EE 590 Direct Research

IRIS Testbed – HDLC python implementation for communication between Pololu m3pi and OpenMote

Dennis Wang @ USC ANRG

12/04/2018

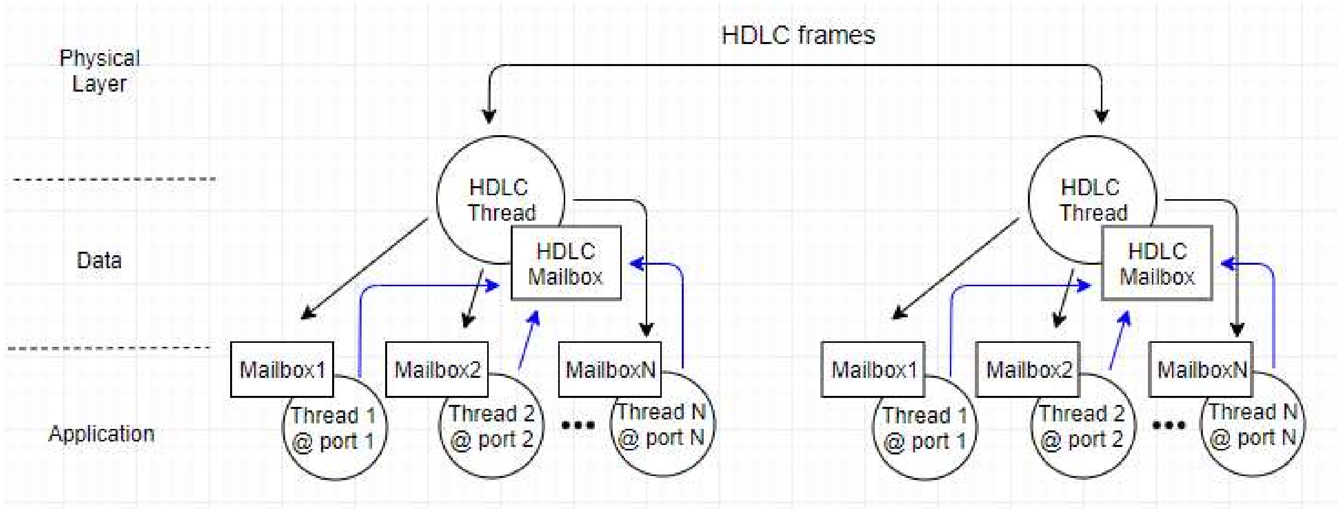
Overview:

In existing C implementation, a thread is dedicated to handle HDLC protocol communication between two devices. Applications that wish to transmit or receive data from the other devices will first register to a list with a port number; an application can only communicate to an application on the other device with the same port number. We can have multiple application threads registering on different ports to communicate with other devices. After registration, the application thread will use different types of messages to communicate with the HDLC thread to perform data transmission. Similarly, the HDLC thread will use another set of messages to notify with the application thread upon data reception. Detail will be discussed in HDLC & Application threads interaction section. The block diagram is shown below

Complication comes when we want to have multi-threading support. Mailboxes (queues) are employed to buffer communication between HDLC thread and application threads. Each application thread also has their own mailbox in case they have more data to send or not able to consume quickly than HDLC thread.

With multi-threading implementation, mailboxes become shared resources. HDLC mailbox is accessed by application threads and HDLC thread itself. Similarly, mailboxes for application threads are accessed by HDLC thread in addition to application threads. Thus, mutex is employed to avoid synchronization issue. In addition to mutex, synchronization between two devices also needs to be taken into account. If one end is sending, UART line will be busy, so the other side must wait.

Furthermore, although critical section of code will be entered when a mutex is acquired by a thread, it can't execute a blocking instruction that relies on the shared resources holding by the other thread. Otherwise, a deadlock can occur. For example, if a HDLC thread acquires the mutex on UART, but if its mailbox is empty, it must not hold the mutex and wait; it should release mutex and let other threads have a chance to run. Thus, current C implementation of the data transmission is in a busy-wait loop to check if UART is available and if there is any data to send to avoid deadlock. For data reception, hardware interrupt will be generated to notify receiving threads.



HDLC & Application threads interaction

Message is a data structure to keep track of the sender thread ID, mailbox of the sender thread, type of message, and a content field that could be used to send or receive from the other side. It is used for HDLC thread to interact with application threads.

Type of messages

Application level

Application thread will handle 3 types of messages in their mailboxes

1. HDLC_RESP_SND_SUCC:
   It is a message from the HDLC thread as a notification that a frame has been transmitted and an ACK is received. Application thread doesn't need to do anything further
2. HDLC_RESP_RETRY_W_TIMEO:
   It is a notification from HDLC thread that a write request failed due to UART is in use. Application thread will wait for a period of time and prepare a HDLC_MSG_SND message with the same data frame into HDLC thread's mailbox to be sent again.
3. HDLC_PKT_RDY:
   It is a notification from HDLC thread that new data has been received, and it is ready for the application thread.

HDLC level

HDLC thread will handle 4 types of messages in its mailbox

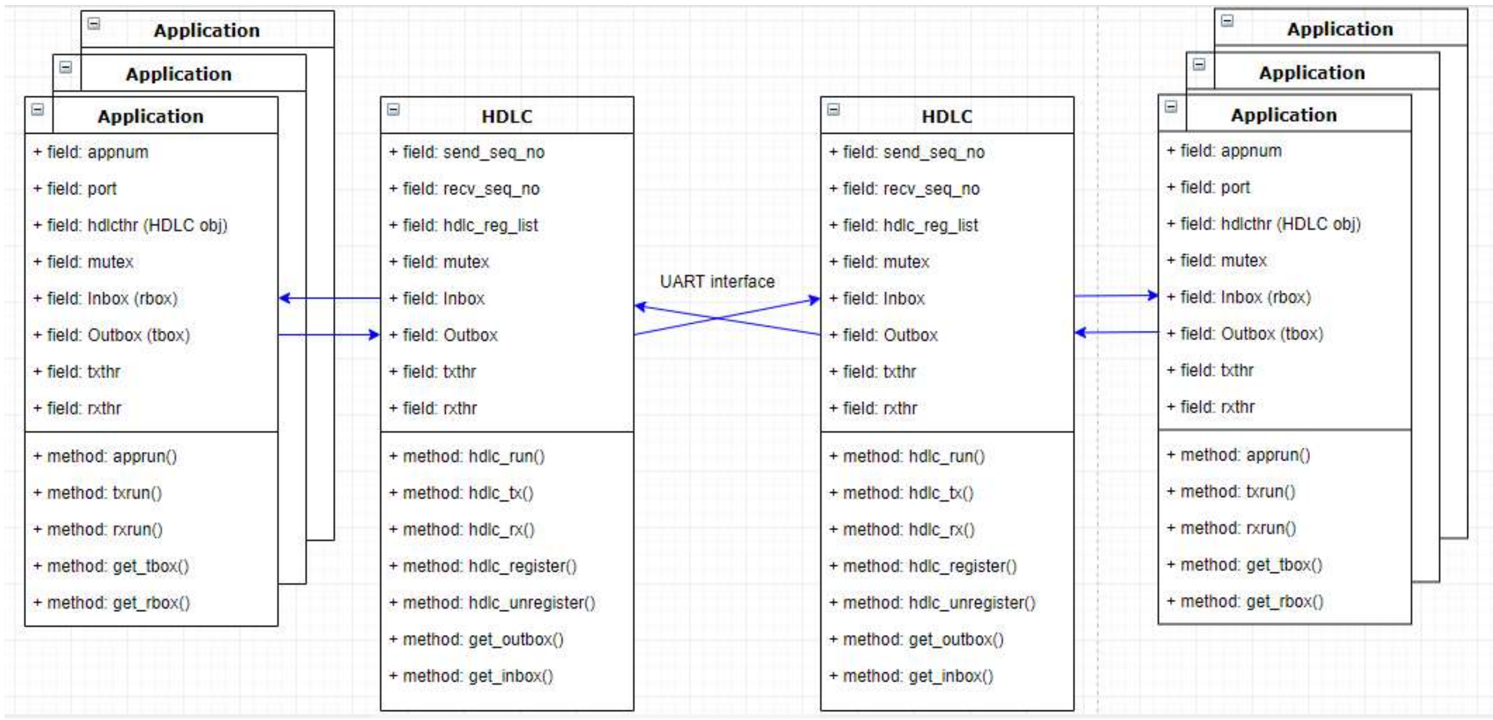
1. HDLC_MSG_RECV:
   Upon a HDLC frame reception, hardware interrupt will be triggered. An interrupt handler will encapsulate the frame with HDLC_MSG_RECV type, and will be placed in the HDLC mailbox. In short, it is a notification that new data frame has arrived from the other end.
   HDLC thread will validate the frame first. If there is no error, the frame could be either data or an ACK from the other end. The data frame and ACK frame can be determine from the HDLC protocol frame type and also from the data size; data size is zero for an ACK frame. If it is a data frame, HDLC thread will prepare a HDLC_MSG_SND_ACK message with the same sequence number in its own mailbox. At the same time, it will find the application thread registered on the port (the port info is embedded within the data) and prepare a HDLC_PKT_RDY message in the corresponding application thread's mailbox. On the other hand, if it is an ACK frame, it will notify the application thread (HDLC will keep track of who was the last sending application thread) with a HDLC_RESP_SND_SUCC message.
2. HDLC_MSG_SND:
   It is a notification that an application thread has a write request.
3. HDLC_MSG_SND_ACK:
   It is a notification that a data frame has been received, and HDLC thread will be responsible to return an ACK frame
4. HDLC_MSG_RESEND:
   When HDLC thread is about to transmit a frame (either data or ACK), but UART is currently busy, HDLC thread leaves a note to itself that this frame needs to be sent at a later time.

Porting to Python

The major difference between C and python is the serial interface library in C can setup an interrupt handler upon receiving frames. The next to best solution for Python is to use a polling loop to check if there is any data on UART line. The tight loop will make sure no data is lost. However, it may result in UART busy all the time. Other than that, Python is rich in package support and flexible for object-oriented programming. With this in mind, implementation is shifted toward object-oriented design. This will allow various number of applications to be registered to a HDLC object easily.

Design considerations

1. Originally a thread is dedicated to handle both transmission and reception. I am proposing to split tasks into RX and TX for HDLC thread and application threads.
   HDLC RX – will perform the task for HDLC_MSG_RECV type of message. Note: it is not necessary to create a HDLC_MSG_RECV message since there is no interrupt handler. However, HDLC RX thread will be in charge the rest of the HDLC_MSG_RECV task discussed in HDLC level section
   HDLC TX – will handle HDLC_MSG_SND, HDLC_MSG_RESEND, HDLC_MSG_SND_ACK types of message
   Application RX – will handle HDLC_PKT_RDY type of message
   Application TX – will handle HDLC_RESP_SND_SUCC, HDLC_RESP_RETRY_W_TIMEO types of message
2. In addition, each mailbox is splitted into two, inbox and outbox.
3. As a result, HDLC is turned into an object which will consist of TX, RX threads, inbox and outbox (with other bookkeeping data members such as a list of entries that registered to HDLC object, a mutex to maintain synchronization between threads and sequence numbers to keep track the frames being transmitted/received).
   Likewise, application is turned into an object as well. It consists of TX, RX threads, inbox and outbox (with additional bookkeeping data members such as port number, app number to check if a frame is for this application object, mutex and the HDLC object it is registering to).
   Inside each class, there will be methods to start the TX, RX threads. The block diagram is shown below.



Testing Plan

1. To avoid hardware issue that may arise with software bugs, it is recommended to fork two processes and a pipe to emulate two devices communicating over a UART line.
2. If the above step can be verified, we can test with two Raspberry Pis running the same code to gain more confident the python code can work with hardware
3. If the above step can be verified, we can run test with a Pololu m3Pi running this code and have an OpenMote running C implementation.

Current status

- HDLC and application classes are implemented
- Code is modified to perform test plan 1. Two processes try to communicate through a pipe. Each process has an HDLC object and two application objects. Current status seems to encounter a deadlock situation. It may be due to UART line is kept busy by the HDLC RX thread most of the time (as explained in Porting to Python section). Further investigation is required.
  Files are under iris-riot/examples/iris_testbed/tests/hdlc_txvr/python_hdlc
  hdlc.py – HDLC class
  run_app.py – Application class
  test_hdlc.py – Two processes with HDLC object and application objects try to communicate through a pipe.
  To run the test:
  python3 test_hdlc.py
- Update as of 01/18/19
  After further analysis, pipe is not ideal to emulate the communication because of two reasons. Pipe has limited built-in read/write support. First, read/write is a blocking call. Second, read/write expects a string as an input; however, messages are prepared in bytes to be sent over HDLC. Thus, another conversion has to be done when sending/receiving through pipe.
  Instead of using pipe to test, we decided to set up a virtual serial bus to run the test. Communication can be successfully emulated; two devices each having two applications threads can send and receive with other side.

- To test with virtual serial bus:
  - `socat -d -d pty,raw,echo=0 pty,raw,echo=0`
    This command will create two virtual devices such as `/dev/pts/5` and `/dev/pts/6`. Everything you write in `/dev/pts/5` will be echoed in `/dev/pts/6` and vice versa. One device can ack as TX line while the other device can ack as RX line.
  - We can then fork a process. Both child and parent processes will have access to TX and RX devices, and we just need to connect the TX of a process to the RX of another process to begin communication. May need to update the virtual device numbers in test_hdlc.py (using the example above, it will be 5 and 6)
  - Run the test as usual:
    python3 test_hdlc.py

- Note: Both python and C implementations of the HDLC thread can only transmit one packet for an application thread until an ACK is received. It is similar to Stop-and-wait ARQ. If the receiver side is slow in response, we can lose some efficiency. However, it guarantees that information is not lost due to dropped packets and that packets are received in the correct order