Developer's Guide To Pacemaker Development

Tutorial 3: Serial Communication

SFWRENG/MECHTRON 3K04
McMaster University

Michael Kehinde
kehindem@mcmaster.ca

Mostafa Ayesh
ayeshm@mcmaster.ca

November 5, 2020

Serial Guide

This document will introduce a method of communicating data between a Simulink model and a computer. This method will use the UART (Universal asynchronous receiver/transmitter) interface on the FRDM-K64F to transmit and receive data to and from a computer using a serial port.

Topics Covered

- What is serial communication and how does it work?
- Configuring UART serial communication
- Designing a serial packet protocol

Prerequisites Read **srsVVI**
Review **Tutorial 1.4**

1 Background

When developing a system that has components that function together for a common purpose, an essential aspect to incorporate is communication. Communication is the process of exchanging information between system components. System components interact with each other by transmitting and receiving digitally coded data. Information in computing systems is digitally coded using binary digits, or bits of 0's and 1's.

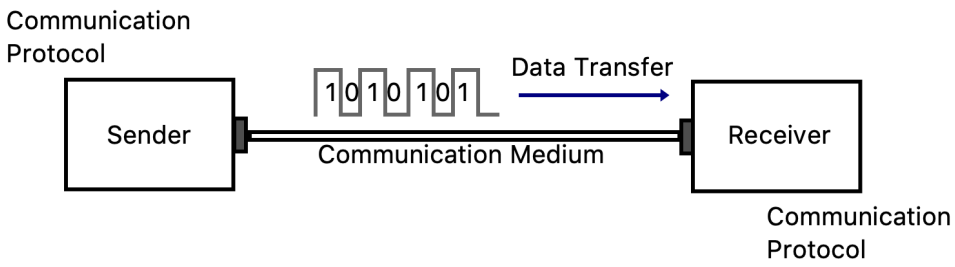

Figure 1: Serial Data Communication

Communication between system components is achieved by implementing a communication protocol over a communication medium. A **communication medium** is a means of linking components together to enable them to interact with each other. Data exchange is achieved according to defined rules known as **communication protocols**. A common protocol must be shared between sender and receiver during data communication.

There are hundreds of communication protocols, however, in general each can be separated into two categories: parallel communication and serial communication.

Parallel communication refers to a class of communication protocols where data is transmitted simultaneously, a group of bits at a time over various parallel paths on the communication medium. Parallel communication contrasts with serial communication, with which data is transmitted sequentially, one bit a time. The main advantages of serial communication include: reduced cost and complexity, and higher data throughput with long distances.

There are a number of serial standards, the most common are I2C and UART. For the purposes of this project, you will be designing and implementing a communication layer in software using the UART serial communication protocol over USB to share information between the DCM and the pacemaker.

1.1 UART Serial Communication Configuration

UART serial communication must be configured properly to establish communication between two components. You will need to keep the following three configurations in mind: transmission speed, data length and serial port.

Transmission speed: the transmission speed is rate at which data is pushed out to the communication channel. Transmission speeds are measured in units of baud, which is the maximum number of bits per second that is capable of being transferred. The sender and receiver must use the same transmission speed to maintain synchronization.

UART uses an asynchronous mode of transmission, which means data can be sent intermittently without the use of a common clock signal to determine when the data is to be sent, received and sampled by the sender and receiver.

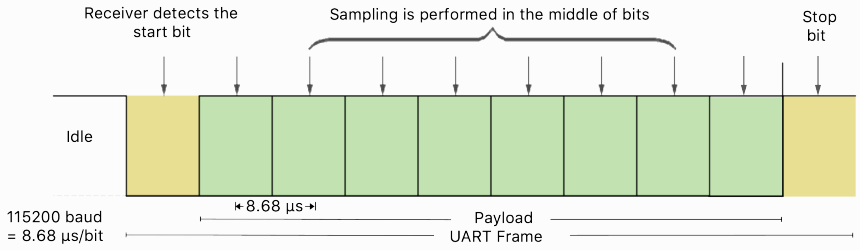

Figure 2: UART Protocol Format

When data is transmitted using UART, the payload is prefixed with a digital low header bit to mark the beginning of the signal and a high stop bit to mark the end. This process is known as framing and is performed inherently by the UART peripherals. The transmission speed is used by sender and receiver to determine the bit time of each bit for sending and sampling the data.

Data length: The sender and receiver must agree on the number of bytes to be transferred and received, respectively. The process described in Defining Data Types can be used to determine how to arrange bytes in a serial packet, what each byte (or group of bytes) mean and how many bytes are to be transferred.

Serial port: the correct serial port must be selected on the sender end and on the receiver end.

In the past, computers had additional physical hardware on the back of the machine called serial ports that provided an interface for serial communication. Nowadays, USB drivers are used to emulate serial ports by mirroring information on the UART to create a virtual serial port over USB. In Windows, device names for serial communication are denoted with "COM" in their name (short for communication). In Unix based systems, the device names begin with "cu." or "tty." When you connect the FRDM-K64F to your computer and try to create a serial environment in software, you will need to select the correct device name for the FRDM-K64F.

1.2 Designing the Packet Protocol

Data that is transferred across the communication channel is nothing but streams of bytes, each byte comprised of 8 bits of 0's and 1's. A stream of bytes make up a serial packet.

In order to achieve efficacious serial communication, a serial packet protocol must be implemented beforehand to provide context on how to interpret the data. A packet protocol specifies how to code and decode raw serial bytes, and defines the format and the order of bytes in a serial packet, as well as the actions taken on the transmission and/or receipt of the packet.

The serial packet protocol is layer of abstraction of the communication protocol. When a packet is transmitted, UART inherently sends each byte from the serial packet one at a time within the payload of a UART frame like the one shown in Figure 2. When frames are received by the receiver, each payload is extracted and stored in the order it was received. One or more bytes in the payload must then be combined appropriately if necessary.

When designing the packet protocol, a table can be used to represent the structure and data contents of a serial packet as shown in sections 5.1.1 and 5.1.2 in the document srsVVI. In the table in section 5.1.1 are different kinds of codes for raw serial data.

Function codes are used to specify action that should be taken upon receipt of a packet. Error detection codes, such as checksums, are used to enforce data integrity by detecting transmission errors due to interference. Sync bytes are used to identify the start of a payload. A good design considers the aspects of reliable data transmission and efficient data transmission.

Note: since we are using a different communication protocol than that which is described in the srsVVI, you may find that some codes may be redundant and/or may be applied at the UART level of abstraction rather than within the serial packet.

1.2.1 Defining Data Types

When we want to transfer data, reasonable data types must be specified to represent the kinds of data that are being sent. The following table describes common data types in Simulink, their size, range and resolution.

Data Type	Description	Size (bytes)	Range	Resolution	C Type
uint8	unsigned integer of 8 bits	1	[0, 255]	1	char
int8	signed integer of 8 bits	1	[-128, 127]	1	signed char
uint16	unsigned integer of 16 bits	2	[0, 65535]	1	unsigned short
int16	signed integer of 16 bits	2	[-32768, 32767]	1	short
single	single precision floating point number	4	[-3.4e38, 3.4e38]	3.4e-38	float
double	double precision floating point number	8	[-1.7e308, 1.7e308]	1.7e-308	double

A single byte is the smallest unit of digital information that can be packed in a serial packet. However, sometimes more than one byte is required to represent the same piece of information.

Suppose you wish to represent the following parameters in a serial packet using as few bytes as possible. What is the minimum number of bytes that can be used? What data types would be used?

Parameter	Min	Max	Unit	Data Type	Number of bytes
LRL					
Amplitude					
Pace Width					
				Total	

Table 1: Data type designation table.

Aside from packet size, are there any other considerations when assessing choices of data types in view of designing for a safety critical system?

1.2.2 Endianness

When it comes to serial communication, developers should be cognizant of endianness. Endianness refers to the storage order of multi-byte data in memory. There are two endian types: big-endian (or *Motorola*) and little-endian (or *Intel*). Computing systems store multibyte values with the most significant byte at a lower address (big-endian based system) or with the most significant byte at a greater address (little-endian based system).

In the figure below, D_0 to D_{31} represent the binary digits of a 4-byte data item. Bits D_0 to D_7 make up the least significant byte and bits D_{24} to D_{31} make up the most significant byte. The value represented by D_0 to D_{31} is interpreted the same way by both systems.

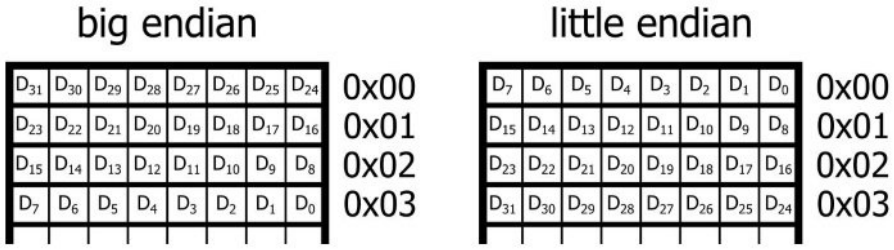

Figure 3: Little Endian and Big Endian Data Storage

Endianness comes into play when serial packets are transmitted between systems or system components with different endian architectures. As discussed above, in order to send information from one component to another, the data is packed into a serial packet. The serial packet is transmitted as a stream of bytes over one or more single-byte transactions. The receiver receives each byte in the byte stream; each byte is stored in the order it is received.

A potential problem can occur if a serial packet has sub-components that are multibyte values. If the multibyte value arrived in the same order for a system with a different endian type, the representation of the data may be compromised.

Most serial software libraries provide support for handling endianness when packing/unpacking bytes. Handling can be implemented on the receiver end to interpret the packet using the intended endian type, or on the sender end by changing the byte order prior to transmission.

2 Tutorial

2.1 Simulink Serial Guide

2.1.1 Simulink Blocks

The following block libraries are used to implement serial communication in Simulink: the Simulink Coder Support Package for NXP FRDM-K64F Board and Embedded Coder.

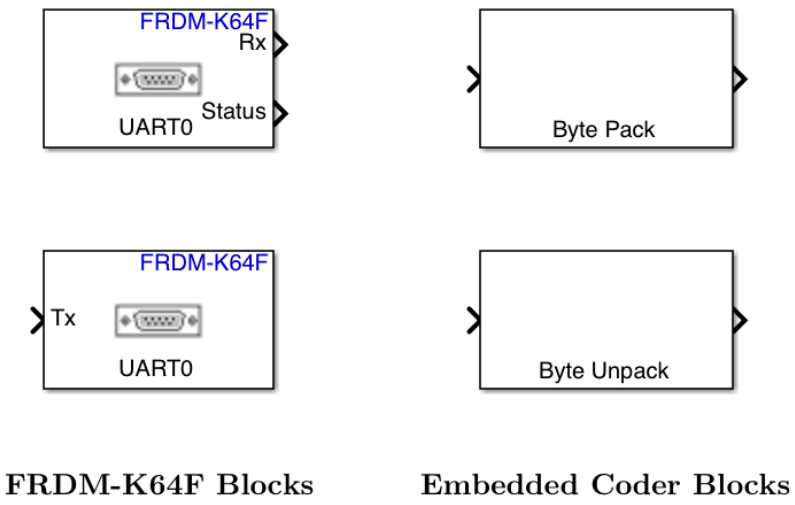

Figure 4: Serial Block Libraries

2.1.1.1 FRDM-K64F Blocks

2.1.1.1.1 Serial Receive

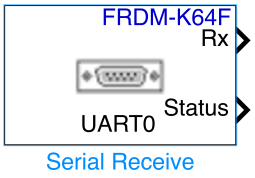

Figure 5: Serial Receive Block

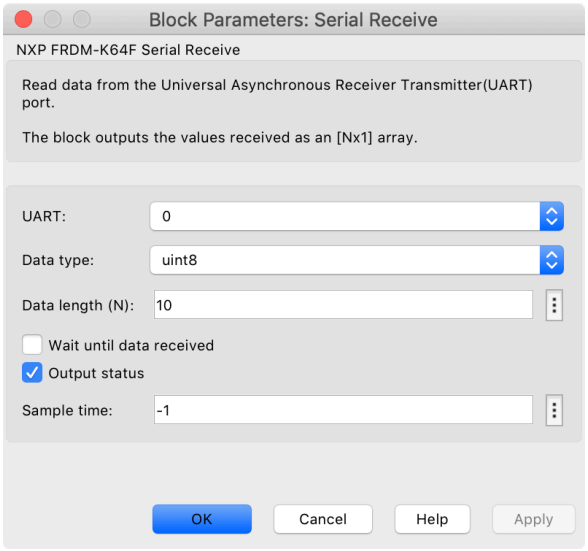

Figure 6: Serial Receive Options

The **Serial Receive** block is used for receiving serial data from the outside into the model. The recommended settings for the block is as follows:

- **UART:** 0 *(USB serial communication)*
- **Data Type:** uint8 *(data received will be an array of bytes)*
- **Data length (N):** *maximum number of bytes to be received*
- **Wait until data received:** unchecked
- **Output status:** checked
- **Sample time:** -1 *(-1 for inherit, i.e. equals the model's step-size)*

Note: if the number of bytes received are more than the data length (N), the extra bytes will be dropped.

2.1.1.1.2 Serial Transmit

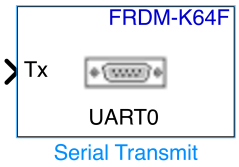

Figure 7: Serial Transmit Block

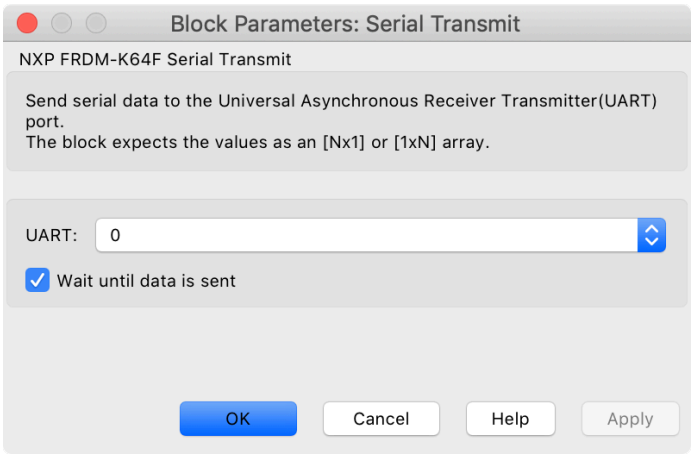

Figure 8: Serial Transmit Options

The **Serial Transmit** block is used for transmitting serial data from the model to the outside. The recommended settings for the block is as follows:

- **UART:** 0 *(USB serial communication)*
- **Wait until data is sent:** checked

2.1.1.1.3 Notes:

- The default baudrate for serial communication is set at **115200**. Make sure to match that in the DCM.
- To view current configurations, refer to Configuration Parameters » Hardware Implementation » Target Hardware Resources » UART0.

2.1.1.2 Byte Packing and Byte Unpacking

2.1.1.2.1 Byte Packing

Byte Packing: Converts data from single/multiple sources/types into an array (vector) of bytes *uint8* (eg. *single* converted to an array of 4 *uint8* bytes).

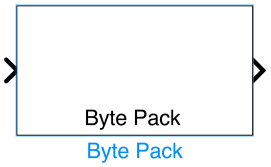

Figure 9: Byte Pack Block

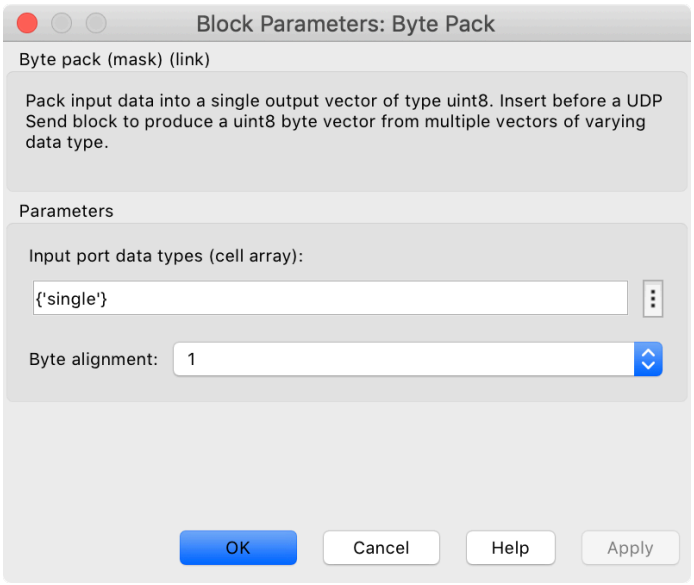

Figure 10: Byte Pack Options

- **Input port data types (cell array):** A cell array with the types of input data
- **Byte Alignment:** Leave unchanged

2.1.1.2.2 Byte Unpacking

Byte Unpacking: Converts an array (vector) of data into single/multiple arrays (vectors) of a specified type.

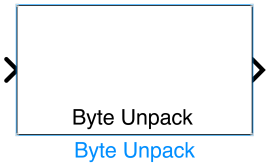

Figure 11: Byte Unpack Block

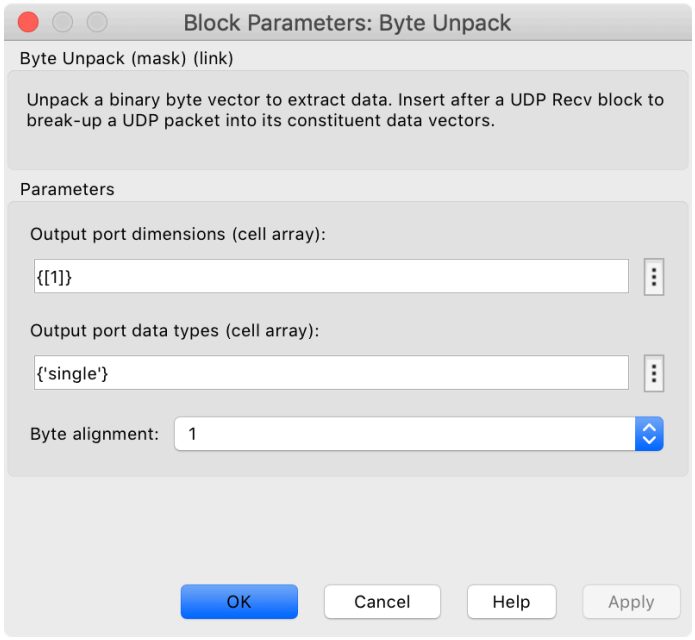

Figure 12: Byte Unpack Options

- **Output port dimensions (cell array):** {{1}} *(leave unchanged)*
- **Output port data types (cell array):** The data type you want convert an array of bytes (uint8) to.

2.1.1.2.3 Byte Packing and Byte Unpacking in Stateflow

Inside a stateflow chart you can use **typecast(data, datatype)** examples:

- **Byte Packing:** if we have a variable *inputdata* of type *single* and we want to convert it to an array of 4 bytes *(uint8)*: **typecast(inputdata, 'uint8')**
- **Byte Unpacking:** if we have an array of 2 bytes *uint8* and we want to convert them into a value of type *uint16*: **typecast(inputdata(1:2), 'uint16')**

2.1.2 Serial Communication Example in Simulink

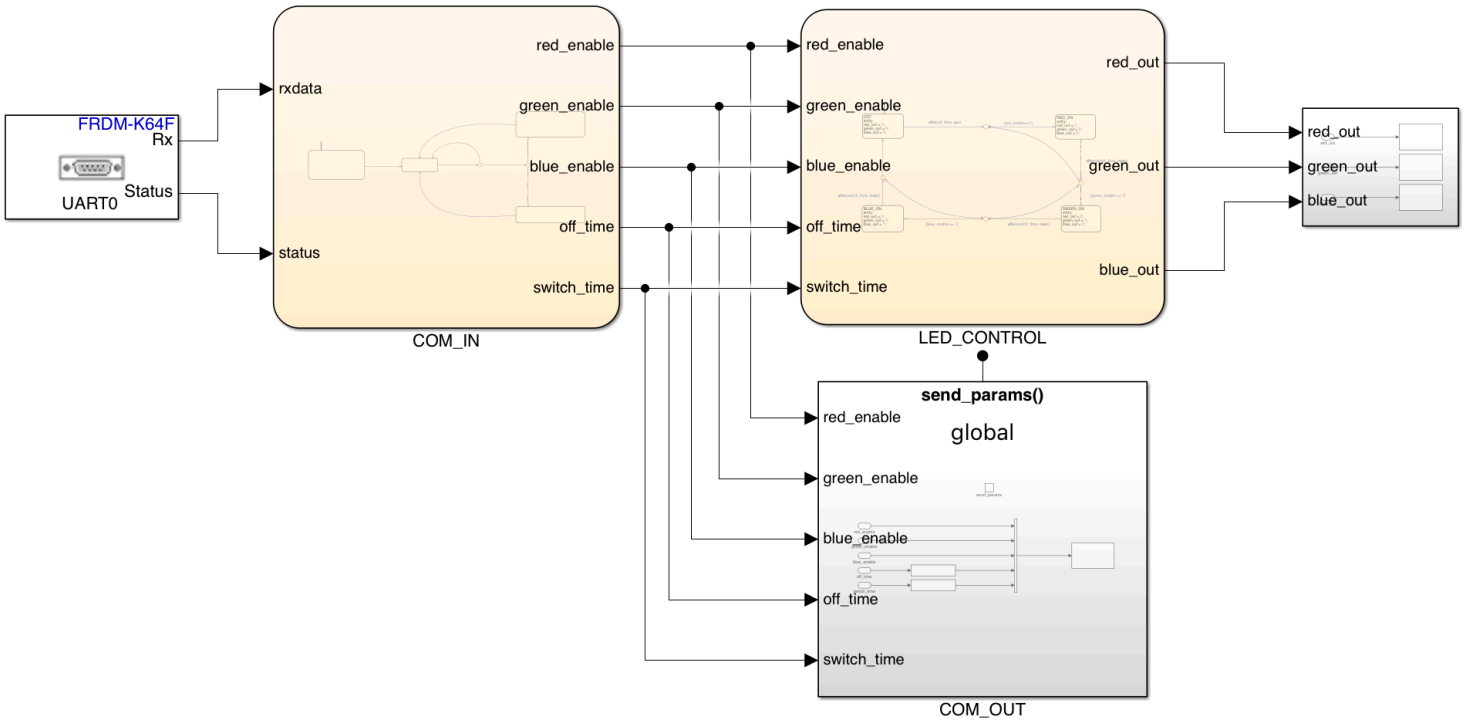

Figure 13: Simulink Model

The data received by the microcontroller from the user interface is constructed as follows (ordered from top to bottom) you can find more on this in the SRSVVI Document:

	Size
SYNC	1 uint8 byte (**CONSTANT** 0x16 signifies beginning of transmission)
FN_CODE	1 uint8 byte (**CONSTANT** 0x55 for setting params, 0x22 for echo params)
RED_ENABLE	1 uint8 byte
GREEN_ENABLE	1 uint8 byte
BLUE_ENABLE	1 uint8 byte
OFF_TIME	4 uint8 bytes (type single)
SWITCH_TIME	2 uint8 bytes (type uint16)
TOTAL	**11 uint8 bytes**

Table 2: Serial Packet Format

Serial Receive Block: Set Data length (N) to total number of uint8 bytes 11.

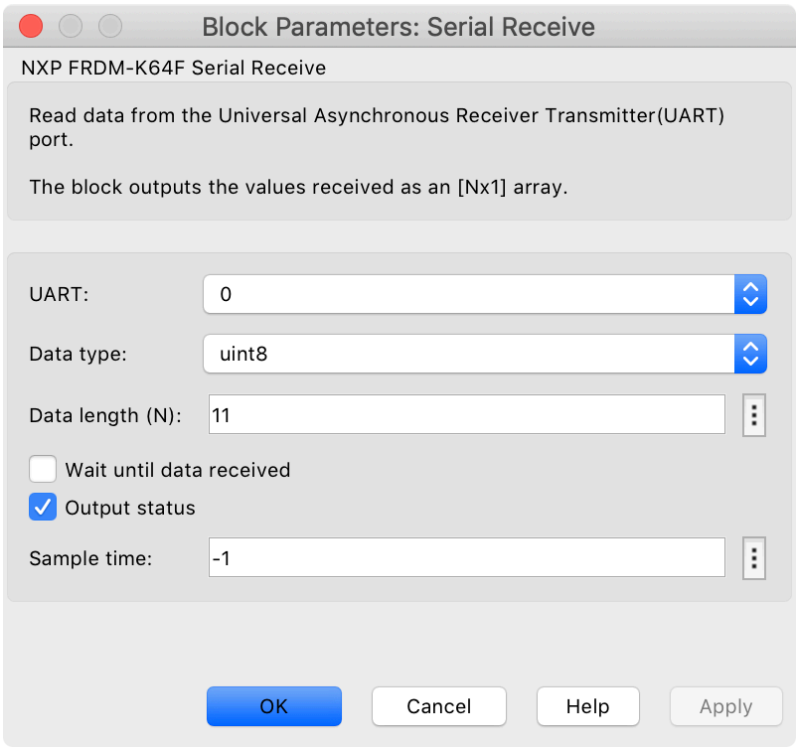

Figure 14: Serial Receive Options

2.1.2.1 COM IN Chart

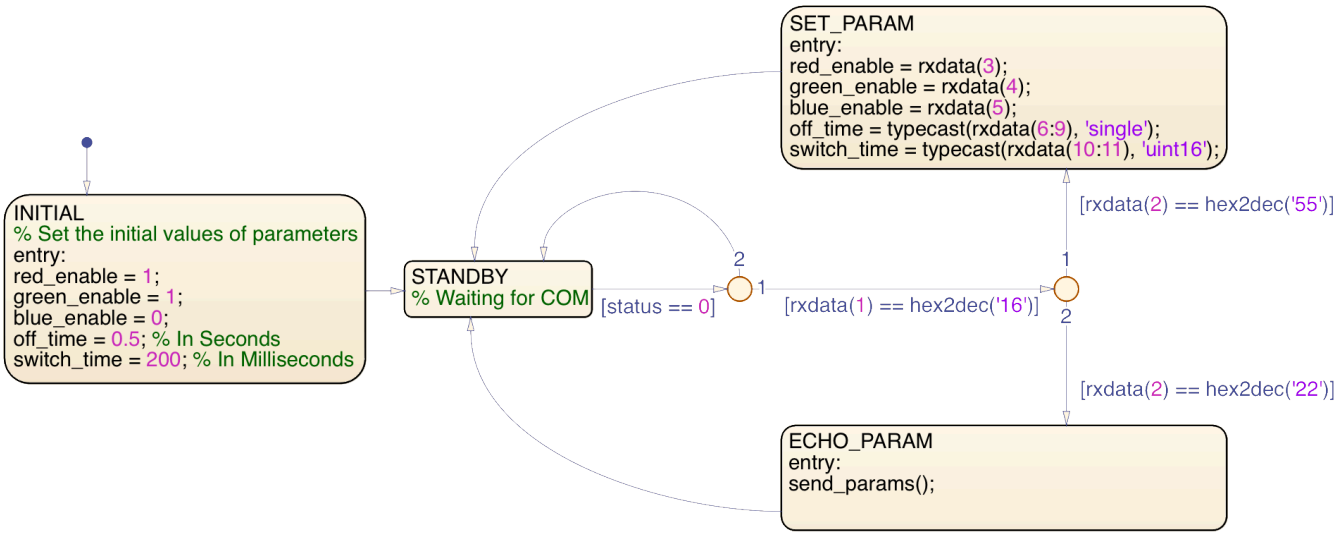

Figure 15: COM_IN Chart

- **Inputs**
  - **rxdata:** type *uint8*
  - **status:** type *uint8*
- **Outputs**
  - **red_enable:** type *uint8*
  - **green_enable:** type *uint8*
  - **blue_enable:** type *uint8*
  - **off_time:** type *single*
  - **switch_time:** type *uint16*
- **States**
  - **INITIAL:** Setting the initial values of the outputs (values before receiving any serial communication
  - **STANDBY:** Waiting for incoming serial communication
  - **SET_PARAM:** Uses the serial input to assign values to the parameters
  - **ECHO_PARAM:** Sends back the current value of the parameters

- **NOTES**
  - From Model Explorer (CTRL + H), check the box that says "**Execute (enter) Chart At Initialization**", in order to set the initial values of the parameters on execution.

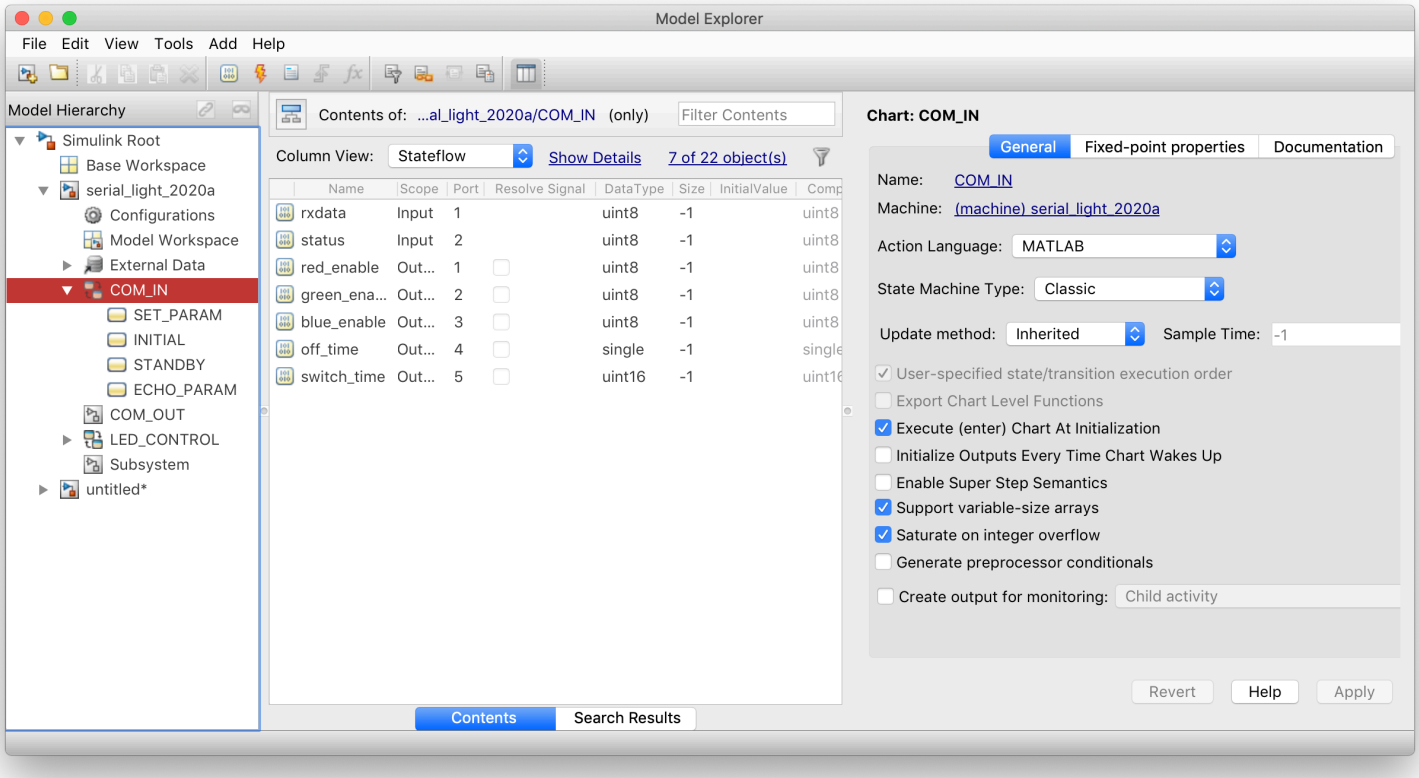

Figure 16: Model Explorer

2.1.2.2 COM OUT Subsystem (Function-Call Subsystem Block)

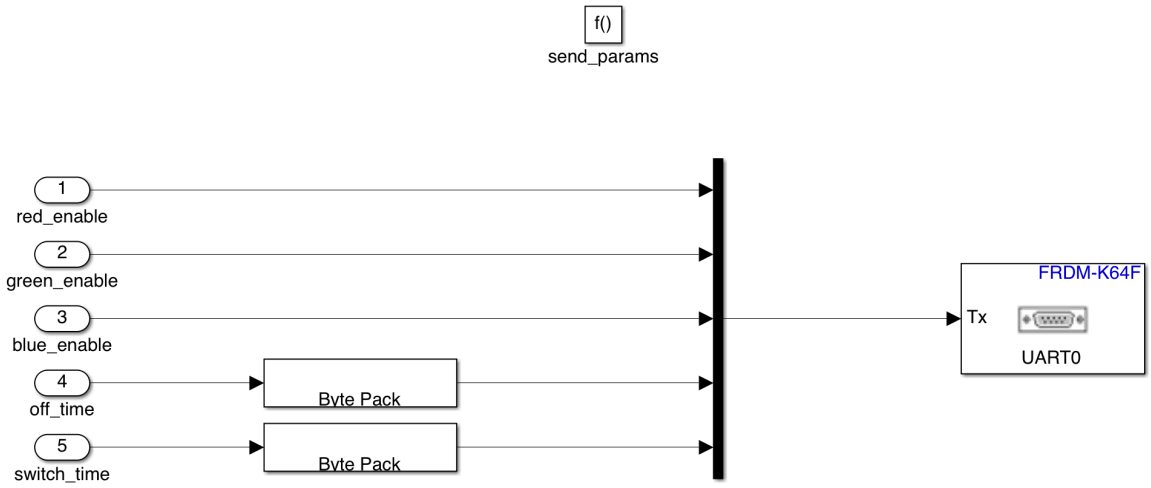

Figure 17: COM_OUT Subsystem

- **Inputs**
  - Same outputs of **COM_IN** chart

- **Trigger Settings**

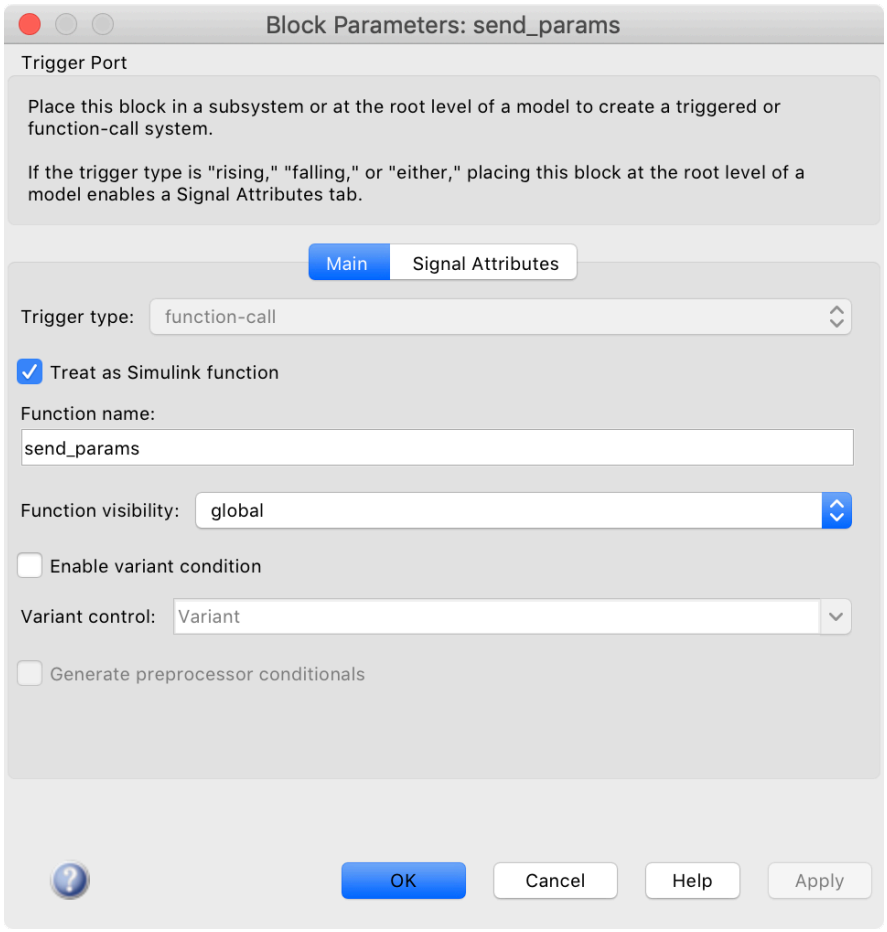

Figure 18: Trigger Options

  - **Treat as Simulink function:** Check the box to be able to call the function inside a Stateflow chart
  - **Function name:** Function name used to call the function inside a Stateflow chart

- **NOTES**
  - Byte Pack converts **off_time (single)**, **switch_time (uint16)** to **uint8** to be sent through serial

2.1.2.3 LED_CONTROL Chart

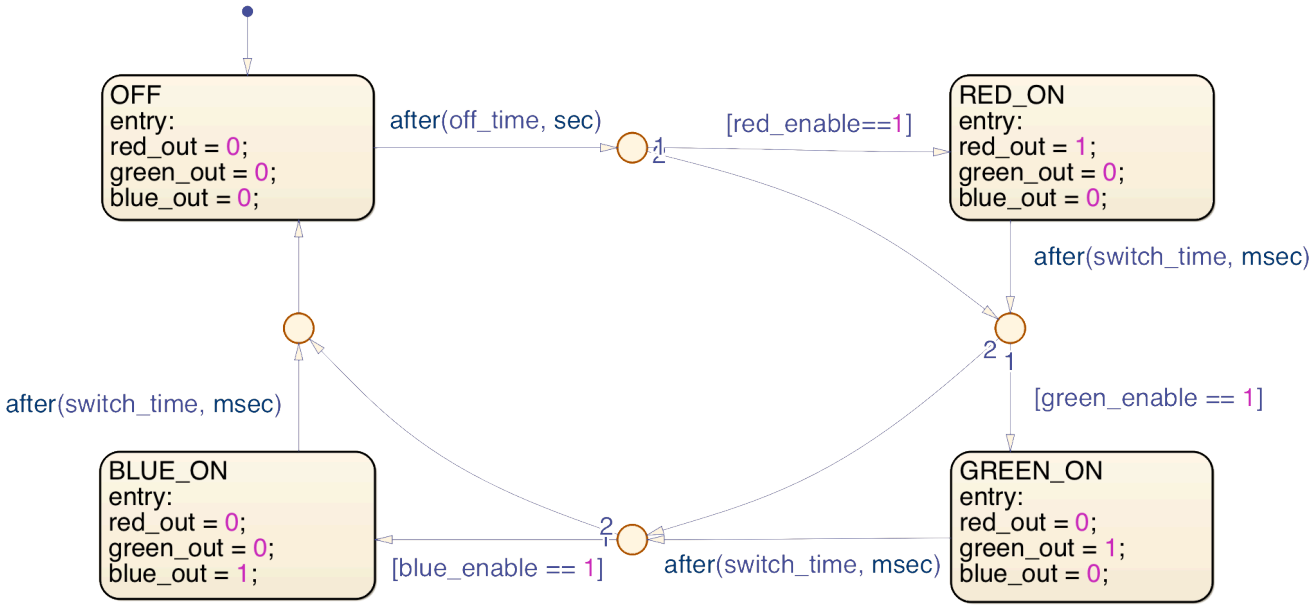

Figure 19: LED_CONTROL Chart

- **Inputs**
  - **red_enable:** type *uint8*
  - **green_enable:** type *uint8*
  - **blue_enable:** type *uint8*
  - **off_time:** type *single*
  - **switch_time:** type *uint16*
- **Outputs**
  - **red_out:** type *boolean*
  - **green_out:** type *boolean*
  - **blue_out:** type *boolean*
- **States**
  - **OFF:** Turn off all LEDs
  - **RED_ON:** Turn on the red LED exclusively
  - **GREEN_ON:** Turn on the green LED exclusively

  - **BLUE_ON:** Turn on the blue LED exclusively

- **NOTES**
  - Respect the transition execution order

2.1.2.4 LED Subsystem

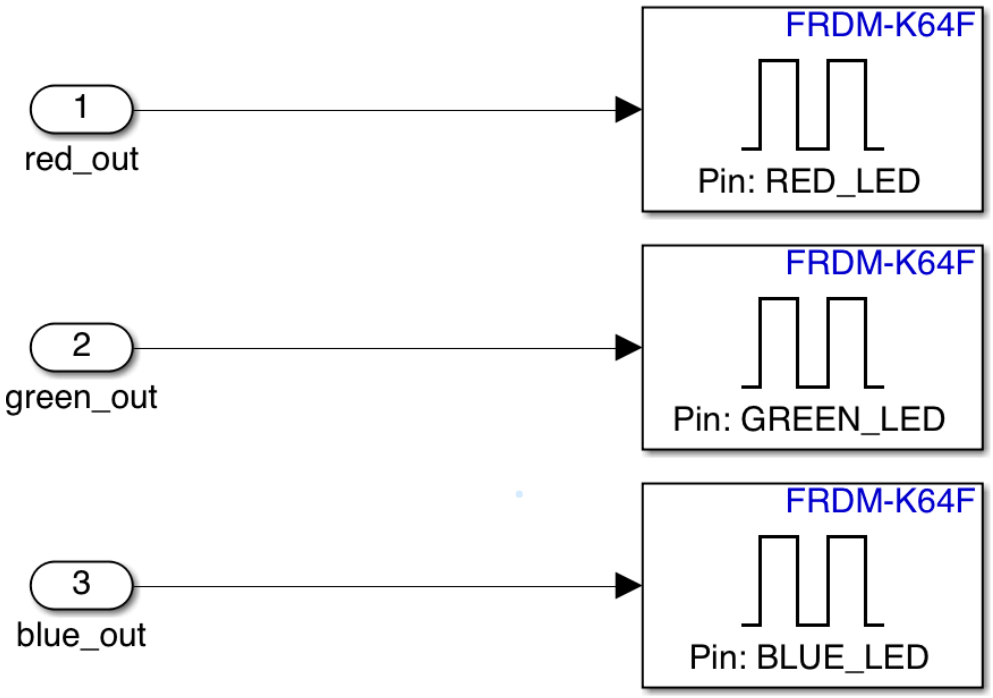

Figure 20: LED Subsystem

- **Inputs**
  - **red_out:** type *boolean*
  - **green_out:** type *boolean*
  - **blue_out:** type *boolean*

2.2 Serial Communication on Computer

1. Write a basic script to configure serial communication as per the specification in Table 2.

2. Transmit several serial packets from your computer to the FRDM-K64F to verify the Simulink model.

   Note: Many programming languages provide support for typecasting and transmitting/receiving serial data. If you are using Python to build your DCM, see `https://buildmedia.readthedocs.org/media/pdf/pyserial/latest/pyserial.pdf` for the Python 3.4 serial communication library and `https://docs.python.org/2/library/struct.html` for the documentation of the struct library for typecasting.

3 Revision History

Version	Date	Modification	Modified by
1.0	Nov. 3, 2019	Initial Simulink Serial Guide	Ayesh, Mostafa
2.0	Nov. 5, 2020	Introduced background material, adapted tutorial content, integrated document template	Kehinde, Michael