

Getting started with Ultra-Wide-Band 3D Positioning DWM1000 and DWM1001 Modules

Posted on [2018-09-192018-09-19](#) by [admin](#)



The **possibilities of accurate 3D tracking are endless**: Drone navigation, logistics, human-machine-interfaces... but unfortunately developing Real Time Location Systems is expensive and takes a lot of time, **or does it?**

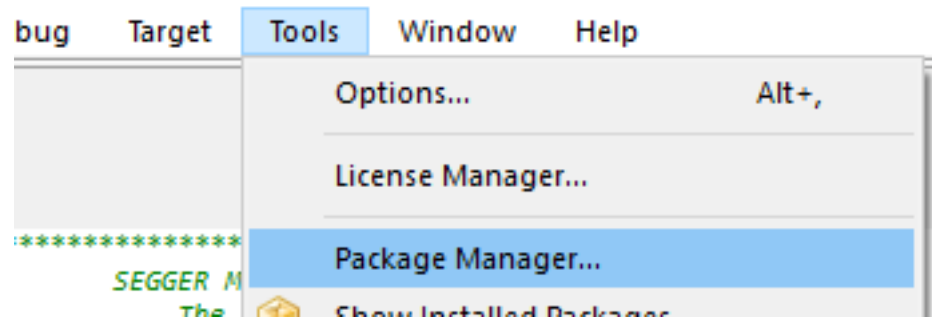
This is how you can start developing your custom applications with *DWM1001* modules for under \$20 in under an hour.

0. Preparing the environment

A screenshot of the Embedded Studio software interface. On the left, there is a logo for 'Embedded Studio' with a large 'C' and '++' symbols. On the right, the text 'Embedded Studio — A Complete All-In-One Solution' is displayed above a list of features.

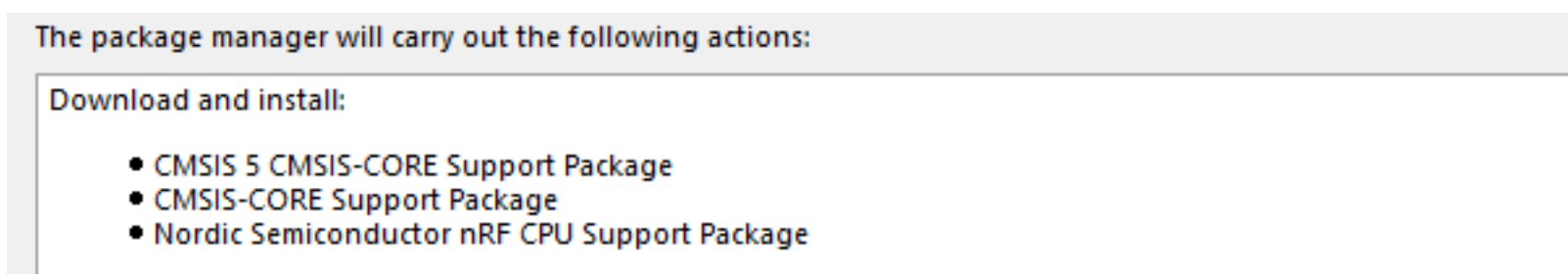
- Professional IDE solution for embedded C/C++ programming
- Cross-Platform: Runs on Windows, macOS, and Linux
- Clang/LLVM, and GCC C/C++ Compilers included
- Highly optimized run-time library for best performance and smallest code size
- Feature-packed debugger with seamless I-Link integration

We will be using [Segger Embedded Studio](#) because it provides a **free license** for Nordic **nRF** family development.



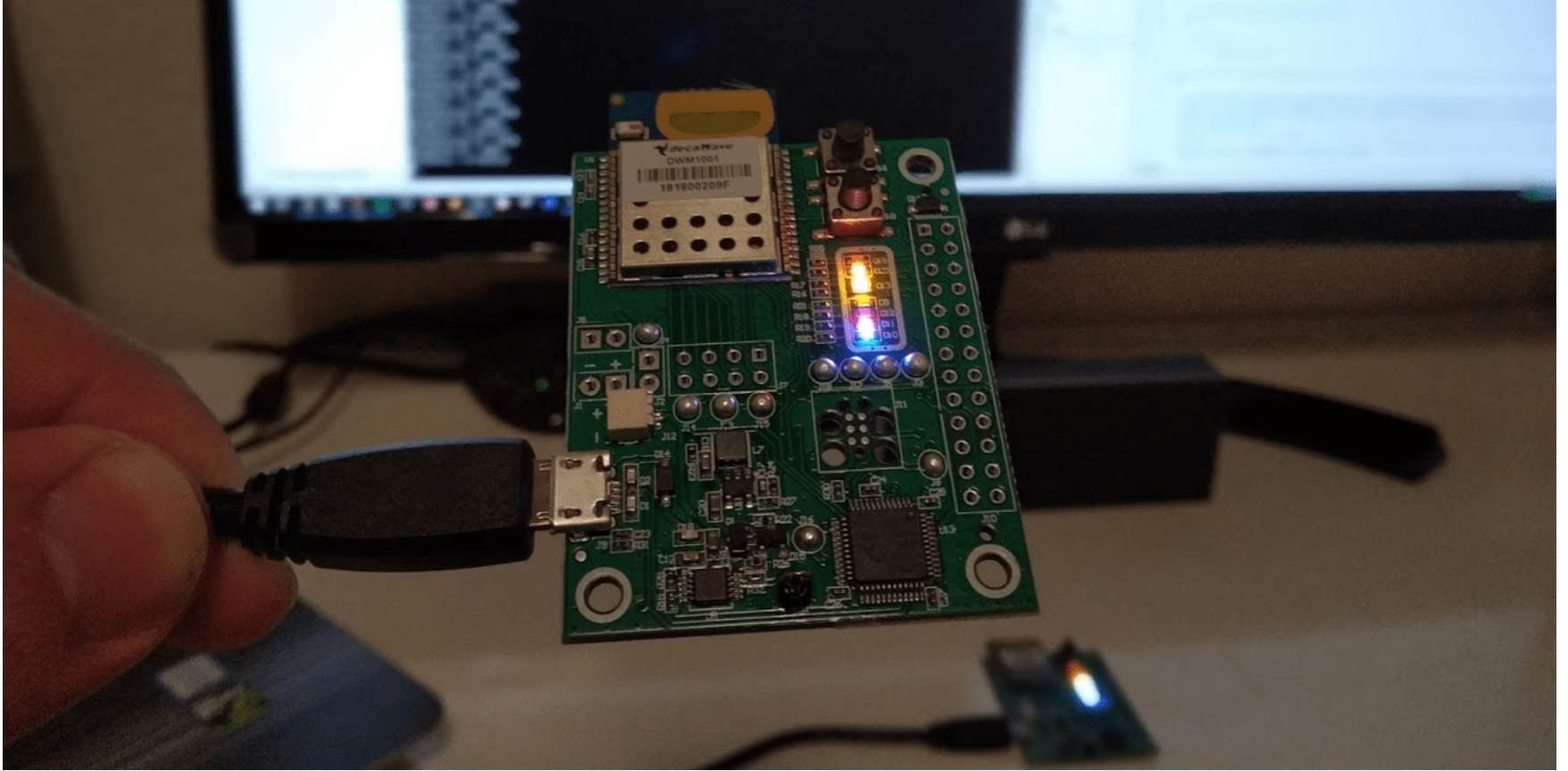
Once installed, we will need the following packages:

- CMSIS 5 CMSIS-CORE Support Package (version 5.02)
- CMSIS-CORE Support Package (version 4.05)
- Nordic Semiconductor nRF CPU Support Package (version 1.06)

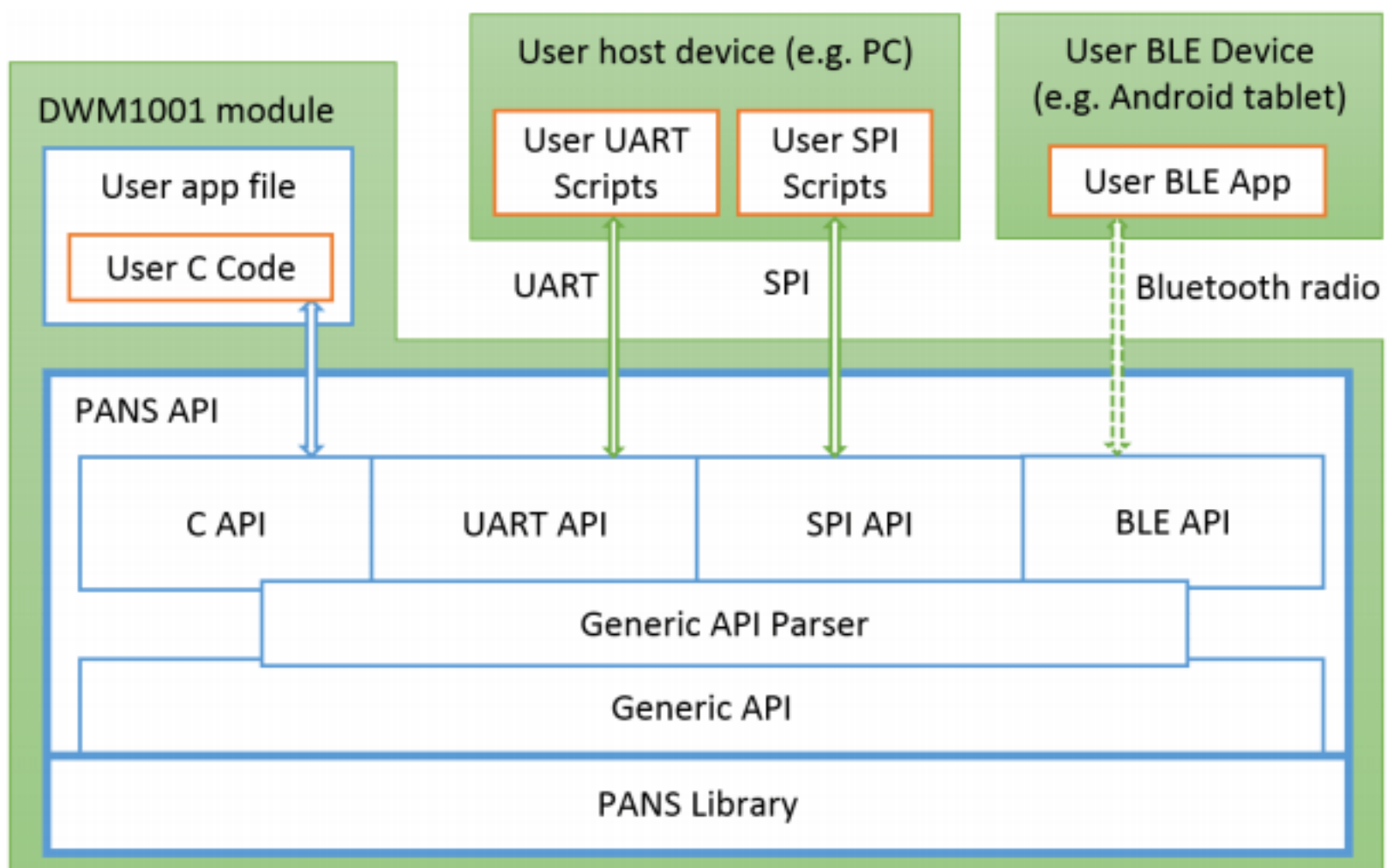


You may encounter **build errors** using the latest SES version, it can be better to install an older one. **Version 3.34a** has been verified to work.

You can build the examples and **start getting measurements** in minutes!



1. Getting to know the API

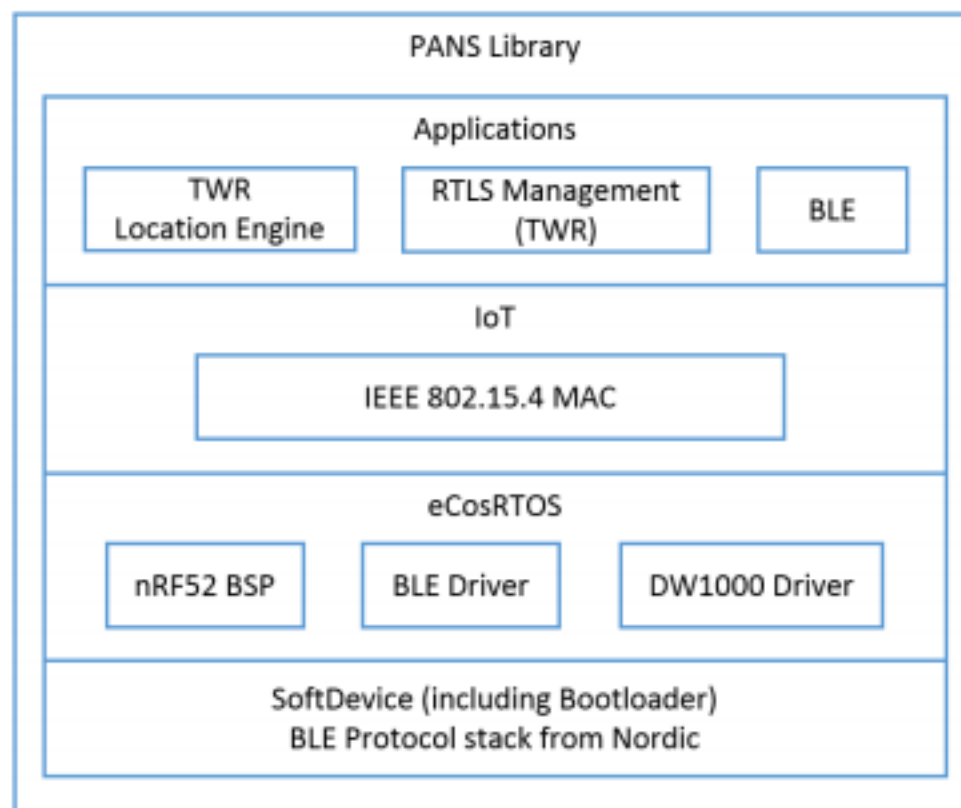


DecaWave provides a very comprehensive API that allows you to integrate your design ideas in a minimal amount of time.

The ways you can access the API are:

- **User C code:** Using the provided toolchain.
- **SPI or UART:** Using Type-Length-Value format.
- **UART:** Shell mode.
- **BLE:** Via services and characteristics. (The documentation on the BLE API is incomplete, but you can check the Android app code for reference).

Even though lots of examples and a very decent amount of documentation is provided; unfortunately, **the PANS API source is not provided**, we would like to see it published to be able to customize or extend it.

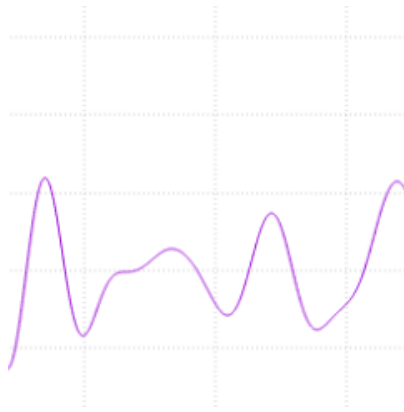


The API structure builds up from the NRF **SoftDevice**, which should be familiar for those of us who have worked with Nordic devices in the past. Just mention that **this is closed-source too**. It includes BLE Central, Peripheral, Broadcaster and Observer roles, and **supports OTA firmware upgrades**. The included version is S132.



The [eCos RTOS](#) includes the **Board Support Package (BSP)**, and all the

drivers for BLE, Accelerometer, and most importantly, the DW1000 driver.



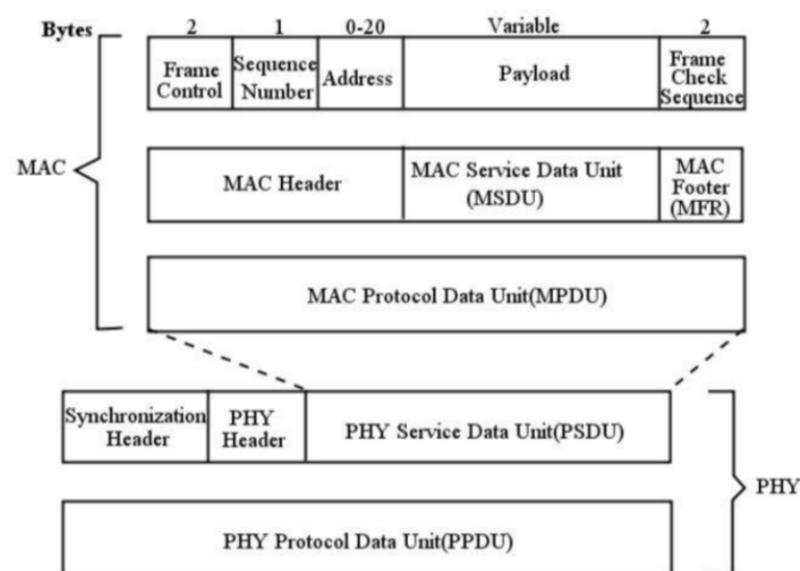
A quick way of knowing if the module is being moved physically is implemented through a component of I2C **Accelerometer** LIS2DH12TR **slave at address 0x19**. You can read it at 0x33 and write it at 0x32.



The **Android App** is really useful for a quick first-time configuration **and for programming the modules over-the-air**.

DRTLS Network

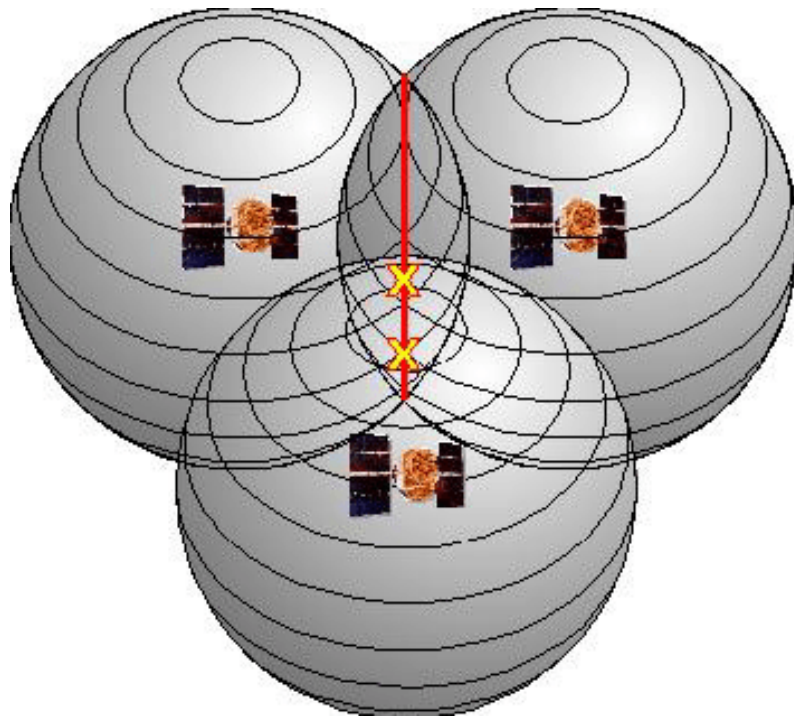
The stack allows discovery, joining and leaving the network. The UWB packets adhere to the standard **802.15.4** format:



Two-Way-Ranging Solver (Location Engine)

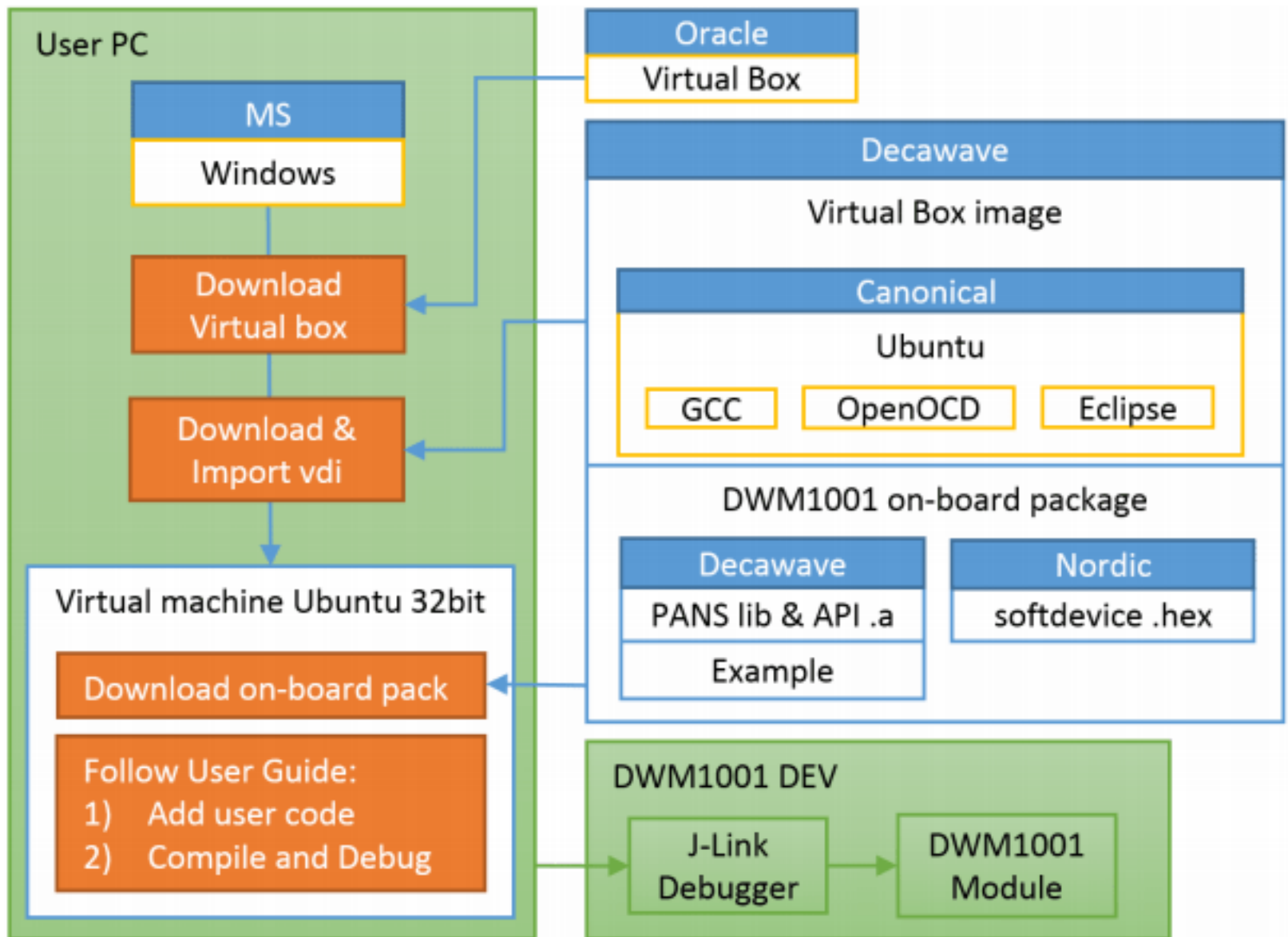
Likely the most important part for us. This part of the API calculates the X,

Y, Z coordinates from the results of TWR. Although it's very useful to get up to speed quickly, **this engine can be disabled to use custom filtering and solving algorithms.**



2. The development toolchain

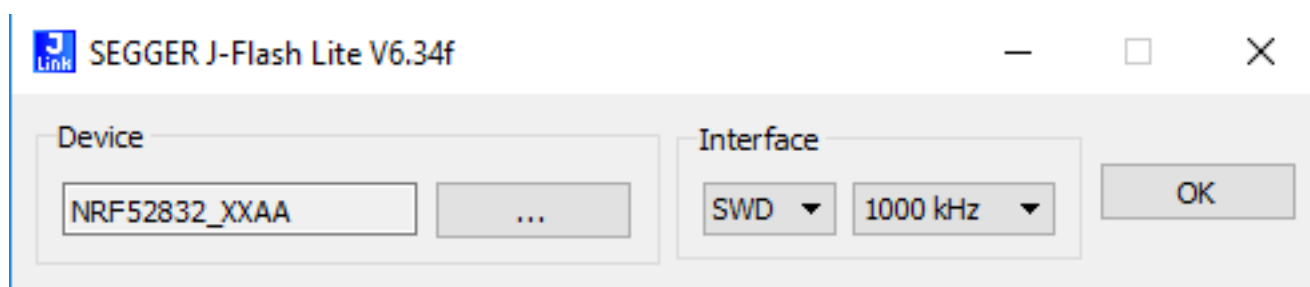
It is really great that DecaWave provides such a comprehensive set of resources **already configured**, [download the toolchain](#) and with this guide, you will be developing in minutes. Kudos to DecaWave for this.

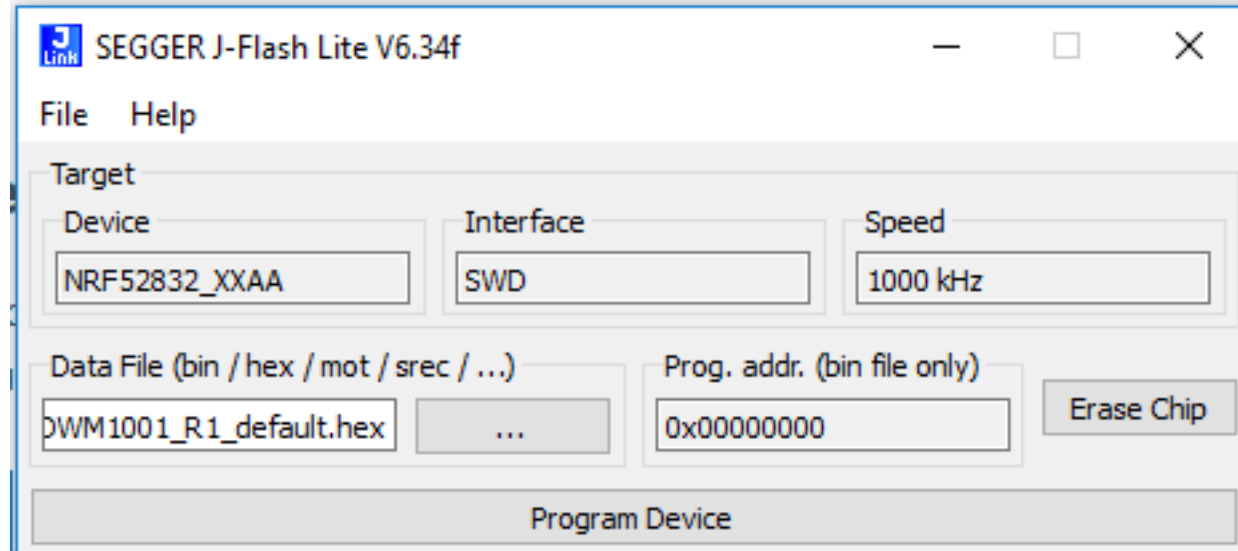


You can see that the PANS API, being closed-source, is provided as a compiled static library *.a*, and the nRF SoftDevice as *.hex*.

Restoring the factory image

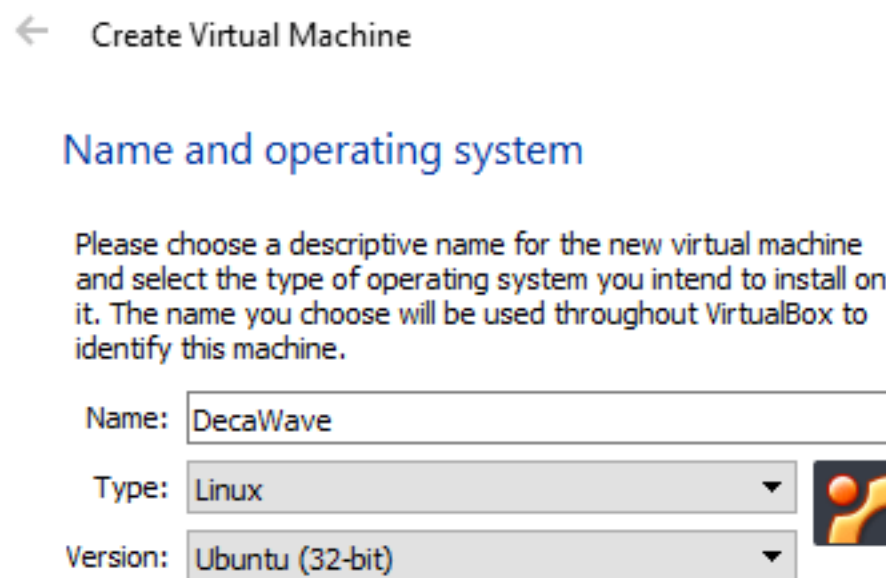
If you ever need to go back to factory settings, download any binary or erase the chip, just use **J-Flash Lite**, which you can download from [J-Link Software and Documentation Pack](#).



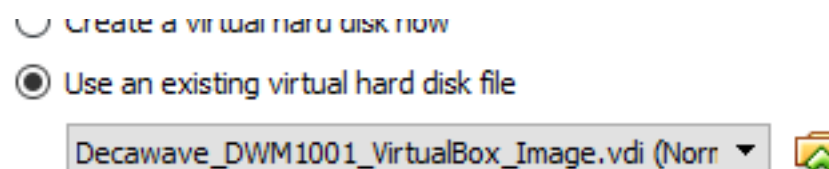


Setting up the development Virtual Machine

You can create a New virtual machine with these settings:



Assign a generous amount of RAM, and **select the provided .vdi to be used as Hard disk.**



Or just Add a new machine and use the default settings by selecting the **.vbox** file.



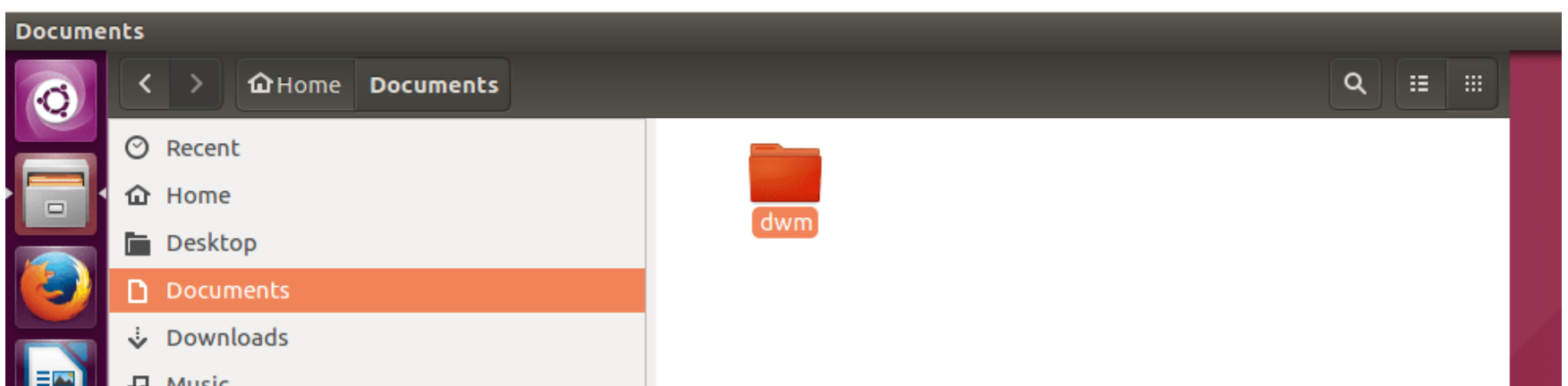
Once ready, you can run it and login: user **dw** password **dw**



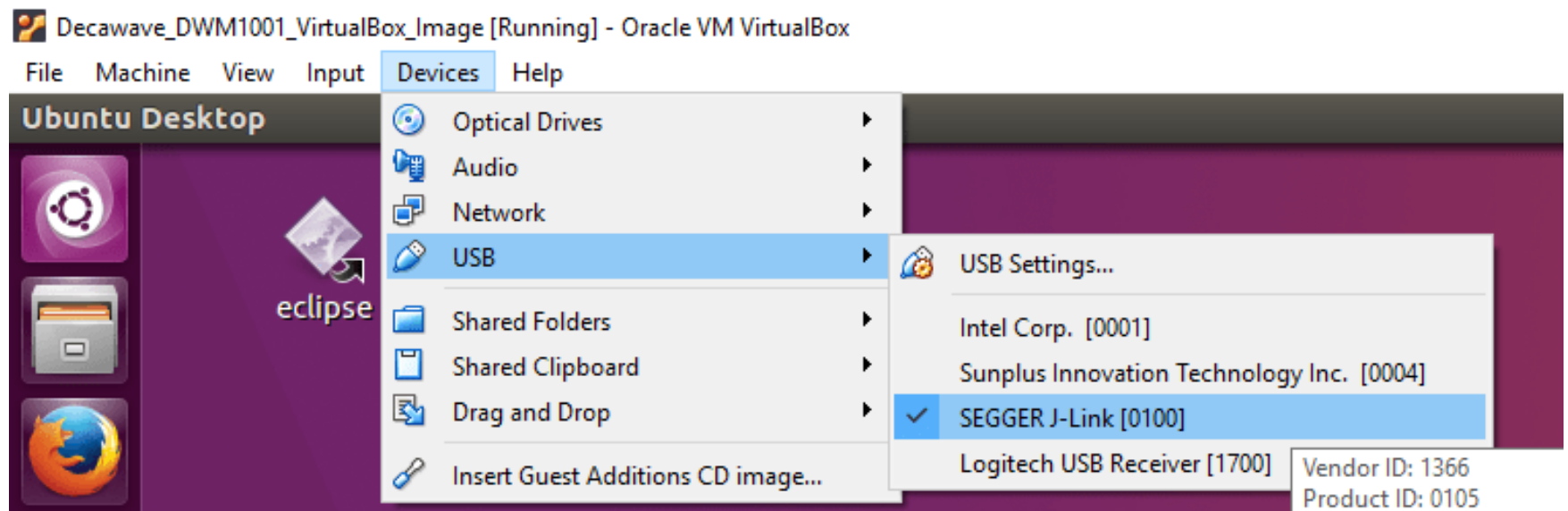
[Download the sources from DecaWave](#). The link may go obsolete, look for a package with a name similar to "DWM1001, DW10001-DEV and MDEK1001 Documents, Source Code, Android Application & Firmware Image".

At this point, the documentation is a bit outdated. You need to find the **dwm** folder which is in *DWM1001_DWM1001-DEV_MDEK1001_Sources_and_Docs_vX/DWM1001/Source_Code/dwm1001_on-board_package_v1p0* and place it **inside /Documents** in the Virtual Machine.

The easiest way to copy the folder into the Virtual Machine is creating a **Shared Folder** (Devices > Shared Folders). Create a new one, point it to a folder in your computer and select **Auto-mount and Permanent**. Reboot the VM and you should see the shared folder as a removable device or inside */media/*



Now, you can **connect the DWM1001-DEV** via micro-USB. For the same purpose, you can connect any J-Link that is connected to a nRF52832.



Make sure the J-Link device is ticked inside the Virtual Machine. That will make the device available inside the VM.

Factory reset

To verify that the setup is working, you can **reset the device to factory settings** with:

```
cd ~/Documents/dwm/examples/dwm-simple
make recover
```

```
dw@dw:~$ cd ~/Documents/dwm/examples/dwm-simple/
dw@dw:~/Documents/dwm/examples/dwm-simple$ make recover
openocd -s /usr/lib/ -f nrf52_swd.cfg -c "init;halt;eraseall;shutdown"
```

It should end after a few seconds and the onboard **LEDs with be lit**.

```
xPSR: 0x01000000 pc: 0x000008e4 msp: 0x20000400
erased address 0x0001f000 (length 4096) in 0.104815s (38.162 KiB/s)
target halted due to debug-request, current mode: Thread
xPSR: 0x01000000 pc: 0x000008e4 msp: 0x20000400
wrote 3876 bytes from file ../../recovery/bootloader_s132.bin in 2.228397s (1.6
99 KiB/s)
shutdown command invoked
dw@dw:~/Documents/dwm/examples/dwm-simple$
```

3. Using your own code

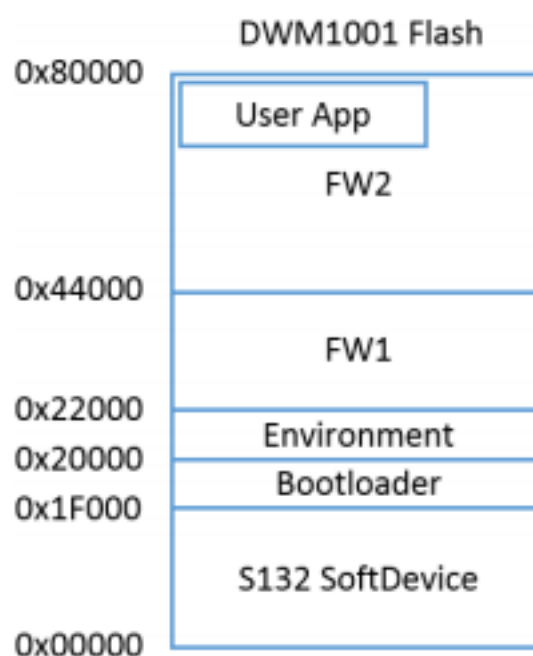


This is no doubt the main purpose of this guide. **As introduced before, there are several ways** to use your own code to control DWM1001 and DWM1001-DEV modules. Let's try an example of each one of them.

3.1 User C code within the DWM1001 firmware

This is the way to go to integrate custom C code that runs on-board while using the high-level PANS API.

The **512KB flash** (0 to 0x80000) is partitioned like this:

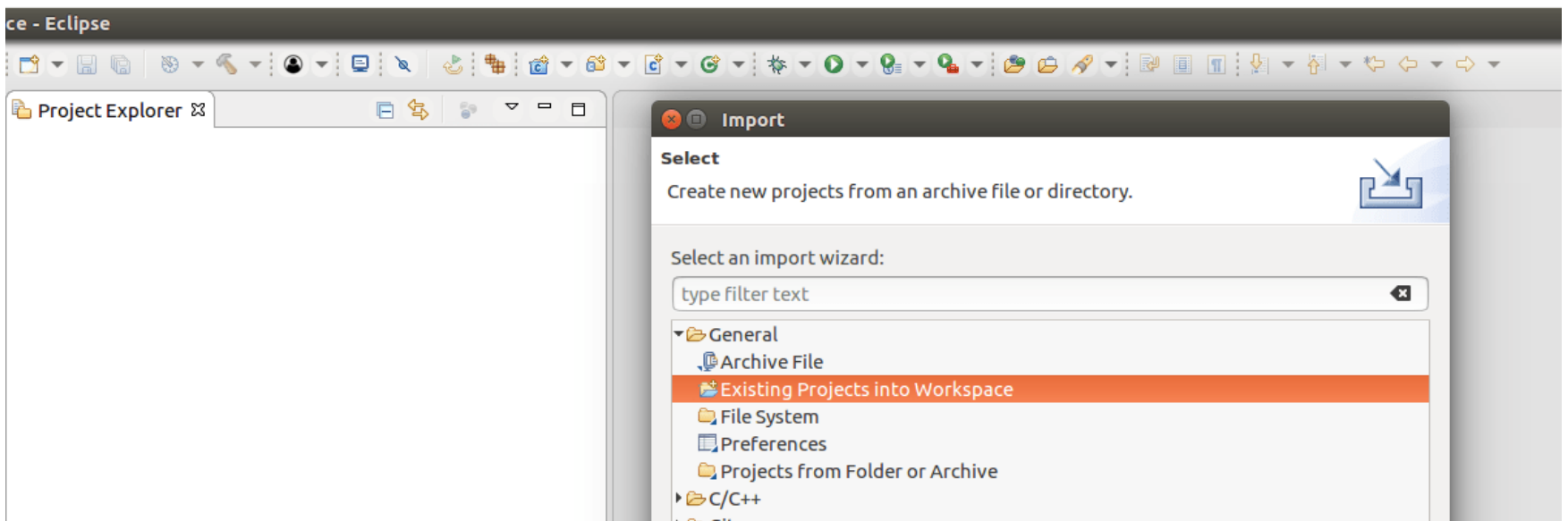


- **SoftDevice:** provides BLE capabilities, you can find more information

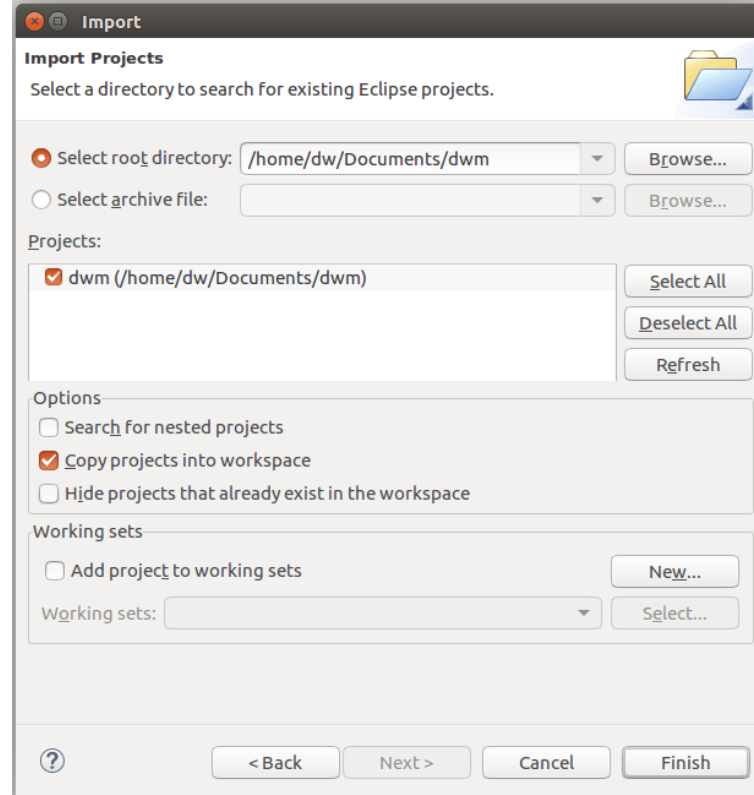
and examples in the [Nordic Semiconductor](#) website.

- **Bootloader**: Allows choosing between FW1 and FW2 during boot.
- The **Environment** area is used for configuration and is therefore **preserved across power cycles**, and even firmware re-flash. To clear it you must issue a full erase.
- **FW1** is used for the **OverTheAir** firmware **update**.
- **FW2** can be up to 240KB in size. This image includes the PANS lib and the **user application**. Here comes one limitation for the user code:

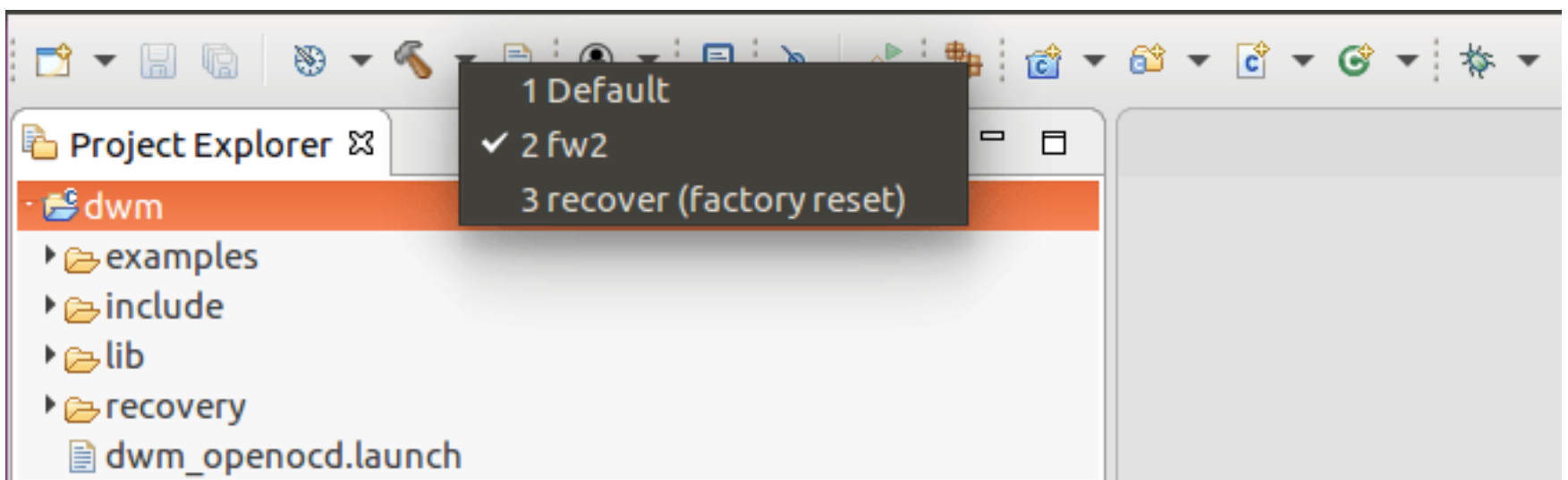
When bundled inside FW2, the user code runs as a thread and can only take up to 60KB in flash, and use 3KB of RAM.



To get started, right-click inside the Project Explorer and import the projects in `/home/dw/Documents/dwm`



Since we want to develop **user code**, and it has to be inside **FW2**, we have to select the second option in the build drop-down menu.



To create user code:

- Add functionalities inside *dwm/examples/dwm-simple/dwm-simple.c*
- All the C API functions that you can use are listed in *dwm/include/dwm.h*

To quickly see how user code is executed, in **dwm-simple.c** add:

...

```
dwm_pos_t pos;
```

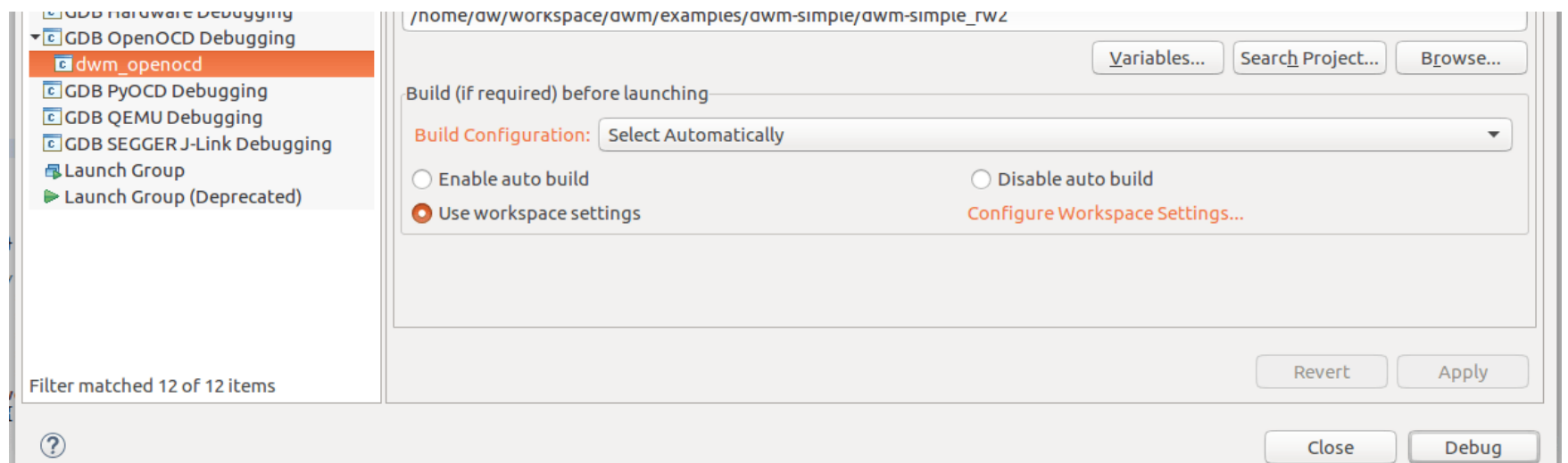
...

```
while (1) {
    /* Thread loop */
    dwm_pos_get(&pos);
    printf("x=%ld, y=%ld, z=%ld, qf=%u \n", pos.x, pos.y, pos.z, pos.qf);
    printf("\t\t Time=%lu \n", dwm_systime_us_get());
}
```

To **download and debug** the firmware right-click the project and select Debug Configurations.



You will find **dwm_openocd**, which by the way is a great on-chip debugger that works with GDB, and even has some interesting Boundary Scan capabilities that can be useful in FPGA design process.



Once you click **Debug**, the firmware will be downloaded and the Debug perspective will be opened. Here you can use all the Eclipse tools to debug your code.

To see the modifications running, disconnect the device from the Virtual Machine, and open a **serial** terminal to the **J-Link COM Port**. **The default baud-rate is 115200 bps**. The added console output should appear. If nothing shows up, start the *Shell Mode* (more on this in a minute) by pressing Enter two times.

```
x=0, y=0, z=0, qf=0
Time=276174342
x=0, y=0, z=0, qf=0
Time=276178122
```

Now you can start tweaking the code and adding more complex functionalities!



One important thing to mention is that this code is executed along with all the functionalities already available in the factory firmware, so you will still be able to use those components too.

It's precisely one of this components, the BLE included in the **SoftDevice**, that **can generate debugging problems**. Since its interruptions have the highest priority, they will conflict with the user interrupt. Disable *dwm_ble_compile()* while debugging or *mask the BLE interrupts* to avoid this.

3.2 Interface the API from an external processor via UART

You can connect to the UART either through the USB-Serial converter, which shows up **as a COM Port** under the J-Link hardware, **or through the pins**:

Pin to use in DWM1001 DEV		Pin to be connected to
Connector J10	Function	
Pin 6	GND	GND
Pin 8	RXD	TXD
Pin 10	TXD	RXD

If you are using a **Raspberry Pi**, it will show up as `/dev/serial0`

The UART works in two modes that can be switched:

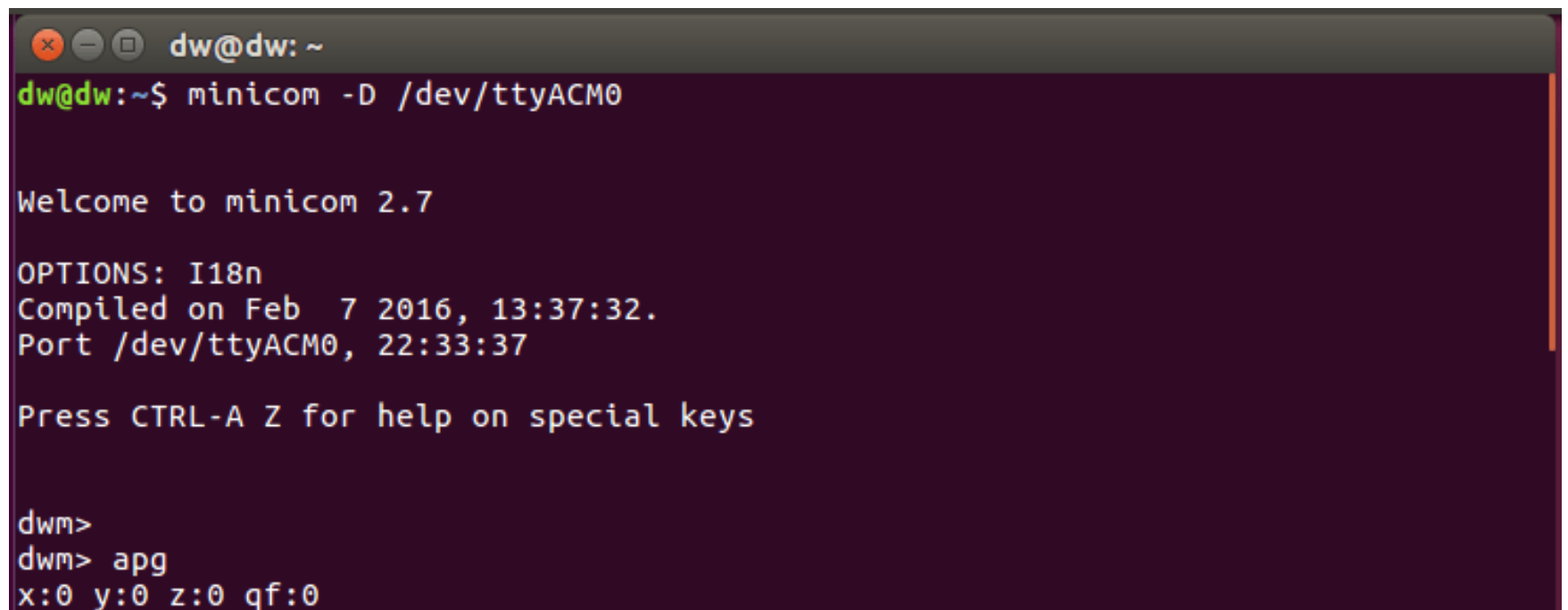
- **GENERIC (default):** Use **T**ype **L**ength **V**alue encoded commands. To switch to Shell Mode, send *Enter* twice [0x0D, 0x0D].
- **Shell Mode:** In this mode, you can interact with the module via cli. To go back to Generic Mode, send *quit*.

Shell mode example

With the J-Link device under the Linux VM, open a serial port with:

```
minicom -D /dev/ttyACM0
```

Enter *Shell Mode* pressing Enter twice. Now, **the same function that we implemented before using User C code**, can be run with the **apg** command:

A screenshot of a terminal window with a dark purple background. The window title bar shows 'dw@dw: ~'. The prompt is 'dw@dw:~\$' and the command 'minicom -D /dev/ttyACM0' has been entered. The output shows 'Welcome to minicom 2.7', 'OPTIONS: I18n', 'Compiled on Feb 7 2016, 13:37:32.', 'Port /dev/ttyACM0, 22:33:37', and 'Press CTRL-A Z for help on special keys'. The prompt has changed to 'dwm>'. The user has entered 'apg' and the output shows 'x:0 y:0 z:0 qf:0'.

```
dw@dw:~$ minicom -D /dev/ttyACM0

Welcome to minicom 2.7

OPTIONS: I18n
Compiled on Feb 7 2016, 13:37:32.
Port /dev/ttyACM0, 22:33:37

Press CTRL-A Z for help on special keys

dwm>
dwm> apg
x:0 y:0 z:0 qf:0
```

Generic mode example

If you have a **Raspberry Pi** you can quickly test the Generic Mode with the host API, as **the HAL is already provided**.

[Download the dwm1001_host_api package](#). (It is included in the complete docs&sources package), and browse the **examples\ex1_TWR_2Hosts\tag** folder from inside the VM. You can copy it to the shared folder we created previously. (Inside the downloaded package, the path will be similar to *DWM1001_DWM1001-DEV_MDEK1001_Sources_and_Docs_v8**DWM1001\Source_Code\DWM1001_host_api\dwm1001_host_api\examples\ex1_TWR_2Hosts\tag*).

Edit the Makefile, for example with

```
nano Makefile
```

Change

```
TARGET = 0  
...  
INTERFACE_NUMBER = 0
```

Make and run, and you should see the output as follows:

```
make  
./tag_cfg
```

```
dwm_init(): dev0  
Opening log file log.txt  
    LMH_UARTRX_Init()...  
    UART: Init start.  
    UART: Init done.  
Setting to tag: dev0.  
dwm_cfg_tag_set(&cfg_tag): dev0.  
Wait 2s for node to reset.  
Comparing set vs. get: dev0.  
  
Configuration succeeded.  
  
Wait 1000 ms...  
dwm_loc_get(&loc):  
    [121,50,251,100]
```

To use a custom platform you need to:

- Provide your own platform **HAL**.
- Assign it to **HAL_DIR** in *dwm1001.mak* (inside DWM1001_host_api\dwm1001_host_api\include\)

3.3 Interface via SPI

The encoding used in **SPI is the same as in UART Generic Mode** so you just need to connect the appropriate pins

Pin to use		Pin to be connected to
Pin number	Function	
Pin 19	MOSI	MOSI
Pin 21	MISO	MISO
Pin 23	SCLK	SCLK
Pin 25	GND	GND
Pin 24	CSN	CSN

And change the Makefile to

```
INTERFACE_NUMBER = 1
```

Conclusion

We’ve seen that the amount of **available resources** is really amazing, you can **get started in minutes**. The amount of ways in which you can **interface** the device allows for this modules to be integrated in almost any kind of system.

Once you have tested the different ways to control them, and you have decided which interface you will use, it’s time to **dive into the API** and the libraries. That’s the best way to create a reasonably highly **custom-tailored solution in the shortest amount of time**.

References

This guide is based on the official documentation from DecaWave, which is

subject to changes. Not many links are provided since they break with each update. It's better to browse <https://www.decawave.com/> and download the latest versions directly. Always refer to the up-to-date official documentation when working on designs.