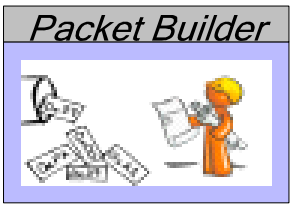
PacketBuilder
1.00

PacketBuilder_1



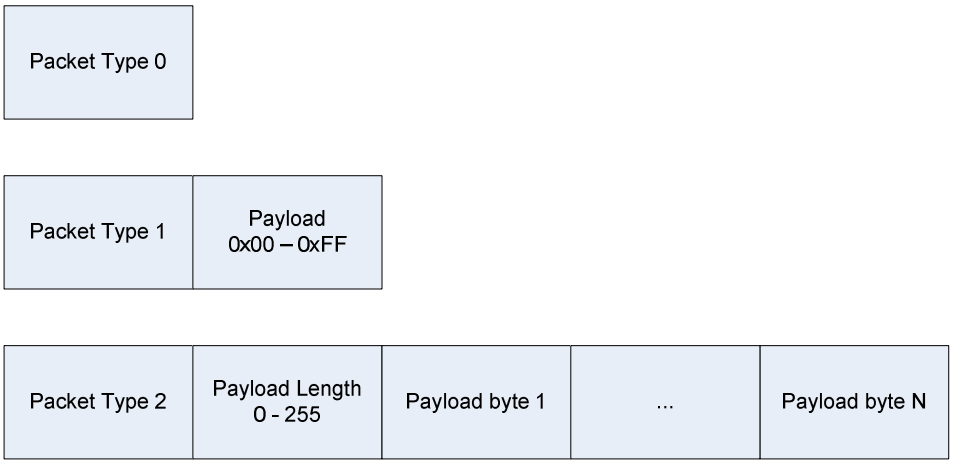
Features

- Adds a simple receiver protocol over any byte-by-byte interface
- 3 packet types (0 byte, 1 byte, and variable length up to 255 bytes)
- Easy to build your own protocol on top

General Description

The PacketBuilder component simplifies the task of receiving data over a byte by byte interface like a UART. The packet builder component receives data and constructs packets based on a simple packet header and data received as a payload for the packet. The user is informed when a fully formed packet is ready to be processed.

The PacketBuilder component provides a low level protocol for accomplishing this task in a simple and efficient manner. To begin, an introduction into the protocol is required. There are 3 packet types, each identified by first receiving a unique packet type identifier:



- Packet type 0 is identified by receiving a `PacketBuilder_1_PACKET_TYPE_0` identifier byte and has no payload. This packet is complete as soon as it receives the packet identifier byte. This packet is useful as a reset command or as a basic communication check.
- Packet type 1 is identified by receiving a `PacketBuilder_1_PACKET_TYPE_1` identifier byte and has a single byte payload. This packet is complete when the payload byte is received. This packet type is useful as a short and quick data packet or command packet.
- Packet type 2 is identified by receiving a `PacketBuilder_1_PACKET_TYPE_2` identifier byte and has a variable length payload packet (1 byte for payload size + up to 255 additional

bytes). After identifying the packet as a type 2 packet, the next byte received is the length of the packet payload in bytes. After receiving the packet length, the payload data is gathered. When all of the bytes in the payload have been received, the packet is complete.

The typical use case would be receiving data over a UART. As data is received, it is passed to the PacketBuilder to construct packets. When a completed packet has been formed, the PacketBuilder returns a non-zero value and the firmware can work on the data in the packet. The PacketBuilder is flexible in that it can work on single bytes at a time, or on larger chunks of received data. When single bytes are received, the PacketBuilder only needs to be called once per byte to determine if a packet has been completed. When more than one byte is received at a time, you may need to call the PacketBuilder multiple times to completely empty the receive buffer. Multiple complete packets may be received with a multiple byte reception and the PacketBuilder will return as soon as a packet is complete, leaving unprocessed data in the receive buffer. To empty the receive buffer, simply keep calling the PacketBuilder until the receive buffer no longer contains data.

Any garbage bytes that precede a valid packet type identifier are ignored. There is no error checking on any of the received data. You can add a checksum or error correction code to a packet type 2 by writing your own protocol layer on top of this very low level protocol.

Parameters and Setup

None

Application Programming Interface

Application Programming Interface (API) routines allow you to configure the component using software. The following table lists and describes the interface to each function. The subsequent sections cover each function in more detail.

By default, PSoC Creator assigns the instance name "PacketBuilder_1" to the first instance of a component in a given design. You can rename it to any unique value that follows the syntactic rules for identifiers. The instance name becomes the prefix of every global function name, variable, and constant symbol.

Function	Description
uint8 PacketBuilder_1_BuildPacket(uint8 snippet_buffer[], uint8 * snippet_length, uint8 packet_buffer[], uint8 * packet_length)	This function takes incomplete snippets of received data from an interface and builds them into complete packets.

uint8 PacketBuilder_1_BuildPacket(uint8 snippet_buffer[], uint8 * snippet_length, uint8 packet_buffer[], uint8 * packet_length)

Description: This function takes snippets of received data from an interface and builds them into complete packets. The **snippet_buffer[]** is filled by you, the user, from data received by your interface. Data received from your interface is placed into the **snippet_buffer[]** and the number of bytes received is written into **snippet_length**. When **PacketBuilder_1_BuildPacket(…)** is called, the **snippet_buffer[]** is emptied of its information and the data is transferred to the **packet_buffer[]**. **snippet_length** is decremented for every byte transferred. When a complete packet is received or the **snippet_length** reaches zero, the function will return. The return value of the function will be non-zero to indicate that **packet_buffer[]** contains a complete packet of length **packet_length**.

Parameters:

- **uint8 snippet_buffer[]** – this is user updated array of bytes received by an interface. It is the raw, unprocessed data from the source.
- **uint8 * snippet_length** – this is a pointer to a uint8 variable that holds the number of unprocessed bytes currently in the **snippet_buffer[]**. This value will be set by you when you receive data to be processed, and it will be modified by the function as it process the data received. If you intend to add data to the **snippet_buffer[]** before **snippet_length** has reached zero, care must be taken to prevent unprocessed data from being overwritten by appending the new data on the end of the unprocessed data and adding the appropriate number of bytes to the **snippet_length**. In this case, **snippet_length** must be increased by the number of new bytes, not simply overwritten.
- **uint8 packet_buffer[]** – this is the array where completed packets will be stored. Incomplete packets will also be stored in this array until they have received all their information. Care must be taken to ensure that the **packet_buffer[]** is of sufficient size to hold the largest expected packet.
- **uint8 * packet_length** – this is a pointer to a uint8 variable that holds the number of bytes currently in the **packet_buffer[]**. When the function returns a non-zero value indicating a complete packet is ready, the **packet_buffer[]** will contain a completed packet of length **packet_length** bytes. When you have processed all the information in the **packet_buffer[]**, reset the value of **packet_length** to zero before calling the **PacketBuilder_1_BuildPacket(…)** function again.

Return Value: **uint8 packet_complete** – returns a non-zero value if a completed packet is ready to be processed. Be sure to reset **packet_length** back to zero before processing the next packet.

Side Effects: None

Defines

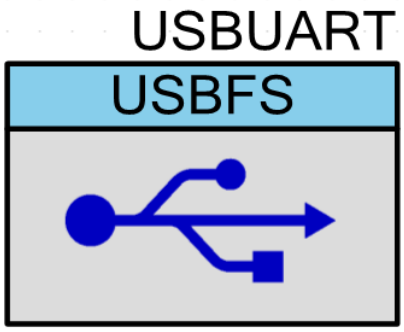
The symbols for each packet type are:

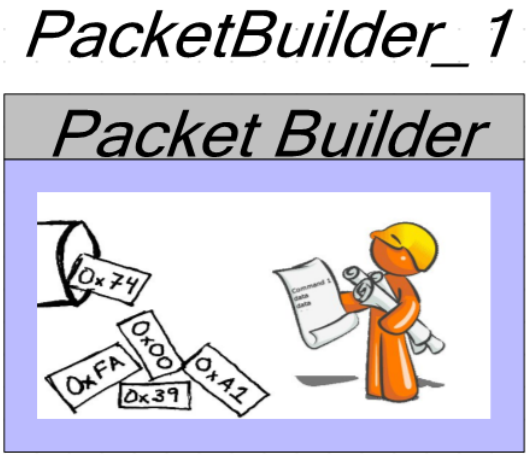
```
#define PacketBuilder_1_PACKET_TYPE_0 '0'
#define PacketBuilder_1_PACKET_TYPE_1 '1'
#define PacketBuilder_1_PACKET_TYPE_2 '2'
```

These can be found in the PacketBuilder_1_PacketBuilder.h file

Sample Firmware Source Code

The following example code shows the packet builder in a real world example. Data was sent from a computer to control the position of two servos. The servo position information was sent as two 16 bit values. These values were stored as position variables for the right and left servo. A special command was sent to indicate when the servo positions should be updated with the values stored in the variables. The schematic consisted of:






```
// ***---------------  example code explanation: -------------------------***
/*
This code uses a USBUART component as the com interface.  The packet structure is as follows:

A packet type 0 (immediate packet) was used as a connection check.  If a packet 0 is received,
the code returns a '!' character to indicate the com interface is functioning and confirms the
packet builder interface is running as expected.

A packet type 1 (short form packet) was used to send a command: when packet 1 is received with
a parameter of 0x00, write the servo position variables into the PWM.  This allows the servo
variables to be updated separately, then both servo positions can be updated at the same time
by issuing a command.

A packet type 2 (long form packet) was used to update two unsigned 16 bit variables.  First, a
long form packet was issued with a length of 3 bytes, then a 'l' (for "left") or a "r' (for
"right") was sent, followed by an unit16, sent MSB first.  (1 byte for a left servo / right
servo indicator, and 2 bytes for the uint16 = 3 bytes total)

The main loop polled to see if any data had come in on the USBUART.  If it had, it dispatched
what it received to the packet builder. The packet builder was called and an if() statement
checked to see if a packet had been completed.  If the packet builder indicated that a packet
was complete, another if statement processed the packet, then reset packet_length and
packet_complete to indicate the packet had been processed.  It repeated this process until the
snippet_length was zero.
*/
// ------------------ begin example code --------------------//
```

```
#include <device.h>

uint8 packet_buffer[32]; // buffer to hold complete packets
uint8 packet_length=0;

uint8 snippet_buffer[8]; // buffer to hold incomplete snippets from the USBUART
uint8 snippet_length = 0;

char szBuffer[8];  // buffer to hold replies to the PC

void main()
{
        uint8 packet_complete; // flag to indicate a completed packet
        uint16 Left_Servo = 36000; // initial servo positions
        uint16 Right_Servo = 36000;

        PWM_Start();
        PWM_WriteCompare1(Left_Servo); // generate the signal for initial servo positions
        PWM_WriteCompare2(Right_Servo);

        CyGlobalIntEnable;

        USBUART_Start(0, USBUART_5V_OPERATION);
        while(!USBUART_bGetConfiguration()); /* Wait for Device to enumerate */
        USBUART_CDC_Init();


    for(;;)
    {

        if(USBUART_DataIsReady() > 0) // DataIsReady returns a non-zero value when data has arrived
        {
                snippet_length = USBUART_GetCount(); //l

                USBUART_GetAll(snippet_buffer);

        }

        // process any completed packets
        while(snippet_length > 0)
        {
                packet_complete = PacketBuilder_1_BuildPacket(snippet_buffer, &snippet_length,
packet_buffer, &packet_length);
                if(packet_complete)
                {
                        if(packet_buffer[0] == PacketBuilder_1_PACKET_TYPE_0)
                        {
                        // send back the reset reply
                                sprintf(szBuffer,"!");
                                USBUART_PutString(szBuffer);
                                while(USBUART_CDCIsReady() == 0);
                        }
                        else if(packet_buffer[0] == PacketBuilder_1_PACKET_TYPE_1)
                        {
                        // packet type 1 with parameter of "0" is the "write servo positions" command
                                if(packet_buffer[1] == 0)
                                {
                                        PWM_WriteCompare1(Left_Servo);
                                        PWM_WriteCompare2(Right_Servo);
                                }
                        }
                        else if(packet_buffer[0] == PacketBuilder_1_PACKET_TYPE_2)
                        // packet type 2 with parameter 'l' or 'r' for left or right indicate a 2 byte
                        uint16 is attached
                        {
                                if(packet_buffer[2] == 'l')
                                {
```

```
                                Left_Servo = (((uint16) packet_buffer[3]) << 8) | packet_buffer[4];
                        }
                        else if(packet_buffer[2] == 'r')
                        {
                                Right_Servo = (((uint16) packet_buffer[3]) << 8) | packet_buffer[4];
                        }
                }

                packet_length = 0; // make sure to reset the packet length!
                packet_complete = 0; // and clear the packet complete flag, since we have finished
processing the packet
            }
        }

}
```