait_myevents(void)
  {
    int             waitlist[6][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 or DEV1. So the     *
 *            awaited code event is RESP_READ and the awaited "reserve"      *
 *            field value may be "asy_iod0" or "asy_iod1". 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" or "asy_iod1". We choose tu 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_iod0" (RED led) "gpio_iod1" (GREEN led) or            *
 *            "gpio_iod2" (YELLOW led)                                       *
 *          - The DRV_KEY_PRESS (RCU key has been pressed) or the            *
 *            DRV_KEY_RELEASE (RCU key has been released)                    *
 *          - 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]  = TICK           ; // an event code TICK
    waitlist[5][2]  = 0              ; // with "reserve" equal to request

    waitevt_task(waitlist ,            // Address of waiting list
                 6        ,            // 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, DRV_KEY_PRESS, DRV_KEY_RELEASE but also  *
 *          the VMK generated event TICK (every second). According to the    *
 *          value of "task_evt.code" but also "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 AYS\DEV0
            ret = EV_ASY1_RCV        ; // and "asy_iod1" foe ASY\DEV1
          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 AYS\DEV0,
            ret = EV_ASY1_SND        ; // "asy_iod1" for ASY\DEV1
          break                      ; //

        case IND_REPORT              : // For a IND_REPORT, the "reserve" field
          if      (res EQ gpio_iod0)   // 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_iod1)   // a IND_REPORT, so here the 3 buttons,
            ret = EV_GPIO1_IN        ; // so the values may be "gpio_iod0",
          else if (res EQ gpio_iod2)   // "gpio_iod1" ou "gpio_iod2" and not
            ret = EV_GPIO2_IN        ; // more
          break                      ; //

        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                       ;

  }
```