USB **Host** Project Tutorial

# Introduction

This document is meant to be a quick start guide on working with the USB Device interface and Ganymede libraries. For in-depth coverage of the inner workings and settings for the USB Device interface and API, please refer to the appropriate Renesas documentation, which can be found by right-clicking on the module in the configuration view, and selecting Module Resources.

This document assumes the reader has already cloned the Tutorials repository and ExampleProject project into their e2 Studio environment, and is familiar with the process of creating their own unique project based on this. For more information on that, please refer to the Project Creation Tutorial located in the Documents folder of the Tutorials Repo. To reach the Tutorials repo, please go to “<https://github.com/NYUAD-LabOps/Tutorials>”.

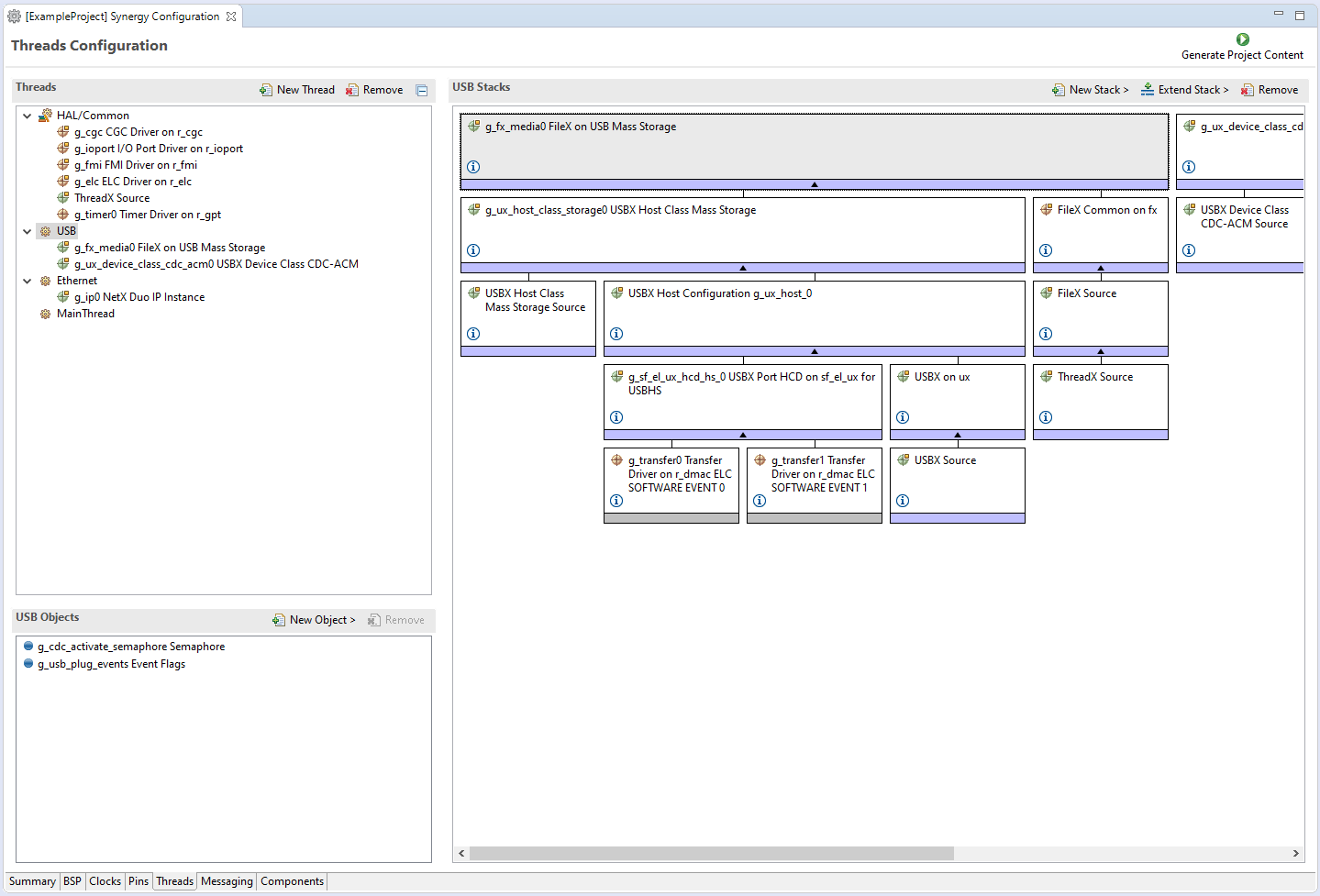
# Preliminary

This section covers knowledge of the Host interface which should be understood by the developer prior to any attempt at working with it.

The Host interface is used with USB plug or stick devices. These are generally some kind of storage device. It may be a mass storage device, or it may be a security dongle. There are wide range of applications, but generally some kind of file storage is involved. This brings us to the Renesas USBX and FileX APIs.

The USBX API is responsible for hardware-level setup, settings, and interaction. The device settings can be modified by opening the configuration.xml file and going to the USB thread, as shown in Figure 1.

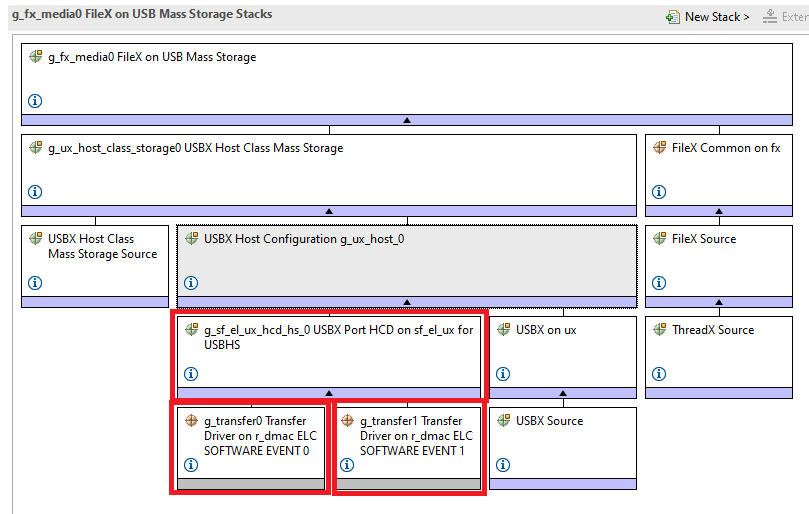
Figure 1



Looking in the Stacks view, we can see both FileX and USBX stack components. While USBX is responsible for the hardware, FileX handles the file system interactions which are necessary for opening media, modifying and saving files, etc… FileX can work with the FAT32 filesystem only.

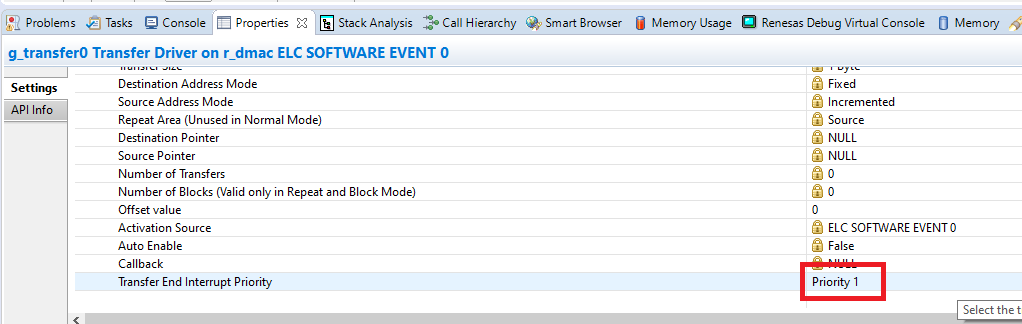
Generally, it should not be necessary to change any of these settings. However, it is important to be aware of the available interrupts and their priority, and how this can affect RTOS operation. Looking at Figure 2, there are three stack components indicated.

Figure 2



Selecting any of these components and looking at their properties in the Properties view, we can see an a modifiable interrupt priority is listed. In the case of the two Transfer Driver components, it’s necessary to scroll to the bottom to see this. Figure 3 shows the interrupt priority selection for “g\_transfer0”.

Figure 3



It’s important to understand the influence of thread and interrupt priority on application behavior, and be aware of the priority of various interface interrupts relative to thread priority. Improper selection can result in high latency or interface failure. For further information about this, please refer to the appropriate Renesas documentation, which can be found by right-clicking on a stack component and selecting Module Resources for module-specific information. For general ThreadX information, search for “X-Ware Docs” on the Renesas website.

After any modification, click the Generate Project Content button to rebuild the project data with the new settings.

Figure 4



# USB Thread

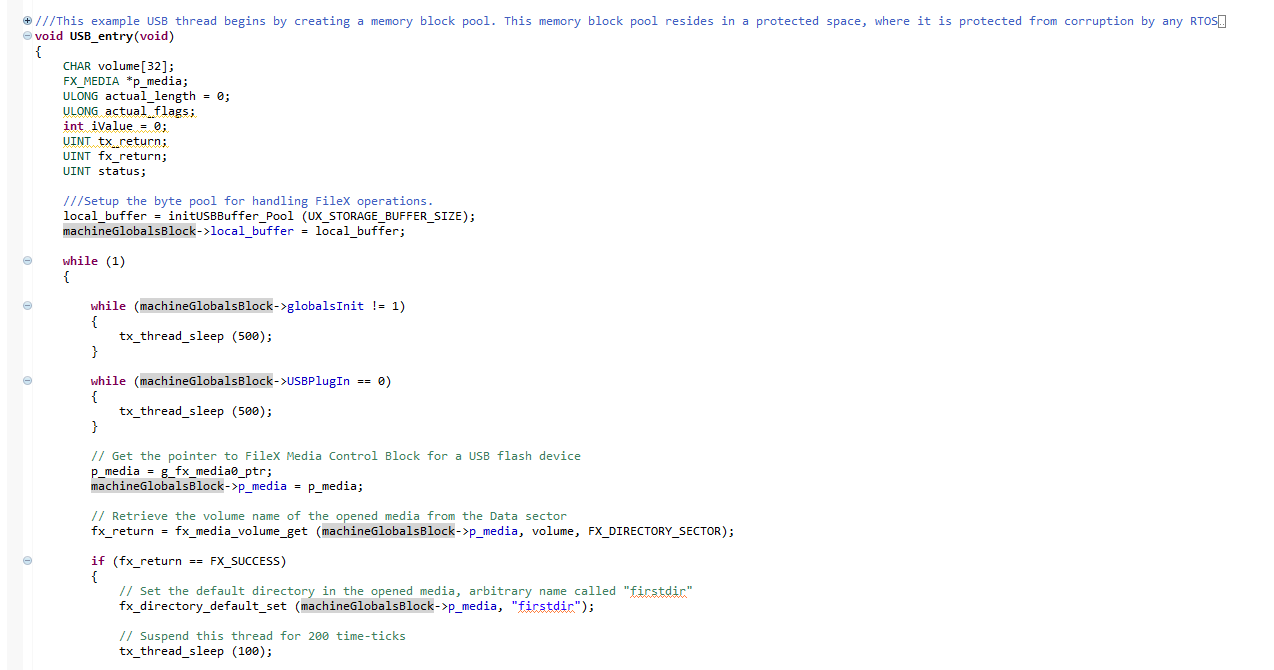
The example project contains a USB thread. This thread contains example code both for the Device and Host interfaces. Open USB.c and scroll down until you see this section of code:



Everything outside of this loop is related to the USB Host interface. For this tutorial we have no interest or need for this. For the sake of our Host example, this loop will need to be removed to allow the final flushing and closing operation for the storage media. Although it’s good practice to do this, it’s not technically necessary that the media be closed prior to the USB stick being removed. It’s a best practice for avoiding data corruption, but it’s essentially the same as using the “Safely Remove Hardware” option in Windows prior to removing a flash drive. Failing to do this will not result in data corruption unless an operation is still in progress.

With that out of the way, let’s go to the beginning of the USB thread, as shown in Figure 5.

Figure 5



Here we see some variables are created initially. Moving on, we see a call to “initUSBBuffer\_Pool” which returns a pointer to “local\_buffer”. This operation creates a memory pool which is used to safely store the contents of the USB Host buffer.