# LiteOS
# Programmer's Guide

Version 1.0
Last updated: Oct 5 2008

This guide illustrates how to program with LiteOS and provides detailed descriptions of the provided libraries and APIs.

Download location: www.liteos.net

## Preface

LiteOS provides a UNIX-like environment for sensor networks, networked embedded devices, and cyber physical systems. It provides a thread-based run-time execution environment for applications. The goal of this User's Guide is to get you familiarized with its environment. For programming in LiteOS, see Programmer's guide.

For updates, check:

**www.liteos.net**

## Mailing lists:

liteos-users@cs.uiuc.edu

http://lists.cs.uiuc.edu/mailman/listinfo/liteos-users

liteos-developers@cs.uiuc.edu

http://lists.cs.uiuc.edu/mailman/listinfo/liteos-developers

## Additional References

- The following are additional documentation, available on www.liteos.org

- LiteOS User's Guide
- LiteOS Application Notes

# LiteOS Kernel Programming

## How to compile the LiteOS kernel for your purpose

LiteOS is developed for embedded applications that use AVR processors. Typical application domains of such processors include sensor network applications that run on MicaZ and Iris motes (Iris support by LiteOS is under development), embedded system applications, and cyber physical systems. LiteOS provides two modes of compilation (details in the programmer's guide), **micaz** and **iris**, for different usage purposes.

To compile the kernel, enter the SourceCode/LiteOS_Kernel directory, and use one of the following commands to build the kernel for different purposes:

*./build micaz for MicaZ nodes*

*./build iris for Iris nodes (under development)*

# LiteOS Applications Programming on MicaZ

## Write "Hello World" in LiteOS

Just like to get started with any other programming environment, we start by giving the classical "hello, world" example. The difference between the LiteOS environment, which runs on motes, and conventional computer environment, is that motes do not have monitors.  Hence, we decide to send strings by radio. This first example does the following:

**The "hello, world" example broadcasts the string over the radio.**

## The Source Code Listing

We get started with the source code listing of HelloWorld, as follows:

```
#include "leds.h"
#include "thread.h"
#include "radio.h"
int main()
```

```
{
   int i;
        while (1)
        {
         radioSend_string("Hello, world!\n");
        greenToggle();
        sleepThread(100);
        }
        return 0;
}
```

If you know C, you will quickly recognize that this is a very familiar program that uses libraries to send out packets. Note that this particular application creates a thread. Therefore, you do not need to worry about the underlying details, such as when the radio finishes transmitting and when the thread wakes up; the kernel handles such details for you.

## Compile this application

**Note:**

Before proceeding with this section, type the command **python** under your cygwin environment. If you have not got python installed, install python under cygwin before continue.

In the LiteOS operating system, there is a directory called **Apps**. Under this directory, there are two directories: **Apps**, which contain the C programs, and **Libraries**, which contain the source code of the libraries.

To set up the compiling environment, first copy the file named *build* to your cygwin home directory, such as usr/local/bin. (or any other directory that cygwin search and locate executables.)

```
cp build /usr/local/bin
```

The purpose of build script, if you open it with a text editor, is to automatically configure this compiling environment and generate the makefile. It also requires that your cygwin has a make environment. Normally when you install cygwin you already have this environment. Otherwise, you need to install it manually.

**IMPORTANT STEP**: Next, navigate to the Apps directory, and open the file named Makefile. Change two parameters:

The CC parameter: change it to the particular AVR-GCC installation path, as specified when you installed WinAVR, on your PC. **Your AVR-GCC should be the version 4.1.1. to correctly compile the applications.** Otherwise, you will meet problems saying the makefile causes errors. To verify the version of your installed GCC, go to the directory of WinAVR, and type `avr-gcc.exe --version` to get its version.

The AVR-SIZE parameter: change it to the particular avr-size.exe location on your PC that comes with WinAVR. **This file should be Version 2.17 or newer.** Similary, navigate to its directory and type `avr-size –version` to get its version number.

Next, navigate to the particular application directory. For example, the HelloWorld directory contains the HelloWorld application. Type the following command:

```
$ build -flash_page=200 -ram_start=2700 -
ram_size=512 HelloWorld.lhex
```

```
LiteOS Build Environment Startss...
AVR Memory Usage
----------------
Device: atmega128

Program:    586 bytes (0.4% Full)
(.text + .data + .bootloader)

Data:       20 bytes (0.5% Full)
(.data + .bss + .noinit)
```

Build complete. The lhex is located in ./build **directory.**We now explain this command.

**$ build -flash_page=200 -ram_start=2700 -ram_size=512 HelloWorld.lhex**

It is important that the parameters should be set correctly. To do this, we describe what they mean in the context of LiteOS.

As described earlier, the build script handles the compilation procedure and generates the lhex file. Note that build requires three parameters: the start page to be used, the start address of RAM, and the RAM usage of this application. Basically, each page contains 256 bytes. Earlier, when you compiled the kernel, you will get a number of bytes used by the kernel, both flash and RAM. The exact number usually changes as we add and remove functionalities. For instance, the most up-to-date version in the SourceForge at the time this document is prepared has the following flash and RAM uses:

(by using ./build micaz in SourceCode/LiteOS_Kernel)

```
Program:   45070 bytes (67.6% Full)

(.text + .data + .bootloader)

Data:      2306 bytes (56.1% Full)

(.data + .bss + .noinit)
```

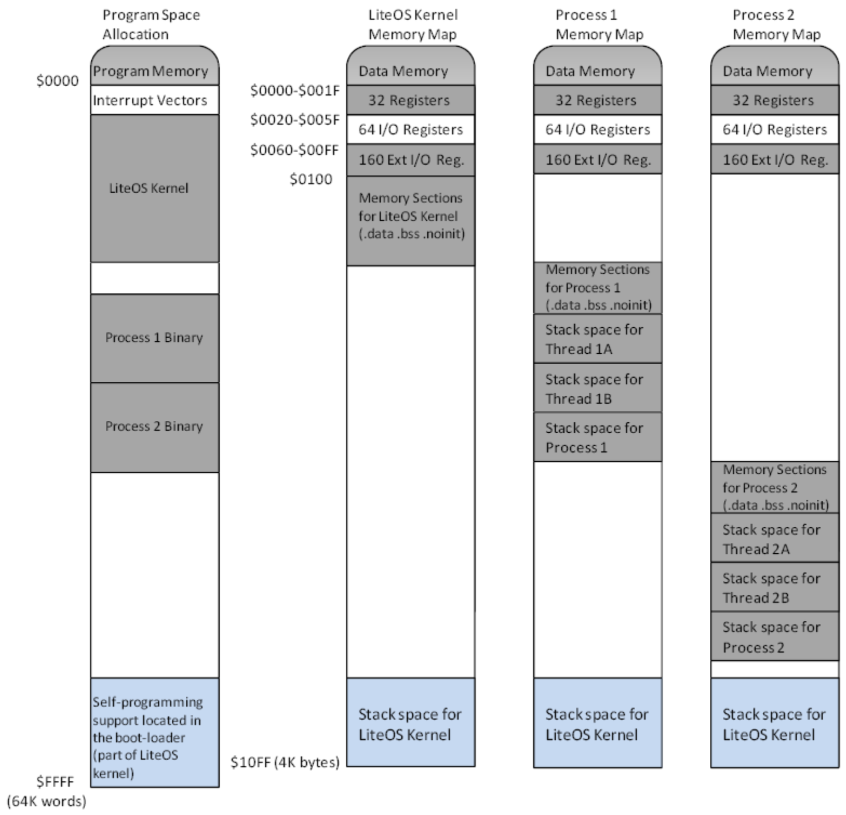
Therefore, you get the number of pages used by the kernel as 45070/256=176 and RAM as 2306. The RAM use has to be added with another 256 bytes since these are reserved by the compiler as register and I/O mappings. Specifically, in this example, we allocate RAM starting with 2700 to user applications.

Not surprisingly, in the previous example, we used pages from 200 and RAM from 2700, hence no conflict will occur. If conflicts do occur, the kernel will refuse to load the user application.

Also note that the kernel also uses stack that grows from the highest RAM address 0x10FF (MicaZ has 4096 bytes of RAM). The stack pointer decreases as PUSH instructions are executed. Please keep sufficient RAM for the kernel stack. In reality, we suggest that address higher than 3650 should be reserved by the kernel.

To verify there are indeed no memory conflicts, the shell also provides the **memory** command for displaying the current memory allocation information. If insufficient memory is allocated for a thread, you will notice one thread in MEMORY_CORRUPTED status when you use the **ps** command.

To better help understand how memory locations are allocated in LiteOS, the following figure shows a typical example.


Program Space Allocation
LiteOS Kernel Memory Map
Process 1 Memory Map
Process 2 Memory Map
$0000
Program Memory
Interrupt Vectors
LiteOS Kernel
Process 1 Binary
Process 2 Binary
Self-programming support located in the boot-loader (part of LiteOS kernel)
$FFFF (64K words)
$0000-$001F
$0020-$005F
$0060-$00FF
$0100
$10FF (4K bytes)
Data Memory
32 Registers
64 I/O Registers
160 Ext I/O Reg.
Memory Sections for LiteOS Kernel (.data .bss .noinit)
Stack space for LiteOS Kernel
Memory Sections for Process 1 (.data .bss .noinit)
Stack space for Thread 1A
Stack space for Thread 1B
Stack space for Process 1
Memory Sections for Process 2 (.data .bss .noinit)
Stack space for Thread 2A
Stack space for Thread 2B
Stack space for Process 2

In this figure, there are two processes residing in the memory space, named as Process 1 and Process 2, respectively. Note that it is critical that their memory addresses do not conflict with each other. Otherwise, read/writes to shared memory locations will lead to semantics errors of both processes.

Another observation is that each process in this figure could create new threads (through the Thread library, as will be described later). In this scenario, process 1 creates threads 1A and 1B, while process 2 creates threads 2A and 2B, respectively. Each thread will share the memory space as its parent process, but owning its own stack. Therefore, it is imperative to allocate the stack space correctly such that different threads will not lead to conflict with each other. To this end, various stack analysis tools have been developed and published in the literature, which can be leveraged to estimate the stack usage situations.

At the end of the compilation, the HelloWorld application uses 586 bytes of program flash, or 3 pages, and 20 bytes of RAM. Note that because of stack usage, you cannot only allocate 18 bytes to the thread. Among other things, each stack must also maintain 32 registers information, as well space for function local variables. Besides using existing stack analysis tools, we use the following simple rule of thumb is to set the ram size:

RAM Size = Static RAM (Data line in compilation) + 50 (the registers and other thread information) + LocalVariableSize (chosen based on the particular program).

Note that we allocated *more than enough* RAM for the HelloWorld application. It needs only about 80 bytes of RAM to run correctly.

Now that you get a build directory created, and a helloworld.lhex file generated, copy this file to the mote and run it. You will see that the "hello, world" message is broadcast for 100 times, and the mote will blink during such broadcast.

## Directory Structure

To help develop applications and avoid resource conflicts, we recommend the following way to organize the directory information. In the Apps directory, for each new application, it is suggested that you create a separate directory. All the source code of the new application should be placed in this new directory. Once you open the Apps directory, you will find that there are already many existing directories, such as BlinkBasic. They are bundled applications with the operating system environment, and are carefully debugged so that they are readily usable.

# The Libraries

In this section, we give a brief introduction of the libraries that are provided by LiteOS. Note that these libraries are still in early stage of development, and please report the bugs you encounter on the LiteOS mailing list.

## ADC.H: Sensors

The adc.h defines functions for controlling the sensors. It allows reading the light, temperature, magnetic, and accelerator sensor values. Note that the magnetic sensor allows reading both the X axis and the Y axis readings. Please refer to CrossBow MTS310 data sheet to interpret the sensor readings.

The return values of these functions are 16-bit integers.

```
//Get the light sensor reading
int get_light();

//Get the temperature sensor reading
int get_temp();

//Get the magnetic sensor reading, X axis.
int get_magx();

//Get the magnetic sensor reading, Y axis.
int get_magy();

//Get the accelerator sensor reading, X axis.
int get_accx();

//Get the accelerator sensor reading, Y axis.
int get_accy();
```

## EEPROM.H: Non-volatile data storage

EEPROM.h contains the definitions for handling the EEPROM on the MCU of the node. In the Atmega128 MCU available on MicaZ, it contains 4096 bytes of EEPROM. Of such bytes, the first 3200 bytes are used by the LiteFS file system to keep directory information. Therefore, they are not available for the user applications to keep non-volatile data. The remaining 896 bytes are accessed through two functions, read and write:

```
//Read from the EEPROM for nBytes bytes, starting from address
//specified by addr, and store them in location buffer.
void readFromEEPROM(uint16_t addr, uint16_t nBytes, uint8_t *buffer);

//Write to the EEPROM for nBytes bytes, starting from address
//specified by addr, and store them in location buffer.
void writeToEEPROM(uint16_t addr, uint16_t nBytes, uint8_t *buffer);
```

Note that, because of the extremely limited size of EEPROM, it is not suggested to keep large volume of data here. Instead, they are intended primarily for application configuration data. For example, a neighborhood management protocol could use EEPROM to keep the neighborhood updating frequency. On the other hand, for large volume of data, such as long-term sensor readings, it is suggested that such data should be kept in local files.

## File.h Local File System operations

The file.h defines function prototypes for controlling the file operations. It allows creating, writing, and reading from files that are stored on local file systems in sensor nodes. Below are the function interfaces of these files:

```
//Open a file using the pathname and the mode as the parameter
MYFILE *mfopen(char *pathname, const char *mode);

//Close an opened file
void mfclose(MYFILE *fp);

//Read from a file and put the read data into a buffer
void mfread(MYFILE *fp, void *buffer, int nBytes);

//Write to a file using a buffer
void mfwrite(MYFILE *fp, void *buffer, int nBytes);

//Modify the file read/write position
void mfseek(MYFILE *fp, int offset, int position);

//Write to a file using data stored in buffer, but uses the '\0' as the end
void mfwrite_withoutlength(MYFILE *fp, void *buffer);
```

An example of using the file operations can be found in SenseAndDeposit application, described later.

## Leds.h Led Operations

The leds.h defines function prototypes for controlling the file operations. It allows setting various Led displays.

```
//Toggle the green LED
void greenToggle();
//Toggle the red LED
void redToggle();
//Toggle the yellow LED
void yellowToggle();
//Turn on the red LED
void redOn();
//Turn off the red LED
void redOff();
//Turn on the green LED
void greenOn();
//Turn off the green LED
void greenOff();
//Turn on the yellow LED
void yellowOn();
//Turn off the yellow LED
void yellowOff();
```

An example of using these functions can be found in BlinkBasic.

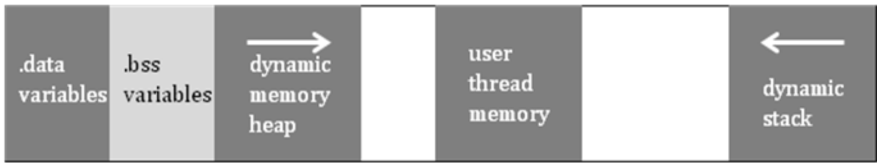
## Malloc.h Dynamic Memory Operations

Dynamic memory is supported in LiteOS through the library malloc. There are two function interfaces provided:

```
//Get a chunk of memory specified by size from the operating system
//Returns NULL if the allocation is unsuccessful
void *malloc(uint16_t size);


//Free the chunk of memory pointed by ptr
void free(void *ptr);
```

To use dynamic memory, it is critical to understand how memory management in LiteOS works. The following figure shows the basic organization of memory space in LiteOS when the dynamic memory is used.



As shown in this figure, when the user thread is loaded into the memory space, whose start address is specified by the parameter ram_start, the dynamic memory is positioned between the end of .bss variables, which are global variables defined by user applications that are not initialized, and the first block of the usr thread memory. Therefore, suppose that the LiteOS kernel takes A bytes, and the first user thread is loaded into position B, then for dynamic memory, we have:

The Maximum Dynamic Memory = B – A

The dynamic memory in LiteOS is implemented in an elastic way. Usually when there is no malloc or free functions used, the overhead of the dynamic memory is 4 bytes, which is a pointer to the first free dynamic memory chunk. When dynamic memory is used, the overhead of maintaining the dynamic memory heap grows linearly with the number of chunks, but is typically small enough (several bytes for a chunk) to be tolerated.

## Radio.h Radio operations

The radio.h defines function prototypes for controlling the radio from the user's perspective. It allows sending and receiving messages through the radio using specified port and destination addresses.

```
//Send out a message using the radio
int radioSend(uint8_t port, uint16_t address, uint8_t length, uint8_t *msg);

//Send out a string using the radio
int radioSend_string(uint8_t *msg);

//Send out an integer which is 16-bit
int radioSend_uint16(uint16_t value);
```

```
//Wait to receive a packet through the radio
int  radioReceive(uint16_t port, uint8_t maxlength, uint8_t *msg);


//Wait for a limited time to receive a packet through the radio
int  radioReceiveTimed(uint16_t port, uint8_t maxlength, uint8_t msg,
uint16_t time);

//Change the radio frequency
void setRadioFreq(uint16_t freq);


//Change the radio channel
void setRadioChannel(uint8_t channel);

//Set the radio output power level
void setRadioPower(uint8_t power);
```

Following is an example of using the above functions to send out multiple packets using different radio channels. Here, the user application must set the channel number between 11 and 26, and uses the setRadioChannel function to change the channel before broadcasting the "hello, world" message.

```
#include "leds.h"
#include "system.h"
#include "liteoscommon.h"
#include "thread.h"
#include "radio.h"
#include "stringutil.h"
#include "file.h"
#include "adc.h"

int main()
{

 uint8_t channel;

 for (channel = 11;channel <25;channel++)
 { setRadioChannel(channel);

 radioSend_string("Hello, world!\n");
 greenToggle();
 sleepThread(1000);

 }

 channel = 11;
 setRadioChannel(channel);
 return 0;
}
```

## Serial.h Serial Port Communication

```
//Send a string over the serial port to PC
int serialSend_string(uint8_t *msg);
//Send an 16-bit integer over the serial port to PC
int serialSend_uint16(uint16_t value);
//Send a specified length message through the serial port to PC
int serialSend(uint8_t length, uint8_t *msg);
//Wait and receive a message from PC through the serial port
void serialReceive(uint16_t port, uint8_t maxlength, uint8_t *msg);
```

## Stringutil.h String operations

The stringutil.h provides basic functions for simple string operations. The interface is as follows:

```
int String_length(char* s);
void mystrncpy(char *dest, const char *src, uint16_t n);
void mystrcpy(char *dest, const char *src);
char* string_split(char ** string, char delimiter);
void String_append(char *base, char *string);
char *String_intToString(int num);
uint16_t hex2value(uint8_t hex);
```

An example of using the string operations can be found in SenseAndDeposit.

## System.h System provided functions

```
typedef uint8_t _atomic_t;
inline _atomic_t _atomic_start(void);
inline void _atomic_end(_atomic_t oldSreg);
int rnd();
```

## Thread.h Thread operations

The thread.h provides basic thread operations as follows:

```
void sleepThread(int milliseconds);
void yield();
thread **getCurrentThread();
uint8_t getCurrentThreadIndex();
void postTask(void (*tp) (void), uint16_t priority);
uint8_t atomic_start(void) ;
void atomic_end(uint8_t oldSreg);
void syscall_postThreadTask();
void debugValue(uint16_t v1, uint16_t v2, uint16_t v3);
```

An example of using the string operations can be found in SenseAndDeposit. The most basic use, of course, is to sleep a thread for a certain period of time.

# Example Applications

This section describes several included applications that are bundled with the programming environment. Please note that as they are, they are still primitive. More advanced applications will be updated in later versions of LiteOS.

## BlinkBasic Application

The Blink application toggles the LED. To make things more interesting, we have also made it sending out data periodically using the light sensor. Please refer to Application Note LITEOS-AN10 on how to display the messages on PC in real time.

//Source code listing. The program is easy to understand.

```
#include "leds.h"
#include "thread.h"
#include "adc.h"
#include "radio.h"

int main()
```

```
{
        //Open the interrupt because by default, the interrupt is disabled
        __asm__ __volatile__("sei" ::);
        int msn;

        for (msn=0;msn<100;msn++)
        {
        greenToggle();
        yellowToggle();
        reading = get_light();
        radioSend_uint16(reading);
        sleepThread(1000);
        }
        return 0;
}
```

## SenseAndDeposit Application

As its name indicates, the Sense Application uses the sensors to execute sampling tasks. It is fully customized in terms of the type of sensor used, the sampling frequency, and the number of samples, by using a local file called **config**. To reduce overhead, we use a very simple syntax for the **config** file as follows:

**Sleeptime : Four Digits using Hex format**

**Number of Samples: Three Digits using Hex format**

**ype of Sensor: One digit using Hex format**

For example, a config file may look like the following:

00640641FFFFFFFFFFFFFFFFFFFFFF

Here, FFFF is used as padding. This configuration means that the sleep time between consecutive samplings is 0x0064, or 100, milliseconds; the number of samples is 0x064, or 100, samples; and the type of sensor used is type I.

Following is a list of sensors and their types:

**Type I: Light**

**Type II: Accelerator**

**Type III: Temperature**

When the application starts, it reads the parameters from the **config** file, and deposits the samples into a local file called **deposit**. This file can then be retrieved back to the base station for further analysis.

At the end of the operation, as shown in the screenshot, there is a file named deposit, whose content is displayed as follows: (first few lines)

```
The reading is 750

The reading is 756

The reading is 751

The reading is 755
```

//Source code listing

```
#include "leds.h"
#include "system.h"
#include "liteoscommon.h"
#include "thread.h"
#include "radio.h"
#include "stringutil.h"
#include "file.h"
#include "adc.h"

 MYFILE *fileptr;
 MYFILE  *configptr;
 uint16_t samples;
 uint16_t reading;
 uint16_t sleeptime;
 uint16_t type;


int main()
{
 uint8_t config[17];
 configptr = mfopen("/config", "r");
 mfread(configptr, config, 16);
 mfclose(configptr);
 config[16]='\0';

 sleeptime =
hex2value(config[0])*1000+hex2value(config[1])*100+hex2value(config[2])
*10+hex2value(config[3]);
 samples = hex2value(config[4])*100 + hex2value(config[5])*10 +
hex2value(config[6]);
 type = hex2value(config[7]);

 fileptr = mfopen("/deposit", "w");
 while (1)
 {

  if (samples >0)
  samples--;
  else
  break;

  sleepThread(sleeptime);

 // type = 1;

  if (type == 1)
  reading = get_light();
  else if (type == 2)
  reading = get_accx();
  else if (type == 3)
  reading = get_temp();
  else
  reading = 0;


  redToggle();

  {{
    char _tempbuffer[32];
    char *_temp1 = "The reading is ";
    char *_temp2 = String_intToString(reading);
    char *_temp3 = "\n";
            uint8_t lengthstring;
    _tempbuffer[0] = '\0';
    String_append(_tempbuffer, _temp1);
    String_append(_tempbuffer, _temp2);
    String_append(_tempbuffer, _temp3);

    lengthstring = String_length(_tempbuffer);

    mfwrite(fileptr, _tempbuffer, lengthstring);
            mfseek(fileptr, lengthstring, 1);

    }}

}


mfclose(fileptr);
return 0;

}
```

# Debugging Your Applications

This section describes debugging your applications. The focus is on using the current debugging commands that are built into the LiteShell to view and display real time debugging information.

## Introduction to the Debugging Commands

LiteOS currently supports the following debugging commands:

debug – for setting up the debugging environment

list -- for listing current variables

print – for printing variable values

breakpoint – for setting up a breakpoint

listbreakpoint – for listing current breakpoints

continue – for continuing from a previous breakpoint

snapshot – for making a snapshot of a thread

restore – for restoring a thread to an earlier image

## The HelloWorld Example

We illustrate how to use the commands through the HelloWorld example. This time, we intentionally insert several useless variables into the source code for later debugging purposes. The revised source code listing is as follows:

```
#include "leds.h"
#include "thread.h"
#include "adc.h"
#include "radio.h"

//This is a version of BlinkBasic that illustrates the debugging interface.
The details are described in the programmer's
//guide on how to debug user applications.

int msn;
typedef struct
{ int a;
        int b;
} structab;

structab cde;
int reading;

int main()
{
 __asm__ __volatile__("sei" ::);
        for (msn=0;msn<300;msn++)
        {
  greenToggle();
  cde.a = 1 ;
  cde.b = 2;
```

```
            yellowToggle();
            reading = get_light();
            radioSend_uint16(reading);
            sleepThread(1000);
            }
            return 0;
}
```

Compile and install this application. Then you use the above debugging commands as follows. The comments are intentionally added to illustrate the purpose of each command.

//Screenshot of the command lists

```
$ls  //list the current directory, where blink is the application
The returned has 2 packets.

dev
blink
Time elapes 500

$ps //get the current active threads


The reply has 2 packets.
sysshell   Active
blink   Sleep
Time elapes 312

//Set up the debug environment. Change the directory
//to the particular directory on your application, in this
//case, the BlinkBasic application


$debug e:/OS/TerminalProject/ProjectsNew/BlinkBasic
Directory error. File does not exist.


//Set up the correct environment
$debug e:/OS/TerminalProject/ProjectsNew/Apps/BlinkBasic

//List the variables that are in the application that
//can be used for debugging
```

```
$list
This variable has name of mythreadserial and size of 2
This variable has name of msn and size of 2
This variable has name of cde and size of 4
This variable has name of reading and size of 2
This variable has name of buffer and size of 32
This variable has name of mythread and size of 2



//List current breakpoints. Note that, if you restart the shell unexpectedly,
//some breakpoints may be lost.
$listbreakpoint
Note that there may be additional breakpoints on the thread that are not
listed.
There are 0 breakpoints.



//Print the variable values in the mote
$print msn
Variable length 2 : 0x88 ,0x0 ,
Time elapes 16

//Print the cde structure values
$print cde
Variable length 4 : 0x1 ,0x0 ,0x2 ,0x0 ,
Time elapes 15

//Set up breakpoints. Note that to do this, open the //generated lss file in
your directory, and find the
//particular address in the assembly code that you want
//to insert breakpoint. The source code is also listed in
//the lss file for your reference.
$breakpoint 1233c
Time elapes 31

//List current thread information.
$ps
The reply has 2 packets.
sysshell   Active
blink   Breakpoint_Reached at 0x01233C
Time elapes 312
```

```
//List current breakpoints that are active
$listbreakpoint
Note that there may be additional breakpoints on the thread that are not
listed.
There are 1 breakpoints.
Breakpoint at 0x01233C


//Continue the thread that with the breakpoint
$continue 1233c
Time elapes 31

//List all the current breakpoints. Once the continue command is used, the
previous breakpoint is removed.
$listbreakpoint
Note that there may be additional breakpoints on the thread that are not
listed.
There are 0 breakpoints.

//Use the ps command to get the current thread status
$ps
The reply has 2 packets.
sysshell   Active
blink   Sleep
Time elapes 313


//Use the snapshot command to get a snapshot of the thread memory
image, in a file named image in the root directory
$snapshot blink
Time elapes 203

$ls
The returned has 3 packets.

dev
blink
image

Time elapes 500

$cp image /c/Temp2/image
Reading port 2 packets
Reading port 2 packets
Reading port 2 packets
Reading port 2 packets
Reading port 2 packets
Reading port 2 packets
Reading port 2 packets
Reading port 2 packets
Reading port 2 packets
Reading port 2 packets
Reading port 2 packets
Reading port 2 packets
Now Patching
Copy finished
Time elapes 2922
```

//Now you can open image file to see the memory situation of the current thread.