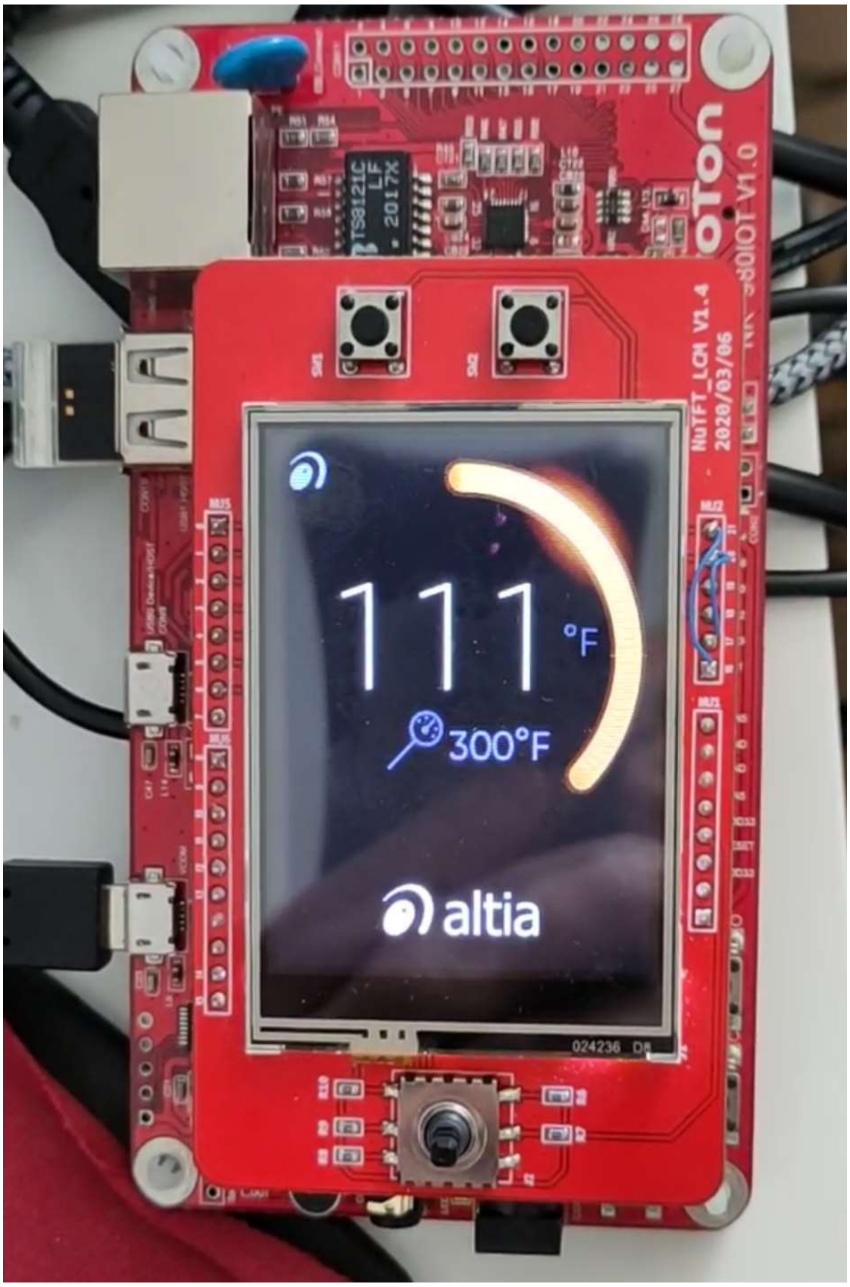
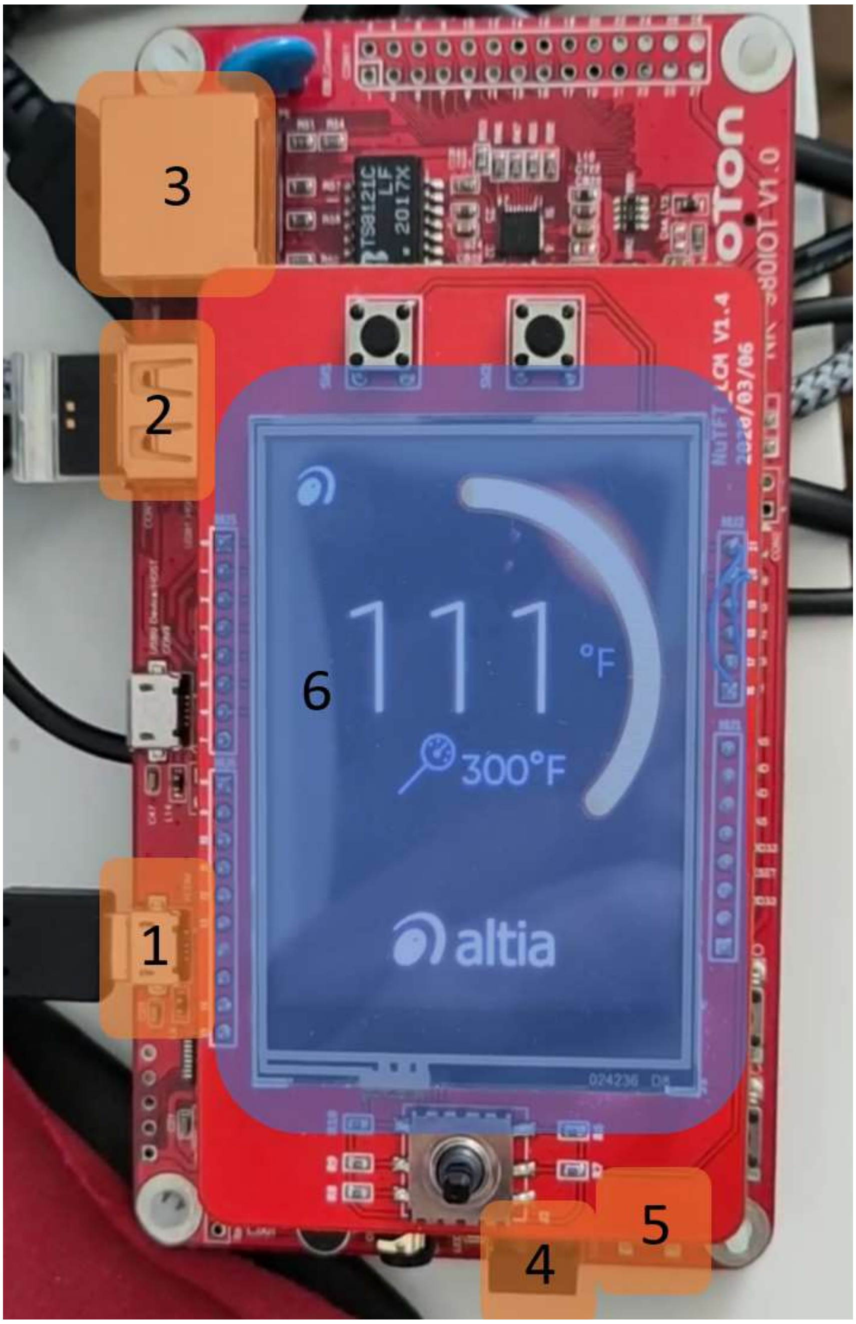
ByteSizedDemo on Nuvoton NuMaker-IIoT-NUC980

Hardware (as provided by Nuvoton)



Connector and Components



1 -> micro-USB connector: connect to USB. When the USB-driver for the HW is installed on the PC, this provides access to the linux console (TeraTerm 11520 8N1)

2 -> USB-A port: Can be used to connect USB-stick (FAT) to the HW. When the Linux image is booted the USB driver of the Linux System is recognizing the plugged in USB-stick.

Use

```
     mount /dev/sda1 /mnt
```

to mount the USB stick to the file system. With

```
     cd /mnt
```

one should be able to go into the USB stick folder/content. From there one is also able to execute built linux applications.

3 -> Ethernet port

4 -> external power supply input connector (5V)

5 -> Reset button (hidden under display shield)

6 -> 240x320 pixel touch display (/dev/fbo ad /dev/input/event0)

How to build the Altia HMI

The ByteSizedDemo project was assigned to the "miniGL SW Render (MASTER) v13.2.1" template (NOT for customers!!!).

Code Gen Options

Important: The FreeType runtime font engine has to be disabled, because first results lead to the HMI crashing (Segmentation Fault) when executed on HW.

Target Toolchain -> gcc_armeabi
Target CPU -> arm926ej-s
Target FPU -> (leave empty) -> requires an adaptation of the altmake_gcc_armeabi.mk make file (since this device does not have an FPU):

```
CDEFINES += \
   MINIGL_INLINE=inline \
   FS_INLINE=inline

CFLAGS += \
   -mcpu=$(strip $(MINIGL_SRT_CPU)) \
   -mthumb

ifeq ($(strip $(MINIGL_SRT_FPU)),)
CFLAGS +=
else
CFLAGS += -mfloat-abi=hard -mfpu=$(strip $(MINIGL_SRT_FPU))
endif  # ifeq ($(strip $(MINIGL_SRT_FPU)),)


CXXFLAGS += \
   -mcpu=$(strip $(MINIGL_SRT_CPU)) \
   -mthumb \
   -fno-exceptions

ifeq ($(strip $(MINIGL_SRT_FPU)),)
CXXFLAGS +=
else
CXXFLAGS += -mfloat-abi=hard -mfpu=$(strip $(MINIGL_SRT_FPU))
endif  # ifeq ($(strip $(MINIGL_SRT_FPU)),)
```

Target Options -> -march=armv5te -mfloat-abi=soft -fno-short-enums

To build the libaltia.a from the generated code Windows version of gcc-arm-none-eabi-6.2.1 compiler was installed and 'set TOOLCHAIN_BASE_PATH= C:\gcc-arm-none-eabi-6.2.1" was set on CMD. Then run altmake.bat.

Build the Executable

When the Altia HMI library is built (one should have a complete 'out' folder holding the built library, the BAM reflash folder and the header files). One can use a bytesizeddemo.c file together with a Makefile to build the bytesizeddemo executable in the Nuvoton Buildroot VMWare environment (https://www.nuvoton.com/resource-download.jsp?tp_GUID=SW1320200406183205):

Makefile:

```
.SUFFIXES : .x .o .c .s

HMI_DIR := .
HMI := ByteSizedDemo

CC := arm-linux-gcc
STRIP := arm-linux-strip
OBJCOPY := arm-linux-objcopy

INCLUDE := -I./out
LIBDIRS := -L./out
LIBS := -laltia
OBJFILES := $(HMI_DIR)/out/reflash/$(HMI)/altia_table_bin.o
$(HMI_DIR)/out/reflash/$(HMI)/images/altia_images_bin.o
$(HMI_DIR)/out/reflash/$(HMI)/fonts/altia_fonts_bin.o

TARGET = bytesizeddemo
SRCS := bytesizeddemo.c

prebuild:
    $(OBJCOPY) -I binary -O elf32-littlearm -B arm --rename-section
    .data=.rodata,alloc,load,readonly,data,contents
    $(HMI_DIR)/out/reflash/$(HMI)/table.bin
    $(HMI_DIR)/out/reflash/$(HMI)/altia_table_bin.o;
    $(OBJCOPY) -I binary -O elf32-littlearm -B arm --rename-section
    .data=.rodata,alloc,load,readonly,data,contents
    $(HMI_DIR)/out/reflash/$(HMI)/images/altiaImageDataPartition0.bin
    $(HMI_DIR)/out/reflash/$(HMI)/images/altia_images_bin.o;
    $(OBJCOPY) -I binary -O elf32-littlearm -B arm --rename-section
    .data=.rodata,alloc,load,readonly,data,contents
    $(HMI_DIR)/out/reflash/$(HMI)/fonts/altiaImageDataPartition0.bin
    $(HMI_DIR)/out/reflash/$(HMI)/fonts/altia_fonts_bin.o;

all:
    $(CC) -Wl,-Map -Wl,bytesizeddemo.map -mthumb -static -
    DMINIGL_UNICODE $(INCLUDE) $(SRCS) $(LIBDIRS) $(LIBS) $(OBJFILES) -
    o $(TARGET)
#        $(STRIP) $(TARGET)

clean:
    rm -f *.o
    rm -f *.x
    rm -f *.flat
    rm -f *.map
    rm -f temp
    rm -f *.img
    rm -f $(TARGET)
    rm -f *.gdb
    rm -f *.bak
```

bytesizeddemo.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <linux/fb.h>
#include <sys/mman.h>
#include <string.h>
#include <linux/input.h>

#include <sys/resource.h>


#include "miniGL.h"
#include "miniGL_software_render.h"
#include "miniGLFunctions.h"

//#define DEBUG

// handle touch

const char *ev_type[EV_CNT] = {
    [EV_SYN]       = "EV_SYN",
    [EV_KEY]       = "EV_KEY",
    [EV_REL]       = "EV_REL",
    [EV_ABS]       = "EV_ABS",
    [EV_MSC]       = "EV_MSC",
    [EV_SW]        = "EV_SW",
    [EV_LED]       = "EV_LED",
    [EV_SND]       = "EV_SND",
    [EV_REP]       = "EV_REP",
    [EV_FF]        = "EV_FF",
    [EV_PWR]       = "EV_PWR",
    [EV_FF_STATUS] = "EV_FF_STATUS",
    [EV_MAX]       = "EV_MAX",
};

const char *ev_code_syn[SYN_CNT] = {
    [SYN_REPORT]    = "SYN_REPORT",
    [SYN_CONFIG]    = "SYN_CONFIG",
    [SYN_MT_REPORT] = "SYN_MT_REPORT",
    [SYN_DROPPED]   = "SYN_DROPPED",
    [SYN_MAX]       = "SYN_MAX",
};

const char *ev_code_abs[ABS_CNT] = {
    [ABS_X]     = "ABS_X",
    [ABS_Y]     = "ABS_Y",
    [ABS_PRESSURE] = "ABS_PRESSURE",
    [ABS_MAX] = "ABS_MAX",
};


const char *ev_code_rel[REL_CNT] = {
    [REL_X]       = "REL_X",
    [REL_Y]       = "REL_Y",
    [REL_Z]       = "REL_Z",
    [REL_RX]      = "REL_RX",
    [REL_RY]      = "REL_RY",
    [REL_RZ]      = "REL_RZ",
    [REL_HWHEEL] = "REL_WHEEL",
    [REL_DIAL]    = "REL_DIAL",
    [REL_WHEEL]  = "REL_WHEEL",
    [REL_MISC]    = "REL_MISC",
    [REL_MAX]     = "REL_MAX",
};

const char *ev_code_key[KEY_CNT] = {
    [BTN_LEFT]     = "BTN_LEFT",
    [BTN_RIGHT]    = "BTN_RIGHT",
    [BTN_MIDDLE]   = "BTN_MIDDLE",
    [BTN_SIDE]     = "BTN_SIDE",
    [BTN_EXTRA]    = "BTN_EXTRA",
    [BTN_FORWARD] = "BTN_FORWARD",
    [BTN_BACK]     = "BTN_BACK",
    [BTN_TASK]     = "BTN_TASK",
    [BTN_TOUCH]    = "BTN_TOUCH",
    [KEY_MAX]      = "KEY_MAX",
};

struct input_event ie;

void print_event(struct input_event *ie)
{
    switch (ie->type) {
    case EV_SYN:
        printf("time:%ld.%06ld\ttype:%s\tcode:%s\tvalue:%d\n",
            ie->time.tv_sec, ie->time.tv_usec, ev_type[ie->type],
            ev_code_syn[ie->code], ie->value);
        break;
    case EV_ABS:
        printf("time:%ld.%06ld\ttype:%s\tcode:%s\tvalue:%d\n",
            ie->time.tv_sec, ie->time.tv_usec, ev_type[ie->type],
            ev_code_abs[ie->code], ie->value);
        break;
    case EV_REL:
        printf("time:%ld.%06ld\ttype:%s\tcode:%s\tvalue:%d\n",
            ie->time.tv_sec, ie->time.tv_usec, ev_type[ie->type],
            ev_code_rel[ie->code], ie->value);
        break;
    case EV_KEY:
        printf("time:%ld.%06ld\ttype:%s\tcode:%s\tvalue:%d\n",
            ie->time.tv_sec, ie->time.tv_usec, ev_type[ie->type],
            ev_code_key[ie->code], ie->value);
        break;
    default:
        break;
    }
}

int touch_fd;
#define TOUCH_DEV "/dev/input/event0"



#define FBDEV "/dev/fb0"
int fbfd;
struct fb_var_screeninfo vinfo;
unsigned short int *displayFramebuffer = NULL;

/* handle FBDEV */
unsigned short int *init_fbdev()
{
    unsigned short int *fbpointer = NULL;
    int fd = open(FBDEV, O_RDWR);

    if(fd >= 0)
    {
        int fbsize;

        ioctl(fd, FBIOGET_VSCREENINFO, &vinfo);

        fbsize = vinfo.xres * vinfo.yres * (vinfo.bits_per_pixel/8);

        fbpointer = (unsigned short int *)mmap(0, fbsize, PROT_READ |
        PROT_WRITE, MAP_SHARED, fd, (off_t)0);
    }

    return fbpointer;
}

void fillFrameBuffer(unsigned short int *fbbuffer, short int color)
{
    int x,y;

    for(y = 0; y < vinfo.yres; y++)
    {
        for(x = 0; x < vinfo.xres; x++)
        {
            *(fbbuffer + y * vinfo.xres + x) = color;
        }
    }
```

```
}

void copyFrameBuffer(unsigned short int *source, unsigned short int
*destination)
{
#if 1
    int x,y;

    for(y = 0; y < vinfo.yres; y++)
    {
        for(x = 0; x < vinfo.xres; x++)
        {
            *(destination + y * vinfo.xres + x) = *(source + y *
            vinfo.xres + x);
        }
    }
#endif
}

/* framebuffer */
#define DISPLAY_WIDTH 240
#define DISPLAY_HEIGHT 320
#define BYTES_PER_PIXEL 2
unsigned char *framebuffer = NULL;

extern const unsigned int
_binary___out_reflash_ByteSizedDemo_table_bin_start;
MINIGL_UINT8 *altiaTable = (MINIGL_UINT8
*)&_binary___out_reflash_ByteSizedDemo_table_bin_start;

extern const unsigned int
_binary___out_reflash_ByteSizedDemo_images_altiaImageDataPartition0_bin
_start;
MINIGL_UINT8 *altiaImages = (MINIGL_UINT8
*)&_binary___out_reflash_ByteSizedDemo_images_altiaImageDataPartition0_
bin_start;

extern const unsigned int
_binary___out_reflash_ByteSizedDemo_fonts_altiaImageDataPartition0_bin_
start;
MINIGL_UINT8 *altiaFonts = (MINIGL_UINT8
*)&_binary___out_reflash_ByteSizedDemo_fonts_altiaImageDataPartition0_b
in_start;

MINIGL_UINT8 * bsp_ReflashQueryString(MINIGL_CONST MINIGL_UINT16 *
string)
{
    MINIGL_CONST MINIGL_UINT8 *address = NULL;

    char local_string[128] = {0};
    MINIGL_UINT32 count = 0;

    if(string == NULL)
    {
        return altiaTable;
    }

    while((count < 128) && (string[count] != 0x0000))
    {
        local_string[count] = (MINIGL_UINT8)string[count];
        count++;
    }

    if(0 == strcmp(local_string, "\\images"))
    {
        address = altiaImages;
    }
#if 1
    else if(0 == strcmp(local_string, "\\fonts"))
    {
        address = altiaFonts;
    }
#endif
    return (MINIGL_UINT8 *)address;
}

void* bsp_GetBackFrameBuffer()
{
#ifdef DEBUG
    printf("bsp_GetBackFrameBuffer() called!\n");
#endif

    return (void *)framebuffer;
}

void updateRectangle(unsigned char *source, unsigned char *destination,
int x0, int y0, int x1, int y1)
{
    int line_cnt;
    int lines = y1 - y0;
    int cpy_length = x1 - x0;

    unsigned char *src = source + y0 * vinfo.xres + x0;
    unsigned char *dst = destination + y0 * vinfo.xres + x0;

    for(line_cnt = 0; line_cnt < lines; line_cnt++)
    {
        memcpy((void *)dst, (void *)src, (size_t)(cpy_length *
vinfo.bits_per_pixel / 8));

        src += vinfo.xres * vinfo.bits_per_pixel / 8;
        dst += vinfo.xres * vinfo.bits_per_pixel / 8;
    }
}
```

```
MINIGL_INT bsp_DriverFlushed(MINIGL_CONST MINIGL_INT16
dirtyExtentCount, MINIGL_CONST BSP_CLIPPING_RECTANGLE *
dirtyExtentList)
{
    int i;

#ifdef DEBUG
    printf("bsp_DriverFlushed() called!\n");

    printf("Source Buffer Address: 0x%x\n", (unsigned int)framebuffer);
    printf("Destination Buffer Address: 0x%x\n", (unsigned
int)framebuffer);

    printf("dirtyExtentCount: %i\n", dirtyExtentCount);
#endif /* DEBUG */

    for(i = 0; i < dirtyExtentCount; i++)
    {
#ifdef DEBUG
        printf("dirtyExtentList[%i].x0 = %i\n", i,
dirtyExtentList[i].x0);
        printf("dirtyExtentList[%i].y0 = %i\n", i,
dirtyExtentList[i].y0);
        printf("dirtyExtentList[%i].x1 = %i\n", i,
dirtyExtentList[i].x1);
        printf("dirtyExtentList[%i].y1 = %i\n", i,
dirtyExtentList[i].y1);
#endif
        updateRectangle((unsigned char *)framebuffer, (unsigned char
*)displayFramebuffer, dirtyExtentList[i].x0, dirtyExtentList[i].y0,
dirtyExtentList[i].x1, dirtyExtentList[i].y1);
    }

//    copyFrameBuffer((unsigned short int *)framebuffer, (unsigned
short int *)displayFramebuffer);
}

void bsp_ErrorHandler(MINIGL_CONST MINIGL_UINT32 errorCode,
MINIGL_CONST MINIGL_UINT32 majorCode, MINIGL_CONST MINIGL_UINT32
minorCode)
{
    printf("bsp_ErrorHandler() called!\n");
    printf("Error Code: %lu\r\n", errorCode);
    printf("Error MajorCode: %lu\r\n", majorCode);
    printf("Error MinorCode: %lu\r\n", minorCode);
}

typedef struct
{
    /*
     * Set once during initialization
     */
    MINIGL_UINT16 dispWidth, dispHeight;       /* dimensions of
associated display */

    /*
     * Event based state updated by bsp_GetEvent()
     */
    MINIGL_UINT16 evFlag;                         /* OR of EVENT_* bits
*/
    MINIGL_UINT32 pen;                       /* pending mouse or touch
*/
    MINIGL_UINT32 x;
    MINIGL_UINT32 y;
} INPUT_DEVICE;

INPUT_DEVICE input_device;

/*
 * Bit masks to use for evFlag field in INPUT_DEVICE struct
 */
#define EVENT_ABS_POS                           0x0001
#define EVENT_BUTTON_PRESSED                    0x0002
#define EVENT_BUTTON_RELEASED                   0x0004
#define EVENT_SYN_DROPPED                       0x0008
#define EVENT_SYN_MT_REPORT                     0x0010
#define EVENT_ABS_MT_ANY                        0x0020
#define EVENT_ABS_MT_POS                        0x0040
#define EVENT_ABS_MT_SLOT                       0x0080
#define EVENT_ABS_MT_PRESSURE                   0x0100

/*
 * Macros to set/clear/test evFlag bit fields in INPUT_DEVICE struct
 */
#define EVENT_FLAG_SET(d,b)                     ((d)->evFlag |= (b))
#define EVENT_FLAG_CLR(d,b)                     ((d)->evFlag &= ~(b))
#define EVENT_FLAG_IS_SET(d,b)                  (0 != ((d)->evFlag &
(b)))
#define EVENT_FLAG_IS_CLR(d,b)                  (0 == ((d)->evFlag &
(b)))

//#define DEBUG_TOUCH
MINIGL_INT bsp_GetEvent(BSP_STIMULUS_EVENT * event, MINIGL_INT32
timeout)
{
    int read_bytes = 0;
    int ret_value = 0;

#ifdef DEBUG_TOUCH
    printf("bsp_GetEvent() called!\n");
#endif  /* DEBUG_TOUCH */

    do
    {
        read_bytes = read(touch_fd, &ie, sizeof(struct input_event));
        if(read_bytes >= 0)
        {
#ifdef DEBUG_TOUCH
            print_event(&ie);
#endif /* DEBUG_TOUCH */

            if((ie.type == EV_SYN) && (ie.code == SYN_REPORT))
            {
                if(EVENT_FLAG_IS_CLR(&input_device, EVENT_SYN_DROPPED))
                {
                    if(EVENT_FLAG_IS_SET(&input_device, EVENT_ABS_POS))
                    {
                        EVENT_FLAG_CLR(&input_device, EVENT_ABS_POS);

                        event->x = input_device.x;
                        event->y = input_device.y;

                        ret_value = 1;
                    }

                    if(EVENT_FLAG_IS_SET(&input_device,
EVENT_BUTTON_PRESSED))
                    {
                        EVENT_FLAG_CLR(&input_device,
EVENT_BUTTON_PRESSED);

                        event->pen = 1;

                        ret_value = 1;
                    }

                    if(EVENT_FLAG_IS_SET(&input_device,
EVENT_BUTTON_RELEASED))
                    {
                        EVENT_FLAG_CLR(&input_device,
EVENT_BUTTON_RELEASED);

                        event->pen = 0;

                        ret_value = 1;
                    }

                    // clear all event flags
                    input_device.evFlag = 0;
                }
            }

            else if((ie.type == EV_SYN) && (ie.code == SYN_DROPPED))
            {
                EVENT_FLAG_SET(&input_device, EVENT_SYN_DROPPED);
            }
            else if((ie.type == EV_ABS) && (ie.code == ABS_X))
            {
                input_device.x = (MINIGL_INT16)((240 / 4096.9) *
ie.value);

                EVENT_FLAG_SET(&input_device, EVENT_ABS_POS);
            }

            else if((ie.type == EV_ABS) && (ie.code == ABS_Y))
            {
                input_device.y = (MINIGL_INT16)((320 / 4096.9) *
ie.value);

                EVENT_FLAG_SET(&input_device, EVENT_ABS_POS);
            }

            else if((ie.type == EV_KEY) && (ie.code == BTN_TOUCH))
            {
                input_device.pen = ie.value;

                if(ie.value == 1) EVENT_FLAG_SET(&input_device,
EVENT_BUTTON_PRESSED);
                if(ie.value == 0) EVENT_FLAG_SET(&input_device,
EVENT_BUTTON_RELEASED);
            }
        }
    } while( read_bytes > 0 );

    return ret_value;
}

int main(void)
{
    const rlim_t kStackSize = 64L * 1024L;   // min stack size = 64 kb
    struct rlimit rl;
    int result;

    result = getrlimit(RLIMIT_STACK, &rl);
    if (result == 0)
    {
        if (rl.rlim_cur < kStackSize)
        {
            rl.rlim_cur = kStackSize;
            result = setrlimit(RLIMIT_STACK, &rl);
            if (result != 0)
            {
                fprintf(stderr, "setrlimit returned result = %d\n",
result);
            }
        }
    }

    // open touch device
    if ((touch_fd = open(TOUCH_DEV, O_RDONLY | O_NONBLOCK)) == -1) {
        printf("Failed to open touch device!");
        while(1);
    }

    // open display device
    displayFramebuffer = init_fbdev();

    framebuffer = (unsigned char *)malloc((size_t)(DISPLAY_WIDTH *
DISPLAY_HEIGHT * BYTES_PER_PIXEL));

    // start Altia HMI
    if(altiaInitDriver() < 0)
    {
        printf("altiaInitDriver() failed!\n");
        while(1);
    }

    /* draw first screen */
    altiaSendEvent(ALT_ANIM(TimerEnable), 0);
    altiaFlushOutput();

    for(;;)
    {
        altiaPending();

        /* run altia HMI */
        altiaSendEvent(ALT_ANIM(do_timer), 1);
            altiaFlushOutput();
    }

    return 0;
}
```

To build the demo run the VMWare Linux PC:

```
> make prebuild
> make all
```

How to Deploy and Run ByteSizedDemo

The ready built demo executable can be downloaded from here:
https://altia.box.com/s/0hxfxrrqhh8u9q2fs0fgyyckxxmmlxv3

Copy the file bytesizeddemo onto an USB-stick and connect the USB stick to USB-port (2), mount the USB-stick to the Linux file system: `mount /dev/sda1 /mnt`

Change tothe folder 'mnt': `cd /mnt`

Run the demo: `./bytesizeddemo`

Remark: The HW comes with an emWIN demo auto-started at boot. To run the Altia demo login to the Linux console (root, No Password) and execute: `killall GUIDemo`

This will stop the GUIDemo and the Altia ByteSizedDemo can be executed.