1. **XAPI Definition**

The XBee Application Programming Interface (XAPI) is a small Arduino-based framework that allows easier manipulation of Digi International's XBee radio hardware. In the development of version 3.0 of the XAPI, two primary features were considered: 1) abstract away the complicated formation of an XBee hardware-based packet, and 2) present the application developer with a framework that is both modular and standardized. In addition to the two primary features, various secondary features are also considered and implemented into the XAPI. One such secondary feature is the usage of a standard packet format for all communications both local and over the radio. In XAPI parlance, the standard packet is called a “TUN packet” [Appendix A]. The name is derived from the fact that the standard packet is TUNneled through the XBee hardware-based packet for remote radio communications. A more detailed description of the TUN packet can be found in section IV of this document.

1. **XAPI Services**

In order to utilize the XAPI, the developer will need to create additional functionality in the form of “services.” It's best to think of a service as a sophisticated interrupt routine in which the “interrupt latch” is called in a software loop rather than through a hardware trigger as is typical of most micro-controllers. As with classical interrupts, a service must consist of a lean, fast code designed to solve a single problem as quickly as possible. If a service is inefficient and doesn't quickly return control to the main loop, the entire XAPI system will take a performance hit or even fail.

Since a service is designed to be executed like an interrupt routine, there's a "philosophical" and "terminology carry-over". Philosophically, the service must be small and quick to execute; that is, the service must also avoid solving complex problems. Rather it is best to use the built-in XAPI internal communications functionality between services when a complex problem requires it. The terminology carry-over is the usage of the word “latch.” In the classical interrupt setting, a latch is a method used by developers to “latch” together multiple interrupt handlers under the same interrupt address. For example, consider an interrupt thrown by a hardware ticker. When the ticker fires an interrupt, the code execution pointer goes to a jump table that has the address of the interrupt handler stored. A simple interrupt handler might be used to produce a “beep” when the ticker fires. But what if the computer also needs to flash a light when the ticker fires an interrupt? In that case, the new interrupt handler stores the current interrupt handler address and then replaces the address with its address in the jump table. So now when the ticker interrupt fires, a light is flashed and a beep is produced. In other words, the basic idea of a latch is a multiple interrupt handler that is executed in succession for a single interrupt firing.

Each service must have a latch that serves as the entry-point into the service. Instead of using a physical interrupt such as the timer interrupt, the Arduino development system uses an infinite loop. All latches are executed from within an infinite loop along with an XAPI latch, which will update the XAPI system. Below is an example of latches used within the Arduino infinite loop.

void loop()

{

xapi.xapi\_latch();

lcd\_service.lcd\_service\_latch();

serial\_service.serial\_service\_latch();

 }

Figure 1: An Arduino infinite loop that is used in a fashion similar to an interrupt call.

The Figure 1 code example shows a total of three latches. The first latch is the actual XAPI system latch. For each loop iteration, the XAPI latch is called to perform the extraction of incoming bytes in order to create a complete, error-free packet. When a packet is created, it is stored into two buffers. One of the buffers, the m\_external\_TUN\_packet, is available to be queried by the services upon the execution of their latches.

The second latch in the loop is the lcd\_service\_latch. As the name suggests, this latch is the entry point into the service that updates the LCD display, which is attached to the Arduino hardware. To update the display, the LCD service must first query the XAPI to see if an external LCD packet has arrived. If the XAPI is holding a completed LCD packet, the LCD service will extract the packet, process it, and then update the actual LCD display module. Below is the LCD latch that must perform these duties.

void LCD\_service::lcd\_service\_latch()

{

// update the button states

  read\_LCD\_buttons();

// process any local LCD message packets

  process\_local\_TUN\_packet();

// process any external LCD message packets

  process\_external\_TUN\_packet(); }

Figure 2: The LCD Service latch. This latch performs three duties before returning control to the main loop.

As illustrated in Figure 2, the LCD latch is relatively simple to understand. The first duty performed by the LCD latch is to process any buttons being pressed. On the LCD hardware, there are about 7 buttons that the user can press and for which this latch will capture those presses. The second duty performed by the latch is to process any “local” TUN packets. A local TUN packet is an LCD TUN packet sent by some other service that seeks to use the services provided by the LCD service module. This scenario may occur if another unrelated service is in an error state, which results in placing a local TUN error message packet into an XAPI designated buffer. The LCD service extracts this local TUN packet and display the error message to the user. The last duty performed by the latch is the extraction of an “external” TUN packet, which exists on the local hardware. External packets are external to the hardware on which the latch is being executed. Therefore, some foreign transmitter is sending an LCD message over XBee radio to the LCD service. As with the local TUN packet, the external TUN packet is extracted from the XAPI and processed, and the message is displayed on the LCD hardware.

The final latch executed in the loop shown in Figure 1 is the serial\_service\_latch. This latch is the entry point to the serial service, which monitors the RS-232 hardware on the Arduino board. As bytes arrive over the serial port, the latch extracts them and attempts to construct a full TUN packet. Once a TUN packet is constructed, the serial service uses a simple switch statement to determine the action to be applied to the recently constructed TUN packet. In order to see the switch statement used, a code trace of the routine assemble\_TUN\_packet(m\_serial.read()) is performed. Once the incoming bytes are read from the serial port and added to the construction of the TUN packet, the latch then checks if there is a local TUN packet in waiting. If so, a switch statement uses the packet type to determine what to do with the local TUN packet. In the event that the TUN packet is of TUN\_TYPE\_LOCAL\_CHAT, the TUN packet is extracted and outputted through the serial hardware. This process can be seen in its entirety by doing a code trace starting with the snd\_local\_TUN\_packet\_via\_serial() routine (see Figure 3).

void Serial\_service::serial\_service\_latch()

{

  uint8\_t packet\_type = 0;

   // see if there is a new byte  if (m\_serial.available() > 0)

  {

  assemble\_TUN\_packet(m\_serial.read());

  }

// process any local serial packets that need to

  // be shipped out over serial.

  // NOTE: the point of this code is to allow other

  // services to ship out packets via serial instead

  // of radio.

  packet\_type = m\_xapi.CONNECT\_local\_TUN\_get\_type();

switch(packet\_type)

  {

  case TUN\_TYPE\_LOCAL\_SERIAL\_DEBUG\_MSG:

  case TUN\_TYPE\_LOCAL\_CHAT:

  snd\_local\_TUN\_packet\_via\_serial();

  break;  } }

Figure 3: The Serial service latch. This latch is called from the Arduino infinite loop.

As can be inferred from the LCD latch and the serial service latch, a typical XAPI service latch must perform the following duties: 1) Query the XAPI for any external messages over radio, 2) Process any external messages, 3) Query the XAPI for any local messages sent by other services, 4) Process any local messages, and 5) Stay simple and easily readable. The latch should be simple and to the point. If the latch appears bloated, consider abstracting some of the logic away into other routines. As mentioned earlier, the service must also be fast as there are other services, which may be time sensitive. The Serial service, for example, is very time-sensitive. If some other service is taking forever, incoming serial bytes may be dropped because the serial service was unable to extract them from the incoming hardware buffer.

1. **Architecture Overview**

The architecture of the XAPI closely resembles that of a simple real-time operating system (RTOS). Just as an RTOS has multiple tasks, the XAPI has multiple “services.” While a simple RTOS may execute its tasks in a round-robin fashion, the XAPI uses an infinite loop to execute each of the services via its “latch.” Next, an RTOS may provide a pipe for inter-task communications. The XAPI does similar inter-service communication via a “local buffer.” Lastly, the XAPI mirrors the modularity of the RTOS. Services can be added and removed from the XAPI as desired.

The XAPI is deliberately small and simple. Only basic framework features are offered to the user of this API. Two of these features include the transmission and reception of external and local TUN packets via XBee hardware. Upon receiving these packets, the XAPI sorts them into one of two buffers: “m\_external\_TUN\_single\_buff” for incoming external TUN packets, and “m\_local\_TUN\_single\_buff” for internal communication between active services. Both of these buffers have public helper functions for which the XAPI 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 placed 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. Below is an image of the XAPI 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 that is used to perform calls on XAPI routines. |

1. **XAPI To XBee Communications**

The primary feature of the XAPI is simplified message passing over XBee radio hardware. Using a standard RS-232 serial connection between the Arduino and the XBee hardware, the XAPI is able to send control and data packets to and from the XBee hardware. But in order for successful communications between the XAPI and the XBee hardware, the XBee must be in “API” mode. The XBee hardware has two possible modes of operation: Application Programming Interface (API) mode, and Attention (AT) mode.

By default, the XBee ships to the consumer in AT mode. Under this mode, the hardware behaves as if it's a classical Hayes modem circa 1990's. Generally, the AT command set is “human readable,” and allows for simple control of the XBee hardware. The developer can even communicate to the XBee hardware using a simple ASCII-based terminal. The downside to the AT command set is the lack of finer hardware [what is meant by finer hardware?] and AODV network management controls. Also, the AT mode does not provide a direct way to communicate with the XBee hardware. Much of the AT command set simply passes messages through the hardware to a single destination. If the application developer needs to fine-tune the performance of the AODV network, the XBee hardware must be in API mode.

XAPI exclusively uses the XBee in API mode. This allows application developers the opportunity to write services that not only process application specific packets, but also fine-tune the XBee and use some of the more powerful features of the hardware. While the API mode is powerful, it is far from user-friendly. For example, all communications between the Arduino and the XBee hardware must be performed using correctly formed packets over a serial connection. Since there are numerous XBee API packet types, determining the correct packet type and properly constructing the packet is time consuming and prone to mistakes. To help the application developer, the “Request to Transmit” API packet type is built into the XAPI. This packet type is most commonly used to allow for the radio transmission of developer-defined data and packets. If more packet types are required, the developer can use the built-in API packet type as an example on how to create new packet types.

In summary, an unmodified XAPI 3.0 framework only communicates to XBee hardware in API mode. The developer is only able to send application-specific messages to and from other XBee equipped Arduino modules via the “Request To Transmit” XBee defined packet type. The developer may add new services to the XAPI to facilitate new XBee hardware packet types as the need arises. It is recommended that the addition of any new packet types come in the form of a service rather than as an addition to the XAPI itself. This approach helps to prevent the XAPI from getting large and complex.

1. **Example Usage**

The XAPI core is of little use 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, an 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 4: 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 that is driven by an 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 4, 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 check if that code is 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 that it is a member of the class in which it is being instantiated. 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 its own payload structures. One of the basic payloads is simply text without any formatting. If this were 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 are the checksum, and finally, the remaining bytes are the payload itself. A full description of the TUN packet is in Appendix B. All services require the use of the “create\_TUN\_packet” routine in the Util class in order to perform communications 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 will 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 that 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 that 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, XAPI provides this functionality 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. Two classes provide this ease of development: 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.

|  |
| --- |
|  |
| Figure 5: 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. 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 that 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,
3. The TUN packet is transferred to the Serial\_service over RS-232 from the PC,
4. 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,
5. The TUN packet is stored within the XAPI buffer m\_local\_TUN\_single\_buff via the XAPI function CONNECT\_local\_TUN\_set\_packet,
6. The LCD\_service eventually queries the local XAPI buffer via the CONNECT\_local\_TUN\_get\_type function,
7. The LCD\_service realizes the packet belongs to it and extracts it using the XAPI function CONNECT\_local\_TUN\_get\_packet,
8. The LCD\_Service extracts the payload from the TUN packet,
9. 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 up to the point that the LCD\_service consumes it. All of the major routines, which touch the TUN packet, are listed.

|  |
| --- |
|  |
| Figure 6: 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. |

1. **XAPI Utilities**

XAPI includes built-in routines to simplify working with the XAPI framework. Located in Util.h and Util.cpp, these routines (collectively called “utilities”) help developers do common XAPI specific duties along with random functionality that has been found useful in day-to-day development. The primary feature provided by the utilities is the simple and error-free construction of TUN packets. Creating TUN packets from scratch can cause system-wide problems due to hard-to-track bugs. Therefore, the utilities abstract this process away from the developer to reduce mistakes. Unfortunately, the utilities provided to the developer are not magic. If a developer needs to create new types of TUN payloads, the Util.h and Util.cpp may need to be updated. This was the case with the LCD service and as such will be used as an example later in this document.

In addition to the easy creation of TUN packets, the utilities provide a collection of routines that can be used for any number of reasons. There are routines that are used to convert integers to their hexadecimal equivalent, for example. There are also routines that can easily extract data from the TUN packets such as the size of the payload, the type ID of the packet, and the actual payload itself. Again, not every possible utility routine has been implemented and future developers are free to add new routines to improve the Util.h and Util.cpp files.

**Creation of an External TUN Packet**

One of the main features of the utilities is the simplified creation of an external TUN packet. Rather than having the developer create one from scratch, various utility routines can be used instead. Consider the following example: A developer needs to send an external TUN packet with the simple LCD payload of “HELLO\_WORLD.” Below is actual code from LCD\_service.cpp, which uses utilities to create the TUN packet.

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;

}

The intent of the above code is to create an external TUN packet that will tell the external LCD service to display a message. Therefore, if the developer wishes to use the above code to send a “HELLO\_WORLD” message to a remote Arduino running the XAPI with the LCD service, here is what the calling code would look like:

lcd\_snd\_EXTERNAL\_message(0x1AFF00BC, 0x0ECA010F, 0xFFFF, 0, 0, “HELLO\_WORLD”);

The above code would send a TUN packet over the radio to the Arduino with the 64-bit XBee address of 0x1AFF00BC:0x0ECA010F, and the 16-bit address of 0xFFFF (which means broadcast). As such an XBee can either use a direct, hardwired 64-bit address or a network-derived 16-bit address. Understanding how the Xbee handles addressing is obviously critical for any XAPI TUN packet code to work. The LCD being used in this example code is generic and specifically built for the Arduino hardware by various vendors. The LCD panel has two lines, 16-characters wide. Therefore, an x value of 0 and a y value of 0 will print “HELLO\_WORLD” on the first line at displacement of 0.

We must now look deeper into the code to see how the LCD service makes use of the utilities to simplify TUN packet creation. To begin, the following three buffers are created:

1. uint8\_t payload\_buff[LARGE\_BUFF\_SZ];
2. uint8\_t TUN\_buff[LARGE\_BUFF\_SZ]; and
3. uint8\_t Xbee\_buff[LARGE\_BUFF\_SZ].

All three of the buffers are of type “uint8\_t” which is exactly the same as a “char” type in this case. All three buffers are simply byte buffers, which should be common to any developer who has worked with C/C++. The size of the buffers is of “LARGE\_BUFF\_SIZE.” This is a #define size that is located in the universal .h file. #defines are used to keep the code consistent and to avoid the usage of random “magic” numbers. Generally speaking, LARGE\_BUFF\_SIZE is usually set to 70. Therefore, each of the three buffers will have room for 70 bytes.

The next line of code creates the TUN payload, and only the payload. This is a critical step that must be understood. The creation of a packet, which can be sent over radio, must be created in a slow, step-by-step process. There is no “big bang” routine that produces the entire TUN packet and subsequent XBee packet in one shot. This slow process is deliberate as it allows for easier tracking of bugs and for the easy creation of new routines and more sophisticated TUN packets. Let's take a closer look at the payload creation code:

// 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);

The comments preceding the code show how the payload section will be constructed. Basically, two ASCII-Hex bytes will be used for the x-position integer, two ASCII-Hex bytes will be used for the y-position, and the “HELLO\_WORLD” will take up the rest of the payload. It can be seen through the call how the arguments are passed. One interesting detail is that a pointer to the payload\_buff is passed along with the max size of that buffer. Sending the max size of the payload\_buff helps prevent buffer overflows deeper in the code.

The next line of code produces the actual TUN packet. Technically, the TUN packet is simply a wrapper around the payload that the external receiving XAPI will use to properly process the payload. Below is the code:

// 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);

The above code is the generic routine to create all TUN packets given a payload. As can be seen from the comments, the format of the resulting TUN\_buff (i.e., the actual TUN packet), is shown. The beginning of the TUN packet is the '$' character. The next two bytes are the ASCII-Hex bytes of the TUN packet type. In this case, the type is of “TUN\_TYPE\_EXTERNAL\_LCD\_MSG.” All TUN packet types are #defines which can be found in the file universial.h. The next two bytes are the ASCII-Hex bytes of the payload size. The next four bytes are the ASCII-Hex of the checksum, and finally, the payload is in the buffer followed by the '%' character.

This complete TUN packet is stored in the “TUN\_buff” buffer with a size of “TUN\_buff\_sz.” The TUN packet is now complete. If this packet was destined for a local service, no further processing would be necessary. Unfortunately, this TUN packet must be sent over the radio to another Arduino using the XBee radio module. Therefore, the TUN packet must be encased within a XBee transmission request packet. The resulting transmission request packet is complicated and not shown in the comments. The critical concept to grasp is that the complete TUN packet is stored within the payload section of the XBee transmission request packet. For this reason the '$' and '%' characters are used in the creation of a TUN packet. Those characters serve as a sentinel value that allows for the easy extraction of the TUN packet from the complicated XBee packet. Below is the code, which will produce the final XBee packet that is ready to be sent over radio:

// 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);

The above code takes the completed TUN packet, which is stored in the TUN\_buff buffer, and encases it within the XBee packet. The final result will be stored in the Xbee\_buff. Again, the Xbee\_buff is of max size “LARGE\_BUFF\_SZ” and the total amount of bytes in the buffer is stored in the variable Xbee\_buff\_sz. Now that a complete XBee packet has been created, it can be sent over the radio via the built-in XAPI routine. The code below shows how to send the XBee packet over the radio:

// Physically ship out completed Xbee packet over radio

m\_xapi.snd\_packet(Xbee\_buff, Xbee\_buff\_sz);

Most of the routines shown above are generic and the developer will use them without modification of the time. The one routine that tends to be application specific is the routine that will create the payload for the TUN packet. In this example, it's the m\_util.construct\_lcd\_payload() routine that will most likely have to be replaced in order to produce unique TUN packets for a given situation. Also, the developer will need to allocate a new TUN type ID for any new TUN packets they need to use. In summary, the developer must follow the steps below in order to create a TUN packet that can be sent over radio to a remote Arduino:

1. Create the TUN payload
2. Encase the TUN payload in a TUN packet
3. Encase the TUN packet in the payload section of the XBee packet
4. Use the XAPI routine to send the XBee packet over radio
5. **The Heartbeat Service – an Example Code**

This section demonstrates the construction of the Heartbeat service. The Heartbeat service creates a “beat” on a regular interval. The purpose of this Heartbeat is to allow the remote embedded system to periodically send a “pulse” over the radio to some other node. The pulse will be interpreted as proof that the remote system is alive and functioning correctly. Since the Heartbeat relies on a periodic pulse, hardware interrupts are used to trigger the sending of a pulse. As of XAPI version 3.0, interrupts and any other sort of additional processing threads are not supported. Therefore, in order to fully implement the Heartbeat service, changes to the XAPI itself are required.

A special heartbeat packet is created with each beat. For simplicity, the heartbeat packet is sent over the Serial Service rather than over the radio using the XAPI framework. The reason behind this design decision is as follows: 1) Changing the Heartbeat Service to use radio rather than the Serial Service is straight forward; 2) Plenty of services already built use the radio, and 3) The Heartbeat Service demonstrates the usage of the local buffer to allow communications between local services on the same Arduino hardware.

The first step to setting up a new service is using the following basic skeleton class:

|  |
| --- |
| #ifndef SKELETON\_SERVICE\_h  #define SKELETON\_SERVICE\_h  #include <arduino.h>  #include <Xapi.h>  #include <Util.h>  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  // General documentation on this service to  // be located in this area.  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  class Skeleton\_service  {  // The private objects and values that are  // passed via the constructor. Additional  // private objects should be allocated in this area.  private:  // A pointer to the XAPI that will be passed in via  // the constructor.  Xapi& m\_xapi;    // A Utility object to aid in packet construction  Util m\_util;    // Private data members to keep object state  private:  int m\_foo;    // Private member functions inaccessible outside of object  private:  void init\_m\_foo();  void inc\_m\_foo();    // Public routines generally executed from the Arduino Sketch  public:  // Constructor generally in the global space of the sketch  Skeleton\_service(Xapi& \_xapi);    // Latch usually located in the "loop()" sketch routine.  void skeleton\_service\_latch();  };  #endif |
| Figure 7: A basic “skeleton” service class (Skeleton\_service.h). The bare essentials are shown and can be modified for performing a specific XAPI service. |

While it is not critical to use the given Skeleton service, it does simplify development and it keeps a sense of coding standards across services. Speaking of coding standards, the reader will notice the case differences between class names and function names. Only the first letter of the class name is capitalized followed by “\_service.” The latch, in comparison, is the service name in all lowercase followed by a “\_service\_latch().” Generally, there are no arguments passed to the latch as service latches are modeled after traditional interrupt latches that generally do not have arguments passed into them. The next major naming convention comes in the form of “m\_.” The “m\_” is appended to all data members of the class. Hence, the “m\_” means “member.” It’s named in this way to help avoid confusion when dealing with data manipulation from within class member functions.

Now that the service coding standards have been discussed, it will be easier to understand the internals of a basic service class. To begin, the first critical private data member is a pointer to the XAPI. This pointer, m\_xapi, will be used to call XAPI routines to survey packets and to ultimately ship packets out via radio. All services must have the same pointer to a single XAPI object. The last critical private data member is the m\_util object. The developer can use the m\_util object to simplify the creation of packets. This object also has member functions that allow for the extraction of data from packets and the conversion of ASCII-Hex numbers into unsigned integers and vice-versa. Basically, the m\_util object is an attempt to simplify the creation of properly formed packets and allow the developer to focus primarily on the duties of the service rather than the construction of the service.

Finally, there are two critical public member routines that must be implemented in each service. The first routine is the constructor. Unlike the default constructor, the critical constructor must accept a pointer to the XAPI. This pointer will then be stored in the m\_xapi private data member. The next critical member routine is the latch. The purpose of the latch is to allow the service to update itself on each iteration of the Arduino “loop()” routine. In the skeleton example given in this section, the “Skeleton\_service(Xapi& \_xapi)” is the constructor, and “skeleton\_service\_latch()” is the latch routine. The next section will provide a rough overview as to how to implement the critical member routines.

Example Service Code

This section provides a very basic example of how to code service member routines. It is assumed the reader understands basic C++ coding conventions, and as such, will not be explained. Rather, this section will just focus on the critical member routines that may prove to be slightly difficult at first attempt. Consider the following body of code:

|  |
| --- |
| #ifndef SKELETON\_SERVICE\_cpp  #define SKELETON\_SERVICE\_cpp  #include <skeleton\_service.h>  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  // Simply init the m\_foo private data member  void Skeleton\_service::init\_m\_foo()  {  // set the starting value for m\_foo  m\_foo = 0;  }  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  // Increment m\_foo in the main loop  void Skeleton\_service::inc\_m\_foo()  {  m\_foo++;  }  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  // Latch to be executed in Arduino main loop.  void Skeleton\_service::skeleton\_service\_latch()  {  // simply increment the foo member function  inc\_m\_foo();  }  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  // Constructor to be called in Arduino global space  Skeleton\_service::Skeleton\_service(Xapi& \_xapi): m\_xapi(\_xapi)  {  // simply init foo  init\_m\_foo();  }  #endif |
| Figure 8: The body of code that defines the Skeleton\_service class. The code in this file (Skeleton\_service.cpp) can be modified along with the class definition code (Skeleton\_service.h) to create a completely new service. |

In Figure 8, the critical member functions skeleton\_service\_latch() and the constructor, are given very basic bodies. The only real job the latch currently does is increment the m\_foo data member via the inc\_m\_foo() routine. The constructor, on the other hand, first copies over the XAPI pointer and then initializes the m\_foo data member via the init\_m\_foo() routine.

As it stands, the Skeleton\_service is complete. An individual can import the files “Skeleton\_service.h” and “Skeleton\_service.cpp” into the Arduino IDE and use it as a library. Even though it’s possible to use the Skeleton\_service outright, it’s suggested that the developer never does this. Rather, the Skeleton\_service is simply a starting point for a more sophisticated service. To illustrate the usage of a complete service, the code below is a fully working Arduino sketch that focuses solely on the inclusion and the proper set-up of the Skeleton service. The reason why this code is included is to give the developer an even clearer understanding as to how to properly initialize the services and how to use the latch in the loop() routine.

|  |
| --- |
| #include <Skeleton\_service.h>  #include <Xapi.h>  #include <Universal.h>  #include <Util.h>  #include <Single\_buff.h>  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  // This file contains example code to exercise  // the Skeleton\_service code.  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  // Init the objects to be used.  // The XAPI framework will communicate to the  // XBee hardware over the default Serial port  Xapi xapi = Xapi(Serial);  // The Skeleton service  Skeleton\_service skeleton\_service(xapi);  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  void setup()  {  // Some objects might require setup.  // Skeleton service does not    // The Serial object must be initialized correctly  // in order to communicate with the XBee hardware.  Serial.begin(9600);  }  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  // Latches go into this section  void loop()  {  // XAPI latch  xapi.xapi\_latch();    // Skeleton service latch  skeleton\_service.skeleton\_service\_latch();  } |
| Figure 9: The complete Arduino sketch that exercises the Skeleton\_service (skeleton\_demo.ino). The purpose of this code is to demonstrate how to properly setup and use a basic service. Notice that the XAPI itself also has a latch that is executed in the loop() routine. |

The Skeleton service can now used as a template for the Heartbeat service. First, changes made to the Skeleton header file are made. These changes include the names of the various methods, the class name, the file name, and the comments. There are also new variables and methods that are necessary for the Heartbeat service to function correctly. Remember: The Skeleton service is simply a framework, not a rigid template that must be followed to determent to the service.

|  |
| --- |
| #ifndef HEARTBEAT\_SERVICE\_h  #define HEARTBEAT\_SERVICE\_h  #include <arduino.h>  #include <Xapi.h>  #include <Util.h>  #include <avr/io.h>  #include <avr/interrupt.h>  #define LEDPIN 13  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  // Without the following webpages, I would not  // have been able to develop the drone heartbeat.  // CREDITS:  // http://blog.oscarliang.net/arduino-timer-and-interrupt-tutorial/  // https://arduinodiy.wordpress.com/2012/02/28/timer-interrupts/  //  // Some code has been directly lifted from the above webpages.  // It is not my intent to take credit for that code.  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  // This service simply does two tasks:  // 1) send out a heartbeat packet  // 2) flash the onboard LED with each heartbeat  //  // The point of this service is to inform  // whomever is listening that the drone is  // alive and still within radio range.  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  class Heartbeat\_service  {  // objects used  private:  Xapi& m\_xapi;  Util m\_util;    // whether or not the heartbeat is active  private:  // Flag to determine if service is active  bool m\_active;    // Clock frequency of Arduino hardware  double m\_clock\_freq;    // Seconds between interrupt firings  double m\_target\_time;    // Internally used for calculating register values  double m\_timer\_res;  // Internally used for calculating register values  double m\_timer\_counts;  // Internally used for calculating register values  const double SCALER = 1024.0;    // Methods for Heartbeat\_services  private:  // Allows interrupt to send pulses  void set\_active();  // Returns true if interrupt is initialized, false if not  bool is\_interrupt\_initalized();  // Calibrates the interrupt registers and turns it on  void init\_and\_fire\_heartbeat();  // Sends out a pulse over the radio  void produce\_heartbeat();  // Constructor  public:  // Constructor that requires clock frequency and firing time  Heartbeat\_service(Xapi& \_xapi, double clock\_freq, double target\_time);  // Latch to be used in the loop() to set registers and turn on interrupt  void heartbeat\_service\_latch();  };  #endif |
| Figure 10: The Heartbeat service header file. This header file is the result of modifying the original Skeleton header file for the purpose of the Heartbeat service. The more critical changes are bolded. |

The above code listing shows the contents of Heartbeat\_service.h. It should be noted that Heartbeat\_service.h only shows a small resemblance to the original Skeleton\_service.h file. The Skeleton files maybe dramatically different than the final product and should only be used as guideline. The reader will notice that the first major departure from the original Skeleton service is the comments in the beginning of the listing. It is here where a simple description and links to source code online can dramatically increase the productivity of future developers when they attempt to add new features to the service or modify already existing code. The above listing gives links to example code and tutorials on how the Arduino interrupt system works. Modifications of the Heartbeat service will require a deep understand of Arduino hardware interrupts.

The next major modification of the Skeleton service is the inclusion of member variables. Again, it should be noted that member variables all have the “m\_” preceding the name. This is simply a coding standard that helps with reader comprehension. Most of the member variables are used internally in other routines to perform calculations necessary for the correct execution of the interrupt mechanism. Therefore, they technically do not have to be member routines, temporary local-scope variables in the methods would have worked just fine for most of them. One member variable, m\_active in particular, is used by multiple member methods to determine if the hardware interrupts are active. When it is false, the service “believes” it is not producing heartbeat pulses. When true, the service “believes” it is producing the pulses to be sent over radio. The reason why the word “believes” is used is because the service doesn’t directly know if an interrupt is firing. Rather, the service can either set or disable the interrupt and the underlying hardware executes independently on another thread.

The next section contains private methods that allow the service to initialize the hardware interrupt and produce the pulse to be sent over radio. The first two methods are mostly “helper” routines. The “set\_active()” routine simply sets the m\_active variable to true. The next method, “is\_interrupt\_initalized()” will be queried by other methods to determine if the hardware interrupt system is firing. The exact mechanics behind the functioning of these methods and all other methods in the Heartbeat service will be demonstrated later in this section.

The majority of the work done in this service is performed by the “init\_and\_fire\_heartbeat()” and “produce\_heartbeat()” methods. The first method takes calculations performed in the constructor to properly set up the hardware interrupt registers. Once the registers are set, the method turns on all hardware interrupts. The second method, produce\_heartbeat(), literally produces the heartbeat in the form of an XBee packet to be set over the radio. This method will be magically called when the hardware interrupt is fired. The reason why it’s considered “magical” is because of some unconventional coding which had to written in order for this service to work.

The last private section contains the constructor and the latch. The Heartbeat constructor contains significantly more code than the Skeleton constructor. The reason behind this is simple. There are calculations that must be performed in order to correctly set the interrupt registers. The constructor takes care of this work along with setting up a pointer to itself. The pointer is part of the “magical” code which had to be written and will be described later in this section. The next method, “heartbeat\_service\_latch(),” is responsible for turning on the heartbeat interrupt and informing the rest of the service that the interrupts are firing. The latch is the main consumer of most of the helper methods of the previous private section in the header file. The reader may have noticed that the latch is executed on every iteration of the Arduino loop(), and as such would it would be redundant to setup the interrupt on every iteration. To remedy this situation, the latch first checks to see if the interrupt hasn’t already been set up on a previous iteration of the loop. If it has, the latch simply exits without doing any significant processing. Below is a listing of the Heartbeat\_service.c file. It contains all the code that makes up the Heartbeat service and the interrupt routine:

|  |
| --- |
| #ifndef HEARTBEAT\_SERVICE\_cpp  #define HEARTBEAT\_SERVICE\_cpp  #include <heartbeat\_service.h>  // Get a pointer to the class  Heartbeat\_service\* hbs;  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  // Creates a heartbeat and places on local buffer  // for the serial service to export to PC  // NOTE: You will need to change it to  // XAPI if intended to go over radio  void produce\_heartbeat()  {  // Need to use a pointer since the interrupt  // routine is NOT part of the class. There may  // be a better method of calling a class routine  // from an interrupt, but I am currently unsure as  // to how to do this.  hbs->produce\_heartbeat();  }  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  // Creates a heartbeat and places on local buffer  void Heartbeat\_service::produce\_heartbeat()  {  // For the payload  uint8\_t payload[] = HEARTBEAT\_TEXT;  uint8\_t payload\_sz = strlen((char\*)payload);    // For the entire packet storage  uint8\_t buff[MED\_BUFF\_SZ];    // Create the packet  m\_util.create\_TUN\_packet( TUN\_TYPE\_EXTERNAL\_HEARTBEAT,  payload, payload\_sz,  buff, MED\_BUFF\_SZ);    // Remove the start and stop sentinal bytes  m\_util.remove\_TUN\_frame(buff);    // Place on the local buffer for the serial service to process  m\_xapi.CONNECT\_local\_TUN\_set\_packet(buff, MED\_BUFF\_SZ);    // Flash the heartbeat LCD  digitalWrite(LEDPIN, !digitalRead(LEDPIN));  }  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  // Interrupt Handler  ISR(TIMER1\_COMPA\_vect)  {  // Output the heartbeat  produce\_heartbeat();  }  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  // Will turn init and turn on interrupts  void Heartbeat\_service::init\_and\_fire\_heartbeat()  {  // Init the LEDPIN  pinMode(LEDPIN, OUTPUT);    // initialize Timer1  // Disable global interrupts  cli();    // Set entire TCCR1A register to 0  TCCR1A = 0;    // Same for TCCR1B  TCCR1B = 0;  // set compare match register to desired timer count:  OCR1A = m\_timer\_counts;  // turn on CTC mode:  TCCR1B |= (1 << WGM12);  // Set CS10 and CS12 bits for 1024 prescaler:  TCCR1B |= (1 << CS10);  TCCR1B |= (1 << CS12);  // enable timer compare interrupt:  TIMSK1 |= (1 << OCIE1A);  // enable global interrupts:  sei();  }  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  // Will return true or false on whether or not  // the interrupt is initalized.  bool Heartbeat\_service::is\_interrupt\_initalized()  {  return m\_active;  }  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  // Simply inform the system that the interrupt is set.  void Heartbeat\_service::set\_active()  {  m\_active = true;  }  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  // Latch will basically only set up the interrupt.  // If interrupt is already set up, the latch will  // basically do nothing.  void Heartbeat\_service::heartbeat\_service\_latch()  {  if (!is\_interrupt\_initalized())  {  init\_and\_fire\_heartbeat();  set\_active();  }  }  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  Heartbeat\_service::Heartbeat\_service(Xapi& \_xapi, double clock\_freq,  double target\_time): m\_xapi(\_xapi), m\_clock\_freq(clock\_freq), m\_target\_time(target\_time),  m\_active(false)  {  // get the timer resolution of the Arduino  m\_timer\_res = 1.0 / (clock\_freq / SCALER);    // To use CTC, you need to determine the number of counts  // needed to get to a one second interval.  // Assuming we keep the 1024 prescaler as before, we’ll calculate as follows:  // (target time) = (timer resolution) \* (# timer counts + 1)  // (# timer counts + 1) = (target time) / (timer resolution)  m\_timer\_counts = m\_target\_time/m\_timer\_res;  m\_timer\_counts--;    // get a pointer so we can call class routines  // via the interrupt routine which is not a class member.  hbs = this;  }  #endif |
| Figure 11: The above is a listing of the “Heartbeat\_service.c” file. This file contains all the code that makes up the Heartbeat service including the code necessary for hardware based interrupts. |

To conclude this section, two more listings will be presented. The first listing contains the Arduino sketch that sets up the Heartbeat service and turns it on. As with the Heartbeat service code, the sketch is simply a modification of the Skeleton sketch presented in the previous section. The last listing is the modified XAPI. Since hardware interrupts effectively create a multi-threaded application, critical sections must now be protected. Without the presented modifications, XAPI behavior would be unpredictable and will probably end up crashing the system.

|  |
| --- |
| #include <Heartbeat\_service.h>  #include <Xapi.h>  #include <Universal.h>  #include <Util.h>  #include <Single\_buff.h>  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  // This file contains example code to exercise  // the Heartbeat\_service code.  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  // Init the objects to be used.  // The XAPI framework will communicate to the  // XBee hardware over the default Serial port  Xapi xapi = Xapi(Serial);  // The Heartbeat service  Heartbeat\_service heartbeat\_service(xapi, 16000000, 0.05);    //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  void setup()  {  // Some objects might require setup.  // Heartbeat service does not    // The Serial object must be initialized correctly  // in order to communicate with the XBee hardware.  Serial.begin(9600);  }  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  // Latches go into this section  void loop()  {  // XAPI latch  xapi.xapi\_latch();    // Heartbeat service latch  heartbeat\_service.heartbeat\_service\_latch();  } |
| Figure 12: An example of the Heartbeat service being used by an Arduino sketch. The lines of code that are significantly different than the original Skeleton sketch are bolded. |

The remainder of this section shows the reader the necessary changes to the XAPI that allow for error-free usage of interrupts. The main issue in the usage of interrupts is that the XAPI doesn’t use atomic methods in the usage of the internal buffers. The internal buffers are used to send messages back and forth between local services. If the Heartbeat service interrupts happens to modify the local packet messaging system as another service is attempting to do the same, the system may fail. The simplest solution to protecting the internal packet critical system is to use mutex’s. Since the Arduino doesn’t supply an operating system, mutex’s are not available. Therefore, all interrupts are simply shut down when any thread is entering a critical area. This is made possible with the usage of the cli() and sei() routines. Below is an example listing of modified XAPI code.

|  |
| --- |
| //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  // The functions below are used to connect to the Xapi\_comms class. The purpose  // of Xapi\_comms is to allow for the easy construction of services. The programmer  // needs to add Xapi\_comms to the service in order to connect to Xapi.  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  // Allows for the distribution of a local TUN packet. Generally services will  // communicate with each other using the LOCAL TUN packet storage  uint8\_t Xapi::CONNECT\_local\_TUN\_set\_packet(uint8\_t\* buff, const uint8\_t buff\_sz)  {  cli();  uint8\_t sz = 0;    sz = m\_local\_TUN\_single\_buff.add\_TUN\_buffer(buff, buff\_sz);    sei();    return (sz);  }  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  // Gets the currently stored external TUN packet.  // Returns: > 0 or 0 depending on size of packet.  // If 0, that means there was no packet  uint8\_t Xapi::CONNECT\_local\_TUN\_get\_packet(uint8\_t\* buff, const uint8\_t buff\_sz)  {  uint8\_t sz = 0;  cli();    if(m\_local\_TUN\_single\_buff.any\_items())  sz = m\_local\_TUN\_single\_buff.pop\_top(buff, buff\_sz);    sei();  return (sz);  }  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  // gets the ID of the external tun packet  uint16\_t Xapi::CONNECT\_local\_TUN\_get\_id()  {  cli();    uint8\_t id = m\_local\_TUN\_single\_buff.get\_buffer\_id();    sei();    return (id);  }  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  // gets the type of the external tun packet  // Returns type or -1 if no packet  int Xapi::CONNECT\_local\_TUN\_get\_type()  {  cli();    uint8\_t type = m\_local\_TUN\_single\_buff.get\_TUN\_type();    sei();    return (type);  }  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  // gets the type of the external tun packet  // Returns type or -1 if no packet  int Xapi::CONNECT\_external\_TUN\_get\_type()  {  cli();    uint8\_t type = get\_TUN\_type();    sei();    return (type);  }  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  // gets the ID of the external tun packet  uint16\_t Xapi::CONNECT\_external\_TUN\_get\_id()  {  cli();    uint8\_t id = m\_external\_TUN\_single\_buff.get\_buffer\_id();    sei();    return (id);  }  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  //\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  // Gets the currently stored external TUN packet.  // Returns: > 0 or 0 depending on size of packet.  // If 0, that means there was no packet  uint8\_t Xapi::CONNECT\_external\_TUN\_get\_packet(uint8\_t\* buff, const uint8\_t buff\_sz)  {  cli();    uint8\_t size = 0;    size = get\_TUN\_packet(buff, buff\_sz);    sei();    return (size);  } |
| Figure 13: An XAPI code snippet showing how to protect the critical areas using the cli() and sei() routines. |

**Appendix A – Utility Helper**

**TUN-centric Routines**

The utilities class (util.cpp) not only allows for easier packet construction, but it also contains routines to simplify the handling of TUN packets. The basic ASCII-Hex TUN packet contains four sections: TUN Type, Payload Size, Checksum, and Payload. Manipulating each one of the four sections directly can lead to difficult-to-track bugs and a possible deviation from the TUN standard. The remainder of this section will go over the most frequently used TUN packet utility routines.

void int\_to\_hex32(uint32\_t value, uint32\_t size, char stor\_buff[]);

Purpose: Convert an unsigned 32-bit integer into ASCII-Hex string.

Arguments:

value: 32-bit unsigned integer value to be converted

size: 32-bit unsigned integer value of the size of stor\_buff

stor\_buff: allocated buffer to store the ASCII-Hex string of value

Output:

Void

void int\_to\_hex(const int value, const int size, char stor\_buff[]);

Purpose: Convert an signed 16-bit integer into ASCII-Hex string. Note: do not use negative integers, only use positive integers.

Arguments:

value: 16-bit signed integer value to be converted

size: 16-bit unsigned integer value of the size of stor\_buff

stor\_buff: allocated buffer to store the ASCII-Hex string of value

Output:

Void

uint8\_t ascii\_hex\_byte\_to\_int(const uint8\_t msb, const uint8\_t lsb);

Purpose: Two ASCII-Hex bytes are used to represent one binary bite. For example, the byte value of 178 would be “B2” in ASCII-Hex. The 'B' is the msb of the ASCII-Hex string, while the '2' is the lsb of the ASCII-Hex string. This routine will convert the ASCII-Hex msb and lsb into a single byte value.

Arguments:

msb: the msb of the ASCII-Hex string.

Lsb: the lsb of the ASCII-Hex string.

Output:

uint8\_t: The byte value of the ASCII-Hex msb combined with the lsb.

uint16\_t hex\_to\_int(int start\_byte, int end\_byte, int size, const uint8\_t\* buffer);

Purpose: Converts a string of ASCII-Hex into an unsigned 16-bit integer. This routine is used by other pre-built routines to extract values from the TUN packets. Therefore, there are many examples as to how this routine is used throughout the util.cpp file.

Arguments:

start\_byte: The starting index into the buffer ASCII-Hex string.

end\_byte: The ending index into the buffer ASCII-Hex string.

Size: The amount of ASCII-Hex bytes between the start\_byte and the end\_byte.

Output:

uint16\_t: The 16-bit unsigned integer value of the ASCII-Hex bytes between start\_byte and end\_byte.

void clean\_packet(uint8\_t\* packet, uint16\_t size);

Purpose: Cleans out a buffer for use.

Arguments:

packet: The pre-allocated buffer.

size: The total size of packet.

Output:

void: The packet buffer is will be wiped clean.

uint8\_t cpy\_buff(uint8\_t\* dest, const uint8\_t\* src, const uint8\_t byte\_cnt);

Purpose: Copies source buffer to destination buffer. Note: since byte\_cnt is an 8-bit number, the largest possible buffer size is 255 bytes.

Arguments:

dest: The destination buffer.

src: The source buffer.

byte\_cnt: the total number of bytes to copy starting from 0.

Output:

uint8\_t: The amount of bytes copied.

Boolean verify\_checksum(const uint8\_t\* packet);

Purpose: Takes an entire TUN packet and verifies the packet is correct via the Checksum value. Note: the '$' and '%' framing bytes must not be present.

Arguments:

packet: The complete TUN packet.

Output:

boolean: True if TUN packet is correct, False if TUN packet is corrupt.

uint8\_t get\_SYS\_packet\_length(const uint8\_t\* packet);

Purpose: Extracts the size of a SYS packet. Note: The SYS packet is not a well-developed idea in the XAPI framework. Using SYS packets will be a feature in later implementations of the XAPI.

Arguments:

packet: The complete SYS packet.

Output:

uint8\_t: The size of the SYS packet.

uint8\_t get\_SYS\_packet\_length(const uint8\_t\* packet);

Purpose: Extracts the size of a SYS packet. Note: The SYS packet is not a well-developed idea in the XAPI framework. Using SYS packets will be a feature in later implementations of the XAPI.

Arguments:

packet: The complete SYS packet.

Output:

uint8\_t: The size of the SYS packet.

uint8\_t create\_TUN\_packet( uint8\_t packet\_type, const uint8\_t\* payload,

uint8\_t payload\_sz, uint8\_t\* buff, uint8\_t buff\_sz);

Purpose: Create a full TUN packet.

Arguments:

packet\_type: 8-bit number representing the TUN packet type.

payload: The byte array representing the payload of the TUN packet.

payload\_sz: unsigned 8-bit size of the payload.

buff: pre-allocated byte array to store the completed TUN packet.

buff\_sz: the 8-bit value of the max size of the pre-allocated buffer.

Output:

uint8\_t: The final size of the buff array used to store the TUN packet.

uint8\_t get\_TUN\_type(const uint8\_t\* packet);

Purpose: Extract the TUN type from a TUN packet.

Arguments:

packet: The complete TUN packet.

Output:

uint8\_t: The returned TUN type.

uint8\_t get\_TUN\_payload\_sz(const uint8\_t\* packet);

Purpose: Extract the TUN payload size from a TUN packet.

Arguments:

packet: The complete TUN packet.

Output:

uint8\_t: The returned TUN payload size.

uint8\_t get\_TUN\_packet\_sz(const uint8\_t\* packet);

Purpose: Extract the TUN packet size from a TUN packet.

Arguments:

packet: The complete TUN packet.

Output:

uint8\_t: The returned TUN packet size.

Boolean is\_TUN\_packet\_framed(const uint8\_t\* packet);

Purpose: Determine if the TUN packet is framed with the bytes '$' and '%.'

Arguments:

packet: The complete TUN packet.

Output:

boolean: True if framed, false if not framed.

Boolean remove\_TUN\_frame(uint8\_t\* packet);

Purpose: Removes the '$' and '%' frame from TUN packet.

Arguments:

packet: The framed TUN packet

Output:

boolean: True if frame removed, false if no frame removed.

packet: the unframed TUN packet.

uint16\_t get\_TUN\_checksum(const uint8\_t\* packet);

Purpose: Extract the 16-bit checksum from TUN packet

Arguments:

packet: The completed TUN packet

Output:

uint16\_t: the 16-bit checksum.

uint16\_t derive\_TUN\_checksum(const uint8\_t\* packet);

Purpose: Given a TUN packet, will generate a checksum value.

Arguments:

packet: The completed TUN packet

Output:

uint16\_t: the 16-bit checksum.

uint8\_t get\_TUN\_payload(const uint8\_t\* packet, uint8\_t\* buff, uint8\_t buff\_sz);

Purpose: Given a TUN packet, will copy the payload into a new buffer.

Arguments:

packet: The completed TUN packet

buff: the pre-allocated buffer to store the TUN packet payload.

buff\_sz: the max size of the buff pre-allocated buffer.

Output:

uint8\_t: the 8-bit size of the payload.

uint8\_t construct\_lcd\_payload( const uint8\_t x, const uint8\_t y, const uint8\_t\* str,   const uint8\_t str\_sz, uint8\_t\* buff,

const uint8\_t buff\_sz);

Purpose: Constructs a TUN payload suited for the LCD service.

Arguments:

x, y: the location on the LCD screen to display the str message.

str: the C-style string that will be displayed on the LCD.

str\_sz: the size of the str.

buff: pre-allocated buffer to store LCD payload.

buff\_sz: the max size of buffer.

Output:

uint8\_t: the 8-bit size of the payload.

uint8\_t create\_TUN\_lcd\_packet(const boolean is\_local, const uint8\_t x,   const uint8\_t y, const uint8\_t\* str,

  const uint8\_t str\_sz, uint8\_t\* buff,

  const uint8\_t buff\_sz);

Purpose: Creates a complete LCD service TUN packet. Note: Please call this routine to create a LCD TUN service packet. The construct\_lcd\_payload(...) routine simply creates the LCD payload, it doesn't create the entire LCD service TUN packet like this routine does.

Arguments:

x, y: the location on the LCD screen to display the str message.

str: the C-style string that will be displayed on the LCD.

str\_sz: the size of the str.

buff: pre-allocated buffer to store LCD payload.

buff\_sz: the max size of buffer.

Output:

uint8\_t: the 8-bit size of the LCD TUN packet.

**Conclusion**

Util.cpp and util.h should not be considered complete or finished. It is in these two files that developers can add routines in which all services will be able to use. Take for instance the LCD service. The individual who developed the LCD service included new routines in the util.cpp to allow other services to easily use the LCD service to display messages. Therefore, the utilities can be considered something of a “commons area.” Whenever a new service needs to share routines with the other services, the new routines should be added to util.cpp and util.h.

**Appendix B – 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 B1: 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 B2: 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 not 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.