Application Note:
TLT-0652-AN-MICROEJ-ECOM-COMM

ECOM Comm

In relation to: MICROEJ products

Features

This Application Note explains how to use the ECOM Comm API to add support for serial communication.

Description

This Application Note assumes the reader wishes to understand how to use the ECOM Comm API to add support for serial communication.

Table of Contents

List of Figures

1 Introduction

1.1 Intended audience

The intended audience for this Application Note are developers who want to add support for serial communication to their MicroEJ® Java platform.

1.2 Scope

This Application Note shows how the ECOM Comm Low-Level API (called LLCOMM) is used to implement a serial communication driver. The Note does not provide an implementation for any specific hardware; it shows the principles of operation of the LLCOMM API.

1.3 Background

The MicroEJ Java platform contains both Java and native libraries that support communication with devices connected via serial ports, usually via UART hardware. The ECOM library is a Java connection framework based on the MIDP Connector pattern. The ECOM Comm library extends ECOM to allow stream-based communication via serial communication ports. Collectively these libraries are known as the "ECOM Comm stack".

To make use of the stack a native driver that understands the specific hardware must be implemented. The driver communicates with the ECOM Comm stack via the LLCOMM API. This Application Note explains how to create such a driver, using the "buffered" variant of the LLCOMM API. In this variant, the stack takes responsibility for buffering between the Java application and the driver.

1.4 Prerequisites

This document assumes the reader is familiar with the process of creating a Java Platform (JPF), and with the principles of operation of UARTs. It also assumes the reader is able to build and deploy a MicroEJ application.

1.5 Files supplied with this Application Note

This application note is packaged with an archive of an Eclipse project in the file `Ecom-Comm-example.zip`, which should be imported in the normal way.

It also contains a number of files that are referenced in the remainder of this Application Note.

2 LLCOMM concepts and operation

2.1 Implementations and connections

- LLCOMM implementation

  Each driver that supports the LLCOMM API is said to be an *LLCOMM implementation*. The implementation must be given a name, defined using a `#define` statement in the driver C file. The ECOM Comm stack can simultaneously access multiple implementations.

- Connection

  A *connection* is a bi-directional data channel between the Java application and a specific serial port. Each LLCOMM implementation can provide one or more connections.

- Application port number

  An *application port number* is a number that identifies a particular serial port provided by the platform. Each connection has a unique application port number. An application specifies a port number when it opens a connection. Application port numbers do not have to start from 0, and they do not need to form a contiguous set.

For example, a UART chip might provide four UARTs, each controlled in the same way (but with different register addresses). Since the code required for these UARTs is the same, a single driver (single LLCOMM implementation) can be created, and four connections defined for that implementation.

If the platform is to support two different UART chips, that require different driver code, then there will be two LLCOMM implementations, each with a distinct name, and a number of connections defined for each one.

In the driver each connection is represented by an instance of a data structure that holds information about the connection. The first parameter of every call made by the stack to the driver is the address of a connection data structure - the requested operation is to be applied to that connection.

2.2 Interactions

In this section we explore the interactions between the ECOM Comm stack and a LLCOMM implementation (i.e. a driver). We will assume there is a single LLCOMM implementation that provides two connections, with platform port ids 0 and 1. The examples assume the use of the "buffered" variant of the LLCOMM API.

When the MicroJVM starts, the stack initializes each connection by calling `LLCOMM_BUFFERED_CONNECTION_IMPL_initialize`.

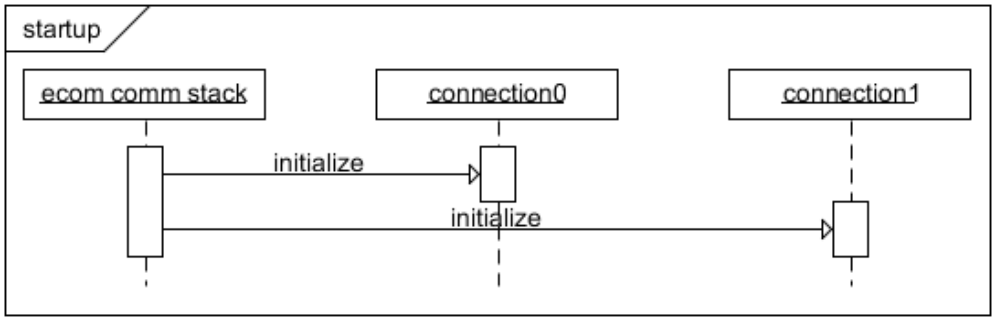

Figure 2.1. Initialization

When the Java application attempts to open a connection by calling `Connector.open` with an appropriate connection string, the stack attempts to find the matching connection provided by the driver by calling `LLCOMM_BUFFERED_CONNECTION_IMPL_getPlatformId()` on each connection in turn. The first such call that returns same platform id than application id is assumed to be the connection that is be-

ing opened (for a better understanding, please refer to *Platform Architecture User Manual*, paragraph *ECOM COM / Comm Port Identifier / Opening Sequence*). The stack then configures the connection by calling `LLCOMM_BUFFERED_CONNECTION_IMPL_configureDevice`, passing in the parameters specified by the application in the connection string.

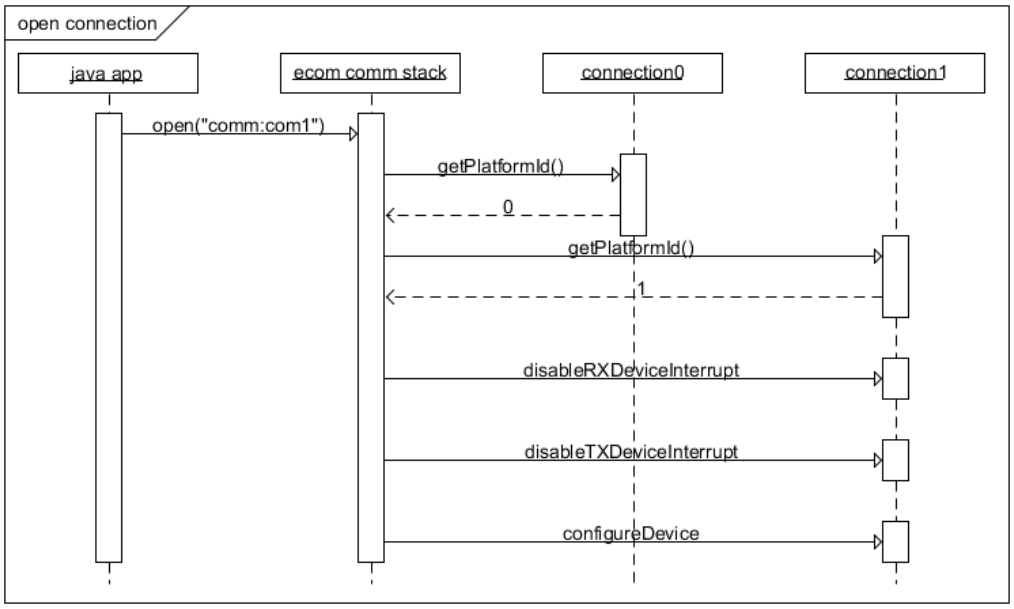

Figure 2.2. Open a connection

When the Java application opens an output stream by calling `conn.openOutputStream()` the stack asks the driver for information about the buffer that it should use to hold outgoing data, and then asks the driver to enable the transmit hardware.

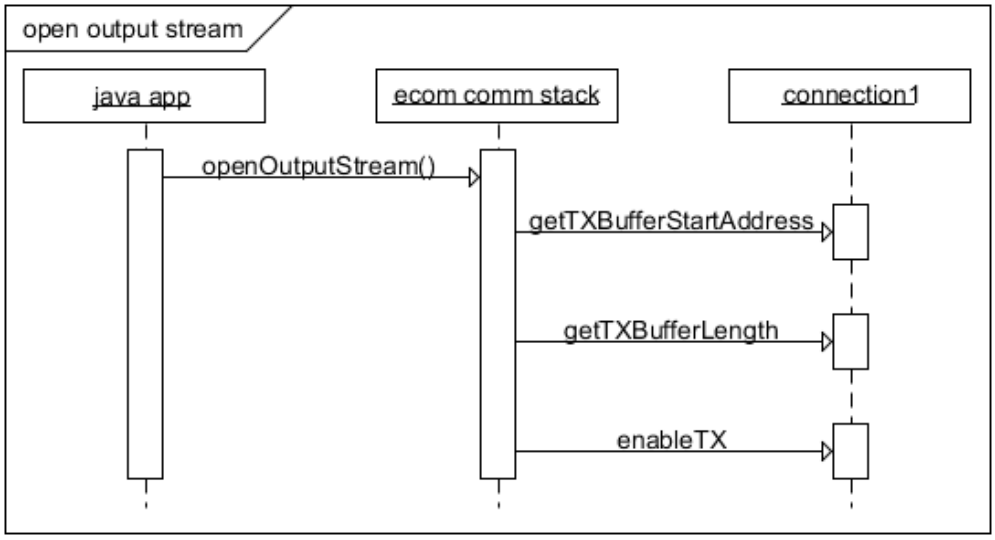

Figure 2.3. Open an output stream

When the Java application writes data to the output stream the stack buffers the data and asks the driver to enable the interrupt that occurs when the hardware can accept data for transmission. As the hardware is initially idle this interrupt will occur almost at once. The driver's interrupt handler will now notify the stack that the hardware is ready to accept data (by calling `LLCOMM_BUFFERED_CONNECTION_transmitBufferReady`), and the stack will respond by passing data to the driver by calling `LLCOMM_BUFFERED_CONNECTION_IMPL_sendData`.

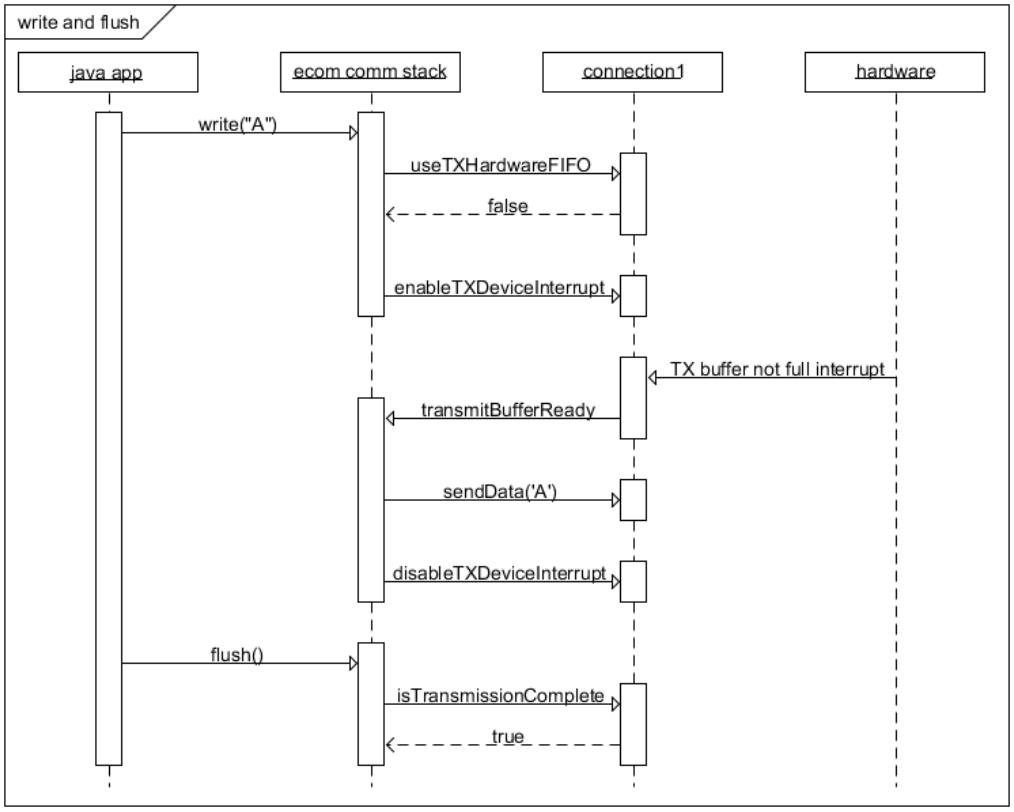

Figure 2.4. Write to an output stream

When the Java application flushes the output stream the stack blocks the caller until the transmit buffer is empty and the driver responds `true` to `LLCOMM_BUFFERED_CONNECTION_IMPL_isTransmissionComplete`.

If the Java application tries to write more data than will fit in the transmit buffer managed by the stack, the calling thread of the application will be blocked until space is available. Therefore it is essential to size the transmit buffer so that the application will perform in an acceptable manner.

Closing the output stream causes the stack to ask the driver to disable the transmit hardware associated with the connection.

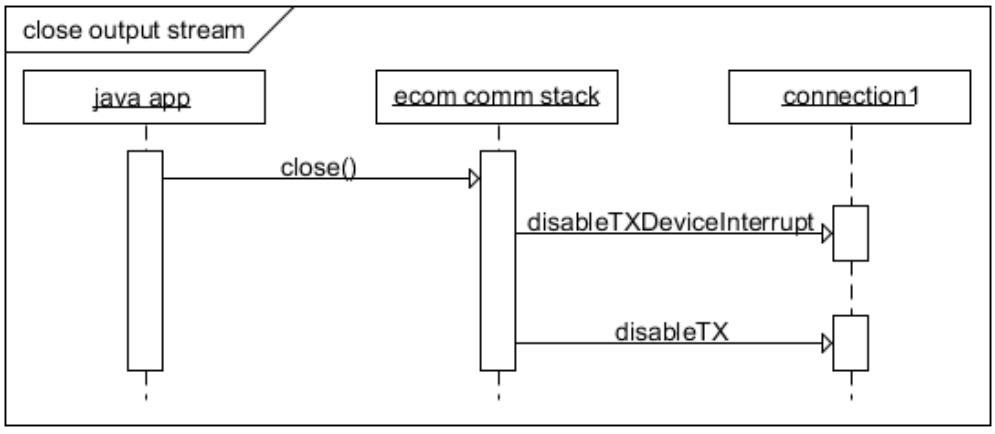

Figure 2.5. Close an output stream

Opening an input stream results in an interaction that is very similar to opening an output stream, except that the stack immediately asks the driver to enable receive interrupts, so that any incoming data can be buffered.

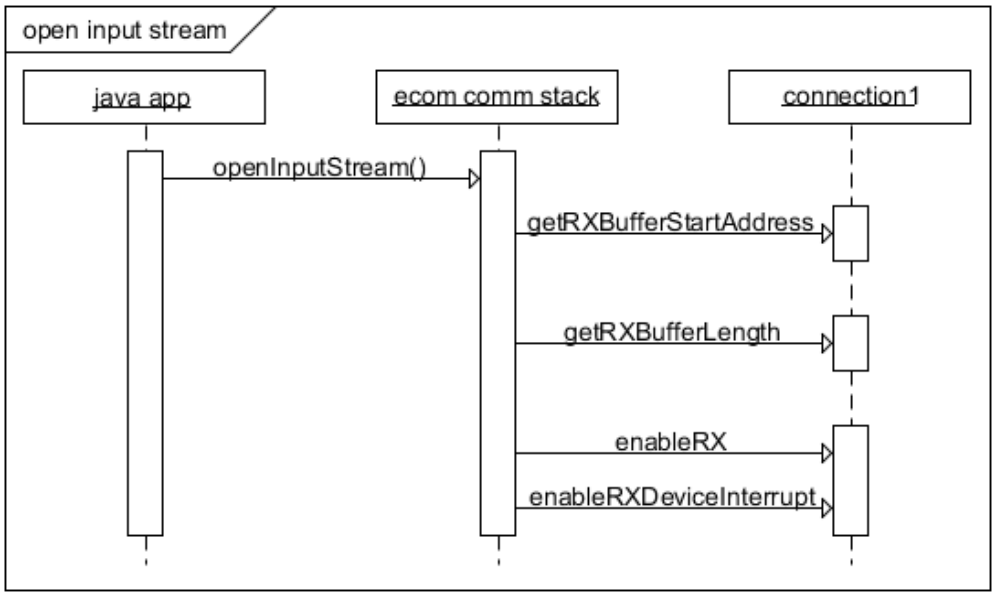

Figure 2.6. Open an input stream

When data arrives the driver's interrupt handler passes the data to the stack by calling `LLCOMM_BUFFERED_CONNECTION_dataReceived`. The stack puts the data in the receive buffer, from where it is retrived when the Java application reads from the stream.

If the driver tries to write more data than will fit in the receive buffer managed by the stack, the data is lost. Therefore it is essential to size the receive buffer so that it can hold all the necessary incoming data. If your usage of serial communication requires flow control you should use the "custom" variant of LLCOMM, which gives the driver complete control of data buffering.

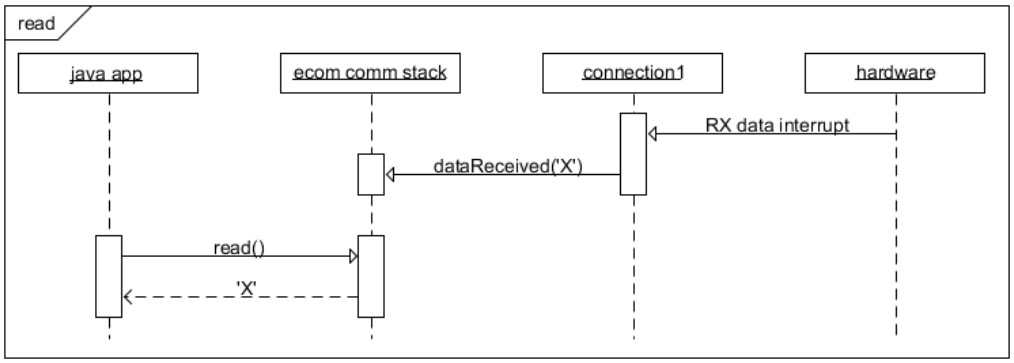

Figure 2.7. Read from an input stream

Closing the input stream causes the stack to ask the driver to disable the receive hardware associated with the connection.

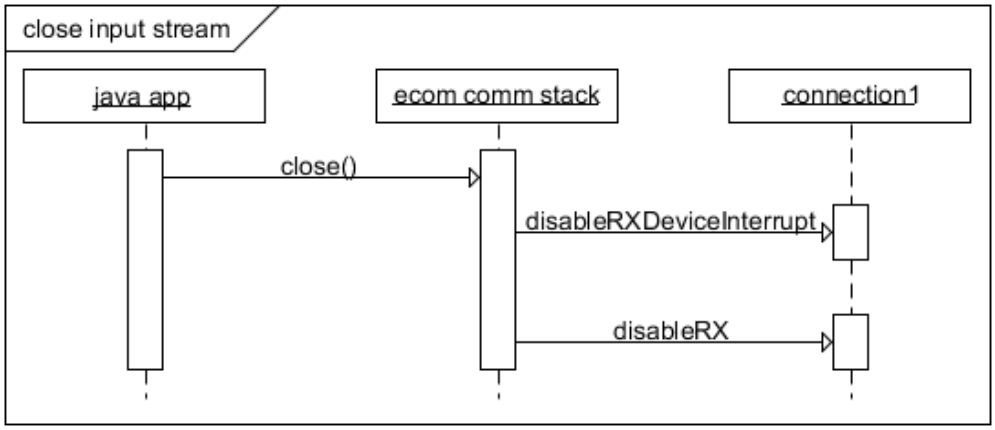

Figure 2.8. Close an input stream

3 Driver implementation

We will now walk through the steps required to create and use an ECOM Comm driver.

This section assumes that you are using a STM3240G-EVAL evaluation board, and have already created a work-in-progress JPF using the `Basic UI` platform example. We strongly recommend that you test your JPF by building and running the example application created with the platform example prior to carrying out the steps defined below.

The steps are:

1. Install the `ECOM` and `ECOM COMM` modules
2. Create the LLCOMM implementation (the driver)
3. Configure the connections
4. Create a Java application to test the driver
5. Update the BSP project and build the executable

If you have not already done so, you should import the Eclipse project supplied in `Ecomm-Comm-example.zip` now (**File** → **Import...** → **General/Existing Projects into Workspace**).

3.1 Install the ECOM and ECOM COMM modules

- Navigate to the `example.platform` file of your JPF, and go to the **Content** tab. In the **Modules** section, check the entire **ECOM** entry.
- As described in the ECOM-COMM module details, the ECOM-COMM module needs to be configured using a `ecom-comm.xml` file. To understand how to configure the module, have a look at the users manual, chapter **Embedded Communications > ECOM Comm > XML File**. The application note provides such a file that contains following configuration :

```
<ecom>
 <comms>
  <comm port="0" nickname="A COMM Port"/>
  <comm port="1" nickname="Another COMM Port"/>
 </comms>
</ecom>
```

Figure 3.1. ECOM Comm configuration

  The configuration describes that JPF exposes a single COMM port whose physical port equals 1.

- Create a `ecom-comm` folder on your JPF configuration project and copy on it the `ecom-comm.xml` file provided.
- Create a `bsp.xml` file in `ecom-comm` folder and add the following tag :

```
<nativeImplementation name="LLCOMM_UART" nativeName="LLCOMM_BUFFERED_CONNECTION" />
```

Figure 3.2. ECOM Comm native implementation configuration

  This tells the stack that there is one implementation of `LLCOMM_BUFFERED_CONNECTION` called `LLCOMM_UART`. The users manual, chapter **Embedded Communications > ECOM Comm > Driver API**, explains in details the purpose of this configuration.

- Build the platform.

3.2 Create the LLCOMM implementation (the driver)

- Copy the supplied `LLCOMM_UART.c` file into the `src` folder of your BSP project.
- Copy the supplied `LLCOMM_UART.h` file into the `inc` folder of your BSP project.
- Ensure you make the modification to the `buttons.c` file in the `src` folder of your BSP project discussed below.

The driver created in this Application Note does not control real UART hardware. It logs calls made to it, and simulates hardware interrupts.

The driver header file (`LLCOMM_UART.h`) defines the data structure for connections, and the signature of the function used by the stack to create instances of the structure.

```
#define LLCOMM_UART_TX_BUFFER_SIZE 50
#define LLCOMM_UART_RX_BUFFER_SIZE 50

typedef struct LLCOMM_UART
{
 struct LLCOMM_BUFFERED_CONNECTION header;
 int32_t platformId;
 EVAL_COM_TypeDef comIndex;
 uint8_t txItEnabled;
 uint8_t txBuffer[LLCOMM_UART_TX_BUFFER_SIZE];
 uint8_t rxBuffer[LLCOMM_UART_RX_BUFFER_SIZE];
} LLCOMM_UART;

void LLCOMM_UART_new(LLCOMM_UART* env);
```

Figure 3.3. Driver header file

The data structure, called `LLCOMM_UART`, must contain as its first element the standard `LLCOMM_BUFFERED_CONNECTION` structure defined by the stack. Subsequent elements are for use by the driver: in this case we add an element to hold the platform port id of the connection, another to hold the index of this port within the driver (typically the index of the UART within the UART package), and we declare the buffers that will be used by this connection for transmission and reception. The addresses of these buffers are returned by the `LLCOMM_BUFFERED_CONNECTION_IMPL_getTXBufferStartAddress` and `LLCOMM_BUFFERED_CONNECTION_IMPL_getRXBufferStartAddress` functions of the driver respectively.

The driver must implement all the functions defined in the `LLCOMM_BUFFERED_CONNECTION_impl.h` file that was copied to the `includeAPIs` folder in the previous step. The figure below shows a subset of the content of the driver; the full implementation can be found in `LLCOMM_UART.c`.

```
#include "LLCOMM_UART.h"
#define LLCOMM_BUFFERED_CONNECTION_IMPL LLCOMM_UART
#include "LLCOMM_BUFFERED_CONNECTION_impl.h"
#include "microej.h"
#include <stdio.h>

#define COM_PORT_COUNT (2)

static LLCOMM_BUFFERED_CONNECTION* connections[COM_PORT_COUNT];

// data used for interrupt simulation only
static uint8_t rx_enabled[COM_PORT_COUNT];
static uint8_t tx_enabled[COM_PORT_COUNT];
static int32_t rx_data[3] = {'A', 'B', 'X'};
static int32_t rx_data_index = 0;

void comm_impl_event(int32_t code) {
    if (code != 0) return;
    if (tx_enabled[0]) {
        printf("Simulating TX interrupt for port 0\n");
        LLCOMM_BUFFERED_CONNECTION_transmitBufferReady(connections[0]);
    } else if (rx_enabled[0]) {
        printf("Simulating RX interrupt for port 0\n");
        LLCOMM_BUFFERED_CONNECTION_dataReceived(connections[0],
                                rx_data[rx_data_index++]);
    }
    if (tx_enabled[1]) {
        printf("Simulating TX interrupt for port 1\n");
        LLCOMM_BUFFERED_CONNECTION_transmitBufferReady(connections[1]);
    } else if (rx_enabled[1]) {
        printf("Simulating RX interrupt for port 1\n");
        LLCOMM_BUFFERED_CONNECTION_dataReceived(connections[1],
                                rx_data[rx_data_index++]);
    }
}

void LLCOMM_BUFFERED_CONNECTION_IMPL_initialize(LLCOMM_BUFFERED_CONNECTION* env)
{
    printf("LLCOMM_BUFFERED_CONNECTION_IMPL_initialize\n");
    if (((LLCOMM_UART*)env)->comIndex >= COM_PORT_COUNT) {
        printf("***** Port index greater than %i\n", COM_PORT_COUNT - 1);
        return;
    }
    // store connection to return it in the interrupt
    connections[((LLCOMM_UART*)env)->comIndex] = env;
}

int32_t LLCOMM_BUFFERED_CONNECTION_IMPL_getPlatformId(LLCOMM_BUFFERED_CONNECTION*
 env)
{
    printf("LLCOMM_BUFFERED_CONNECTION_IMPL_getPlatformId returns %i\n",
                                    ((LLCOMM_UART*)env)->platformId);
    return ((LLCOMM_UART*)env)->platformId;
}
```

Figure 3.4. COMM implementation

We will now discuss some parts of the driver in more detail.

```
#define LLCOMM_BUFFERED_CONNECTION_IMPL LLCOMM_UART
```

Defines the name of this implementation of `LLCOMM` to be `LLCOMM_UART`.

```
static LLCOMM_BUFFERED_CONNECTION* connections[COM_PORT_COUNT];
```

Declares space to hold the address of the connection data structure, indexed by the port index. This information is needed by the interrupt handler: when it calls the stack it must pass the address of the relevent connection as a parameter. The entries in this structure are set by the `LLCOMM_BUFFERED_CONNECTION_IMPL_initialize` function.

The `LLCOMM_BUFFERED_CONNECTION_IMPL_getPlatformId` function returns the platform port id of the connection identified by the `env` parameter. As we saw in the earlier interaction diagrams, this is how the stack finds the correct low-level connection when the application attempts to open a Java connection.

The `comm_impl_event` function plays the role of the interrupt handler - we will discuss shortly how it gets called in this simulation. The `tx_enabled` and `rx_enabled` data structures contain flags indicating whether the connection's tx and rx interrupts are enabled. The flags are set by the `enableTXDeviceInterrupt` and `enableRXDeviceInterrupt` functions, and cleared by the `disableTXDeviceInterrupt` and `disableRXDeviceInterrupt` functions. For each connection, if the TX interrupt is enabled the stack is notified that data may be transmitted. If the RX interrupt is enabled the next simulated input data value is sent to the stack (In this incomplete implementation we assume that TX and RX will not both be enabled).

We will use buttons on the STM3240G-EVAL board to simulate UART interrupts. The addition of one line to the `buttons.c` file supplied as part of the platform example (in the `src` folder of the BSP project) will ensure that `comm_impl_event` is called each time a button is released.

```
static void BUTTONS_event(Button_TypeDef Button) {
    ...
    if (button_pressed) {
        if (!BUTTON_PRESSED[Button])    {
            ...
        }
    } else {
        // button is released
        if (BUTTON_PRESSED[Button]) {
            // button was previously pressed, so this is a release event
            if (LLINPUT_sendButtonReleasedEvent(MICROUI_EVENTGEN_BUTTONS, Button)) {
                // the event has been queued: we can store the new button state
                BUTTON_PRESSED[Button] = MICROEJ_FALSE;
ADDED-->        comm_impl_event(Button);
            } else {
                ...
            }
        }
    }
}
```

Figure 3.5. Modification to buttons.c

The parameter is the button number; on the STM3240G-EVAL this will be 0 for the **Wakeup** button, 1 for the **Tamper** button, and 2 for the **Key** button. The `comm_impl_event` function ignores any button except **Wakeup**.

3.3 Configure the connections

- Copy the supplied `ecom_comm.c` file into the `src` folder of your BSP project.

The driver we have implemented can support two connections, with port ids of 0 and 1. We must now provide information to the stack about those connections. This configuration information could be in the driver source file, but it is conventional to keep it separate. This separation is more convenient when there is more than one LLCOMM implementation.

```
#include "LLCOMM_UART.h"
#include <stdint.h>
#include "LLCOMM_impl.h"
#include "microej.h"

// IO structs
static struct LLCOMM_UART conn0;
static struct LLCOMM_UART conn1;

LLCOMM_UART* createLLCOMM_UART(LLCOMM_UART* LLCOMM_UART, int32_t platformId,
 EVAL_COM_TypeDef comIndex)
{
 LLCOMM_UART_new(LLCOMM_UART);
 LLCOMM_UART->platformId = platformId;
 LLCOMM_UART->comIndex = comIndex;
 LLCOMM_UART->txItEnabled = MICROEJ_FALSE;
 return LLCOMM_UART;
}

void LLCOMM_IMPL_initialize(void)
{
 LLCOMM_addConnection((LLCOMM_CONNECTION*)createLLCOMM_UART(&conn0, 0, UART0_INDEX));
 LLCOMM_addConnection((LLCOMM_CONNECTION*)createLLCOMM_UART(&conn1, 1, UART1_INDEX));
}
```

Figure 3.6. Connection configuration

The link between connections and the stack is via the `LLCOMM_IMPL_initialize` method. This function is the first function called by the stack. Connections, seen as a data structure, can be added to a pool of `LLCOMM_CONNECTION` thanks to the `LLCOMM_addConnection` method.

The declarations at the top of the file (`conn0` and `conn1`) reserve space for the connection data structures. As we saw earlier, the format of the structure is defined in the driver header file.

The remaining code ensures that the data structures are correctly initialized, with the appropriate platform port id and com index set in each.

3.4 Create a Java application to test the driver

A test application is supplied as part of the `EcomComm-example` project, which you should have already imported into your workspace. The source is in `EcomCommExample.java`.

A launch configuration called `EcomCommExample EmbJPF`, supplied with the project, can be used to build the application. Check the launch configuration to ensure it has no errors (**Run** → **Run Configurations...**).

Ensure that the **Enable comm connections** option is checked and the application port number corresponds to the one used by the Java application (i.e. application port "1" for physical port "1").

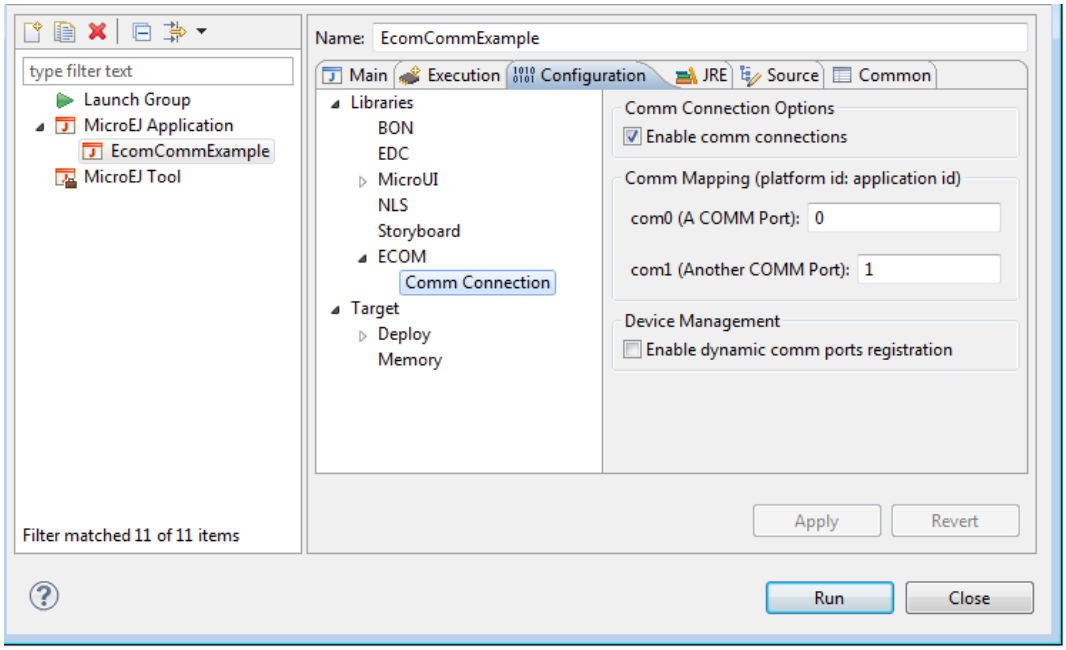

Figure 3.7. Enable comm connections option

Run the launch to create the `SOAR.o` file.

The key part of the java code is in the `launch` method, shown below :

```
private void launch(String connectionString) throws IOException {
    connection = openConnection(connectionString);

    out = openOutputStream(connection);

    out.write(OUTPUT_STRING.getBytes());
    showMessage("Written data");
    out.flush();
    showMessage("Flushed data");

    closeOutputStream(out);

    in = openInputStream(connection);

    int input;
    do {
        input = in.read();
        showMessage("Read data: " + new String(new byte[] {(byte) input}));
    } while (input != 'X');

    closeInputStream(in);

    closeConnection(connection);
}
```

Figure 3.8. Java application code

The connection string used is `"comm:com1"`. The first part of this string, `comm`, indicates that the required connection is an ECOM Comm connection. The second part, `"com1"` identifies the required port. The numeric part is used as the application port id.

Note how the input stream is read until an `'X'` character is received - the RX interrupt simulation in the driver sends an `'X'` as its last data value.

3.5 Update the BSP project and build the executable

Add the `ecom_comm.c` and `LLCOMM_UART.c` files, previously copied into the `src` folder of your BSP project, to the list of source files to be built.

In µVision, select the `MicroEJ` group, right-click, select **Add Files to Group 'MicroEJ'...** and navigate to the two files.

You should now be able to build and deploy the executable.

```
START
VM START
LLCOMM_IMPL_initialize
LLCOMM_BUFFERED_CONNECTION_IMPL_initialize
'conn0' connection added
LLCOMM_BUFFERED_CONNECTION_IMPL_initialize
'conn1' connection added
LLCOMM_BUFFERED_CONNECTION_IMPL_getPlatformId returns 0
LLCOMM_BUFFERED_CONNECTION_IMPL_getPlatformId returns 1
LLCOMM_BUFFERED_CONNECTION_IMPL_disableRXDeviceInterrupt
LLCOMM_BUFFERED_CONNECTION_IMPL_disableTXDeviceInterrupt
LLCOMM_BUFFERED_CONNECTION_IMPL_configureDevice
[LOG] Comm connection opened
LLCOMM_BUFFERED_CONNECTION_IMPL_getTXBufferStartAddress
LLCOMM_BUFFERED_CONNECTION_IMPL_getTXBufferLength
LLCOMM_BUFFERED_CONNECTION_IMPL_enableTX
[LOG] Output stream opened
LLCOMM_BUFFERED_CONNECTION_IMPL_useTXHardwareFIFO
LLCOMM_BUFFERED_CONNECTION_IMPL_enableTXDeviceInterrupt
[LOG] Written data
```

Figure 3.9. Expected output - part 1

When the application is executed you should see the output shown in Figure 3.9. At this point all the output data (3 characters) has been placed by the ECOM Comm stack into the connection's transmit buffer, and the stack is waiting to be notified by the interrupt handler that data can be written to the hardware. The Java application is blocked waiting for the `flush()` to complete.

You can simulate the interrupt by pressing the **Wakeup** button. Each time you press the button a character will be sent by the stack to the driver. After three presses all the data has been sent, the `flush()` completes and the Java application closes the output stream (Figure 3.10).

```
Simulating TX interrupt for port 1
LLCOMM_BUFFERED_CONNECTION_IMPL_sendData: 49
LLCOMM_BUFFERED_CONNECTION_IMPL_useTXHardwareFIFO
Simulating TX interrupt for port 1
LLCOMM_BUFFERED_CONNECTION_IMPL_sendData: 50
LLCOMM_BUFFERED_CONNECTION_IMPL_useTXHardwareFIFO
Simulating TX interrupt for port 1
LLCOMM_BUFFERED_CONNECTION_IMPL_sendData: 51
LLCOMM_BUFFERED_CONNECTION_IMPL_useTXHardwareFIFO
LLCOMM_BUFFERED_CONNECTION_IMPL_disableTXDeviceInterrupt
LLCOMM_BUFFERED_CONNECTION_IMPL_isTransmissionComplete
[LOG] Flushed data
LLCOMM_BUFFERED_CONNECTION_IMPL_disableTXDeviceInterrupt
LLCOMM_BUFFERED_CONNECTION_IMPL_disableTX
[LOG] Output stream closed
```

Figure 3.10. Expected output - part 2

The application now opens the input stream and reads data from it. It blocks until data is available. Pressing the **Wakeup** button now simulates the "data received" interrupt - each time it is pressed another character is sent to the stack, and to the application. On the third press the `'X'` character is sent, and the application closes the input stream and the connection (Figure 3.11).

```
LLCOMM_BUFFERED_CONNECTION_IMPL_getRXBufferStartAddress
LLCOMM_BUFFERED_CONNECTION_IMPL_getRXBufferLength
LLCOMM_BUFFERED_CONNECTION_IMPL_enableRX
LLCOMM_BUFFERED_CONNECTION_IMPL_enableRXDeviceInterrupt
[LOG] Input stream opened
LLCOMM_BUFFERED_CONNECTION_IMPL_disableRXDeviceInterrupt
LLCOMM_BUFFERED_CONNECTION_IMPL_enableRXDeviceInterrupt
Simulating RX interrupt for port 1
[LOG] Read data: A
LLCOMM_BUFFERED_CONNECTION_IMPL_disableRXDeviceInterrupt
LLCOMM_BUFFERED_CONNECTION_IMPL_enableRXDeviceInterrupt
Simulating RX interrupt for port 1
[LOG] Read data: B
LLCOMM_BUFFERED_CONNECTION_IMPL_disableRXDeviceInterrupt
LLCOMM_BUFFERED_CONNECTION_IMPL_enableRXDeviceInterrupt
Simulating RX interrupt for port 1
[LOG] Read data: X
LLCOMM_BUFFERED_CONNECTION_IMPL_disableRXDeviceInterrupt
LLCOMM_BUFFERED_CONNECTION_IMPL_disableRX
[LOG] Input stream closed.
[LOG] Comm connection closed
```

Figure 3.11. Expected output - part 3

4 Document History

Date	Revision	Description
September 26th, 2013	A	First release
July 31th, 2014	B	Product version 3.0.0 compatibility
October 02nd, 2014	C	Product version 3.1.0 compatibility

Headquarters
11, rue du chemin Rouge
44373 Nantes Cedex 3
FRANCE
Phone: +33 2 40 18 04 96
www.is2t.com