We will integrate Bluetooth Low Energy (ble) software with bsp and rtx.

We will use the following directory structure:

proj

<nrf> 🡨 nRF5 SDK downloaded from Nordic site

ble 🡨 BLE example from <nrf>, modified for microbit

rtx 🡨 RTX OS files, from an earlier exercise

bsp 🡨 BSP files from earlier exercise

main.c 🡨 your code

Create a directory proj in your working area to begin.

# Part 1: Download nRF5 SDK and make modifications for Microbit:

1. Download nRF5 SDK from:

<https://www.nordicsemi.com/Products/Development-software/nRF5-SDK/Download>

(Download only SDK. No need to download various softdevice s\*.zip files.)

1. Unzip DeviceDownload.zip. You will get nRF5\_SDK\_x.zip file. (x = version)

Unzip nRF5 zip file inside proj folder. We will refer to this folder as <nrf>.

git add <nrf>

git commit -m "downloaded from nordicsemi website"

1. Change gcc path and version in <nrf>/components/toolchain/gcc/Makefile.posix
2. We need to add Microbit board file in <nrf>/components/boards directory. Copy one of the existing board files (say, pca10040) and make changes as per Microbit schematics. (I have done so and attached microbit.h here).

git add microbit.h

1. Modify nrflib/components/boards/boards.h to include Microbit. Add the following lines after arduino\_primo board:

#elif defined (BOARD\_MICROBIT)

#include "microbit.h"

1. Change NRF52832 to NRF52833 in modules/nrfx/mdk/nrf.h:

/\* Redefine "old" too-generic name NRF52 to NRF52833\_XXAA to keep backwards compatibility. \*/ 🡨 change here

#if defined (NRF52)

#ifndef NRF52833\_XXAA 🡨 change here

#define NRF52833\_XXAA 🡨 change here

#endif

#endif

1. git commit -a -m "changes for microbit"

# Part 2: Build a BLE (Bluetooth Low Energy) application

Reference reading for softdevice: (also available in Teams:Files/Ref)

<https://infocenter.nordicsemi.com/index.jsp?topic=%2Fsds_s113%2FSDS%2Fs1xx%2Fs113.html>

1. Copy <nrf>/examples/ble\_peripheral/ble\_app\_uart as directory ble inside proj directory.
2. We will use files from pca10100 board (it uses NRF52833, same as Microbit). We will simplify the directory structure and remove the unnecessary files.

cd ble

mv pca10100/s113/armgcc/\* . # Makefile and linker script

mv pca10100/s113/config/sdk\_config.h . # Configuration

rm -rf pca\* ble\_app\_uart.eww # remove other boards and tools

1. Copy the correct softdevice hex file and remove others:

cp <nrf>/components/softdevice/s113/hex/s113\_nrf52\_7.2.0\_softdevice.hex hex

rm hex/ble\*.hex # remove other soft devices

git add .

git commit -m "copied from ble\_app\_uart example"

1. Modify configuration: pca10100 uses a crystal for low frequency clock used by BLE. Microbit uses internal RC oscillator. We need to reflect this change in sdk\_config.h.

#define NRF\_SDH\_CLOCK\_LF\_SRC 0 # change from 1 to 0

#define NRF\_SDH\_CLOCK\_LF\_RC\_CTIV 16 # change from 0 to 16

#define NRF\_SDH\_CLOCK\_LF\_RC\_TEMP\_CTIV 2 # change from 0 to 2

#define NRF\_SDH\_CLOCK\_LF\_ACCURACY 1 # change from 7 to 1

1. Make the following changes in Makefile:

Change BOARD\_PCA10100 to BOARD\_MICROBIT

Change SDK\_ROOT := ../../../../../.. to (path of nRF5 SDK)

Change PROJ\_DIR := ../../.. to .

INC\_FOLDERS: change line containing ../config to $(PROJ\_DIR)

After the following line:

CFLAGS += -Wall -Werror

Add the line below:

CFLAGS += -Wno-array-bounds

(We ignore this warning as the build fails with array-bounds error. Maybe the array-bounds

check was introduced in a later version of the tool-chain.)

1. Run make. It should generate \_build/nrf52833\_xxaa.hex.
2. git commit -a -m "ble changes for microbit"

## Test the application:

1. We need a tool to merge soft device and application hex files. To do so, download command line tools from <https://www.nordicsemi.com/Products/Development-tools/nrf-command-line-tools/download#infotabs>

Download tar.gz and untar in /opt/. Add /opt/nrf-command-line-tools/bin to PATH in your .bashrc.

1. Once done run:

mergehex -m hex/s113\_nrf52\_7.2.0\_softdevice.hex \_build/nrf52833\_xxaa.hex -o MICROBIT.hex

Copy MICROBIT.hex to the target hardware.

(You can write a small script build.sh that runs make and mergehex and then copies MICROBIT.hex to Microbit drive.)

1. To test, download nRF Toolbox from Google store.
2. Run UART service and select Nordic\_UART device. It opens up a message window where you can type messages that will show up in UART terminal window and vise-versa.

# Part 3: Convert the ble\_uart app into a library.

1. First, let us create the interface file ble\_uart.h with the following lines:

#ifndef BLE\_UART\_H

#define BLE\_UART\_H

typedef void (\*ble\_recv\_handler\_t)(const uint8\_t data\_buf[], uint32\_t len);

extern void ble\_init(ble\_recv\_handler\_t handler);

extern void ble\_send(uint8\_t data\_array[], uint16\_t len);

#endif /\* BLE\_UART\_H \*/

It is a simple interface. Init takes a callback function as a parameter that is called whenever a block of data is received on BLE interface. And a function is provided to send a block of data to BLE.

1. Next, we will change main.c to incorporate the above interface. The example code does a crisscross: data from UART is sent to BLE and data from BLE is sent to UART. We will replace the UART part with functions to send/receive data to/from the BLE interface.

Attached file ble\_uart.c contain these changes.

Currently we have:

uart\_evt\_handle (from UART) 🡪 ble\_nus\_data\_send (to BLE)

nus\_data\_handler (from BLE) 🡪 uart\_putc (to UART)

Instead, we will have:

uart\_evt\_handle: dummy

ble\_send (from application) 🡪 ble\_nus\_data\_send (to BLE)

nus\_data\_handler (from BLE) 🡪 ble\_recv\_handler (to application)

1. Rename main.c to ble\_uart.c. Then make the following changes:

* Add these two lines after all the include files:

#include "ble\_uart.h" // interface functions

ble\_recv\_handler\_t m\_ble\_recv\_handler; // receive callback handler

* Rename function main to void ble\_ init(ble\_recv\_handler\_t handler);

Remove the main loop at the end of all the initializations.

// Enter main loop.

for (;;)

{

idle\_state\_handle();

}

Add the line:

m\_ble\_recv\_handler = handler;

1. Modify uart\_event\_handle code to implement ble\_send:

void uart\_event\_handle(app\_uart\_evt\_t \*p\_event) { } 🡨 make empty

void ble\_send(char data[], int len)

Instead of reading a character from UART, read it from data[].

1. Modify nus\_data\_handler code to call receive handler instead of echoing characters on a terminal.
2. Write a main.c to test the code. Add main.c to Makefile.
3. Run UART utility in nRF Toolbox. You should see the characters you send getting echoed back.
4. Save the changes in git

git status # make sure other files are not accidentally changed

git commit -a -m "ble\_uart library"

(Take a coffee break. The next step is involving and requires concentration.)

# Part 4: Integrate BSP

1. Copy bsp lib from ex4-int. (The current bsp may also work, but we start from a known point.)

cp -r ex4-int/lib bsp

git add bsp

git commit -m "bsp added"

1. Delete the interrupt related code from gpio.c – all the lines after “Interrupt related configuration”. (BLE code is using GPIO interrupt for detecting button-press to come out of sleep mode.)
2. Get ld files:

cp ble/ble\_app\_uart\_gcc\_nrf52.ld ble\_uart.ld

cp <nrf>/modules/nrfx/mdk/nrf\_common.ld common.ld

Change the line “INCLUDE nrf\_common.ld” to “INCLUDE common.ld” in ble\_uart.ld file.

Remove the following line from ble\_uart.ld file:

GROUP(-lgcc -lc -lnosys)

(We are using ld as linker, NRF Makefile uses gcc as linker. We are linking the libraries by providing their path in the linker command.)

1. Changes in startup file:
   1. Rename ResetISR 🡪 Reset\_Handler (as used in common.ld)
   2. Rename the section name of vector table from .vectors to .isr\_vector (as used in common\_ld)
   3. The linker file expects the stack to be allocated in .stack section. (In our case, we had allocated through linker command file.) We can change either the linker command file common.ld or change startup.c. Here, we will modify startup.c.

Add stack just before the vector table:

#define STACK\_SIZE (8192/4) // stack size in long words

\_\_attribute\_\_ ((section(".stack")))

uint32\_t stack[STACK\_SIZE];

Change the vector table entry for initial SP to:

(pfn\_t) (long) (stack + STACK\_SIZE),

* 1. Add these handlers in the vector table:
     1. RTC1\_IRQHandler: used during Bluetooth pairing
     2. SWI2\_EGU2\_IRQHandler: used by BLE SoftDevice to send data to the app
  2. BLE uses some initialized sections before and after .data section. So crt\_init should copy memory after end of text (\_\_etext) and before bss (\_\_bss\_start\_\_).

// Copy the data segment initializers from flash to SRAM.

pui32Src = &\_\_etext;

pui32Dest = &\_\_data\_start\_\_;

while(pui32Dest < &\_\_bss\_start\_\_)

\*pui32Dest++ = \*pui32Src++;

1. Since we will use serial functions from BSP, we should remove UART related code from ble. From ble/ble\_uart.c, remove uart\_init function definition and its call inside ble\_init. Also remove uart\_event\_handle.
2. Copy main.c from ble directory for testing.

cp ble/main.c main.c

1. Change the function main in main.c to test coexistence of bsp with ble:

int main(void)

{

/\* BSP initializations before BLE because we are using printf from BSP \*/

board\_init();

ble\_init(ble\_recv\_handler);

/\* Greetings \*/

printf("hello, world!\n");

audio\_sweep(500, 2000, 100);

while (1)

;

}

## Build and test

1. Build ble library inside ble folder:

Write the following script build\_lib.sh to build a library:

EXPORT PATH=$PATH:<path-to-arm-gcc>/bin

make

rm -f libble\_uart.a

arm-none-eabi-ar -r libble\_uart.a \_build/nrf52833\_xxaa/\*.o

arm-none-eabi-ar -d libble\_uart.a gcc\_startup\_nrf52833.S.o

arm-none-eabi-ar -d libble\_uart.a system\_nrf52833.c.o

Running “make” creates all the object files required for the application in \_build/nrf52833\_xxaa directory as \*.c.o files. We simply add these files in a library. (Jai Hanuman!)

Since we are using our own startup code, we have removed both startup and system files from ble\_uart library.

1. Build bsp library using build\_lib.sh.
2. Write the main build.sh script. Copy your earlier build.sh script and make the following changes:
   1. Add BLEDIR=ble
   2. Add -I$BLEDIR to CFLAGS
   3. Change linker command file: -T ble\_uart.ld
   4. Change entry point to Reset\_Handler from ResetISR.
   5. Add $BLEDIR/libble\_uart.a ***after*** $BSPDIR/libbsp.a. Add after, because if there is a clash in the symbol, we want to select from BSP.
   6. Add library: “$ARMGCC/arm-none-eabi/lib/thumb/v7e-m+fp/hard/libnosys.a”
   7. Add lines to merge hex files.

That’s it, hopefully 😊

Build and run the BLE loopback test as in Part 3.

1. Once running, verify changes (git status) and commit with an appropriate comment.

# Part 5: Integrate RTX

1. Copy rtx directory from your earlier project.

cp -r <path-to-rtx> rtx

git add rtx

git commit -m "added rtx"

1. Modify bsp/startup.c to match the handler names used in rtx: SVC\_Handler, PendSV\_Handler, SysTick\_Handler.

Add these entries in the vector table in place of IntDefaultHadler.

1. Remove timer.c and timer.h (git remove). Also remove these from build\_lib.sh.
2. BLE softdevice is implemented using SVCall and requires priority to be 4. To do so, change the following line in rtx\_core\_cm.h.

SCB->SHP[7] = (uint8\_t)(0xFCU << n); 🡨 Change 0xFEU to 0xFCU to change priority from 6 to 4.

1. Add RTX in build.sh.

Add RTXDIR variable.

Add -I$RTXDIR to CFLAGS

Add $RTXDIR/librtx.a to linker command line.

## Testing with RTOS:

Write main.c using multithreading code.

ble\_recv\_handler (callback from SWI2 ISR):

Copy data to message buffer

Send message to waiting thread

Thread:

Recv message

Print the string on UART

Echo the string to BLE

Once tested, verify and add changes to git.