

Abstract

Project: eZ2944

BMZ is an operating system for embedded microprocessors. BMZ is a brand new operating system, developed from scratch exclusively for the Zilog 2004 Flash Nets competition. BMZ is an acronym for “Bare Metal Zilog”. The name accentuates both the nature of BMZ as a tool for programmers who are faced with the special challenges of programming on bare metal, and the microprocessor family that hosts this first version of BMZ. BMZ seeks to provide developers of simple embedded applications with a simple yet powerful and complete set of operating system services, and a framework to organize their work

A sample terminal server application has been developed to show off the features of BMZ. The application was chosen to avoid custom hardware, but at the same time to make full use of the resources built into the Zilog eZ80 F91 processor. The application performs the useful task of making the F91’s UARTs available as resources to remote users who connect using the F91’s ethernet hardware and a TCP/IP stack implemented in software using BMZ.

The remote users use any telnet application on their networked desktop computer to access the terminal server. Once the telnet user has connected to the terminal server, they interact with equipment connected to the UART they have selected in exactly the same way as if they were directly connected to that equipment with an asynchronous terminal. The terminal server effectively connects up to two items of legacy equipment to an ethernet network.

Of course the terminal server is capable of serving two different users who are independently accessing different UARTs simultaneously.

The terminal server runs entirely within the F91’s on-chip memory, no external hardware is required beyond an ethernet PHY and RS-232 line drivers.

Although the terminal server is a useful enough application, it serves only as a convenient demonstration of the BMZ operating system. BMZ is the real focus of this project.

BMZ is more than an operating system, it is also an embedded framework. A framework provides application writers with a methodical way of developing their applications in accordance within a well understood pre-existing structure. The idea is to free application writers to focus on developing the unique functionality required by their application. The framework supports the unique functionality by providing a comfortable place for it to live. One way to think about framework based software development is that it involves an adjustment from working in a primarily “I call the system” mode, to a primarily “The system calls me” mode.

Frameworks are a very popular part of desktop software development, BMZ seeks to bring a simple and yet powerful framework approach to development of embedded applications with the Zilog eZ80F91 processor.

A BMZ system comprises a set of independent tasks. Tasks run in response to events, each task is effectively a state machine that performs its processing only when an event of interest to it occurs. The application writer specifies the number and names of the tasks, and writes event handlers for the tasks. The framework instantiates the tasks and delivers the events to be processed to them. BMZ is designed so that the application writer specifies the behavior of the system in a direct and straightforward manner. There is no need to write “boilerplate code”. Boilerplate code is characteristic of many desktop framework systems. Typically, vendors of these systems deliver “Wizards” to automate writing the tedious boilerplate code. Instead of this approach, BMZ seeks to keep everything that is not unique to the application out of the way of the application programmer and inside BMZ itself.

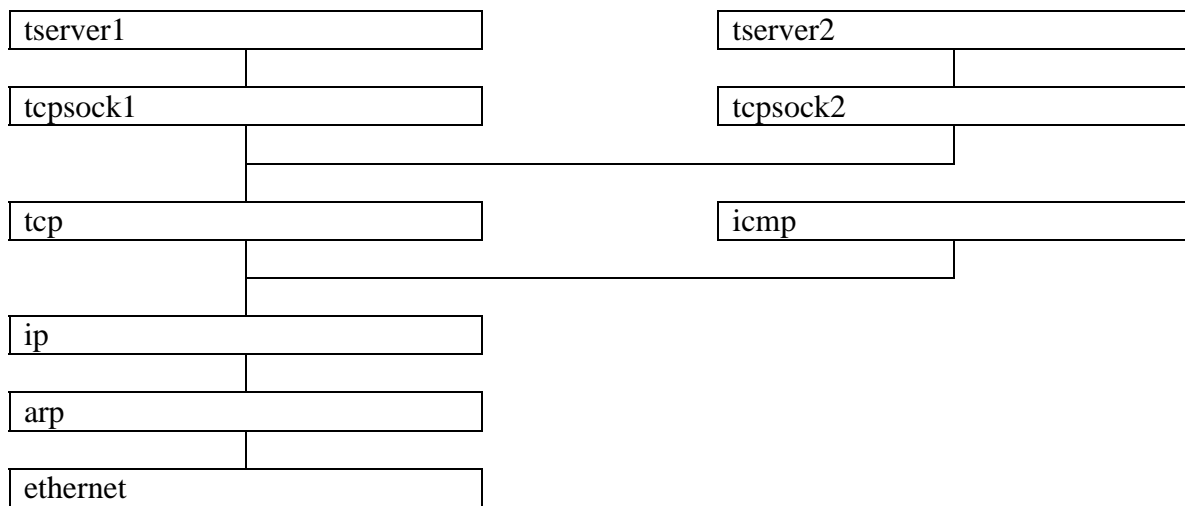
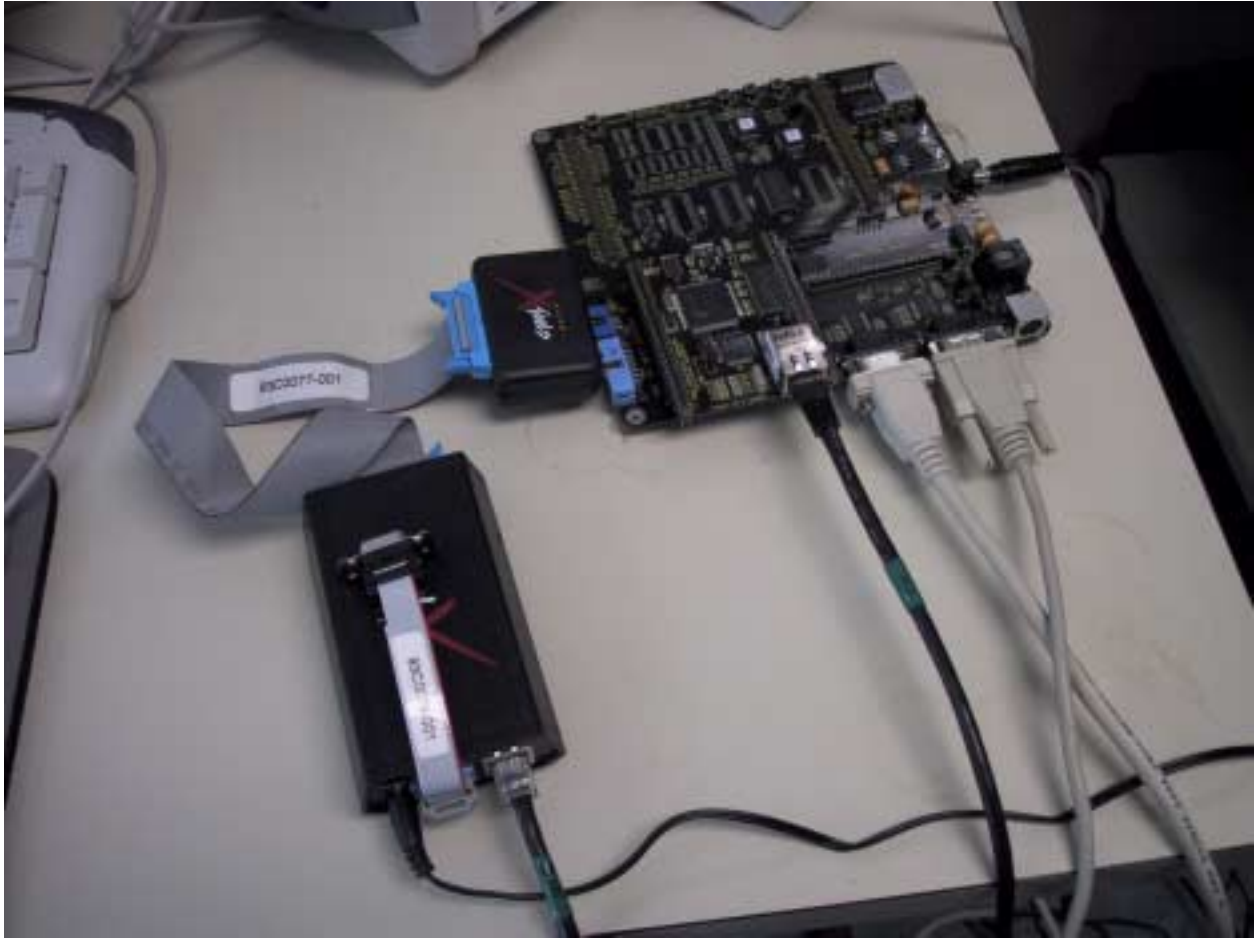


Figure 1. Terminal Server software block diagram

A block diagram of the terminal server application structure is shown above. Each block is a BMZ task. The diagram reflects the fact that much of the terminal server program is involved with implementing the TCP/IP protocol suite. BMZ is well suited to implementing communication protocols. Each of ip, tcp, arp and icmp are acronyms for fundamental parts of TCP/IP. Ethernet is an ethernet driver implemented as a BMZ task. Tcpsock1 and tcpsock2 are tcp “sockets” or independent connections. A tserver (terminal server application) task is connected to each socket.



Example:

```
/*
*****
* project.c
* Project description and startup
*****
*/

#include <stdio.h> // for NULL
#include "bmz.h"

// Instance data
typedef struct
{
    bool    led_on;    // true if led on
    u16     timeout;   // flash time
    TIMER   timer;    // BMZ timer
} LED;

// Define TASKIDs, starting from 1
#define TASKID_LED1 1
#define TASKID_LED2 2

// Assume this exists somewhere, to physically drive a led
extern void led_driver( byte led_id, bool led_on );

/*
*****
*/
```

```

* Init handler
*****/
void *led_init( byte **addr_mem, ul6 *addr_len )
{
    LED    *z;
    byte    id;
    byte    *memory = *addr_mem;
    ul6     memlen = *addr_len;

    // Check we have enough memory
    if( memlen < sizeof(LED) )
        bmz_panic_memory("led");
    else
    {
        // Allocate it
        z      = (LED *)memory;
        memory += sizeof(LED);
        memlen -= sizeof(LED);

        // Task LED1 flashes led 1 once every 2 seconds
        if( bmz_get_current_taskid() == TASKID_LED1 )
        {
            id = 1;
            z->timeout = 2;
        }

        // Task LED2 flashes led 2 once every 3 seconds
        else
        {
            id = 2;
            z->timeout = 3;
        }

        // Turn on the appropriate led
        z->led_on = true;
        led_driver( id, z->led_on );

        // Reset and start timer
        timer_reset( &z->timer, id );
        timer_start_seconds( &z->timer, z->timeout );
    }

    // Update input parameters and return instance ptr
    *addr_mem = memory;
    *addr_len = memlen;
    return( z );
}

/*****
* Timeout handler, toggle led state and restart timer
*****/
void led_timeout( byte timer_id )
{
    LED *z = bmz_get_current_instance();
    z->led_on = (z->led_on ? false : true);
    led_driver( timer_id, z->led_on );
    timer_start_seconds( &z->timer, z->timeout );
}

/*****
* Describe tasks
*****/

```

```

static const TASK_DESCRIPTOR task_descriptors[] =
{
    // Led1 flash task
    {
        TASKID_LED1,          // TASKID
        led_init,            // init handler
        NULL,                // idle handler
        led_timeout,        // timeout handler
        NULL,               // down handler
        NULL,               // up handler
        0,                  // mq down depth
        0,                  // mq up depth
        0,                  // pool nbr
        0,                  // pool len
        0,                  // pool offset
        TASKID_NULL        // share pool of this TASKID
    },

    // Led2 flash task
    {
        TASKID_LED2,          // TASKID
        led_init,            // init handler
        NULL,                // idle handler
        led_timeout,        // timeout handler
        NULL,               // down handler
        NULL,               // up handler
        0,                  // mq down depth
        0,                  // mq up depth
        0,                  // pool nbr
        0,                  // pool len
        0,                  // pool offset
        TASKID_NULL        // share pool of this TASKID
    }
};

/*****
 * Main entry point
 *****/
int main()
{
    static byte buf[5500]; // all the spare memory
    byte *memory = buf;
    ul6 memlen = sizeof(buf);

    // Always initialize BMZ first
    bmz_init();

    // Define the system
    bmz_define_system( task_descriptors, // array of task descriptors
                      2,                // nbr of task descriptors
                      &memory,
                      &memlen );

    // Run the system
    bmz_run();
    return(0);
}

```