1. Introduction

1.1 What is HDLC protocol

HDLC protocol is a general purpose protocol which operates at the data link layer of the OSI reference model. Since it is a layer 2 protocol, it uses the service of physical layer for obtaining data and establishing the communication. It depends upon the type of HDLC protocol that we use that whether the physical layer provides best effort service or reliable service with acknowledged data transfer. HDLC is one of the most commonly used protocol at layer 2. HDLC is a reliable protocol that provides us with full duplex communication and flow control facility.

Each piece of data is encapsulated into an HDLC frame with both header and a trailer. The header contains an HDLC address and HDLC control field. The trailer contains a Cyclic Redundancy Check (CRC) which detects if any errors have occurred during the transmission.

The HDLC address field is usually a single byte long. The address format is shown in figure 1. A Service Access Point (SAP) is usually set to zero, but used in some variants of HDLC to identify one of a number of data link protocol entities. A command/response bit is used to indicate whether the frame relates to information frames being sent from the node or received by the node.

Fig.1 Format of Address Byte(s) in HDLC

The frames are separated by HDLC flag sequences which are transmitted between each frame and whenever there is no data to be transmitted.

1.2 Why to use HDLC

HDLC is a protocol developed by International Organisation for Standardisation (ISO). It is used widely all across the world because it supports both half duplex and full duplex communication lines, peer to peer and multipoint networks, and switched and non switched channels. The procedures defined in HDLC are designed to permit synchronous, code transparent data transmission.

1.3 HDLC operation modes

HDLC offers three different modes of operation. These modes of operation are

(a) Normal Response Mode: This is the mode in which primary station initiates transfer to secondary station. The secondary station can initiate only a response when, and only when it is instructed to do so. After receiving permission from primary station, the secondary station initiates it's transmission. This transmission from secondary station to the primary station may be much more than just an acknowledgement of a frame. It may be more than one information frame. Once the last frame is transmitted by the secondary station, it must wait once again for explicit permission to transfer anything from the primary station. Normal Response mode is used only within an unbalanced configuration.

(b) Asynchronous Response Mode: In this mode primary station does not initiate transfers to the secondary station. In fact, the secondary station does not have to wait to receive explicit permission from the primary station to transfer any frames. However some limitations do exist. Due to the fact that this mode is Asynchronous, the secondary station must wait until it detects an idle channel, before it can transfer any frames. This is when the link is operating at half-duplex. If the link is operating at full-duplex, the secondary station can transmit at any time.

(c) Asynchronous Balanced Mode: This mode uses combined stations. There is no need for permission on any part of any station in this mode. This is because the combined stations do not require any sort of instructions to perform any task on the link.

Normal Response Mode is used most frequently in multi-point lines, where the primary station controls the link. Asynchronous Response Mode is better for point to point links, as it reduces overhead. Asynchronous Balanced is not used widely.

1.4 HDLC Frame Structure

Since it is a data link layer protocol, HDLC uses term 'frame' to indicate an entity of data transmitted from one station to another. The figure 2 below shows different components of an HDLC frame:

Fig.2 HDLC Frame

Field Name	Size
Flag Field (F)	8 bits
Address Field(A)	8 bits
Control Field (C)	8 or 16 bits
Information Field (I)	Variable, not used usually
Frame Check Sequence (FCS)	16 or 32 bits
Closing Flag Field (F)	8 bits

Table 1 HDLC Frame components

2. Current Implementation of HDLC Protocol

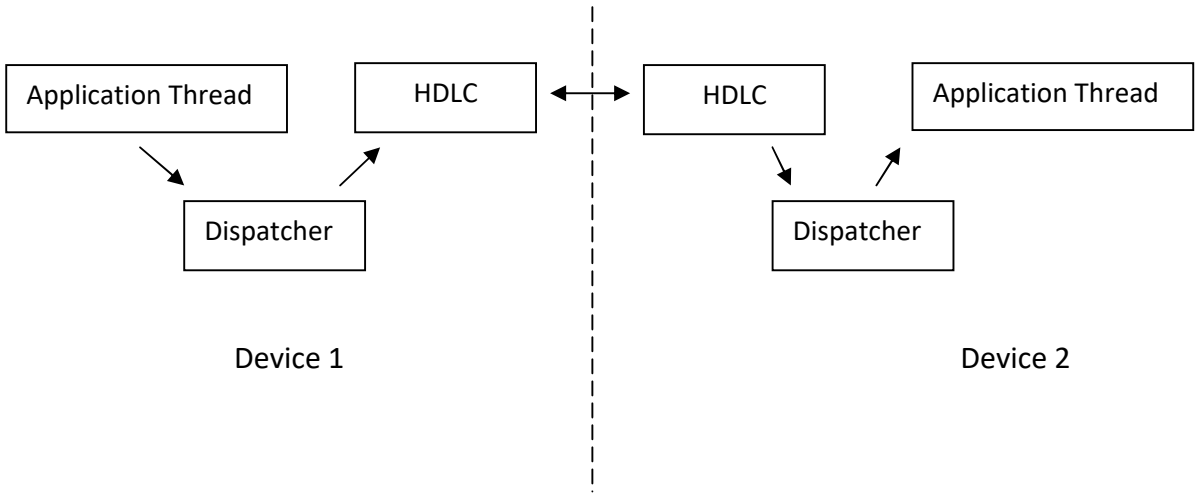

Fig.3 Working of HDLC protocol

Figure 3 above show the current implementation of HDLC protocol. In the newer more stable version, HDLC and dispatcher are merged together for simplicity. The dispatcher has buffer associated with it to store the data if the rate at which HDLC thread is transmitting data is less than the rate at which application thread is supplying the data.

2.1 Dispatcher

To implement the functionalities of dispatcher, we use functions like dispatcher_init(), get_dispatcher_mailbox(), dispatcher_register() and dispatcher_unregister().

Dispatcher_init() initialises the dispatcher and returns the mailbox address of initialised dispatcher. Data in dispatcher is stored using data structure mailbox. Mailbox is used in multitasking C programs. Mailboxes provide a means of passing messages between tasks for data exchange or task synchronisation. MAILBOX is a typedef defined in mtasker.h . It declares and allocates memory in common RAM for a 32-bit variable. To create a Mailbox called temp, execute MAILBOX temp = 0; It is important to explicitly initialise the resource variable to contain zero as part of the application program's initialization after each startup. Failure to do so could leave garbage in the mailbox that precludes any task from writing valid data into the mailbox. After initialising the mailbox, only the routines like TRY_TO_SEND, SEND, RECEIVE etc should be used to modify the contents of the mailbox.

The get_dispatcher_mailbox function returns current mailbox of the dispatcher. Register and unregister functions are used to delete and prepend an entry into the dispatcher.

2.2 HDLC Thread

The hdlc thread uses different functions of file hdlc.c for it's functionality. Since HDLC thread reads as well as sends the data, functions for both reading and writing data onto the serial pins are implemented. Function like rx_cb() is used to read any available data. After reading the data it is stored in a data structure called circular buffer. The term circular buffer refers to an area of memory which is used to store the incoming data. When the buffer is filled, new data is written starting at the beginning of the buffer and overwriting the old. If the data is in correct format then it is stored in the hdlc mailbox. Routine _hdlc_receive is used to continuously listen the serial pins for any available data. If the data is in correct format, it returns otherwise continues listening. Similar to dispatcher, there are functions like get_hdlc_malbox used to return the mailbox associated with hdlc. write_hdlc routine is used to send data onto the serial pins. hdlc_nit is used to initialise a mailbox for current hdlc thread.

3. Thread Implementation on Intel Edison

Since Yocto runs inside Intel Edison, it can support programming in any language like C, C++ and Java. Hence the thread implementation is pretty much the same way as for a general computer. we discuss an example of thread implementation in python.

Figure 4 shows a very basic implementation of threads in python for Intel Edison. We define a function called worker which would be called whenever the thread starts. We create a thread called t and start it. On the standard output console, we can see "worker" printed again and again.

```
import threading

def worker():
    """thread worker function"""
    print 'Worker'
    return

threads = []
for i in range(5):
    t = threading.Thread(target=worker)
    threads.append(t)
    t.start()
```

Fig.4 Threads for Edison

4. Serial Communication for Intel Edison

For serial communication on Intel Edison library libmraa is used. Libmraa is a C/C++ library with bindings to Java, Python and JavaScript to interface with IO on Galileo, Edison & other platforms, with a structured API where port name/numbering matches the board that you are on. The use of libmraa does not tie you to a specific hardware with board detection done at runtime. You can create portable code that will work across the supported platforms. The libmraa is open source (https://github.com/intel-iot-devkit/mraa) and has over 400 fork requests. It is most widely used library on Intel devices.