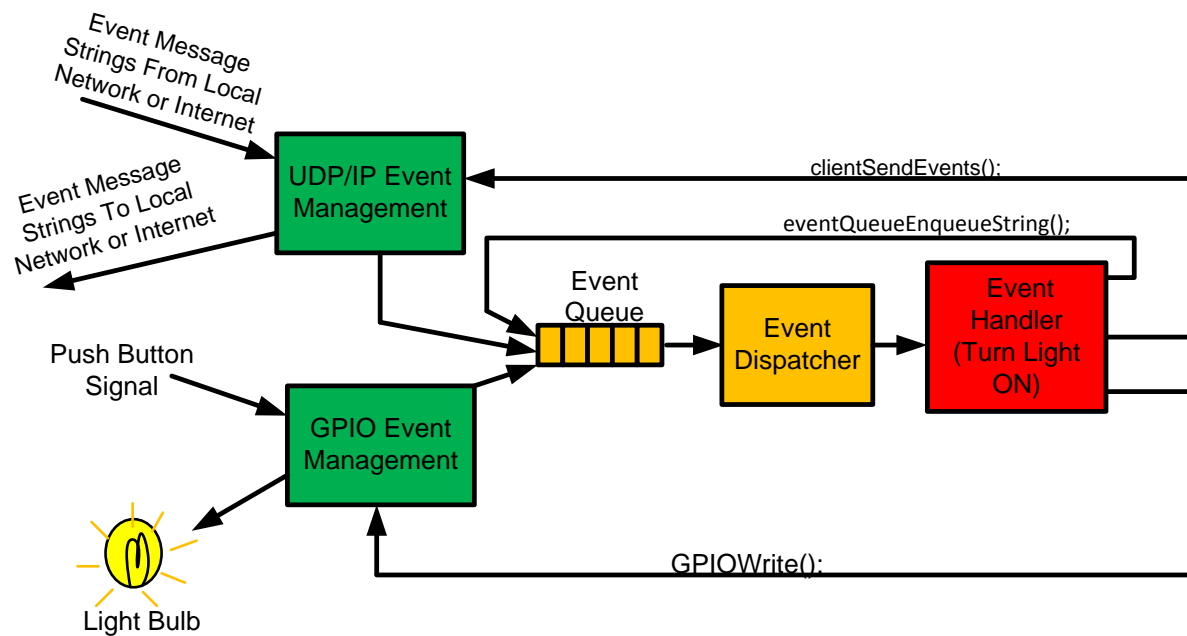


Event Dispatcher Framework Library

Version 1.01



Contents

- 1.1 Introduction
- 1.2 Structure of an Event Dispatcher Framework Library base Application
- 2.1 Event Dispatcher Management
- 2.2 Event Messages
- 2.3 User Defined Behaviors
- 2.4 Event Dispatcher Management API
- 2.5 Sample Program using the Event Dispatcher Management API
- 3.1 GPIO Event Management
- 3.2 GPIO Event Messages
- 3.3 GPIO Event Message API
- 3.4 Sample Program With GPIO Event Management
- 4.1 UDP Event Management
- 4.2 UDP Event Management API
- 4.3 Sample Program with UDP Event Management
- 4.4 Testing the UDP Event Message String Test Program
- 5.1 Time Event Management

Appendix A Compiling Event Dispatcher Framework Applications

- A.1 Introduction
- A.2 Building for a Raspberry Pi tar

1.1 Introduction

The purpose of the Event Dispatcher Framework Library is to provide a generic framework for developing specific purpose event driven control software. This framework was written to have generic control functionality that can be extended by adding user code. The library provides optional UDP/IP network functionality suitable for inter process communication, distributed processing and web interfacing. The framework was first developed for use in embedded system based hardware projects designed and built by students; therefore it was designed with simplicity in mind. It is also cross platform and may be implementing on extremely resource constrained systems. The framework is provided for free as an open source library for any purpose under a BSD like License.

The library is made up of one block for event dispatcher management and three optional blocks for GPIO event management, time event management and UDP/IP event management. Each of these blocks may be used in an application by including the appropriate headers in the application code.

1.2 Structure of an Event Dispatcher Framework Library base Application

An event dispatcher framework based application is a made up of a collection of user defined event handlers which are managed by the Event Dispatcher Framework library, together forming an event driven application. In this style of application, event messages that are generated as the result of I/O, time based and network events cause the event dispatcher to select the appropriate user defined event handlers (functions) to react to each event message.

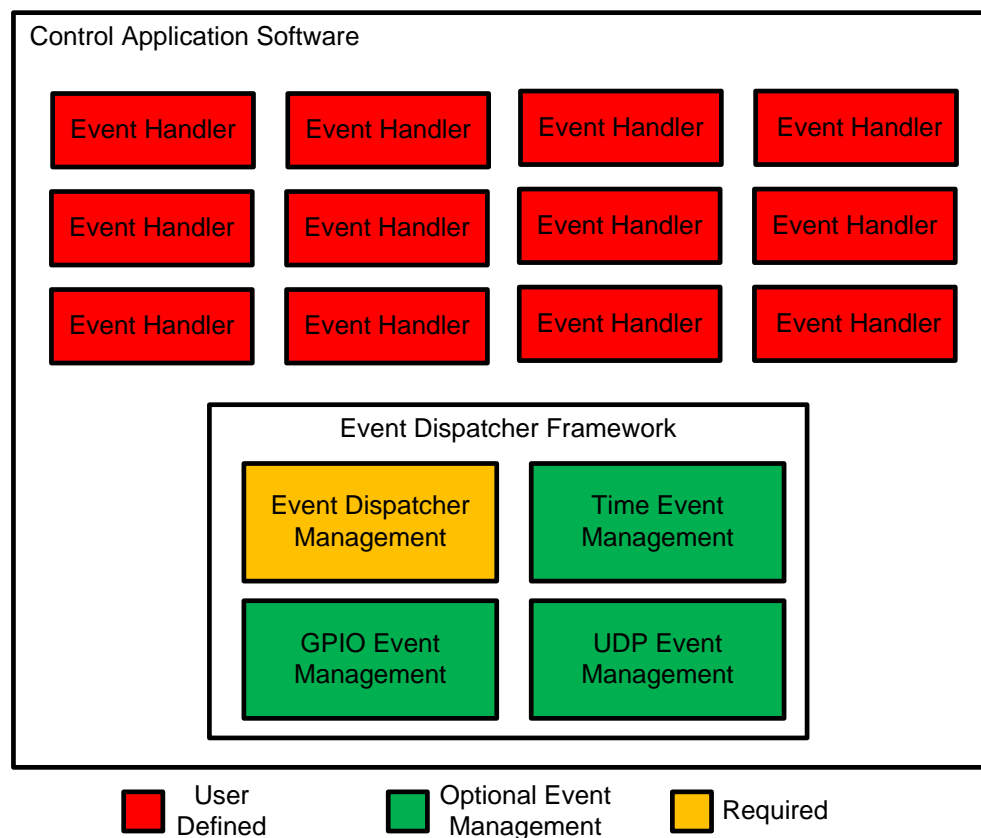


Figure 1.1, Structure of an Event Dispatcher Framework Based Application

2.1 Event Dispatcher Management

Event dispatcher management works by queuing event messages in an event queue, then allowing the dispatcher to match the first event message in the queue to an event handler using a table. Each time the dispatcher finds an event message in the queue that matches an event handler, the dispatcher calls the matching event handler to react or process the corresponding event message.

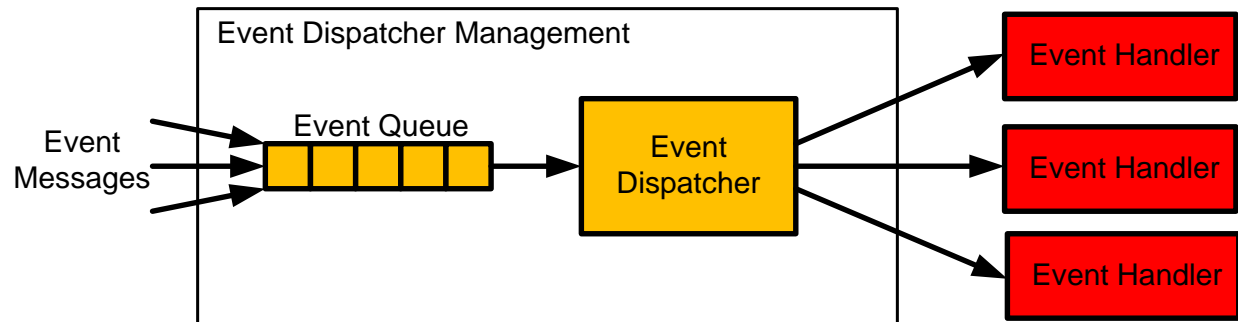


Figure 2.1, Event Dispatcher Management

2.2 Event Messages

In this framework Event Messages are alpha-numeric strings which are generated whenever an event occurs. When an event message string is generated it is pushed into the event queue where it waits its turn to be handled by an event handler. Some examples of events that may occur include key strokes, pushbutton events, external sensor events and timed events.

2.3 User Defined Behaviors

The behavior of the event handlers is defined by the user in source code. When an event occurs, such as I/O, network and timed events, the event's handlers may respond to the event messages by processing information, acting on output signals or even generating new events, including network events.

Example 2.1 The drawing below depicts the how a piece of event driven software may be structured to switch on a light when it receives an event message telling it to do so.



Figure 2.2, Event Driven System to Switch on Light

2.4 Event Dispatcher Management API

The API for Event Dispatcher Management is made up of the following functions:

eventQueueInit();

The above function is called to initialize (clear) the event queue.

eventHandlerFrameworkInit();

The above function is called to initialize the table that stores the handler function pointers and matching event message pairs.

addEventHandler("<event message string>", <handlerFunctionPtr>);

The above function adds event message string / handler function pointer pairs to a table which tells the dispatcher which handlers should be called in response to each event message type. Note that more than one handler may respond to each event and that each handler may respond to more than one type of event.

eventQueueEnqueueString("<event message string>");

Whenever an event is detected by the application software and event message string is generated, the resulting event string should be placed onto the event queue using this function to give the event dispatcher and event handler an opportunity to respond to the event message.

eventHandlerDispatcher();

This function is intended to sit inside of an infinite loop inside the application software once all of the initialization is done. Each time the above function is called it checks to see if there are any event messages in the event queue. If there is a message, then the message is matched to any messages in the dispatchers table that corresponds to an event handler function causing that event handler to run.

eventQueueEmpty();

The above function returns a non-zero value if the event queue is not empty.

2.5 Sample Program using the Event Dispatcher Management API

The sample test program below demonstrates how to initialize event dispatcher management, how to add event handlers, how to push an event message string onto the event queue and how the event handlers get dispatched. Once initialized, the sample test program sits in a loop that waits for a user to type in an event message string that gets placed in the event queue, causing the correct event handler to be called once `eventHandlerDispatcher()` to be called:

```
/* maintest1.c */
#include <stdio.h>
#include <string.h>
#include "ef.h"

void testeh1();
void testeh2();

int main()
{
    char eventMesageString[EVENTSIZEMAX];

    /* Initialize event queue and event dispatcher. */
    eventQueueInit();
    eventHandlerFrameworkInit();

    /* Add event handler / event message string pairs. */
    addEventHandler("HELLO", testeh1);
    addEventHandler("BYE", testeh2);

    while(1)
    {
        /* Enter an event message string at the keyboard. */
        printf("Type in HELLO or BYE.\n");
        scanf("%s", eventMesageString);

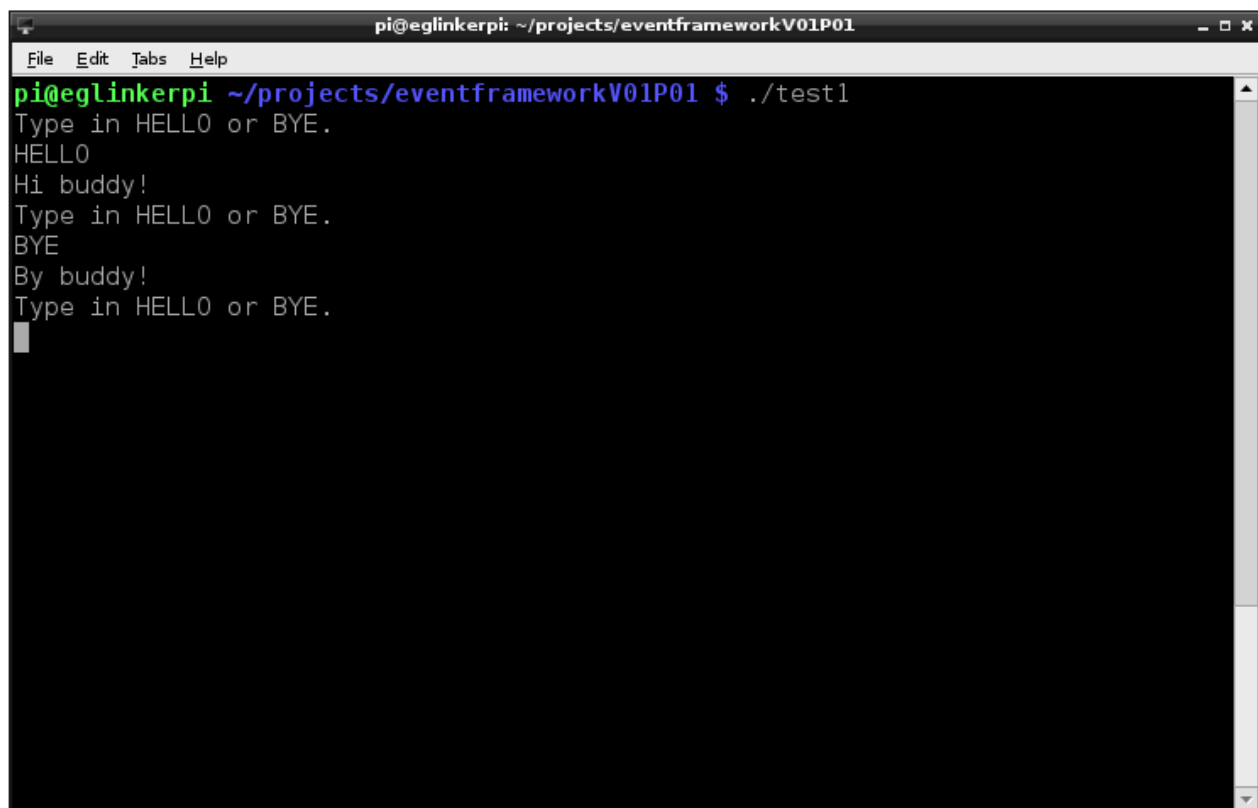
        /* Enqueue the event message string. */
        eventQueueEnqueueString(eventMesageString);

        /* Keep looping and dispatching events until */
        /* event queue is empty. */
        while(!eventQueueEmpty())
        {
            eventHandlerDispatcher();
        }
    }
    return(1);
}
```

```
void testeh1() /* Event handler that responds to HELLO. */
{
    printf("Hi buddy!\n");
    return;
}

void testeh2() /* Event handler that responds to BYE. */
{
    printf("By buddy!\n");
    return;
}
```

When this program is run in a terminal window it looks something like this:



```
pi@eglinkerpi: ~/projects/eventframeworkV01P01
File Edit Tabs Help
pi@eglinkerpi ~/projects/eventframeworkV01P01 $ ./test1
Type in HELLO or BYE.
HELLO
Hi buddy!
Type in HELLO or BYE.
BYE
By buddy!
Type in HELLO or BYE.
█
```

3.1 GPIO Event Management

GPIO event management occurs in the Event Dispatcher Framework Library whenever GPIO pins are initialized as inputs or output, controlled or to generate GPIO event messages using the Event Management API. Once a GPIO signal is configured as an input or an output, any change in its logic state causes a GPIO event message to be automatically be put on the event queue, this way changes in the input or output state can be handled by an event handler.

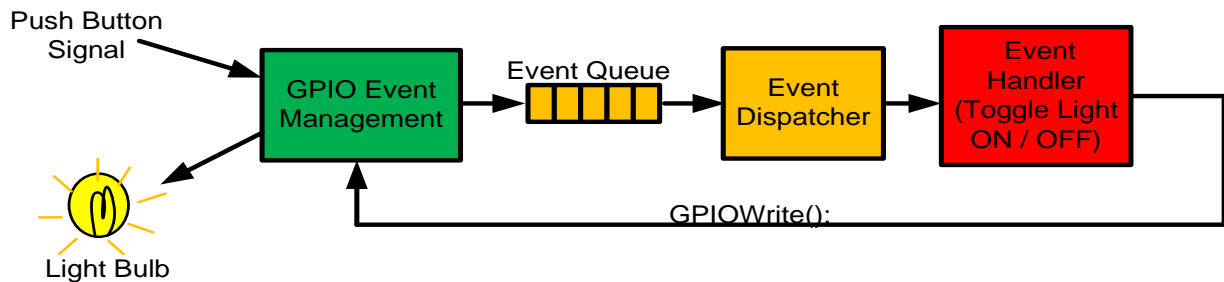


Figure 3.1, GPIO Event Management Example

3.2 GPIO Event Messages

There are two types of GPIO event messages, rising edge event messages and falling edge event messages. For rising edge events, the event message strings begin with the string GPIORISE followed by the numeric value of the GPIO as shown in the examples below:

GPIORISE0, GPIORISE1, GPIORISE2,

For falling edge events, the event message strings begin with the string GPIOFALL with the numeric value of the GPIO as shown in the examples below:

GPIOFALL0, GPIOFALL1, GPIOFALL2,

3.3 GPIO Event Message API

The GPIO event message API is cross platform. The API is made up of the following functions:

GPIOInit();

This function initializes the GPIO Event Management Library.

GPIOOutputMode(<GPIO #>);

This function causes a GPIO pin to be initialized as an output. It also sets up the library to automatically generate a GPIORISE or GPIOFALL event message whenever the output pin state is changed.

GPIOWrite(<GPIO #>, <value>);

Forces a specified GPIO pin to change from the current state to the specified state.

GPIOInputMode(<GPIO #>);

This function causes a GPIO pin to be initialized as an input. It also sets up the library to automatically generate a GPIORISE or GPIOFALL event message whenever the input pin state is changed.

int GPIORead(<GPIO #>);

Reads the current state of the specified GPIO and returns its value.

3.4 Sample Program With GPIO Event Management

This sample program toggles GPIO pin 6 on or off whenever the input signal at pin 5 rises from LOW to HIGH. See Figure 3.1 for pictorial representation.

```
#include <stdio.h>
#include <string.h>
#include "ef.h"

void GPIO5RisingEventHandler();

int main()
{
    /* Event dispatcher init */
    eventQueueInit();
    eventHandlerFrameworkInit();

    GPIOInit(); /* GPIO init */

    /* Initialize GPIO6 as output and switch to off. */
    GPIOOutputMode(6);
    GPIOWrite(6, 0);

    GPIOInputMode(5); /* Initialize GPIO5 as input. */

    /* This installs the handler that flicks GPIO ON or OFF */
    /* whenever the GPIO 5 rising event occurs. */
    addEventHandler("GPIORISE5", GPIO5RisingEventHandler);

    while(1)
    {
        /* Check to see if on of the initialized GPIO pins */
        /* change state, then automatrally adds a GPIO rising */
        /* or GPIO falling event message to the event queue occurs. */
        GPIOEdgeEventDetect();

        /* Keep looping and dispatching events if event queu */
        /* is not empty. */
        while(!eventQueueEmpty())
        {
            eventHandlerDispatcher();
            printf("Event log: %s\n", auxEventString);
            fflush(stdout);
        }
        delay(10);
    }

    return(1);
}
```

```
/* This is the event handler that flips the state of GPIO6 */  
/* whenever the GPIO5 experiences a rising edge event. */  
void GPIO5RisingEventHandler()  
{  
    if(GPIORead(6))  
    {  
        GPIOWrite(6, 0);  
    }  
    else  
    {  
        GPIOWrite(6, 1);  
    }  
    return;  
}
```

4.1 UDP Event Management

UDP event management allows applications based on the Event Dispatcher Framework Library and other UDP/IP enabled applications to exchange event message strings. Messages can be exchanged between programs on the local host, across a network or across the internet. Each program that uses UDP Event Management opens a UDP/IP socket port when it is initialized. Once initialize, the application receives all incoming event messages through this single server port and automatically places the event message strings into the event message queue.

The figure below shows an example of how Event Dispatcher Framework Library base programs may be integrated locally and across a network :

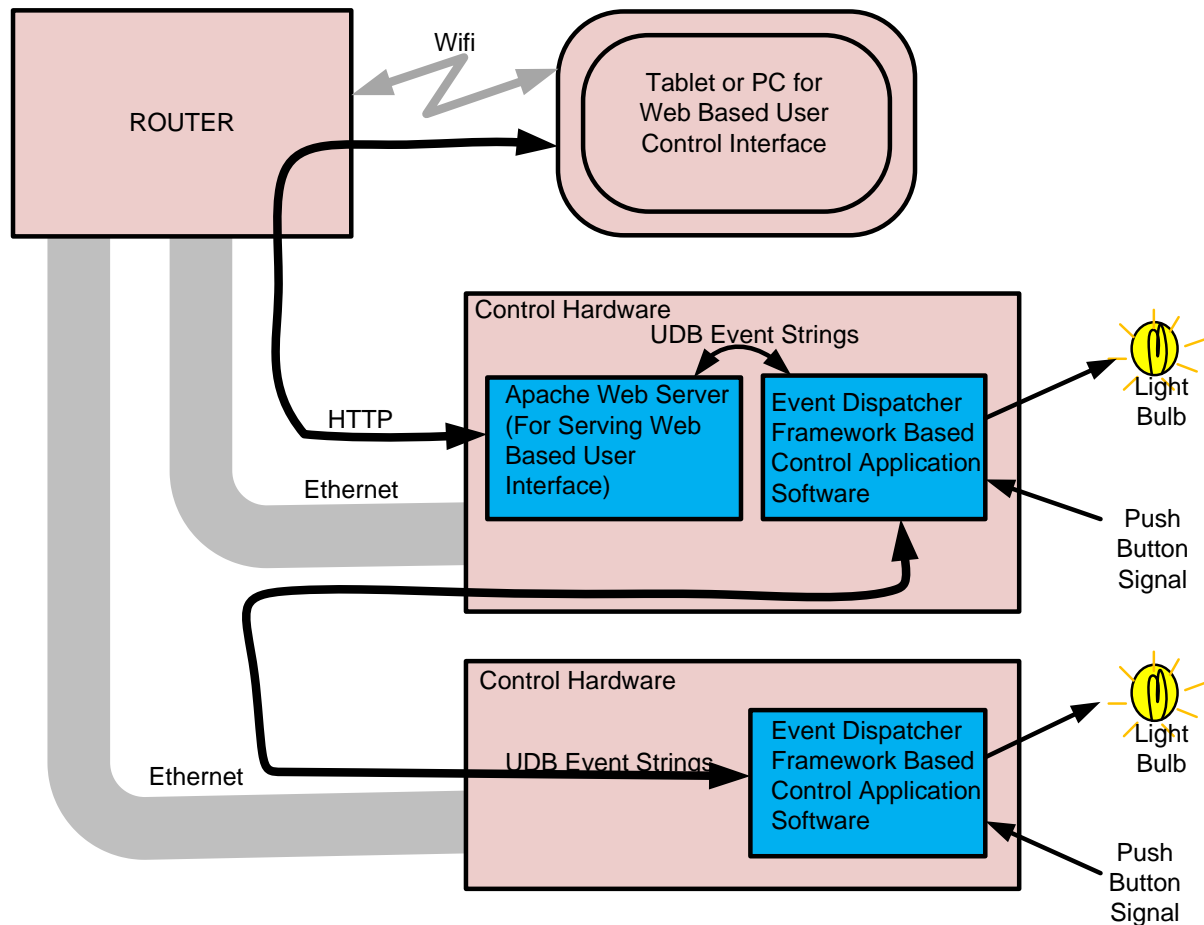


Figure 4.1, UDP Event Management and Integration of Control Software Across Networks

The figure below shows an example of how UDP Event Management is related to the rest of a control application program which is developed using the Event Dispatcher Framework Library. For the sake of simplicity only a single event handler is shown in the diagram, in practice an unlimited number of event handlers may be used in a single control application program:

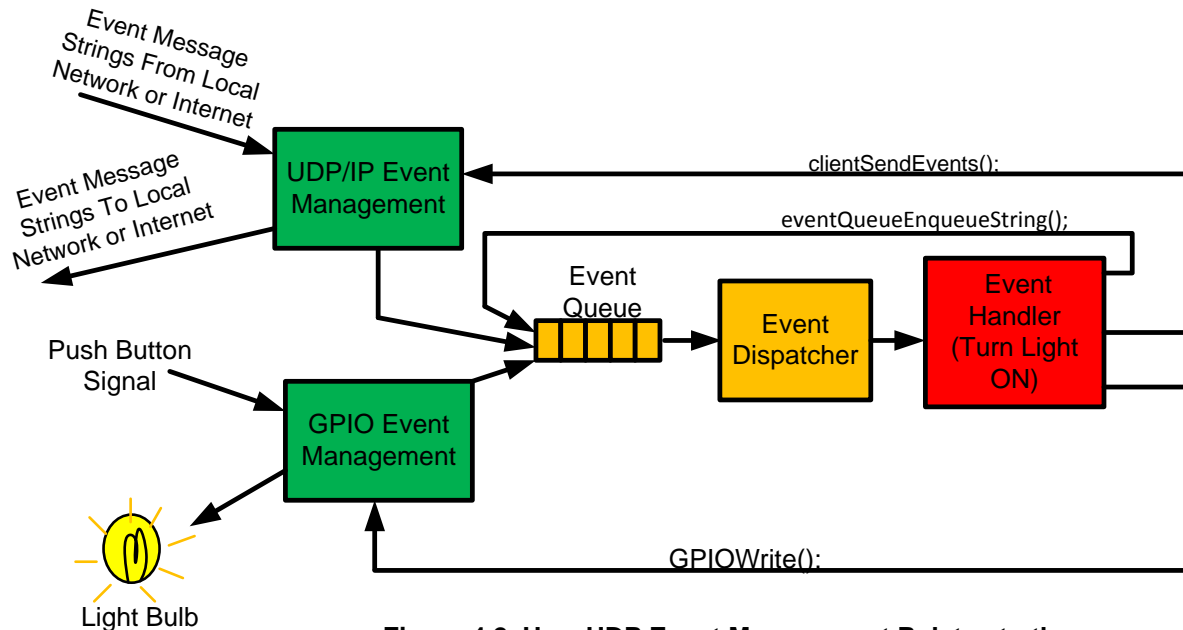


Figure 4.2, How UDP Event Management Relates to the Rest of an Application Program Internally

4.2 UDP Event Management API

The UDP event management API has the smallest number functions of all the optional blocks in the framework:

UDPEventsInit(<port #>);

This function is called to initialize UDP event management and to select the port number of the application program server socket where it will receive UDP event packets from. Once the UDP event management block is initialized it is immediately ready to begin receiving UDP Event Message Strings.

serverRecieveEvents();

This function should be placed in the same infinite loop as the event dispatcher, `eventHandlerDispatcher()`. The purpose of this function is to process the incoming UDP event message strings and place them into the event queue.

clientSendEvents("<Destination IP Address>", <Destination Port>, <event string>);

This function is used to send a UDP event string to another application program running on the local host, across a net work or on the internet. The packet is sent through a client socket port which was initialized at the same time as the server socket.

4.3 Sample Program with UDP Event Management

This sample program can use UDP event strings received through the server socked port to control a light attached to GPIO6, it can also control the light through a rising edge event caused by a bush button attached to GPIO5. It can also start up an mp3 player when commanded to do so by UDP event string:

- GPORISE5 - Event from GPIO5 that causes light to toggle.
- LED TOGGLE - UDP event string that causes light to toggle.
- LED ON - UDP event string that causes light to turn on.
- LED OFF - UDP event string that causes light to turn off.
- PLAY MUSIC - UDP event string that causes an mp3 file to play.

```
#include <stdio.h>
#include <string.h>
#include "ef.h"
#include "lcdwrapper.h"
#include "UDPEvents.h"

void GPIO5RisingEventHandler();
void LEDOff();
void LEDOn();
void playMusic();

int main()
{

    char tempString[EVENTSIZEMAX];
    char testString[64];
    int i;

    printf("Event queue init\n");
    eventQueueInit();

    printf("Event dispatcher init.\n");
    eventHandlerFrameworkInit();

    printf("GPIO init\n");
    GPIOInit();

    printf("UDP Event init\n");
    UDPEventsInit(8888);

    clientSendEvents("127.0.0.1", 45000, "TESTEVENT");

    GPIOOutputMode(6);
    GPIOWrite(6, 0);

    GPIOInputMode(5);
```

```

printf("Start event dispatcher test.\n");

addEventHandler("GPIORISE5", GPIO5RisingEventHandler);
addEventHandler("LED TOGGLE", GPIO5RisingEventHandler);
addEventHandler("LED ON", LEDOn);
addEventHandler("LED OFF", LEDOff);
addEventHandler("PLAY MUSIC", playMusic);

while(1)
{
    GPIOEdgeEventDetect();

    serverRecieveEvents();

    while(!eventQueueEmpty())
    {
        eventHandlerDispatcher();
        printf("Event log: %s\n", auxEventString);
        fflush(stdout);
    }
    delay(10);
}

printf("End event dispatcher test.\n");

return(1);
}

void GPIO5RisingEventHandler()
{
    if(GPIORead(6))
    {
        GPIOWrite(6, 0);
        clientSendEvents("127.0.0.1", 45000, "LED OFF");
    }
    else
    {
        GPIOWrite(6, 1);
        clientSendEvents("127.0.0.1", 45000, "LED ON");
    }
    return;
}

```

```

void LEDOn()
{
    GPIOWrite(6, 1);
    return;
}

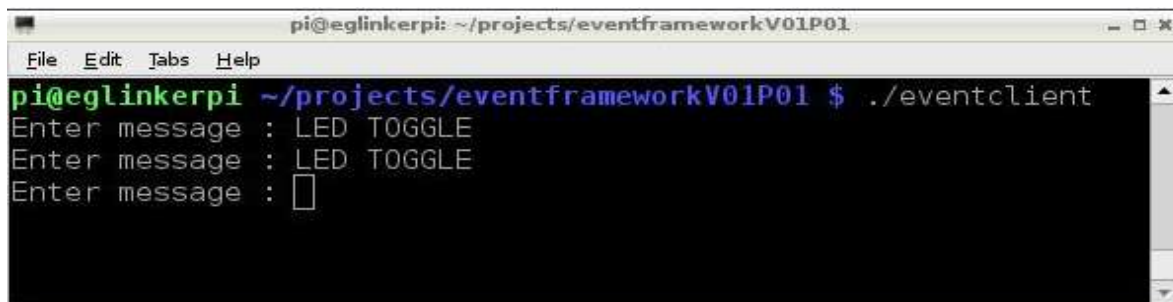
void LEDOff()
{
    GPIOWrite(6, 0);
    return;
}

void playMusic()
{
    system("ffplay Thirtyfive.mp3 &");
    return;
}

```

4.4 Testing the UDP Event Message String Test Program

A separate client software program to send UDP/IP message string packets and a separate UDP/IP event logger are required to full test the above test program. The figure below shows how an event client that was used to send two test messages to the above Event Dispatcher Framework Library based test program.

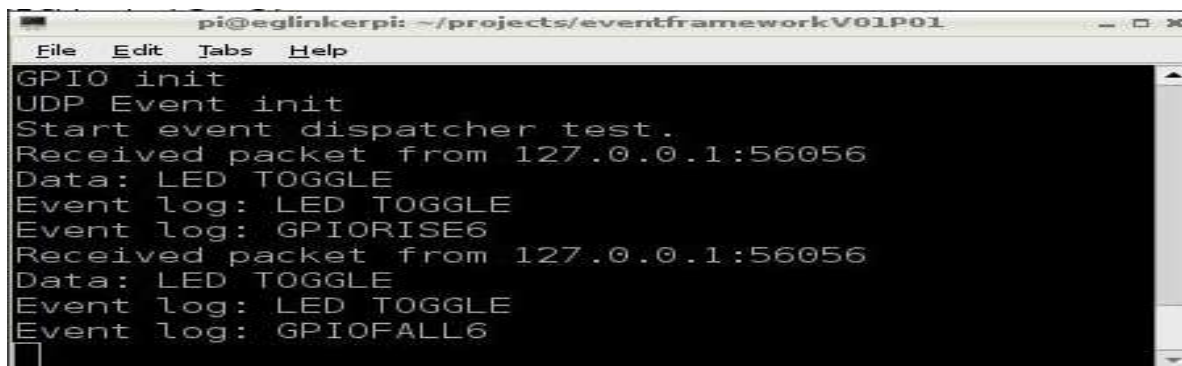


```

pi@eglinkerpi: ~/projects/eventframeworkV01P01
File Edit Tabs Help
pi@eglinkerpi ~/projects/eventframeworkV01P01 $ ./eventclient
Enter message : LED TOGGLE
Enter message : LED TOGGLE
Enter message : 

```

The figure below shows how the test program initializes and reacts to receiving the event message strings.

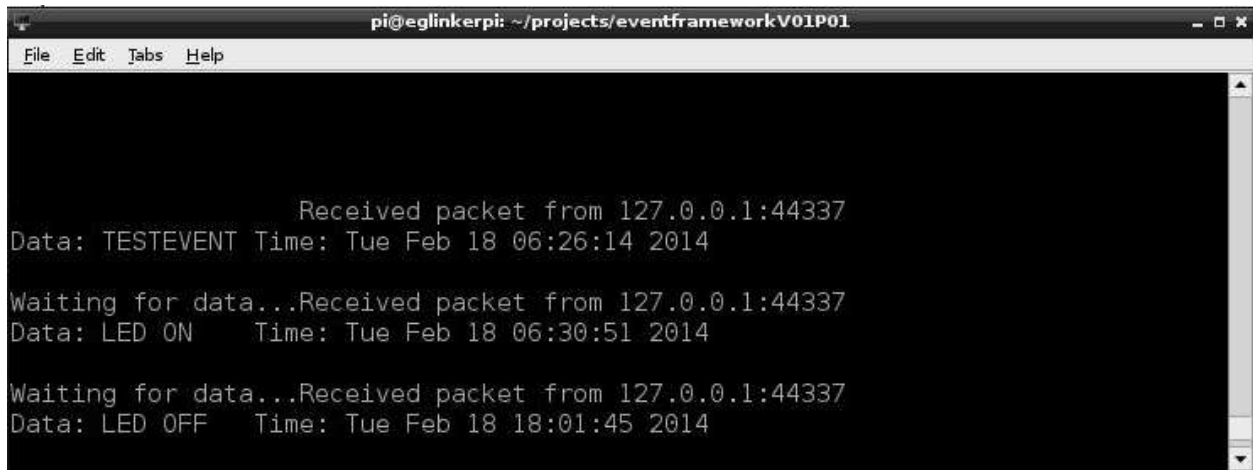


```

pi@eglinkerpi: ~/projects/eventframeworkV01P01
File Edit Tabs Help
GPIO init
UDP Event init
Start event dispatcher test.
Received packet from 127.0.0.1:56056
Data: LED TOGGLE
Event log: LED TOGGLE
Event log: GPIORISE6
Received packet from 127.0.0.1:56056
Data: LED TOGGLE
Event log: LED TOGGLE
Event log: GPIOFALL6

```

The figure below shows how the UDP event logger reacts to the event message string packets received from the example test program, this verifies that the test program does in fact successfully transmit the event message strings using UDP event message string functionality.

A screenshot of a terminal window titled "pi@eglinkerpi: ~/projects/eventframeworkV01P01". The window has a menu bar with "File", "Edit", "Tabs", and "Help". The terminal output shows three received packets from 127.0.0.1:44337. The first packet contains "Data: TESTEVENT Time: Tue Feb 18 06:26:14 2014". The second packet contains "Data: LED ON Time: Tue Feb 18 06:30:51 2014". The third packet contains "Data: LED OFF Time: Tue Feb 18 18:01:45 2014". Each packet is preceded by the text "Waiting for data...".

```
pi@eglinkerpi: ~/projects/eventframeworkV01P01
File Edit Tabs Help

Received packet from 127.0.0.1:44337
Data: TESTEVENT Time: Tue Feb 18 06:26:14 2014

Waiting for data...Received packet from 127.0.0.1:44337
Data: LED ON Time: Tue Feb 18 06:30:51 2014

Waiting for data...Received packet from 127.0.0.1:44337
Data: LED OFF Time: Tue Feb 18 18:01:45 2014
```


5.1 Time Event Management

Time event management is not currently implemented.

Appendix A

Compiling Event Dispatcher Framework Applications

A.1 Introduction

The Event Dispatcher Framework Library can be used in applications by linking your application to your own source code when it is compiled. Alternatively, the source code of the library can simply be added to your project and compiled together with your, which is recommended. The library is cross platform targeted, so make sure that the correct source files or libraries are used with the selected target.

A.2 Building for a Raspberry Pi target with Raspberrian Linux OS using Library Source.

Note that on the Raspberry Pi target with Raspberrian OS the library is dependent on the wiringPi library, so make sure that it is installed. To compile an application in a single line at the command prompt that does not use UDP Event Management, including the library source, type the following:

```
gcc -o <ouput file name> ef.c gpio.c <application source files.c> -lwiringPi
```

In the case of the first two test example application in this manual type the following:

```
gcc -o test1 ef.c gpio.c maintest1.c -lwiringPi
```

and

```
gcc -o test2 ef.c gpio.c maintest2.c -lwiringPi
```

To compile an application in a single line at the command prompt that does use UDP Event Management, including the library source, type the following:

```
gcc -o <ouput file name> ef.c gpio.c UDPEvent.c <application source files.c> -lwiringPi
```

In the case of the third test example application in this manual (using UDP Event Management) type the following:

```
gcc -o test3 ef.c gpio.c UDPEvent.c maintest3.c -lwiringPi
```

