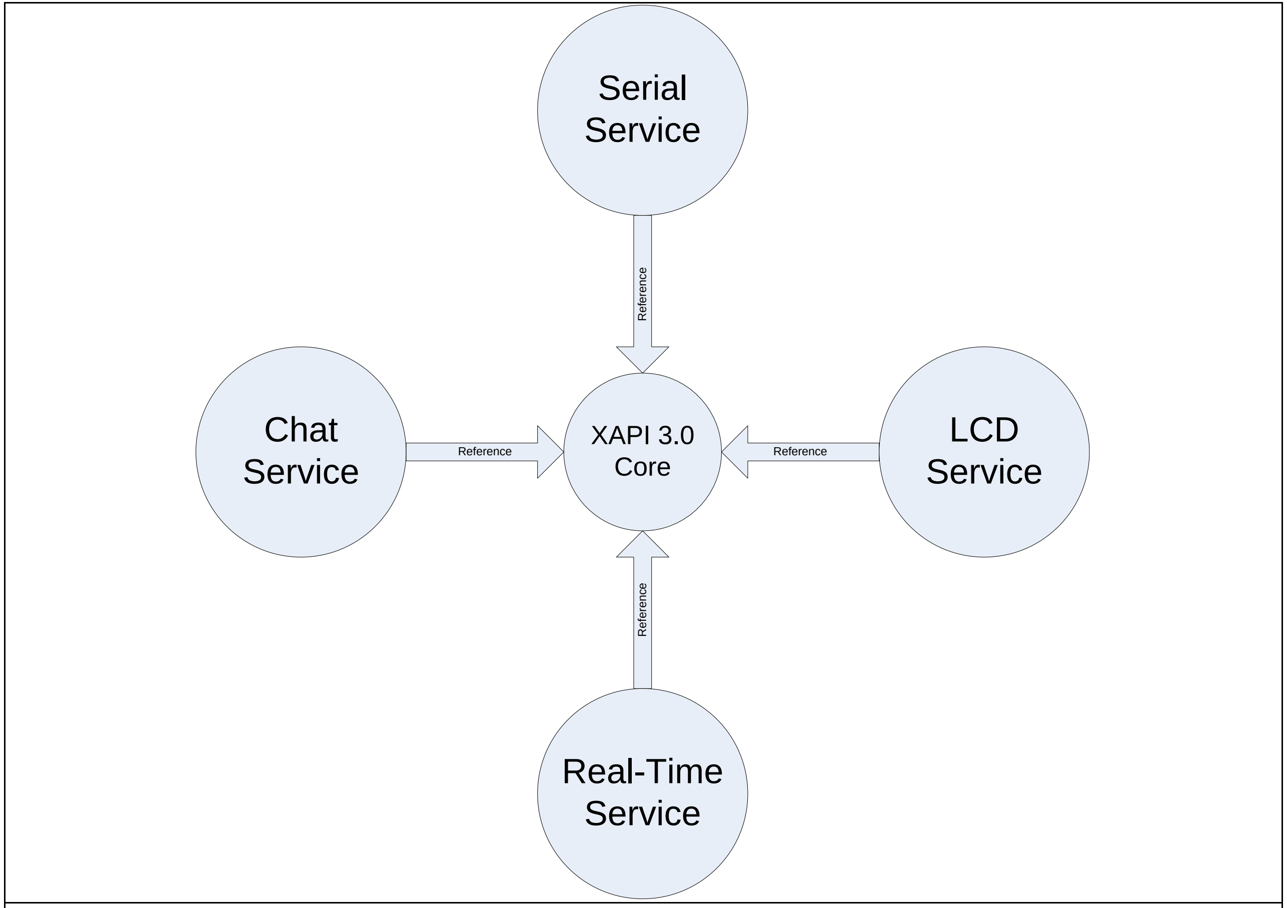
Architecture Overview

The XAPI core is deliberately small and simple. Only basic services are offered to the user of this API which include the transmission and receiving of external and local TUN packets via XBee hardware. Technically external SYS packets are also buffered in the XAPI, but as of version 3.0, external SYS packets are not processed in any significant manor. TUN packets, on the other hand, are sorted into one of two buffers: “m_external_TUN_single_buff” for incoming external TUN packets, and “m_local_TUN_single_buff” for internal, cooperative communication between active services. Both of these buffers have public helper functions which the services must use in order to query the buffers and extract the TUN packet if it belongs to them.

Even without services, the bare XAPI is still useful to developers who need easy access to XBee hardware. The bare XAPI can transmit TUN packets because the XAPI has a simple service built in: the construction and transmission of a “ZigBee Transmit Request” packet. It is within the payload section of this packet that a TUN packet is stuffed into and transmitted. Developers will generally find this service inadequate and will need to develop more specialized services. Before the construction of additional services is covered, it's best to first visualize the XAPI system to see how services “snap-in” to the XAPI core. Below is an image of an XAPI core and several common services:



A graphical representation of the XAPI system. In this drawing, the XAPI core is partnered with four independent services. Each service does a single job and uses the XAPI core to transmit and receive both external and local TUN packets. Each service has a “reference” to the XAPI which is used to perform calls on XAPI routines.

Example Usage

The XAPI core is not much without services. Therefore, this section will illustrate the various ways that a service can use the XAPI core to communicate to other external and local services. The easiest usage of the XAPI core is to transmit a TUN packet to an external service. To illustrate how this is done, actual code from the LCD_service will be examined.

```
//*************************************************
//*************************************************
// This routine simplifies the process of sending
// a message from one LCD to another via the
// XBee modules. Consider using this routine
// instead of trying to send a message using
// all of the individual steps.
// Incoming:
//          addrLSB: The lower-half 32-bit of the address.
//          x: Column of the LCD panel (values 0-15)
//          y: Row of the LCD panel (values 0-1)
//          msg: the actual message in standard
//      c-type string with null termination.
// Returns:
//   The size of the complete packet to be sent
uint8_t LCD_service::lcd_snd_EXTERNAL_message(   const uint32_t addrMSB,
                                                  const uint32_t addrLSB,
                                                  const uint16_t addr16,
                                                  const uint8_t x,
                                                  const uint8_t y,
                                                  const uint8_t* msg)
{
        uint8_t payload_buff_sz = 0;
        uint8_t TUN_buff_sz = 0;
        uint8_t Xbee_buff_sz = 0;

        uint8_t payload_buff[LARGE_BUFF_SZ];
        uint8_t TUN_buff[LARGE_BUFF_SZ];
        uint8_t Xbee_buff[LARGE_BUFF_SZ];

        // produce the following:
        // [X:2][Y:2][STRING] = [PAYLOAD]
        payload_buff_sz = m_util.construct_lcd_payload(x, y, msg, strlen((const char*)msg),
                                payload_buff, LARGE_BUFF_SZ);

        // produce the following (a TUN packet):
        // $[TYPE:2][PAYLOAD_SZ:2][CHECKSUM:4][PAYLOAD]%
        TUN_buff_sz = m_util.create_TUN_packet( TUN_TYPE_EXTERNAL_LCD_MSG,  payload_buff,
                                payload_buff_sz, TUN_buff, LARGE_BUFF_SZ);

        // produce the following (a complete Xbee packet)
        // [PREAMBLE][%TUN_PACKET%][CHECKSUM]
        Xbee_buff_sz = m_xapi.construct_transmit_req( addrMSB, addrLSB, addr16,TUN_buff,
                                                  TUN_buff_sz, Xbee_buff, LARGE_BUFF_SZ);

        // Physically ship out completed Xbee packet over radio
```

```
		m_xapi.snd_packet(Xbee_buff, Xbee_buff_sz);

		return Xbee_buff_sz;
}
```

Figure $0x01: C++ code snippet from the LCD_service class. This code snippet sends out a plain-text string message to be displayed on an external LCD which is being driven by a LCD_service on a remote XAPI enabled device. Usage of the Util class to simplify the creation of TUN packets and other code intensive tasks is also demonstrated.

As can be seen in figure $0x01, the LCD_service only needs four lines of code to send a TUN packet to an external service. The four critical lines of code are in bold. Before the lines of code are described in detail, the Util class must be mentioned. The Util class is an attempt to collect code which is useful to many services. Usage of the Util class across multiple services should cut down on the introduction of new bugs and aid in the correction of old bugs. Instead of attempting to write new code to do such things as integer to hex conversions, creation of TUN packets, checksum calculations, and so forth, it's best to see if that code isn't already available in the Util class. If it isn't and the new code could potentially be useful to multiple other services, consider adding it to the Util class. The instantiation of the Util class in this service is "m_util." The "m_" simply means it is a member of the class in which it is being instantiated in. This naming convention is seen throughout the XAPI.

The first bolded line of code constructs the payload section of the TUN packet. There is no "standard" payload across the services. Rather, each service can define their own payload structures. One of the basic payloads is simply text without any formatting. If this is the case, the first line of code would not be needed as the payload is already formatted. In the case of the LCD_service, the m_util function "create_TUN_packet(…)" is used. This function will convert the x and y values to ASCII hex and merge the result with the text string to be displayed on the LCD. Since any service can request to have a string shown on the LCD, it was decided that create_TUN_packet(…) should reside in the Util class rather than being in the LCD_service class. When complete, this function will modify the "payload_buff" buffer and return the size of the modified payload_buff buffer. If x=0x01, y=0x02, and str="HELLO_WORLD", the payload_buff will contain the ASCII text "0102HELLO_WORLD" and payload_buff_sz would equal 15. Note that the max amount of characters that payload_buff size can hold would be LARGE_BUFF_SZ which at the time of this writing is of value 70.

The second bolded line of code takes the buffer which was created previously and wraps it in a complete TUN packet. The format of the TUN packet is standardized to contain the following: The first two bytes are the ASCII hex number of the type. The second two bytes is the payload size. The next four bytes is the checksum, and finally, the remaining bytes are the payload itself. A full description of the TUN packet is in section $0x04. All services will need to make use of the "create_TUN_packet" routine in the Util class in order to perform any sort of communication with the external world or other local services. In this snipped of code, the completed TUN packet will be stored within the "TUN_buff" buffer. The new size of this buffer will be returned in "TUN_buff_sz." The TUN packet is now technically complete and can be sent locally to another service. Since this TUN packet is intended to be sent over the radio to an external service, it still needs to be wrapped in the XBee system packet 0x10, "ZigBee Transmit Request" packet.

The third bolded line wraps the completed TUN packet into a ZigBee Transmit Request packet.

The TUN packet now resides within the payload section of the Transmit Request packet. This step is necessary as the XBee hardware only understands packets which are formatted in a specific way. The "XBee Module Manual" contains a listing of all possible native packet formats if the reader is interested in learning more about them as they are generally complex and difficult to describe properly. It should be noted the routine which encapsulates the TUN packet into a SYS packet is provided by the XAPI itself. Providing the formatting for this type of packet is the only SYS packet the XAPI provides. If the developer needs additional types of SYS packets, they will have to write new services to address those needs. The completed SYS packet is written to the "Xbee_buff" buffer and the size of the buffer is stored in "Xbee_buff_sz." The final bolded line ships out the packet over radio. Again, this functionality is provided by the XAPI and not the Util class.

In summary, the developer only needs four lines of code to transmit a TUN packet over the ZigBee protocol via XBee radio hardware. This ease of development is provided by two classes: The Util class for various often-used routines, and the XAPI itself. The next section will describe how the XAPI allows services to communicate to other local services. In this situation, the radio hardware is not used. Rather, services directly connected to each other via wires and other electronics can coordinate and solve more complex problems using various features of the XAPI. The coordination of the LCD_service and the Serial_service will be examined in such a way that development of new services using local communications should be relatively easily done.

Example Usage: Local Communications

The pre-built LCD_service and Serial_service will be used in this section to demonstrate local communications between services. Before local communications are described in detail, the internal XAPI buffer utilized for local communications will be described.

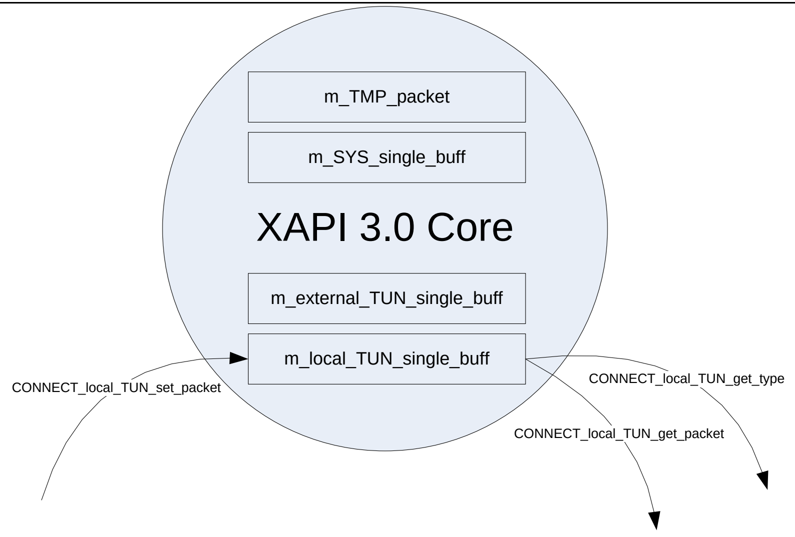

m_TMP_packet
m_SYS_single_buff
XAPI 3.0 Core
m_external_TUN_single_buff
m_local_TUN_single_buff
CONNECT_local_TUN_set_packet
CONNECT_local_TUN_get_type
CONNECT_local_TUN_get_packet

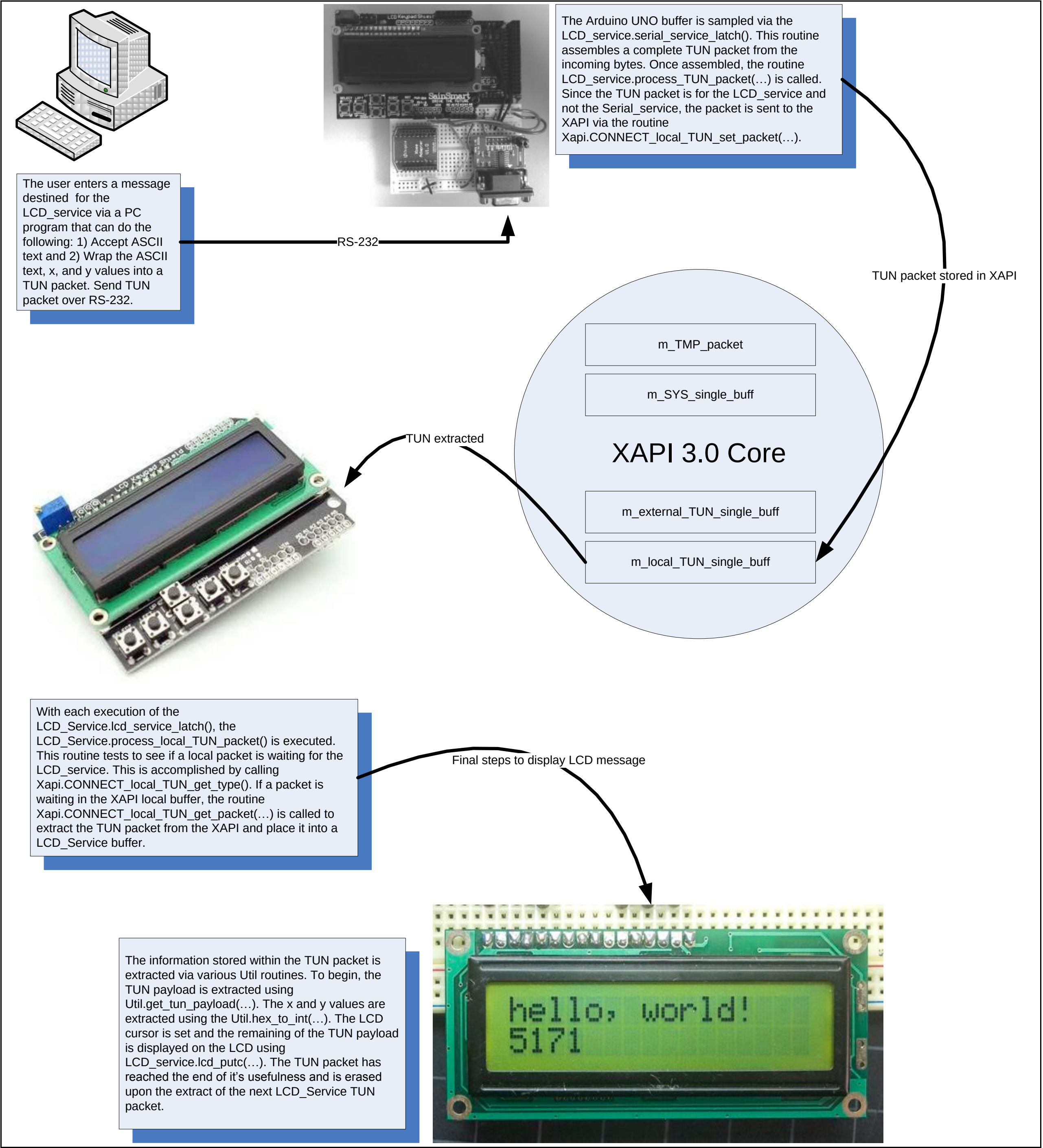
Figure $0x05: The various buffers which allow for local and external communications. The bottom two buffers operate solely on TUN packets. The top two buffers operate on SYS packets. Most services will deal exclusively with the bottom two buffers, the "m_local_TUN_single_buff" and "m_external_TUN_single_buff" buffers.

The XAPI has four buffers which allow for smooth packet storage and extraction. The bottom two buffers, “m_local_TUN_single_buff” and “m_external_TUN_single_buff” buffers are intended to be used heavily by the services. The m_local_TUN_single_buff allows for local messages to be passed between the services and as such will be the focus of this section.

Communications between local services is exclusively TUN packet based. If the reader is unclear about the internals of the TUN packet, please review section $0x11. The process of local communications between the Serial_service and the LCD_Service is as follows:

1) The user types a message into a PC terminal which is connected over RS-232 to the Serial_service.
2) Before the message is transmitted to the Serial_service, the message must be placed into the payload section of a TUN packet.
4) The TUN packet is transferred to the Serial_service over RS-232 from the PC.
5) The Serial_service receives the packet and realizes it is not the intended target of the TUN packet via the TUN type number stored within the packet.
6) The TUN packet is stored within the XAPI buffer m_local_TUN_single_buff via the XAPI function CONNECT_local_TUN_set_packet.
7) The LCD_service eventually queries the local XAPI buffer via the CONNECT_local_TUN_get_type function.
8) The LCD_service realizes the packet belongs to it and extracts it using the XAPI function CONNECT_local_TUN_get_packet.
9) The LCD_Service extracts the payload from the TUN packet.
10) Stored in the TUN packet are the following fields: X, Y, and a string. The string is then shown on the LCD using the X and Y as LCD position coordinates.

Below is a different view of the described process. The LCD TUN packet is traced from the PC all the way till it is finally consumed by the LCD_service. All of the major routines which touch the TUN packet are listed.



The user enters a message destined for the LCD_service via a PC program that can do the following: 1) Accept ASCII text and 2) Wrap the ASCII text, x, and y values into a TUN packet. Send TUN packet over RS-232.

The Arduino UNO buffer is sampled via the LCD_service.serial_service_latch(). This routine assembles a complete TUN packet from the incoming bytes. Once assembled, the routine LCD_service.process_TUN_packet(…) is called. Since the TUN packet is for the LCD_service and not the Serial_service, the packet is sent to the XAPI via the routine Xapi.CONNECT_local_TUN_set_packet(…).

With each execution of the LCD_Service.lcd_service_latch(), the LCD_Service.process_local_TUN_packet() is executed. This routine tests to see if a local packet is waiting for the LCD_service. This is accomplished by calling Xapi.CONNECT_local_TUN_get_type(). If a packet is waiting in the XAPI local buffer, the routine Xapi.CONNECT_local_TUN_get_packet(…) is called to extract the TUN packet from the XAPI and place it into a LCD_Service buffer.

The information stored within the TUN packet is extracted via various Util routines. To begin, the TUN payload is extracted using Util.get_tun_payload(…). The x and y values are extracted using the Util.hex_to_int(…). The LCD cursor is set and the remaining of the TUN payload is displayed on the LCD using LCD_service.lcd_putc(…). The TUN packet has reached the end of it's usefulness and is erased upon the extract of the next LCD_Service TUN packet.

Figure $0xAA: Trace of a local LCD TUN packet from PC to LCD_Service. The many steps and major routines called are described in this illustration. This trace can easily be generalized to allow for other services to send and receive local TUN packets.

TUN Packet Description

The TUN packet is the only means of communication between both local and external services. The TUN packet frame is small and the payload is technically limited to 255 bytes max. In a normal execution, the payload should be limited to 40-55 bytes max. This is because the buffer space allocated within the Arduino tends to be around 70 bytes for both the TUN frame and payload. A buffer of 255+ bytes within the Arduino will quickly drain the available SRAM of the micro-controller. Large TUN packets are thus infeasible.

There are other reasons why the developer should keep their TUN packets small. First off, small TUN packets are faster than large TUN packets during buffer copies and transmissions over RS-232. Secondly, local TUN packets will undergo several buffer copying episodes. The smaller the TUN packet, the quicker buffer copying will commence and complete. Lastly, the smaller the TUN packet, the less resources are needed to process the packet. Keeping the TUN packet small will help avoid excessive memory drains and buffer overruns. In summary, if the developer needs to send large amounts of data via the TUN packet, it's safer and less stressful to the Arduino to use multiple small TUN packets rather than one large single TUN packet.

The name "TUN" may seem odd but it makes sense when one learns of this history of this packet. Originally, only XBee API system packets (SYS) were used. These packets proved difficult to properly capture due to the lack of a complete "frame." There is no sentinel value at the trailing end of the SYS packet which makes it almost impossible to know when the packet is completely captured. To remedy this problem, a tunneled packet is used. The TUNneled packet has a complete frame surrounding it. There is a beginning sentinel byte ad an ending sentinel byte. When placed inside of the payload section of the SYS packet, the TUN packet can easily be filtered out of the SYS packet and processed. Since this packet has proven successful and relatively easy to manage, it has been re-purposed for local communications between services. Below is an illustration of the TUN packet format. The frame which makes the packet particularly easy to manage is highlighted along with sentinel bytes and the payload size.

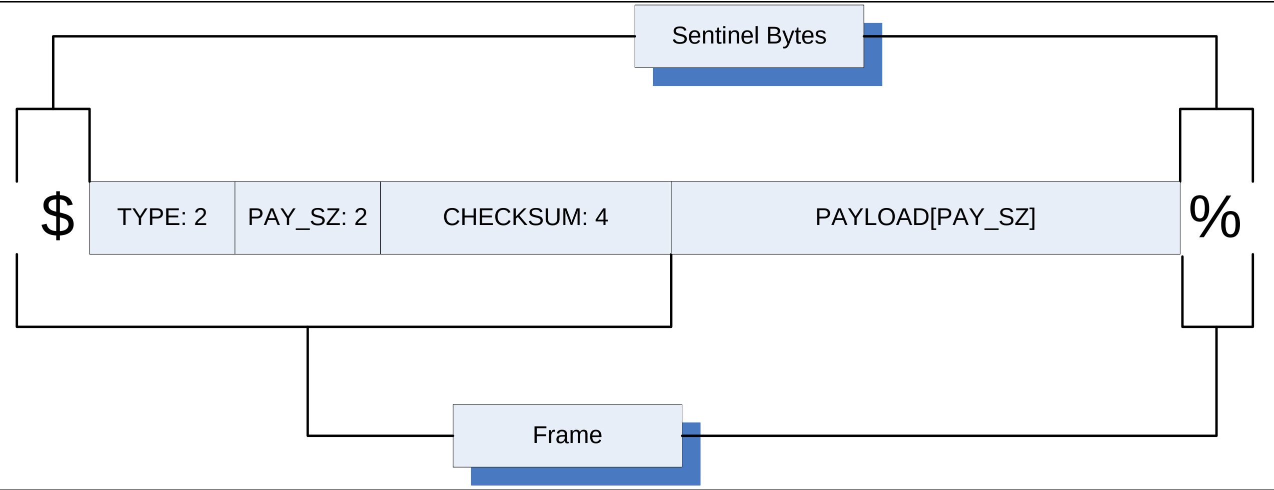


Figure $0x11: The TUN packet format. This illustration shows the how the frame encompasses the payload and how the sentinel bytes mark the beginning and end of the packet. The checksum section of the packet allows for the receiver to determine if the packet arrived without error. The numbers following the title of the section is the number of bytes that section takes. The TYPE and PAY_SZ sections, for example, are 2 ASCII-hex bytes. The CHECKSUM section is 4 ASCII-hex bytes. The PAYLOAD section is PAY_SZ bytes long.

TUN Packet Construction

The Util class offers many routines to aid the developer in using TUN packets. The most generic routine is Util.create_TUN_packet(…). The developer simply calls this routine with the necessary arguments and a buffer large enough to contain an entire complete TUN packet. The routine will fill the empty buffer with the complete packet and return the size of the packet to the caller. This generic routine is only sufficient if the payload doesn't have a specific format. If the TUN packet has a complex payload with a specific format, a routine such as Util.create_TUN_lcd_packet(…). This routine will allow the user to easily create a TUN packet in which the payload is formatted correctly for the LCD_Service to process. Bellow is an illustration of a TUN packet which has a formatted payload destined for the LCD_Service:

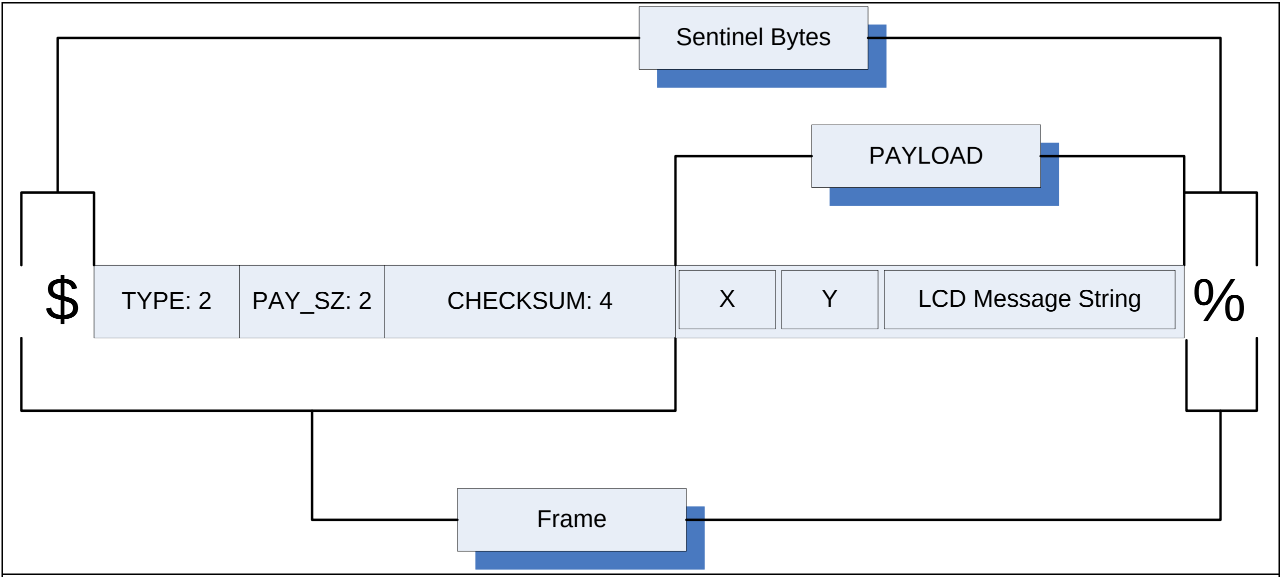


Figure $0x1a: A LCD_Service formatted TUN packet. The payload of this packet has a particular format: The first two ASCII-hex bytes is the X location on the LCD panel. The second two ASCII-hex bytes are the Y location on the LCD panel. The final bytes are the actual human-readable text message which will be displayed on the LCD panel.

It may be strange that the LCD_Service TUN payload format is located in the Util class and no the LCD_Service. The reason for this is easy to understand: Other services can make use of the LCD_Service only if they have access to the packet format. Placing all of the TUN packet formats in the Util class makes it easy for both local and external communications between services.

External LCD TUN Trace

This section will provide the reader with a complete trace of an external LCD_Service TUN packet. The sender of the packet is generic and could be any service. This is an attempt to show that communications between any two services can be simple if some basic steps are followed. Also, this trace should further imprint upon the reader the importance that common functionality between different services can be provided via the Util class.

To begin, please refer to Figure $0x01 and the text explaining Figure $0x01. While this code is clearly contained within the LCD_Service class, it could be any class. There is no code within the

LCD_Service.lcd_snd_EXTERNAL_message(…) routine that is strictly located only within the LCD_Service class. The critical code which allows for the construction and transmission of the message comes from other classes. The first two critical lines of code (bolded lines) comes from the Util class (m_util). The next two critical lines of code (bolded lines) comes from the XAPI class (m_xapi) itself. Any service can follow Figure $0x01 to send external messages to other services. If the target service requires a specially formatted TUN packet payload, the m_util.construct_lcd_payload(…) routine can be replaced with a routine that performs the correct payload formatting. It is suggested that any new routine which performs payload formatting is located within the Util class.

It needs to be made clear that the TUN packet is placed within the payload of the SYS packet prior to going airborne. The receiver must strip away the SYS packet to expose the embedded TUN packet. This process is automatically done by the XAPI in the routine Xapi.TUN_filter_packet(…). When a complete TUN packet has been filtered out of the SYS payload, the routine m_external_TUN_single_buff.add_TUN_buffer(…) is called. This routine stores the completed TUN packet in a buffer where it will be waiting to be extracted by a service.

The process and routines used to extract an external TUN packet by a service are nearly identical to the extraction of local TUN packets. The first step is the latch. The LCD_Service.serial_service_latch() is executed in a loop in the Arduino loop() routine. On each iteration of the loop, the external TUN buffer is queried via the routine Xapi.CONNECT_external_TUN_get_type(). If the returned type matches the type the LCD_Service is interested in, the TUN packet is extracted from the XAPI with the routine Xapi.CONNECT_external_TUN_get_packet(…). Once extracted, the external TUN packet is processed within the LCD_Service.display_TUN_packet(…). This routine basically extracts the payload from the TUN packet and processes the information stored within to display a string on the LCD panel.

Class Descriptions

This section contains a summarized description of all XAPI classes and demonstration services.

Xapi.cpp/Xapi.h: The XAPI core. This class manages all communications both local and external. This class is intended to be small with only minimal features. Additional features are obtained with the addition of new services.

Util.cpp/Util.h: This class contains routines which are intended to be shared between all services and the XAPI itself. This class helps to avoid the introduction of bugs when new services are created or new features are added to the XAPI core by re-using well-tested code.

Universal.h: This header contains all the #defines used by all services and the XAPI. This file is intended to be a one-stop storage for all critical values. When there are new services added, the Universal.h file should be updated with any new values that the new service relies upon. A common addition would be the TUN_types of the TUN packets used by a new service.

Single_buff.cpp/Single_buff.h: A sophisticated storage buffer used by the XAPI to store the various packets which are used by the XAPI and the services. Services may also use this class for their buffer management.

LCD_service.cpp/LCD_service.h: An example service that uses the Arduino LCD hardware. This service also contains routines necessary for the usage of the built-in switches.

Serial_service.cpp/Serial_service.h: An example service that uses the extra serial hardware available on the more powerful Arduino boards. This service works particularly well with Arduino MEGA. It should be noted that this service will not work on the Arduino UNO since there are not enough serial ports on the micro-controller.