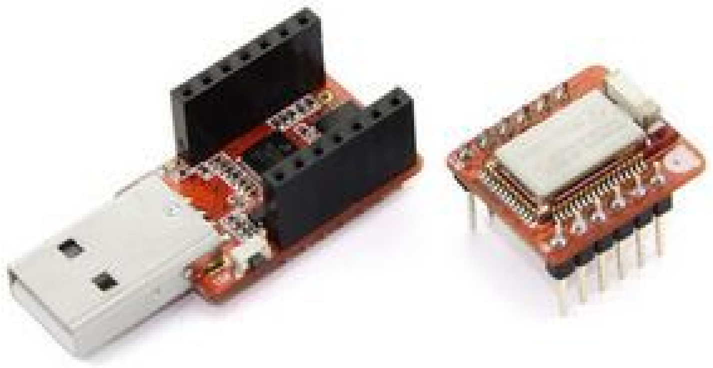


The BLE Nano and MK20 USB programming board from Red Bear Labs is a Bluetooth Smart (also known as Bluetooth 4.1 Low Energy) development board. It utilizes a Nordic Semiconductor nRF51822 chip, which is an ARM Cortex-M0 SoC which supports both BLE Central and Peripheral Roles and runs at 16MHz. The product page can be found here:http://redbearlab.com/blenano/

For more information on Bluetooth Smart technology and protocols, see https://learn.adafruit.com/introduction-to-bluetooth-low-energy/introduction

PROGRAMMING: There are three main ways of programming the BLE Nano

Arduino Library: RBL has created an Arduino Library for the nRF51822. At this stage there are several examples but only BLE Peripheral ROles are supported.

mbed's Bluetooth Low Energy API: the mbed platform is a series of free software libraries developed by ARM. There is an online compiler and several libraries specific to the BLE Nano platform. The software is highly abstracted and makes it very easy to program the Nano. Unfortunately as of 5/29/15 there is still only support for BLE Peripheral roles. The forums suggest they are working on adding Central Role capability, but there's nothing out yet. RBL offers an excellent guide on how to use the Nano with mbed: http://redbearlab.com/getting-started-nrf51822

Nordic nRF51822 BLE SDK: The official firmware development SDK for use with ARM GCC. For access to the full power of the nRF51822 chip, including BLE Central role, this is what you must use. The documentation available on RBL's site is from years ago and will not work with the BLE Nano! The various libraries needed are continually changing versions and breaking each other, so I will be noting versions of everything that I got to work together. Luckily, most of the downloads include links to previous versions if you're reading this from far in the future.

A note on supported platforms: There are Windows, Mac, and Linux versions of the compile and flashing processes. If I manage to get the Windows version running I will come back and update this document, but for now only the Linux version works on my machine, running Ubuntu 14.04 LTSx64.

Nordic SDK in Linux:

First, assemble your Nanos using the supplied headers and then follow this guide https://developer.mbed.org/platforms/RedBearLab-BLE-Nano/ (at the bottom) on flashing firmware. You will need to flash the interface firmware only once, and you should test every Nano you plan on using with the "Out of box" steps. Note that the LED on the BLE Nano (not the one on the MK20 USB Board) is on the opposite side from the nRF51 chip, most likely facing the USB dongle and difficult to see (thanks for that).

The toolchain executables are 32-bit apps, so you will need the 32-bit libraries as well as srec_cat:

```
sudo apt-get install lib32z1 lib32ncurses5 lib32bz2-1.0 srecord
```

Get the GNU Tools for ARM Processors from here: https://launchpad.net/gcc-arm-embedded/+download. For my installation I needed to use gcc-arm-none-eabi-4_8-2014q1-20140314-linux.tar.bz2. You can try to use the latest version, and if it will not work there will be an error during compile that will tell you which version the applications are looking for. You can have multiple versions installed. To install:

```
$ cd /usr/local

$ sudo tar xjf ~/Downloads/gcc-arm-none-eabi-4_8-2014q1-20140314-linux.tar.bz2
```

You can now test if the compiler is working appropriately:

```
$ /usr/local/gcc-arm-none-eabi-4_8-2014q1/bin/arm-none-eabi-gcc --version
```

This should result in the printout of some version information. Next, download the nRF51 SDK from Nordic. For me this was found here: https://developer.nordicsemi.com/nRF51_SDK/nRF51_SDK_v8.x.x/ and was the 8.1.0 version. Then we install the SDK:

```
$ cd

$ mkdir nRF51_SDK_8.1.0

$ cd nRF51_SDK_8.1.0

$ unzip ~/Downloads/nRF51_SDK_8.1.0_b6ed55f.zip

$ NRF51_SDK=$HOME/nRF51_SDK_8.1.0
```

Next we install a custom board file, created by James Wilcox for the RBL community. Download the file here: http://redbearlab.zendesk.com/attachments/token/fQCYu6RkcjOKZiOAxL0hfcDK9/?name=custom_board.h. Place the file in with the other boards by running:

```
$ cp ~/Downloads/custom_board.h $NRF51_SDK/examples/bsp
```

Next, we will modify the Makefile and linker script to allow for compiling for the Nano.

```
$ cd $NRF51_SDK/examples/ble_peripheral/ble_app_hrs/pca10028/s110/armgcc

$ gedit Makefile
```

Find 'CFlags += -DBOARD_PCA10028' and change it to 'CFlags += -DBOARD_CUSTOM' and save.

```
$ gedit ble_app_hrs_gcc_nrf51.ld
```

Find 'RAM (rwx) : ORIGIN = 0x20002000, LENGTH = 0x6000'. Subtract 0x4000 from the LENGTH variable, changing it to 'LENGTH = 0x2000' and save. This needs to be done because the latest SDK expects the Nano to have 32KB of RAM instead of the 16KB it has. Now, we can compile!

```
$ make all
```

If you encounter errors, look at the logs and determine their source. For me, that was an incorrect version of GCC tools. Otherwise, we can move on to uploading to the Nano. With the Nano plugged in using the MK20 USB Dongle:

```
$ cd _build

$ SOFTDEVICE=$NRF51_SDK/components/softdevice/s110/hex/s110_softdevice.hex

$ OUTPUT=output.hex

$ INPUT=nrf51422_xxac_s110.hex

$ srec_cat $SOFTDEVICE -intel $INPUT -intel -o $OUTPUT -intel --line-length=44

$ cp $OUTPUT /media/$USER/MBED
```

This will combine the sample heartrate monitor application with the S110 softdevice and flash the Nano. If it worked, the Nano's LED will now be flashing slowly. To further test, you can download Nordic's "nRF Master Control Panel" app for iOS and Android. This will let you scan for available Bluetooth peripherals, and the Nano should show up as "Nordic_HRM". Connecting to Nordic_HRM will let you see that there are two services running, a Heart Rate Monitor that counts up and down every second and a Battery Status service that counts every two seconds. Next, we perform a very similar operation to get another Nano set up as a master, or Central Device.

BLE Nano as Central Device (Linux)

The app we are going to install on the Nano will collect the HRM messages from the Nano we flashed above and output them over UART. If you are starting here from scratch, you will need to follow the installation steps from above, including the creation of the temporary system variable $NRF51_SDK.

```
$ cd $NRF51_SDK/examples/ble_central/ble_app_hrs_c/pca10028/s120/armgcc

$ gedit Makefile
```

Find 'CFlags += -DBOARD_PCA10028' and change it to 'CFlags += -DBOARD_CUSTOM' and save.

```
$ gedit ble_app_hrs_c_gcc_nrf51.ld
```

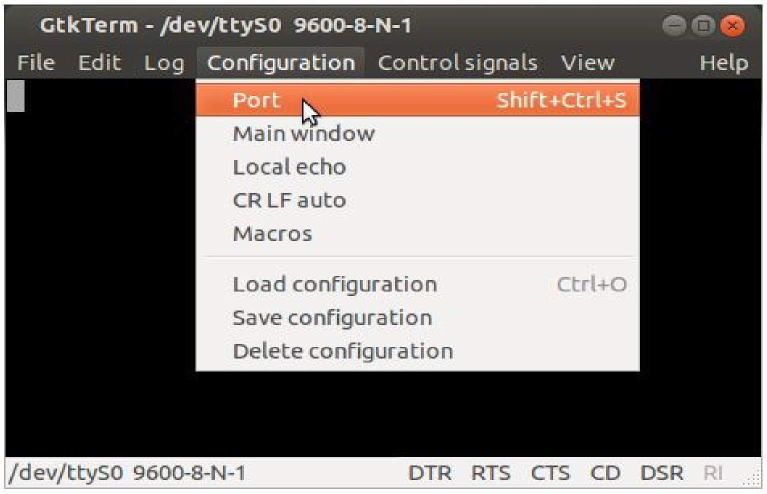
Find 'RAM (rwx) : ORIGIN = 0x20002800, LENGTH = 0x5800'. Subtract 0x4000 from the LENGTH variable, changing it to 'LENGTH = 0x1800' and save. This needs to be done because the latest SDK expects the Nano to have 32KB of RAM instead of the 16KB it has. Now, we can compile!

```
$ make all
```

If you encounter errors, look at the logs and determine their source. For me, that was an incorrect version of GCC tools. Otherwise, we can move on to uploading to the Nano. With the new Nano plugged in using the MK20 USB Dongle:

```
$ cd _build

$ SOFTDEVICE=$NRF51_SDK/components/softdevice/s120/hex/s120_softdevice.hex
$ OUTPUT=output.hex

$ INPUT=nrf51422_xxac_s120.hex

$ srec_cat $SOFTDEVICE -intel $INPUT -intel -o $OUTPUT -intel --line-length=44

$ cp $OUTPUT /media/$USER/MBED
```

This will flash the Nano with the BLE Heart Rate Collector Example program. To test this program, we will need some additional setup. First, get the old Nano that we had flashed with the S110 softdevice and the Heart Rate Application you could read from your phone. Using a power supply or battery, hook up the Vin and Gnd pins on the Nano with 3.3-13V input. This will let it act as a Beacon separate from the computer (note that the program will enter a sleep mode after 3 minutes of operation, at which point if it's not been connected to you will need to reset the Nano). Then, make sure the new Nano we just flashed with S120 and the collector example is plugged into the computer using the MK20 USB board. To see the interaction between these two devices, we need to monitor the UART communication coming from the Collector Nano. I like to use GTKterm, which can be installed from the Ubuntu Software Center. After installation, you can run it by calling:
$ sudo gtkterm



With GTKterm running, go to Configuration->Port. The UART settings are 38400 baud, 8 data bits, 1 stop bit, no parity, and HW flow control with RTS/CTS. Setting these will allow you to see the UART info coming from the Nano plugged into the computer (you will need to find out what port the nano is connected to on the computer).

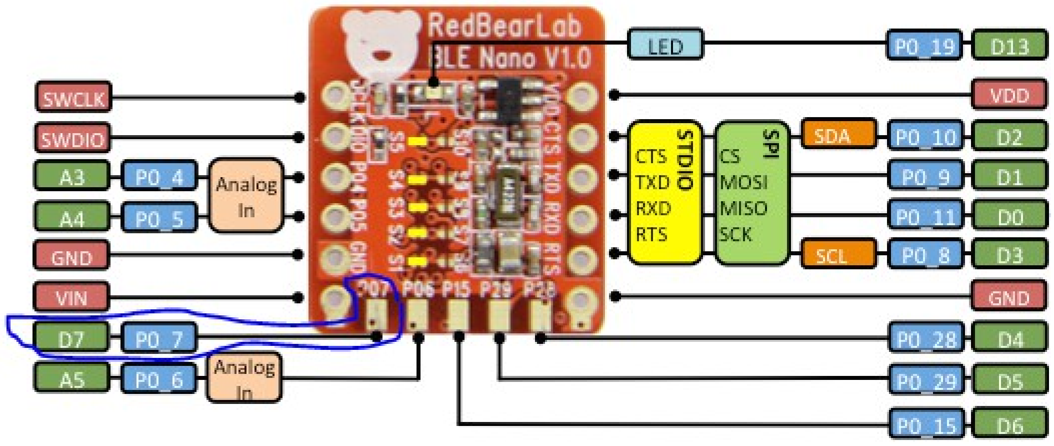
If you power on the Sensor Nano, it will begin Broadcasting. The Collector Nano in the computer will connect to the Sensor Nano, and begin sending information to GTKterm showing the Heart Rate count and the Battery count as well. Congratulations! You have now set up a Master/Slave connection with the RBL BLE Nanos!

Adding user input to connected BLE Nanos

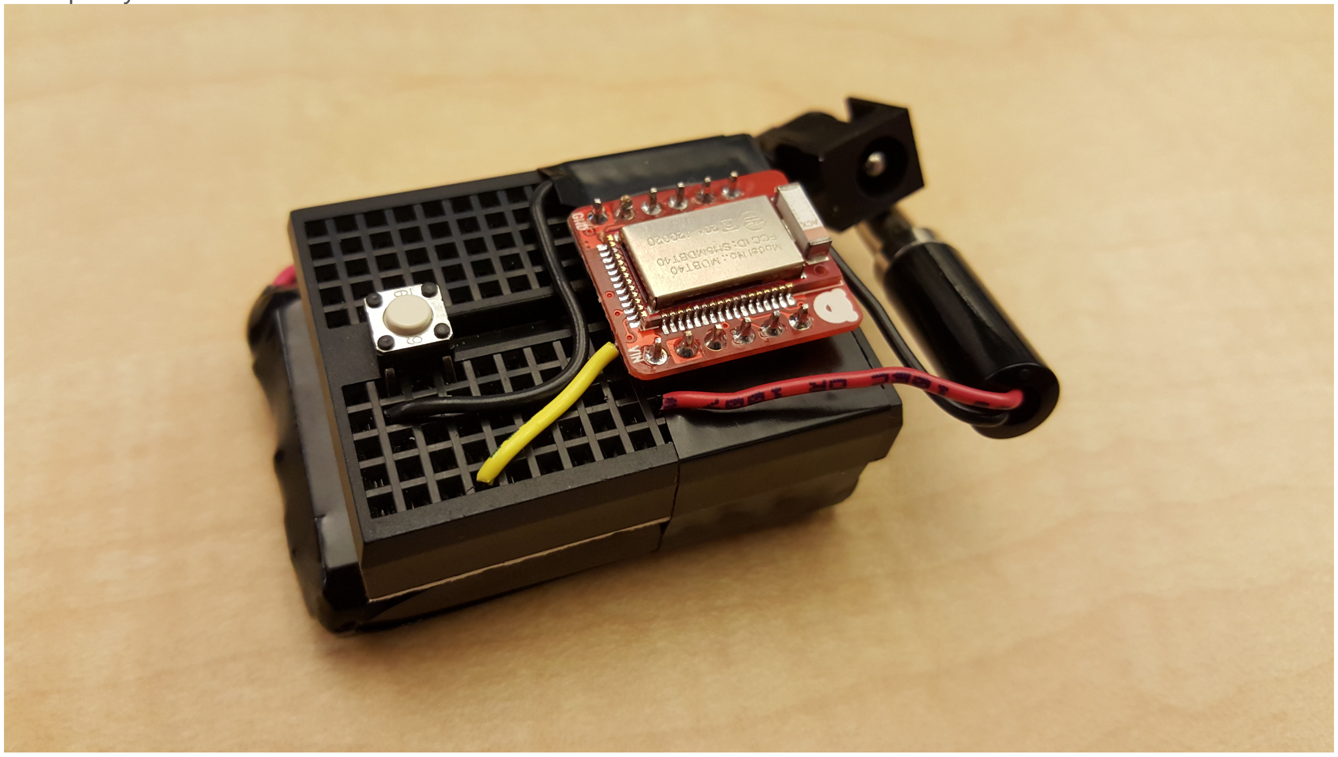
For an example of user input on the peripheral side being reflected on a connected central device, we will utilize the Multilink Example form the Nordic SDK.

SLAVE:

The slave\peripheral\server role in this example will be played by a BLE Nano wired up with an additional button as an input. Looking at the pinout diagram for the BLE Nano, we see that there are a number of I/O lines on the bottom of the chip ready to be soldered to:



You will need to solder a wire to P07 (circled in the photo) so we can use this input. Using a breadboard or other means of connecting wires together, you will need to run a wire from P07, through a momentary push-button switch and then to ground. For additional freedom, wire up a mobile power source like a battery to the system by running the leads to VIN and GND on the chip. For me, that was a 3.7V LiPo battery using barrel jack connectors. An example system is shown below:



In order to access the new button input we've added to the board, we need to modify the custom_board.h file used by the Board Support Package (BSP) to map board-specific events like button presses. We will need new h-files for our slave configuration. Using the $NRF51_SDK variable from the first example, go to:

```
$ cd $NRF51_SDK/examples/bsp
```

```
$ gedit boards.h
```

And find the series of #elif statements. Add the following before the #else:

```
#elif defined(BOARD_NANO_P)

    #include "nano_periph.h"
```

These lines will allow us to treat the Nanos like they are different boards, one equipped with a button and one equipped with an led. Now to make copies of the custom_board.h file:

```
$ cp custom_board.h nano_periph.h

$ gedit nano_periph.h
```

I highly recommend changing the include guard at the top and bottom of the file to be more specific to the new file we've created. Change this from "REDBEAR_NANO_H__" to "REDBEAR_NANO_P_H__" in all three locations (even though one is a comment, better to be thorough). We can leave the LED sections the same because we are not adding an LED. We are adding a button however, so you will need to replace the section of the file that begins with the comment "// there are no buttons on this board" and has two #define statements under it with the following:

```
#define BUTTON_0    7 // Button on I/O p0_7

#define BUTTON_PULL NRF_GPIO_PIN_PULLUP

#define BSP_BUTTON_0 BUTTON_0

#define BUTTON_START BUTTON_0

#define BUTTON_STOP  BUTTON_0

#define BUTTONS_LIST { BUTTON_0 }

#define BUTTONS_NUMBER 1

#define BSP_BUTTON_0_MASK (1<<BUTTON_0)

#define BUTTONS_MASK (BSP_BUTTON_0_MASK)
```

This changes the board-specific mapping to include a button, as if it came with one on the PCB. Finally, we can follow the same process from the other examples to download our example code to the Nano.

```
$ cd
$NRF51_SDK/examples/ble_central/ble_app_multilink_peripheral/pca10028/s110/armg
cc

$ gedit Makefile
```

Find 'CFlags += -DBOARD_PCA10028' and change it to 'CFlags += -DBOARD_NANO_P' and save.

```
$ gedit ble_app_multilink_peripheral_gcc_nrf51.ld
```

Find 'RAM (rwx) : ORIGIN = 0x20002000, LENGTH = 0x6000'. Subtract 0x4000 from the LENGTH variable, changing it to 'LENGTH = 0x2000' and save. This needs to be done because the latest SDK expects the Nano to have 32KB of RAM instead of the 16KB it has. Now, we can compile!

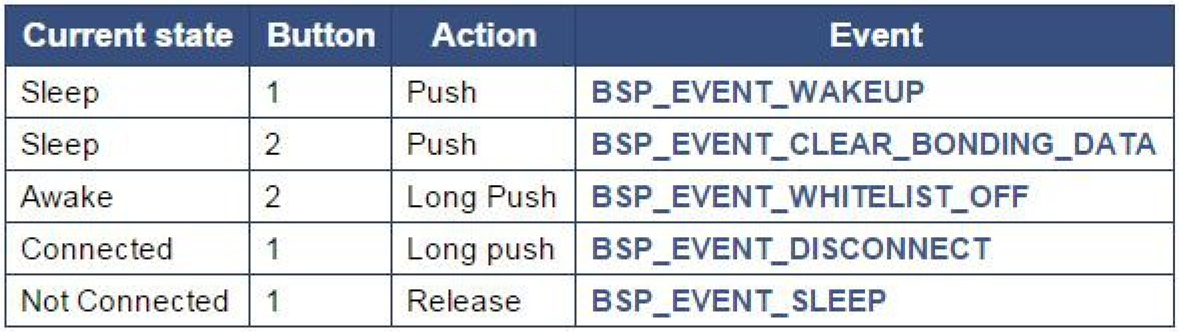
```
$ make all
```

If you encounter errors, look at the logs and determine their source. For me, that was an incorrect version of GCC tools. Otherwise, we can move on to uploading to the Nano. With the new Nano plugged in using the MK20 USB

Dongle:

```
$ cd _build

$ SOFTDEVICE=$NRF51_SDK/components/softdevice/s110/hex/s110_softdevice.hex

$ OUTPUT=output.hex

$ INPUT=nrf51422_xxac_s110.hex

$ srec_cat $SOFTDEVICE -intel $INPUT -intel -o $OUTPUT -intel --line-length=44

$ cp $OUTPUT /media/$USER/MBED
```

This will flash the Nano with the Multilink Peripheral example. When you plug the Nano back into the breadboard and power it, the button will change the state of the Nano and it will broadcast this change to the central device connected to it. Fair warning, the BSP BLE Button Module has a few special button press events mapped according to the following table:

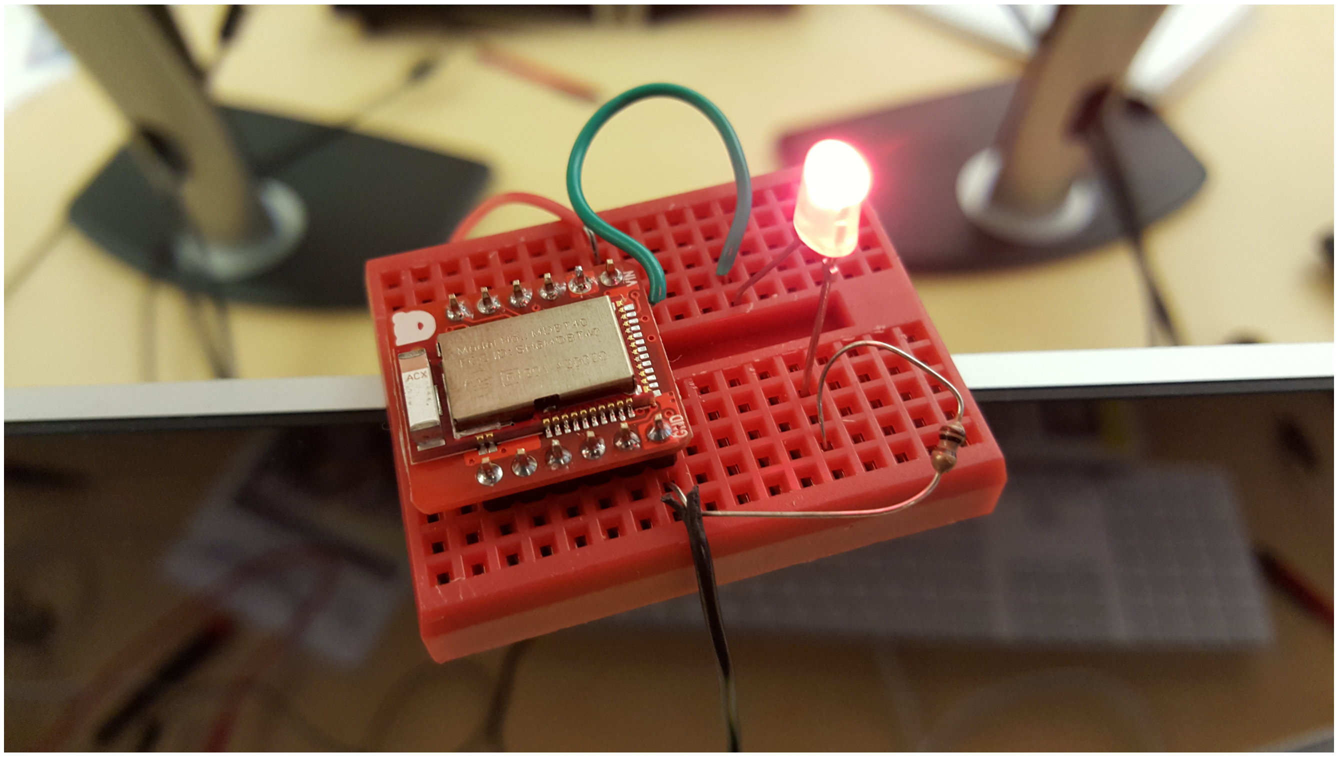


Current state	Button	Action	Event
Sleep	1	Push	BSP_EVENT_WAKEUP
Sleep	2	Push	BSP_EVENT_CLEAR_BONDING_DATA
Awake	2	Long Push	BSP_EVENT_WHITELIST_OFF
Connected	1	Long push	BSP_EVENT_DISCONNECT
Not Connected	1	Release	BSP_EVENT_SLEEP

The important one to notice on here is the "Connected" state followed by a "Long push" on button 1. This means that if you hold down the button we connected to the Nano for a full second, it will cause the board to disconnect from the master/central device. Next, we get the central device running.

MASTER:

The master/central/client role in this example will be played by a BLE Nano wired up with an additional LED as an output. Looking at the pinout diagram for the BLE Nano you will need to solder a wire to P07 (circled in the photo) so we can use this output. Using a breadboard or other means of connecting wires together, you will need to run a wire from P07, through an LED (flat side is ground!) and resistor pair, then to ground. For additional freedom, wire up a mobile power source like a battery to the system by running the leads to VIN and GND on the chip.



In order to access the new LED output we've added to the board, we need to modify the custom_board.h file used by the Board Support Package (BSP) to map board-specific events like button presses. We will need new h-files for our master configuration. Using the $NRF51_SDK variable from the first example, go to:

```
$ cd $NRF51_SDK/examples/bsp
```

```
$ gedit boards.h
```

And find the series of #elif statements. Add the following before the #else:

```
#elif defined(BOARD_NANO_C)

    #include "nano_central.h"
```

These lines will allow us to treat the Nanos like they are different boards, one equipped with a button and one equipped with an led. Now to make copies of the custom_board.h file:

```
$ cp custom_board.h nano_central.h
```

```
$ gedit nano_central.h
```

I highly recommend changing the include guard at the top and bottom of the file to be more specific to the new file we've created. Change this from "REDBEAR_NANO_H__" to "REDBEAR_NANO_C_H__" in all three locations (even though one is a comment, better to be thorough). We can leave the button sections the same because we are not adding a button. We are adding an LED however, so you will need to replace the section of the file between the include guard and the comment "// there are no buttons on this board" with the following:

```
#define LEDS_NUMBER 2

#define LED_OB       19 // Onboard LED

#define LED_ADD      7  // LED soldered to I/O P0_7
```

```
#define LED_START   LED_ADD

#define BSP_LED_0   LED_ADD

#define BSP_LED_1   LED_OB

#define LED_STOP    LED_OB

#define BUTTONS_LIST {}

#define LEDS_LIST { BSP_LED_0, BSP_LED_1 }

#define BSP_LED_0_MASK (1<<BSP_LED_0)

#define BSP_LED_1_MASK (1<<BSP_LED_1)

#define LEDS_MASK    (BSP_LED_0_MASK | BSP_LED_1_MASK)

#define LEDS_INV_MASK LEDS_MASK
```

This changes the board-specific mapping to include another LED, as if it came with one on the PCB. Finally, we can follow the same process from the other examples to download our example code to the Nano.

```
$ cd
$NRF51_SDK/examples/ble_central/ble_app_multilink_central/pca10028/s120/armgcc

$ gedit Makefile
```

Find 'CFlags += -DBOARD_PCA10028' and change it to 'CFlags += -DBOARD_NANO_C' and save.

```
$ gedit ble_app_multilink_central_gcc_nrf51.ld
```

Find 'RAM (rwx) : ORIGIN = 0x20002800, LENGTH = 0x5800'. Subtract 0x4000 from the LENGTH variable, changing it to 'LENGTH = 0x1800' and save. This needs to be done because the latest SDK expects the Nano to have 32KB of RAM instead of the 16KB it has. Now, we can compile!

```
$ make all
```

If you encounter errors, look at the logs and determine their source. For me, that was an incorrect version of GCC tools. Otherwise, we can move on to uploading to the Nano. With the new Nano plugged in using the MK20 USB Dongle:

```
$ cd _build

$ SOFTDEVICE=$NRF51_SDK/components/softdevice/s120/hex/s120_softdevice.hex

$ OUTPUT=output.hex

$ INPUT=nrf51422_xxac_s120.hex

$ srec_cat $SOFTDEVICE -intel $INPUT -intel -o $OUTPUT -intel --line-length=44

$ cp $OUTPUT /media/$USER/MBED
```

This will flash the Nano with the Multilink Central example. When you plug the Nano back into the breadboard and power it, the LED should power on. With the Slave Nano configured properly and powered, pressing the button should toggle the LED on the Master Nano. This wireless user input should work up to 100ft line-of-sight.