ByteSizedDemo on Nuvoton NuMaker-IIoT-NUC980

**Hardware (as provided by Nuvoton)**

기憎날:서 
• 년-_그`•`-궈이OT VI .0 

**Connector and Components**

T%121C 
• 2ΟΙ7Χ 
NuTFT νι.4 

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

#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\_bin\_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](file:///\\images)"))

{

address = altiaImages;

}

#if 1

else if(0 == strcmp(local\_string, "[\\fonts](file:///\\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.