Azure Sphere SDK: First Look

By Sean D. Liming and John R. Malin
Annabooks – www.annabooks.com

January 2019

Reports of 100+ billion devices connected to the Internet by 2030 have helped spawn renewed interest in Embedded Systems, specifically those devices which have Internet connection capability that can connect to the cloud. This class of embedded devices is now being called the Internet of Things (IoT). Devices with processors that can power PCs, tablets, and Phones are the first wave of devices that have connected to the cloud services. Cloud service providers are not focusing on these devices, alone, but looking to go smaller. Microcontrollers (MCUs), computers on a chip, are in everything from toys to appliances to industrial controls. There are new efforts to connect MCUs to the cloud. Google has made investments in home automation. Amazon has taken stewardship of the popular FreeRTOS kernel, which runs on many MCUs, and created Amazon Web Services (AWS) FreeRTOS. The surge toward those 100+ billion Internet-connected devices has begun.

Outside of the .NET Micro Framework effort, Microsoft has never really focused on the MCU device market. Recognizing an opportunity, Microsoft took some time to figure out a solution. With all the internet security breaches in the news, Microsoft wanted to make sure that MCU devices connecting to the cloud had the best in class security. Microsoft researched came up with *The Seven Properties of Highly Connected Devices*. The research led to creating a custom silicon solution with a security subsystem built in that implements certificate security from hardware to application. The final result is a technology called Azure Sphere. Microsoft is working with different silicon partners to create a variety of Azure Sphere MCUs. The first silicon to implement Azure Sphere is the MediaTek MT3620. An MT3620 Reference Development Board (RDB) is now available along with a public preview of the Azure Sphere SDK.

The purpose of this document is to provide a high-level overview of Azure Sphere, the MT3620 RDB, and what we have experienced, so far, with the Azure Sphere SDK.

Note: The Azure Sphere SDK is in preview as of this writing, so we will only cover what is currently supported. More features and capabilities will be made available in future releases. GPIO and UART are the only pin header IOs supported as of this writing.

Azure Sphere from 1000 Feet

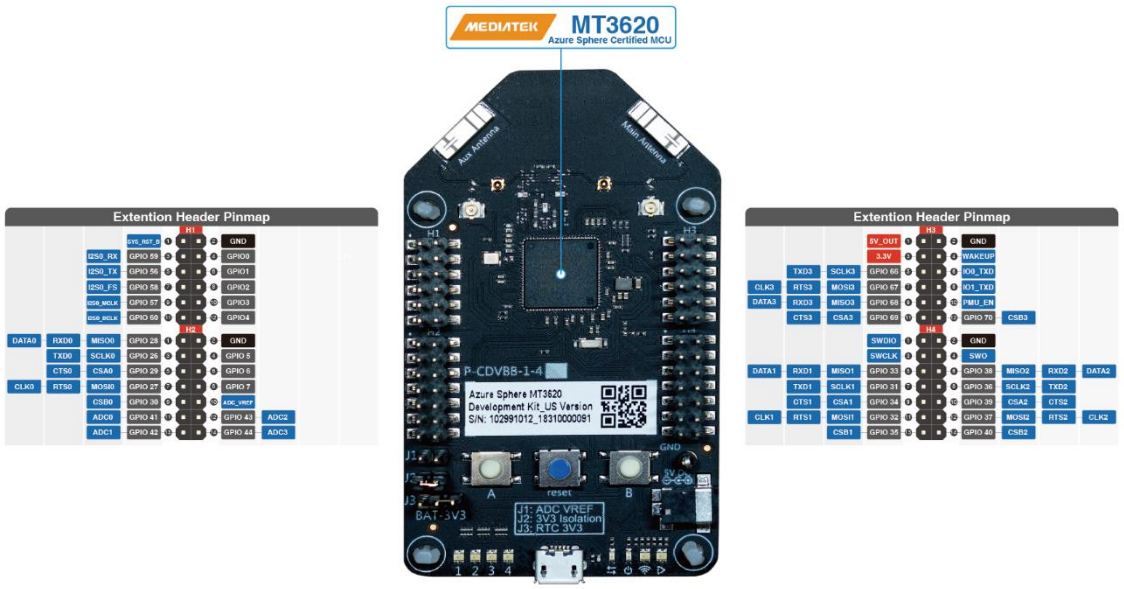
The online Azure Sphere documentation covers the background and details of the technology. An implementation for the Azure Sphere MCU consists of the following:

- Application Processor ARM Cortex-A running a custom Linux kernel (Azure Sphere OS).
- Microsoft Pluton security subsystem, which was developed out of the research of *The Seven Properties of Highly Connected Devices.* This subsystem is the hardware-based secured root of trust for Azure Sphere. This contains a security processor core with all the encryption and cryptography support.
- Real-time processor(s) ARM Cortex-M I/O subsystem that can be configured by the silicon vendor.
- Integrated 16MB (minimum) flash and 4MB (minimum) RAM.
- Connectivity / Communication – all Azure Sphere MCUs will come with wireless Internet connection support.
- Multiplexed I/O – Azure Sphere will support a variety of I/O capabilities such as GPIO, PWM, ADC, UART, SPI, I2C. Silicon vendors can configure how the I/O is implemented.
- Firewalls – each of the above subsystems is protected by a firewall to provide security of access to the I/O only from the subsystem to which it is mapped.

For the application processor, a single application runs in an application container, which provides access to Wi-Fi, I/O, Azure IoT, and the C library. Other OS services allow for over the air (OTA) updates, networking management, device authentication, and application management. All of these features run on a custom Linux kernel running in supervisor mode, which runs on top of a security monitor that protects MCU resources.

MT3620 Chip Basics and the Reference Design Board (RDB)

MediaTek assisted with custom silicon for the research in *The Seven Properties of Highly Connected Devices.* Out of this effort is the first Azure Sphere MCU, the MT3620. MT3620 supports GPIO, UART, I2C, SPI, I2S, PWM and ADC. Only GPIO and UARTs are supported in the current Azure Sphere SDK preview. The MT3620 RDB was created and made available to get developers familiar with Azure Sphere. The board has several buttons and LEDs available for program access. 4 groups of 12-pin headers provide access to the different I/Os from the MCU.



Azure Sphere MT3620
Development Kit_US Version
S/N: 102991012_18310000091

Extention Header Pinmap

				H1			
			SYS_RST_B	1	2	GND	
		I2S0_RX	GPIO 59	3	4	GPIO0	
		I2S0_TX	GPIO 56	5	6	GPIO1	
		I2S0_FS	GPIO 58	7	8	GPIO2	
		I2S0_MCLK	GPIO 57	9	10	GPIO3	
		I2S0_BCLK	GPIO 60	11	12	GPIO4	
				H2			
DATA0	RXD0	MISO0	GPIO 28	1	2	GND	
	TXD0	SCLK0	GPIO 26	3	4	GPIO 5	
	CTS0	CSA0	GPIO 29	5	6	GPIO 6	
CLK0	RTS0	MOSI0	GPIO 27	7	8	GPIO 7	
		CSB0	GPIO 30	9	10	ADC_VREF	
		ADC0	GPIO 41	11	12	GPIO 43	ADC2
		ADC1	GPIO 42	13	14	GPIO 44	ADC3

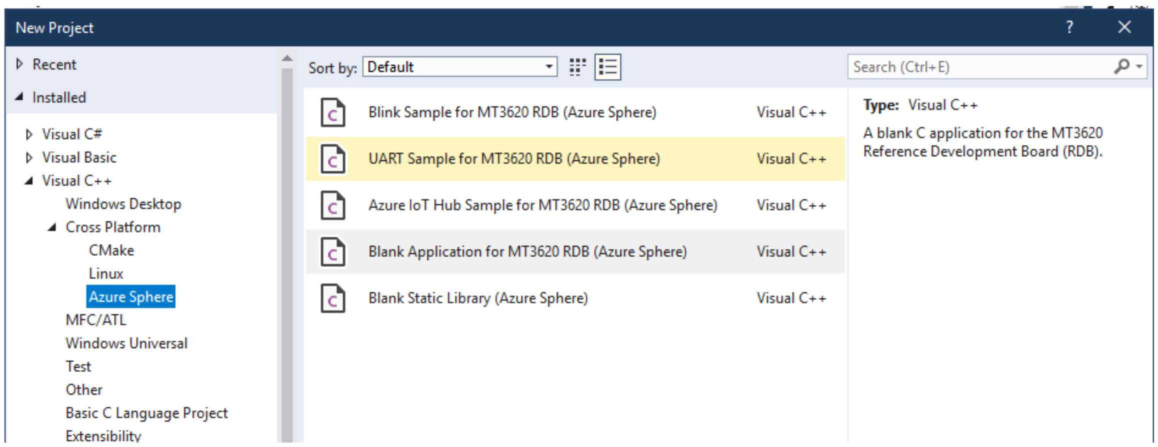
Extention Header Pinmap

				H3					
			5V_OUT	1	2	GND			
			3.3V	3	4	WAKEUP			
	TXD3	SCLK3	GPIO 66	5	6	IO0_TXD			
CLK3	RTS3	MOSI3	GPIO 67	7	8	IO1_TXD			
DATA3	RXD3	MISO3	GPIO 68	9	10	PMU_EN			
	CTS3	CSA3	GPIO 69	11	12	GPIO 70	CSB3		
				H4					
			SWDIO	1	2	GND			
			SWCLK	3	4	SWO			
DATA1	RXD1	MISO1	GPIO 33	5	6	GPIO 38	MISO2	RXD2	DATA2
	TXD1	SCLK1	GPIO 31	7	8	GPIO 36	SCLK2	TXD2	
	CTS1	CSA1	GPIO 34	9	10	GPIO 39	CSA2	CTS2	
CLK1	RTS1	MOSI1	GPIO 32	11	12	GPIO 37	MOSI2	RTS2	CLK2
		CSB1	GPIO 35	13	14	GPIO 40	CSB2		

Note: [2018/09/10] The current Azure Sphere software release does not support all features of the MT3620 hardware.
The following are not yet supported in software:
- 2 x ARM Cortex-M4 with FPU
- ADC, I2C, I2S, PWM and SPI peripheral interfaces (GPIO and UART are supported)
- Wi-Fi 802.11a (b/g/n are supported)
- RTC with clock selection and battery backup

Azure Sphere SDK and the C Programming Language

Over the past 25 years, Microsoft has worked very hard to develop Win32, .NET, C#, and now UWP. The goal was to provide high-level language support that abstracts much of the memory management tasks and to make GUI programming simpler. Azure Sphere doesn't have a GUI and doesn't support any of the modern high-level languages. The application programming language for Azure Sphere is C. More specifically, Azure Sphere supports a variant of POSIX C. One interesting deviation is that "printf" function is replaced with a "log_debug" function to send messages to the debug serial output rather than to a display. The SDK adds the Azure Sphere libraries and example projects to Visual Studio 2017.


New Project
Recent
Installed
Visual C#
Visual Basic
Visual C++
Windows Desktop
Cross Platform
CMake
Linux
Azure Sphere
MFC/ATL
Windows Universal
Test
Other
Basic C Language Project
Extensibility
Sort by: Default
Search (Ctrl+E)
Blink Sample for MT3620 RDB (Azure Sphere) Visual C++
UART Sample for MT3620 RDB (Azure Sphere) Visual C++
Azure IoT Hub Sample for MT3620 RDB (Azure Sphere) Visual C++
Blank Application for MT3620 RDB (Azure Sphere) Visual C++
Blank Static Library (Azure Sphere) Visual C++
Type: Visual C++

A blank C application for the MT3620 Reference Development Board (RDB).

The online getting started guide (https://azure.microsoft.com/en-us/services/azure-sphere/get-started/) provides the basic steps to set up and run applications on the MT3620 RDB. The example projects cover GPIO, serial, and connecting to the Azure IoT HUB. The "Blank Application for MT3620 RDB" provides a basic starting point to develop your own application. When the application is run as is, it sends HelloWorld messages out the debug port which can then be seen in the Visual Studio output pane. Below is the code listing for the Blank Application's main.c file, and here is a general overview of the Blank Application:

1. Lines 1-5 include the standard header libraries. If you want to see all the libraries available just type in #include and IntelliSense will show all the available header files.
2. Line 11 includes the MT3620 RDB specific header to map the IO to names useful to code with.
3. Line 19, Lines 24-28, and Lines 37-41 create a termination request event. Linux kernel can raise this signal to terminate the application.
4. Lines 44-48 – The main loop which checks for the termination and outputs the HelloWorld message. The loop keeps the application alive. If there was no loop the application would just terminate. Within the loop, you can then add your own code to perform custom actions.

```
1.  #include <signal.h>
2.  #include <stdbool.h>
3.  #include <stdlib.h>
4.  #include <string.h>
5.  #include <time.h>
6.  
7.  // applibs_versions.h defines the API struct versions to use for applibs APIs.
8.  #include "applibs_versions.h"
9.  #include <applibs/log.h>
10. 
11. #include "mt3620_rdb.h"
12. 
13. // This C application for the MT3620 Reference Development Board (Azure Sphere)
14. // outputs a string every second to Visual Studio's Device Output window
15. //
16. // It uses the API for the following Azure Sphere application libraries:
17. // - log (messages shown in Visual Studio's Device Output window during
    debugging)
18. 
19. static volatile sig_atomic_t terminationRequested = false;
20. 
21. /// <summary>
22. ///     Signal handler for termination requests. This handler must be async-
    signal-safe.
23. /// </summary>
24. static void TerminationHandler(int signalNumber)
```

```
{
    // Don't use Log_Debug here, as it is not guaranteed to be async signal safe
    terminationRequested = true;
}

/// <summary>
///     Main entry point for this sample.
/// </summary>
int main(int argc, char *argv[])
{
    Log_Debug("Application starting\n");

    // Register a SIGTERM handler for termination requests
    struct sigaction action;
    memset(&action, 0, sizeof(struct sigaction));
    action.sa_handler = TerminationHandler;
    sigaction(SIGTERM, &action, NULL);

    // Main loop
    const struct timespec sleepTime = {1, 0};
    while (!terminationRequested) {
        Log_Debug("Hello world\n");
        nanosleep(&sleepTime, NULL);
    }

    Log_Debug("Application exiting\n");
    return 0;
}
```

Another very important file in the project is the App_manifest.json. Below is the code listing for this file:

```
{
 "SchemaVersion": 1,
 "Name" : "BlankApp",
 "ComponentId" : "553a769c-f3ea-44f8-b115-0a37f02d5e30",
 "EntryPoint": "/bin/app",
 "CmdArgs": [],
 "TargetApplicationRuntimeVersion": 1,
 "Capabilities": {
  "AllowedConnections": [],
  "Gpio": [],
  "Uart": [],
  "WifiConfig": false
 }
}
```

Since security is a key foundation of Azure Sphere, applications are limited to accessing only the IO that they need. Within the App_manifest.json, you set up what hardware you want the application to have access to. It is easy to forget that this has to be set up when accessing hardware. If the application tried to access IO that is not set up in the json file, a permission denied error will be shown in the debug output:

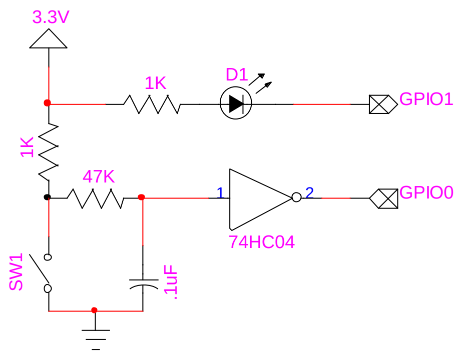
```
Opening GPIO
ERROR: Could not open GPIO0: Permission denied (13).
```

USB Power HUB

Some laptops, like the Surface, have limited power output capability of their USB ports. Since there could be various devices connected to the MT3620 RDB pin headers drawing current, we used a powered USB hub (https://www.sabrent.com/product/HB-UMLS/4-port-usb-2-0-hub-power-switches-black/) to power the Azure Sphere during our testing.

GPIO – Latency Test

The Blinky application demonstrates GPIO input and output using the onboard buttons and LEDs. Knowing interrupt latency is an issue for some projects, so we modified the Blinky application so that the GPIO0 pin is used as the input and GPIO1 pin is used as the output. We could then use an Oscilloscope to measure the interrupt latency. A 74HC04 was used for a debounce circuit to send an input signal to GPIO0, and an LED pulled to 3.3V was used for the GPIO1 output. The 3.3V supply was from the 3.3 Pin on Header 3. Here is the circuit:



Note: A diode would normally be put in parallel of the 47KΩ resistor, but the circuit worked fine as is.

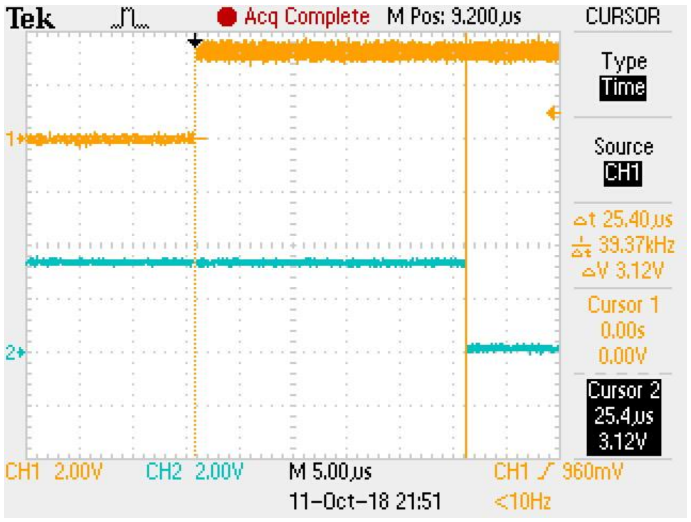
Interrupt handling at the application level is actually polling using the epoll Linux system call. GPIO pins are linked to event handlers using the epoll_ctl call in the separate epoll_timerfd_utilities.c. Here is the main.c application code listing used to perform the test:

```c
#include <signal.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <errno.h>
#include "epoll_timerfd_utilities.h"

// applibs_versions.h defines the API struct versions to use for applibs APIs.
#include "applibs_versions.h"
#include <applibs/log.h>
#include <applibs/gpio.h>

#include "mt3620_rdb.h"

// This C application for the MT3620 Reference Development Board (Azure Sphere)
// outputs a string every second to Visual Studio's Device Output window
//
// It uses the API for the following Azure Sphere application libraries:
// - log (messages shown in Visual Studio's Device Output window during debugging)

static int gpio0Fd = -1;
static int gpio1Fd = -1;
static int epollFd = -1;
static int gpioButtonTimerFd = -1;
static GPIO_Value_Type outValue = 0;
static GPIO_Value_Type buttonState = GPIO_Value_High;
static volatile sig_atomic_t terminationRequested = false;

/// <summary>
///     Signal handler for termination requests. This handler must be async-signal-safe.
/// </summary>
static void TerminationHandler(int signalNumber)
{
    // Don't use Log_Debug here, as it is not guaranteed to be async signal safe
    terminationRequested = true;
}
static void ButtonEventHandler()
{
    // Check for a button press
    GPIO_Value_Type newButtonState;
    int result = GPIO_GetValue(gpio0Fd, &newButtonState);
    if (result != 0) {
        Log_Debug("ERROR: Could not read button GPIO: %s (%d).\n", strerror(errno), errno);
        terminationRequested = true;
        return;
    }

    if (newButtonState != buttonState) {

        if (newButtonState == GPIO_Value_High) {
            // If the button has just been pressed, change the GPIO0 state
            GPIO_GetValue(gpio1Fd, &outValue);
            if (outValue == 0) {
                GPIO_SetValue(gpio1Fd, GPIO_Value_High);
            }
            else {
                GPIO_SetValue(gpio1Fd, GPIO_Value_Low);
            }
        }
    }

    buttonState = newButtonState;
}

/// <summary>
///     Main entry point for this sample.
/// </summary>
int main(int argc, char *argv[])
{
    Log_Debug("Application starting\n");

    // Register a SIGTERM handler for termination requests
    struct sigaction action;
    memset(&action, 0, sizeof(struct sigaction));
    action.sa_handler = TerminationHandler;
    sigaction(SIGTERM, &action, NULL);

    Log_Debug("Opening GPIO\n");
    gpio1Fd = GPIO_OpenAsOutput(MT3620_GPIO1, GPIO_OutputMode_PushPull, GPIO_Value_Low);
    if (gpio1Fd < 0) {
        Log_Debug("ERROR: Could not open GPIO1: %s (%d).\n", strerror(errno), errno);
        terminationRequested = true;
    }
    gpio0Fd = GPIO_OpenAsInput(MT3620_GPIO0);
    if (gpio0Fd < 0) {
        Log_Debug("ERROR: Could not open GPIO0: %s (%d).\n", strerror(errno), errno);
        terminationRequested = true;
    }

    Log_Debug("Create Epoll File Descripter and create handler\n");
    epollFd = CreateEpollFd();
    if (epollFd < 0) {
        terminationRequested = true;
    }
    struct timespec buttonPressCheckPeriod = { 0, 1000000 };
    gpioButtonTimerFd = CreateTimerFdAndAddToEpoll(epollFd, &buttonPressCheckPeriod,
        &ButtonEventHandler, EPOLLIN);
    if (gpioButtonTimerFd < 0) {
        terminationRequested = true;
    }
    Log_Debug("Ready\n");
    // Main loop
    while (!terminationRequested) {
        if (WaitForEventAndCallHandler(epollFd) != 0) {
            terminationRequested = true;
        }
    }
    Log_Debug("Application exiting\n");
    return 0;
}
```

Here is the App_manifest.json code listing showing GPIO 0 and 1 made available for the application:

```
{
  "SchemaVersion": 1,
  "Name" : "MyLatency",
  "ComponentId" : "2bde7b5a-7931-405e-875b-2cbb0b64fc38",
  "EntryPoint": "/bin/app",
  "CmdArgs": [],
  "TargetApplicationRuntimeVersion": 1,
  "Capabilities": {
    "AllowedConnections": [],
    "Gpio": [0,1],
    "Uart": [],
    "WifiConfig": false
  }
}
```

Connecting an O-scope to the input and output and configuring the scope to single signal trigger, the results of a press of the debounce circuit button shows the delay between the input signal going high and the response to the output.



Performing several runs of this test shows that the response was around 25µs. Quite a few times less than 25µs, and once in a while above 25µs (around 28 to 30µs). The application was running in debug mode so the performance will be different when running in release mode. A key to this test is the checking of the button state. Since there are two events with each press of the button: high and low, the "buttonstate" check helps to avoid a change on the output when the input signal goes low.

Serial – Serial Out to LCD

The "UART Sample for the MT3620 RDB" uses a loopback to test serial input and output. We decided to create a serial output application. Since Azure Sphere doesn't support a GUI, some devices might use a 2-line Serial LCD for text output. We used an older Serial LCD from Sparkfun (https://www.sparkfun.com/products/retired/9066). Sparkfun has a few other 3.3V Serial LCD solutions available. The application is a variation of the "UART Sample for the MT3620 RDB" project. Below is the code listing.

1. Lines 75-87 make all the available Serial LCD command available as constant char arrays. The 0x00 is for termination.
2. Lines 91-97 configure the UART3
3. Lines 107-114 exercise some of the commands and send two messages to the Serial LCD. The \0 is required in the message to terminate the call.

```
1.  #include <signal.h>
2.  #include <stdbool.h>
3.  #include <stdlib.h>
4.  #include <string.h>
5.  #include <time.h>
6.  #include <errno.h>
7.  #include <unistd.h>
8.  
9.  // applibs_versions.h defines the API struct versions to use for applibs APIs.
10. #include "applibs_versions.h"
11. #include <applibs/log.h>
12. #include <applibs/uart.h>
13. #include <applibs/gpio.h>
14. 
15. #include "mt3620_rdb.h"
16. 
17. // This C application for the MT3620 Reference Development Board (Azure Sphere)
18. // outputs a string every second to Visual Studio's Device Output window
19. //
20. // It uses the API for the following Azure Sphere application libraries:
```

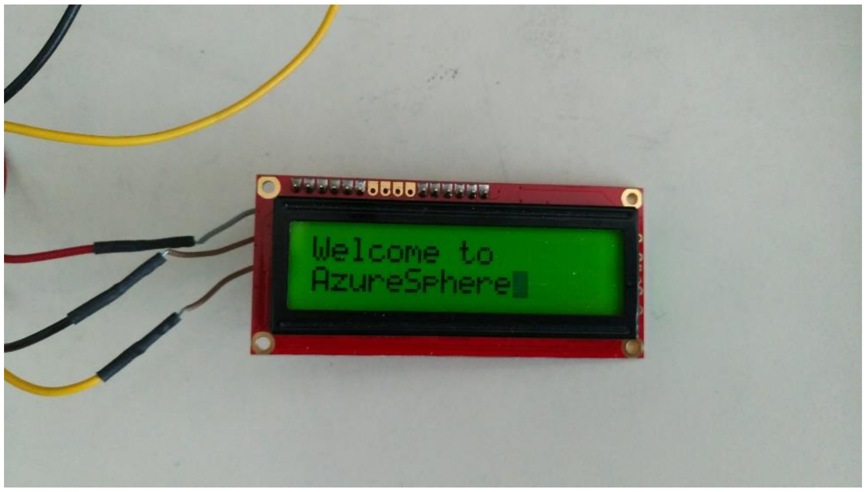
```
// - log (messages shown in Visual Studio's Device Output window during debugging)


static int uartFd = -1;
static volatile sig_atomic_t terminationRequested = false;


/// <summary>
///   Signal handler for termination requests. This handler must be async-signal-safe.
/// </summary>
static void TerminationHandler(int signalNumber)
{
  // Don't use Log_Debug here, as it is not guaranteed to be async signal safe
  terminationRequested = true;
}

static void SendUartMessage(int uartFd, const char *dataToSend)
{
  size_t totalBytesSent = 0;
  size_t totalBytesToSend = strlen(dataToSend);
  int sendIterations = 0;
  while (totalBytesSent < totalBytesToSend) {
    sendIterations++;

    // Send as much of the remaining data as possible
    size_t bytesLeftToSend = totalBytesToSend - totalBytesSent;
    const char *remainingMessageToSend = dataToSend +totalBytesSent;
    ssize_t bytesSent = write(uartFd, remainingMessageToSend, bytesLeftToSend);
    if (bytesSent < 0) {
      Log_Debug("ERROR: Could not write to UART: %s (%d).\n", strerror(errno), errno);
      terminationRequested = true;
      return;
    }

    totalBytesSent += (size_t)bytesSent;
  }

  Log_Debug("Sent %zu bytes over UART in %d calls\n", totalBytesSent, sendIterations);
}

/// <summary>
///   Main entry point for this sample.
/// </summary>
int main(int argc, char *argv[])
{
  Log_Debug("Application starting\n");

  // Register a SIGTERM handler for termination requests
  struct sigaction action;
  memset(&action, 0, sizeof(struct sigaction));
  action.sa_handler = TerminationHandler;
  sigaction(SIGTERM, &action, NULL);

  //Define the control commands for the Sparkfun Serial LCD
  char LCDCLEAR[] = { 0xFE, 0x01, 0x00 };
  char LCDLINE1[] = { 0xFE, 0x80, 0x00 };
  char LCDLINE2[] = { 0xFE, 0xC0, 0x00 };
  char LCDMOVECURR1[] = { 0xFE, 0x14, 0x00 };
  char LCDMOVECURL1[] = { 0xFE, 0x10, 0x00 };
```

```c
  char LCDSCROLLRIGHT[] = { 0xFE, 0x1C, 0x00 };
  char LCDSCROLLLEFT[] = { 0xFE, 0x18, 0x00 };
  char LCDVISUALON[] = { 0xFE, 0x0C, 0x00 };
  char LCDVISUALOFF[] = { 0xFE, 0x08, 0x00 };
  char LCDCURON[] = { 0xFE, 0x0E, 0x00 };
  char LCDCUROFF[] = { 0xFE, 0x0C, 0x00 };
  char LCDBLINKON[] = { 0xFE, 0x0D, 0x00 };
  char LCDBLINKOFF[] = { 0xFE, 0x0C, 0x00 };


  // Create a UART_Config object, open the UART and set up UART event handler
  UART_Config uartConfig;
  UART_InitConfig(&uartConfig); //Setup basic config before device specific config
  uartConfig.baudRate = 9600;
  uartConfig.flowControl = UART_FlowControl_None;
  uartConfig.parity = UART_Parity_None;
  uartConfig.stopBits = UART_StopBits_One;
  uartConfig.dataBits = UART_DataBits_Eight;

  //Open UART 3 (ISU3)
  uartFd = UART_Open(MT3620_RDB_HEADER3_ISU3_UART, &uartConfig);
  if (uartFd < 0) {
    Log_Debug("ERROR: Could not open UART: %s (%d).\n", strerror(errno), errno);
    terminationRequested = true;
  }

  //now send the LCD commands and message
  SendUartMessage(uartFd, &LCDCLEAR);
  SendUartMessage(uartFd, &LCDCURON);
  SendUartMessage(uartFd, &LCDBLINKON);
  const char *messageToSend = "Welcome to\0";
  SendUartMessage(uartFd, messageToSend);
  SendUartMessage(uartFd, &LCDLINE2);
  const char *messageToSend2 = "AzureSphere\0";
  SendUartMessage(uartFd, messageToSend2);

  // Main loop
  const struct timespec sleepTime = {1, 0};
  while (!terminationRequested) {
    Log_Debug("Hello world\n");
    nanosleep(&sleepTime, NULL);
  }

  Log_Debug("Application exiting\n");
  return 0;
}
```

Here is the App_manifest.json code listing showing UART3 made available to the application:

```json
{
  "SchemaVersion": 1,
  "Name" : "MySerialLCDazs",
  "ComponentId" : "19d4a0fb-294d-4b08-96ca-ae366eeb1602",
  "EntryPoint": "/bin/app",
  "CmdArgs": [],
  "TargetApplicationRuntimeVersion": 1,
  "Capabilities": {
    "AllowedConnections": [],
    "Gpio": [],
    "Uart": ["ISU3"],
    "WifiConfig": false
  }
}
```

When the application runs, the message appears on the Serial LCD display.



Multithreaded Applications using pthread

Since Azure Sphere implements POSIX-C, another fun test was to try running a multi-threaded application using pthread (https://en.wikipedia.org/wiki/POSIX_Threads) / (https://computing.llnl.gov/tutorials/pthreads/). The following code listing has a main thread and 2 separate threads to increment / decrement 3 integer variables.

1. Lines 33-39 and Lines 41-46 are the separate functions for incrementing x and decrementing z respectively.
2. Lines 67-73 and Lines 74-80 create the threads to the functions.
3. Main thread will run and the application waits for the two other threads to finish.

```
1.  #include <signal.h>
2.  #include <stdbool.h>
3.  #include <stdlib.h>
4.  #include <string.h>
5.  #include <time.h>
6.  #include <pthread.h>
7.  #include <unistd.h>
8.  #include <errno.h>
9.  
10. // applibs_versions.h defines the API struct versions to use for applibs APIs.
11. #include "applibs_versions.h"
12. #include <applibs/log.h>
13. 
14. #include "mt3620_rdb.h"
15. 
16. // This C application for the MT3620 Reference Development Board (Azure Sphere)
17. // outputs a string every second to Visual Studio's Device Output window
18. //
19. // It uses the API for the following Azure Sphere application libraries:
20. // - log (messages shown in Visual Studio's Device Output window during
    debugging)
21. 
22. static volatile sig_atomic_t terminationRequested = false;
23. 
24. /// <summary>
25. ///     Signal handler for termination requests. This handler must be async-
    signal-safe.
```

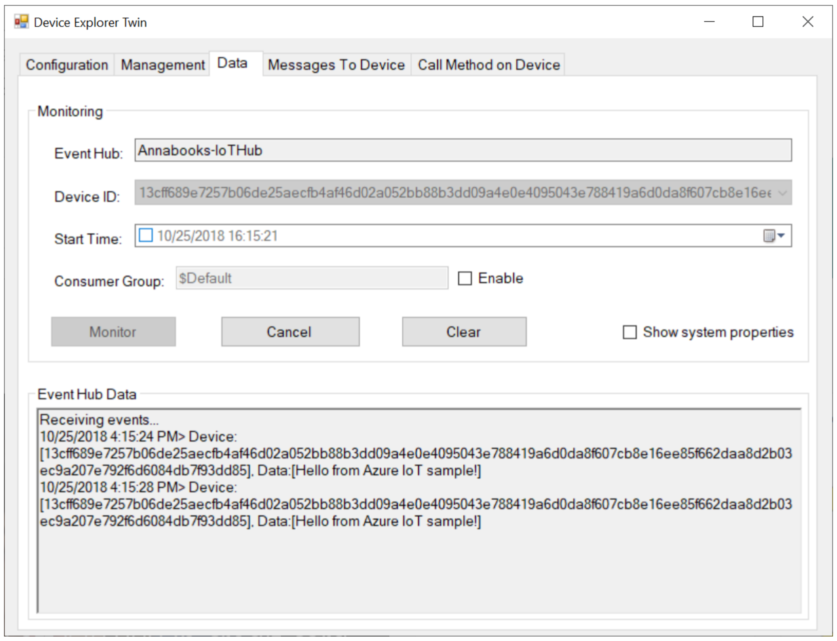
```
/// </summary>
static void TerminationHandler(int signalNumber)
{
    // Don't use Log_Debug here, as it is not guaranteed to be async signal safe
    terminationRequested = true;
}

void *inc_x(void *x_void_ptr) {

    int *x_ptr = (int *)x_void_ptr;
    while (++(*x_ptr) < 15);
    Log_Debug("x increment finished\n");
    return NULL;
}

void *dec_z(void *z_void_ptr) {
    int *z_ptr = (int *)z_void_ptr;
    while (--(*z_ptr) > 25);
    Log_Debug("z decrement finished\n");
    return NULL;
}


/// <summary>
///     Main entry point for this sample.
/// </summary>
int main(int argc, char *argv[])
{
    Log_Debug("Application starting\n");

    // Register a SIGTERM handler for termination requests
    struct sigaction action;
    memset(&action, 0, sizeof(struct sigaction));
    action.sa_handler = TerminationHandler;
    sigaction(SIGTERM, &action, NULL);

    //setup the variables
    int x = 0, y = 0, z=100;
    Log_Debug("x: %d, y: %d, z: %d\n", x, y, z);


    // Second thread
    pthread_t inc_x_thread;
    // create a second thread which executes inc_x(&x)
    if (pthread_create(&inc_x_thread, NULL, inc_x, &x)) {
        Log_Debug("Error creating thread %s (%d).\n", strerror(errno), errno);
        return 1;
    }
    // Third thread
    pthread_t dec_z_thread;
    // create a third thread which executes dec_z(&z)
    if (pthread_create(&dec_z_thread, NULL, dec_z, &z)) {
        Log_Debug("Error creating thread %s (%d).\n", strerror(errno), errno);
        return 1;
    }

    // increment y to 100 in the first thread
    while (++y < 100);
    Log_Debug("y increment finished\n");


    // wait for the second thread to finish
    if (pthread_join(inc_x_thread, NULL)) {
            Log_Debug("Error joining thread %s (%d).\n", strerror(errno), errno);
            return 2;
        }

        // wait for the thrid thread to finish
        if (pthread_join(dec_z_thread, NULL)) {
            Log_Debug("Error joining thread %s (%d).\n", strerror(errno), errno);
            return 2;
        }

        // show the final results
        Log_Debug("\nx: %d, y: %d, z: %d\n", x, y, z);


        Log_Debug("Application exiting\n");
        return 0;
}
```

Here is the output when running the application under the debugger:

```
Application starting
x: 0, y: 0, z: 100
x increment finished
y increment finished
z decrement finished

x: 15, y: 100, z: 25
Application exiting

Child exited with status 0
```

Azure IoT HUB / Device Explorer

Azure Sphere wouldn't be Azure Sphere without Azure. The last test we ran was to connect to Azure IoT HUB following the "Azure IoT HUB Sample for the MT3620 RDB" project. After going through the online documents to set up an IoT HUB, a provisioning service, installing Device Explorer locally, and the certificate verification with the device, we were able to connect the Device Explorer to the MT3620 RDB via Azure.



Event Hub: Annabooks-IoTHub

Device ID: 13cff689e7257b06de25aecfb4af46d02a052bb88b3dd09a4e0e4095043e788419a6d0da8f607cb8e16e

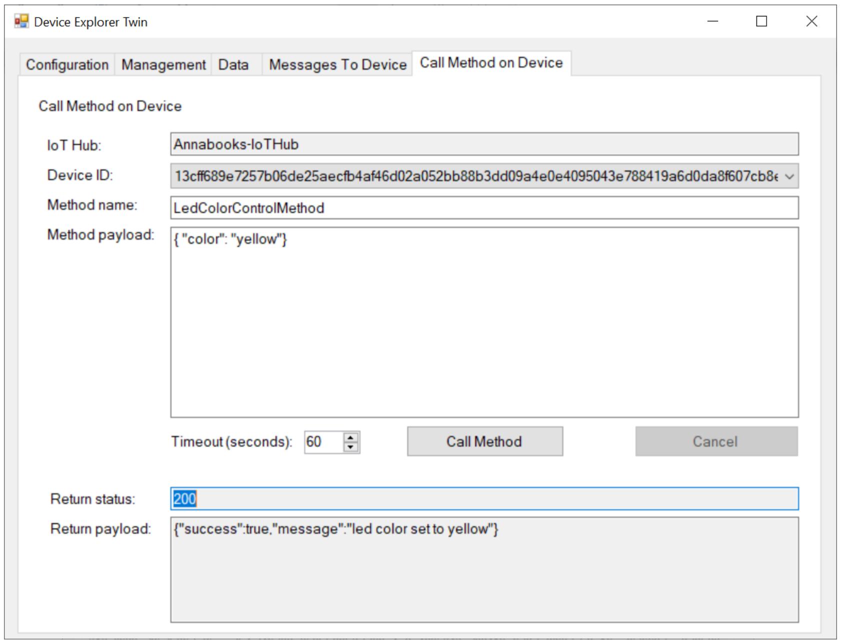
Start Time: ☐ 10/25/2018 16:15:21

Consumer Group: $Default ☐ Enable

☐ Show system properties

```
Receiving events...
10/25/2018 4:15:24 PM> Device:
[13cff689e7257b06de25aecfb4af46d02a052bb88b3dd09a4e0e4095043e788419a6d0da8f607cb8e16ee85f662daa8d2b03
ec9a207e792f6d6084db7f93dd85], Data:[Hello from Azure IoT sample!]
10/25/2018 4:15:28 PM> Device:
[13cff689e7257b06de25aecfb4af46d02a052bb88b3dd09a4e0e4095043e788419a6d0da8f607cb8e16ee85f662daa8d2b03
ec9a207e792f6d6084db7f93dd85], Data:[Hello from Azure IoT sample!]
```

We could also remotely change the color of the LED per the online documents.



Call Method on Device

IoT Hub: Annabooks-IoTHub
Device ID: 13cff689e7257b06de25aecfb4af46d02a052bb88b3dd09a4e0e4095043e788419a6d0da8f607cb8e
Method name: LedColorControlMethod
Method payload: { "color": "yellow"}
Timeout (seconds): 60
Return status: 200
Return payload: {"success":true,"message":"led color set to yellow"}

Conclusion: Our Take

One could go through a SWOT analysis or simply list the pros and cons of Azure Sphere, but we will just provide our viewpoint of this preview release. First, we have been through many preliminary, beta, and preview programs with Microsoft. Although, we are limited to GPIO, UART, and WIFI, Azure Sphere SDK preview has been a pleasant surprise. No major crashes of the development system and the MT3620 RDB appears to be a solid platform. Microsoft might be learning some lessons from the past when taking a product from R&D into production. Testing the other IO and the final release will be important next steps.

Second, Azure Sphere is pushing for the service model. In the ’90s, there was always humorous comments about surfing the web on your appliances. Microsoft has used the washing machine as an Azure Sphere example. Is there really a need to connect your washing machine to the Internet? With so many companies moving to a service model, will anyone be willing to pay for a washing machine service to monitor system health? Maybe, Azure Sphere would be a good solution and service for laundromats and hotel guest laundry, but home consumers might not see the cost benefit. There are many solutions beyond appliances that might take advantage of this solution. Manufacturing control systems, building temperature controls and security, farming machinery, and traffic control systems, and even public utilities could benefit from this solution.

Finally, in comparison to Windows IoT Enterprise, Windows IoT Mobile, and Windows IoT Core, Azure Sphere is the smallest embedded/IoT offering from Microsoft. Will Windows IoT Core and Azure Sphere cannibalize each other’s market space? That remains to be seen. Azure Sphere will beat any hardware cost, but Windows IoT Core supports a GUI. With other players getting into the low-end MCU game, Azure Sphere could carve out some space in the 32-bit MCU market if it gets the proper internal development effort, backing, and execution.

As a side note, we do find it interesting that some folks at Microsoft reached out to us with concern about our *Open Software Stack for the Intel® Atom™ Processor* book that covered creating a Linux distribution with the Yocto Project. All the while, Microsoft was working on Azure which runs on Linux, and now we have Azure Sphere. *Interesting*….

Resources

https://azure.microsoft.com/en-us/services/azure-sphere/

https://docs.microsoft.com/en-us/azure-sphere/

https://www.mediatek.com/products/iot/azure-sphere

https://d86o2zu8ugzlg.cloudfront.net/mediatek-craft/documents/mt3620/MediaTek-MT3620-Product-Brief-May2018.pdf

https://github.com/MicrosoftDocs/azure-sphere-issues/issues