AN1114: Integrating v2.x Silicon Labs *Bluetooth* ® Applications with the Micrium RTOS

This application note provides background information on the system architecture and event-based communication between the RTOS and the Bluetooth application. It then discusses user-defined tasks and describes how to customize an application.

KEY POINTS

- Prerequisites
- Intertask communication and task descriptions
- Application integration using specific example tasks
- Customization instructions

1. Introduction

This application note describes how to integrate a Silicon Labs Bluetooth application with an RTOS (real-time operating system), using the **SOC-Thermometer-RTOS** example in Simplicity Studio as an illustration. As of Silicon Labs Bluetooth SDK version 2.6.0.0, the adaptation layer has been designed to work with Micrium OS. To work with any other RTOS, the OS should have the following features:

- Tasks with priorities
- Flags for triggering task execution from interrupt context
- Mutexes

The solution places the handling of Bluetooth stack events into its own task, allowing the application to run other tasks when no Bluetooth events are pending. When no tasks are ready to run, the application will sleep.

1.1 Prerequisites

You should have

- A general understanding of RTOS concepts such as tasks, semaphores and mutexes.
- A working knowledge of Bluetooth Low Energy communications.
- A Wireless starter kit with a Blue Gecko or Mighty Gecko radio board
- Installed and be familiar with using the following:
  - Simplicity Studio v4.1.4 or above
  - IAR Embedded Workbench for ARM (IAR-EWARM) (optional - only use the version that is compatible with the SDK version, as listed in the SDK release notes). May be used as a compiler in the Simplicity Studio development environment as an alternative to GCC (The GNU Compiler Collection), which is provided with Simplicity Studio. Again, use only the GCC version that is compatible with the SDK version, as listed in the SDK release notes.
  - Bluetooth SDK v2.6.0 or above

If you need to familiarize yourself with any of these concepts, the following may be useful:

- *UG103.14: Bluetooth LE Fundamentals*
- *QSG139: Getting Started with Bluetooth Software Development*
- µCOS-III Real Time Kernel for an overview of RTOS fundamentals

2. System Architecture

The **SOC-Thermometer - RTOS** example application requires several tasks in order to operate

- Bluetooth start task
- Link layer task
- Bluetooth host task
- Idle task
- Bluetooth application task

These have been implemented for the Micrium RTOS for you.

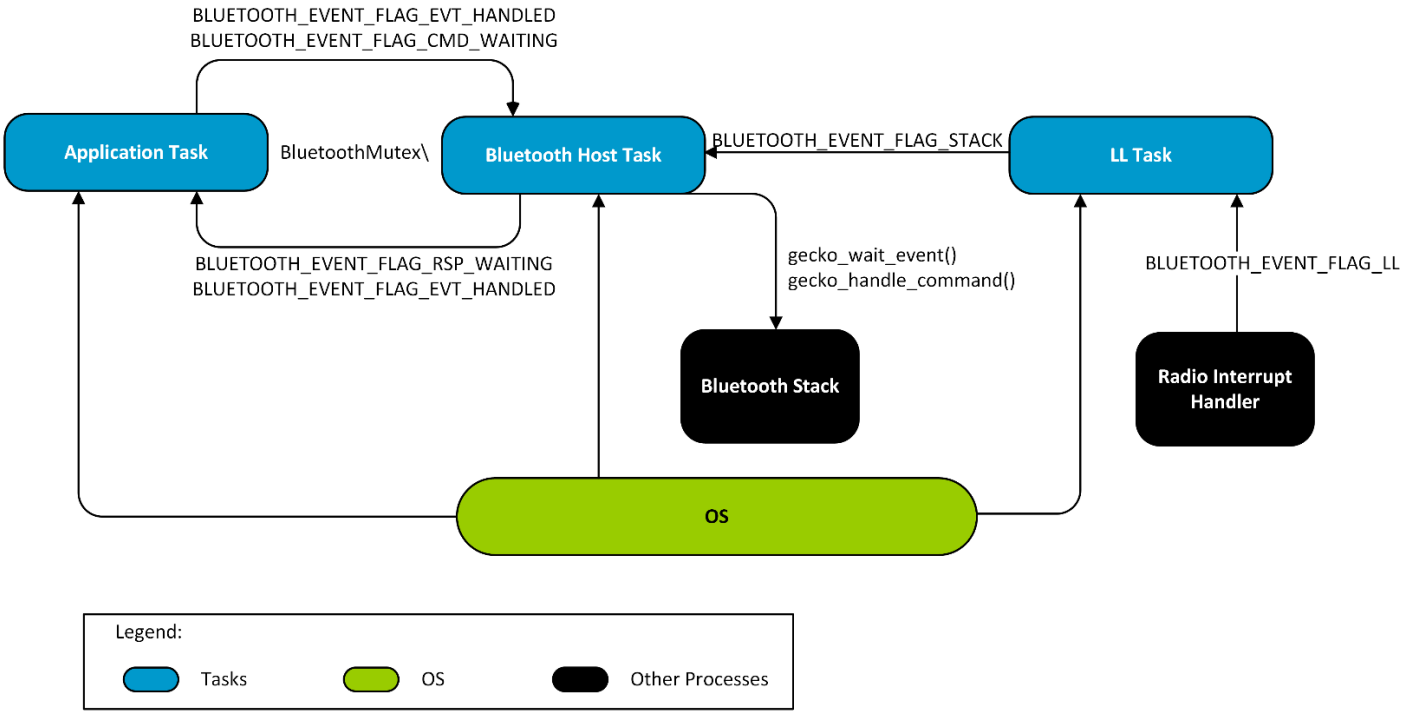
2.1 Inter-Task Communication

Before describing the tasks, it is important to understand how the tasks communicate with each other. The tasks in this application synchronize with each other through the use of a number of flags. These flags are summarized in the following table:

FLAG	Sender	Receiver	Purpose
`BLUETOOTH_EVENT_FLAG_STACK`	Link Layer task	Bluetooth Task	Bluetooth stack needs an update, call `gecko_wait_event()`
`BLUETOOTH_EVENT_FLAG_LL`	Radio interrupt	Link Layer Task	Link Layer needs an update, call `gecko_priority_handle()`
`BLUETOOTH_EVENT_FLAG_CMD_WAITING`	Application Task	Bluetooth Task	Command is ready in shared memory, call `gecko_handle_command()`
`BLUETOOTH_EVENT_FLAG_RSP_WAITING`	Bluetooth Task	Application Task	Response is ready in shared memory
`BLUETOOTH_EVENT_FLAG_EVT_WAITING`	Bluetooth Task	Application Task	Event is ready in shared memory
`BLUETOOTH_EVENT_FLAG_EVT_HANDLED`	Application Task	Bluetooth Task	Event is handled and shared memory is free to use for next event

The following diagram illustrates how these flags are used in synchronizing the tasks.

In addition to these flags, a mutex is used by the gecko command handler to make it thread-safe. This makes it possible to call BGAPI commands from multiple tasks.



2.2 Bluetooth Start Task

The purpose of this task is to prepare the application to run the Bluetooth stack by creating the Bluetooth host and link layer tasks. The application task configures the Bluetooth stack before calling `Bluetooth_start_task` which initializes the stack by calling `gecko_stack_init()`. Each class or group of functions in the BGAPI has an initializer function that must be called. Once this initialization is complete this task terminates.

2.3 Link Layer Task

The purpose of the link layer task is to update the upper link layer. The link layer task waits for the `BLUETOOTH_EVENT_FLAG_LL` flag to be set before running. The upper link layer is updated by calling `gecko_priority_handle()`. The BLUETOOTH_EVENT_FLAG_LL flag is set by `BluetoothLLCallback()`, which is called from a kernel-aware interrupt handler. This task is given the highest priority after the Bluetooth start task.

2.4 Bluetooth Host Task

The purpose of this task is to update the Bluetooth stack, issue events, and handle commands. This task has higher priority than any of the application tasks, but lower than the link layer task.

2.4.1 Updating the Stack

The Bluetooth stack must be updated periodically. The Bluetooth host task reads the next periodic update event from the stack by calling `gecko_can_sleep_ticks()`; the stack is updated by calling `gecko_wait_event()`. This allows the stack to process messages from the link layer as well as its own internal messages for timed actions that it needs to perform.

2.4.2 Issuing Events

The Bluetooth host task sets the BLUETOOTH_EVENT_FLAG_EVT_WAITING flag to indicate to the Bluetooth application task that an event is ready to be retrieved. Only one event can be retrieved at a time. The BLUETOOTH_EVENT_FLAG_EVT_WAITING flag is cleared by the application task when it has retrieved the event. The BLUETOOTH_EVENT_FLAG_EVT_HANDLED flag is set by the application task to indicate that event handling is complete.

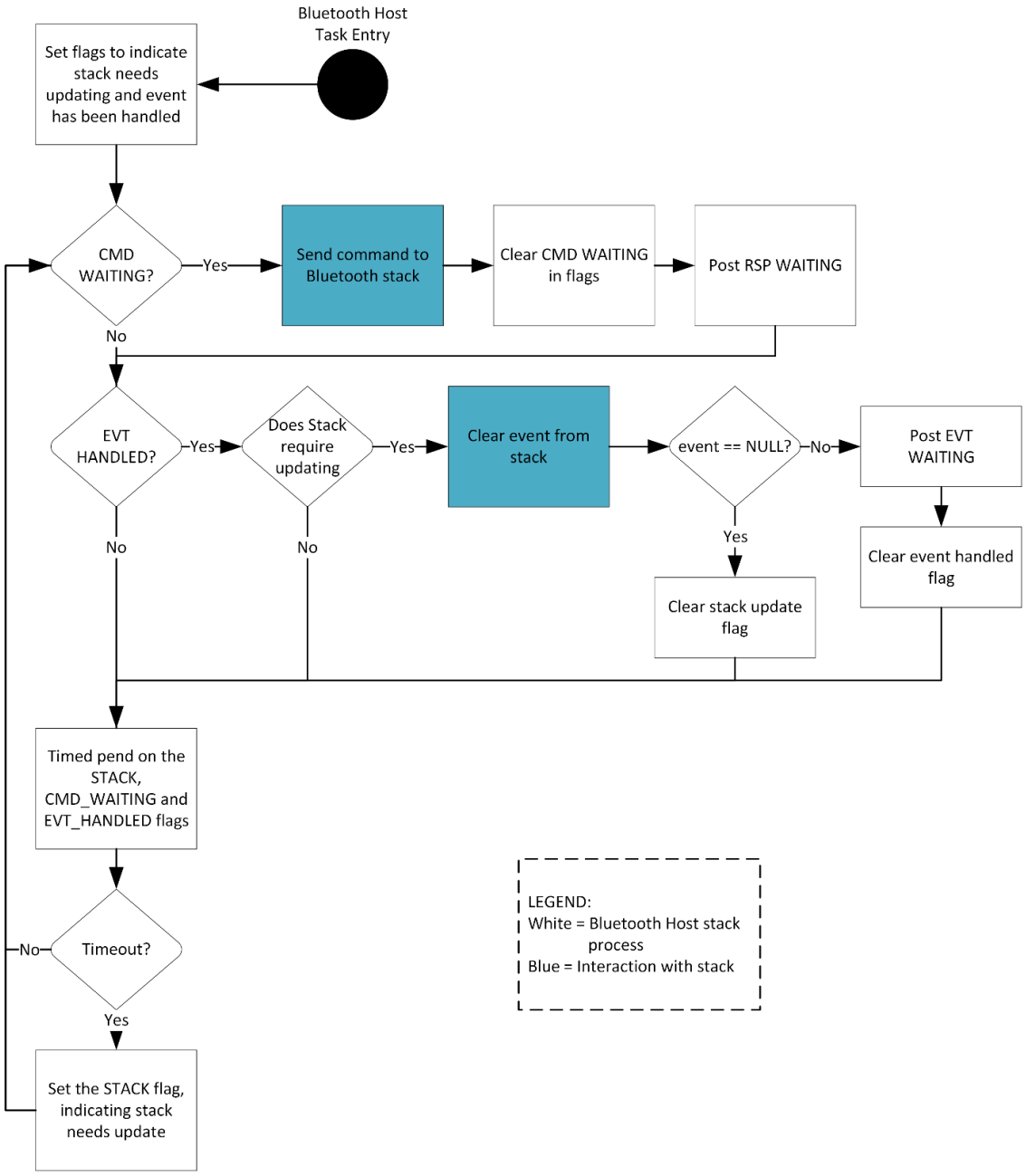
2.4.3 Command Handling

Commands can be sent to the stack from multiple tasks. Responses to these commands are forwarded to the calling task. Commands and responses are synchronized with the `BLUETOOTH_EVENT_FLAG_CMD_WAITING` and `BLUETOOTH_EVENT_FLAG_RSP_WAITING` flags and the BluetoothMutex mutex.

Commands are prepared and sent to the stack by a helper function called `rtos_gecko_handle_command()`. This function is called by any of the BGAPI functions and is made re-entrant through the use of a mutex. The function starts by pending on the mutex. When it gains control of the mutex the command is prepared and placed into shared memory, then the `BLUETOOTH_EVENT_FLAG_CMD_WAITING` flag is set to indicate to the stack that a command is waiting to be handled. This flag is cleared by the Bluetooth host task to indicate that the command has been sent to the stack and that it is now safe to send another command.

Then execution pends on the `BLUETOOTH_EVENT_FLAG_RSP_WAITING flag`, which is set by the Bluetooth host task when the command has been executed. This indicates that a response to the command is waiting. Finally, the mutex is released.

The following diagram illustrates how the Bluetooth Host task operates.



1. On task startup, the `BLUETOOTH_EVENT_FLAG_STACK` is set to indicate that the stack needs updating and the `BLUETOOTH_EVENT_FLAG_EVT_HANDLED` flag is set to indicate that no event is currently being handled.
2. Next, if the `BLUETOOTH_EVENT_FLAG_CMD_WAITING` flag is set, `gecko_handle_command()` is called to handle the command.

3. Then, if the `BLUETOOTH_EVENT_FLAG_STACK` and the `BLUETOOTH_EVENT_FLAG_EVT_HANDLED` flags are set, `gecko_wait_event()` is called to get an event from the stack. If an event is found waiting, the `BLUETOOTH_EVENT_FLAG_EVT_WAITING` flag is set and the `BLUETOOTH_EVENT_FLAG_EVT_HANDLED` flag is cleared to indicate to the application task that an event is ready to be handled and to the Bluetooth host task that an event is currently in the process of being handled. Otherwise, the `BLUETOOTH_EVENT_FLAG_STACK` flag is cleared to indicate that the stack does not require updating.
4. At this point, the task checks to see if the stack requires updating and whether any events are waiting to be handled. If no events are waiting to be handled and the stack does not need updating then it is safe to sleep and a call to `gecko_can_sleep_ticks()` is made to determine how long the system can sleep for. The Bluetooth host task then does a timed pend on the `BLUETOOTH_EVENT_FLAG_STACK`, `BLUETOOTH_EVENT_FLAG_EVT_HANDLED` and `BLUETOOTH_EVENT_FLAG_CMD_WAITING` flags.
5. If the timeout occurs and none of the flags are set in the time determined in step 4, then the `BLUETOOTH_EVENT_FLAG_STACK` is set to indicate that the stack requires updating.
6. Steps 2 – 5 are repeated indefinitely.

2.5 Idle Task

When no tasks are ready to run, the OS calls the idle task. The idle task puts the MCU into lowest available sleep mode, EM2, by default.

3. Application Integration

This section describes the user-defined tasks and how they are used to implement a sample Bluetooth device.

3.1 The Bluetooth Application Task

The purpose of the Bluetooth application task is to handle events sent by the Bluetooth stack. This task pends on the `BLUETOOTH_EVENT_FLAG_EVT_WAITING` flag. This flag is set by the Bluetooth host task to indicate that there is an event waiting to be handled. Once this flag has been set, `BluetoothEventHandler()` is called to handle the event. Finally, the `BLUETOOTH_EVENT_FLAG_EVT_HANDLED` flag is set to indicate to the Bluetooth host task that the event has been handled and the Bluetooth application task is ready to handle another event. This task has a lower priority than the Bluetooth host and link layer tasks.

3.1.1 Bluetooth Event Handler

The Bluetooth event handler, part of the Bluetooth Application task, takes a pointer to an event and handles it accordingly. A full list of events can be found in the Bluetooth API reference guide mentioned in the additional reading section below. Some events triggered by the stack are mainly informative and do not required the application to do anything. As this is a simple application, only a small set of events are handled. The events handled in this example are as follows

- system_boot

  This event indicates that the Bluetooth stack is initialized and ready to receive commands. This is where we set the discoverability and connectability modes.
- gatt_server_characteristic_status

  This event indicates a status change of a characteristic, it is triggered when client configuration has changed. When a client requests notifications or indications of a characteristic, `OSFlagPost()` is called here to set the HTM_STATUS_NOTIFY_ACTIVE_FLAG. This flag is used to indicate to the thermometer task that notifications should be sent periodically. This flag will be cleared when the client wants to stop listening to notifications of this characteristic.
- gatt_server_user_write_request

  This event is triggered when a write request is made for a characteristic that has the user type.
- le_connection_closed

  This event is triggered when a connection is closed. Advertising is restarted in this event to allow future connections.

3.2 The Thermometer Task

The thermometer task runs at 1-second intervals by using an RTOS timer to delay itself. This is done through a call to `OSTimeDlyHMSM()`. When the timer expires, the task increments a counter from a low value (20) to a high value (40) and then starts over. After incrementing the counter, the simulated temperature data is converted from a floating point value into a stream of bytes. Next, a call to `gecko_cmd_gatt_server_send_characteristic_notification()` is made to send the indication to any listening Bluetooth client. The process repeats indefinitely. The code is shown for reference in the following figure.

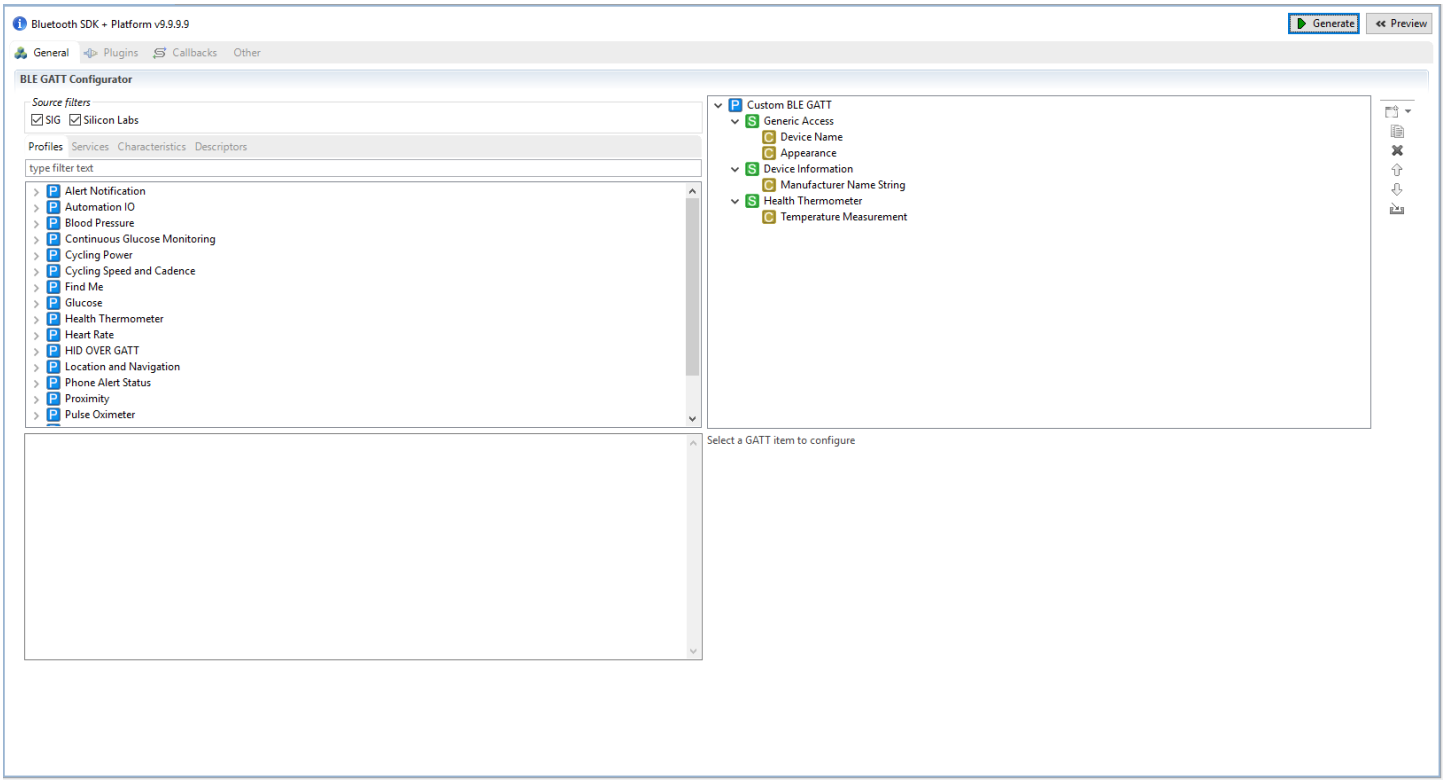
```
/* Task body, always written as an infinite loop.       */
while (DEF_TRUE) {
  OSTimeDlyHMSM(0, 0, 1, 0, OS_OPT_TIME_DLY | OS_OPT_TIME_HMSM_NON_STRICT, &err);

  temperature_counter++;
  if (temperature_counter > 40) {
    temperature_counter = 20;
  }

  uint8_t temp_buffer[5];
  bg_thermometer_create_measurement(temp_buffer,
                                    bg_uint32_to_float(temperature_counter, 0),
                                    0);
  gecko_cmd_gatt_server_send_characteristic_notification(0xff, gattdb_temp_measurement, 5, temp_buffer);
}
```

3.3 Customizing the Application

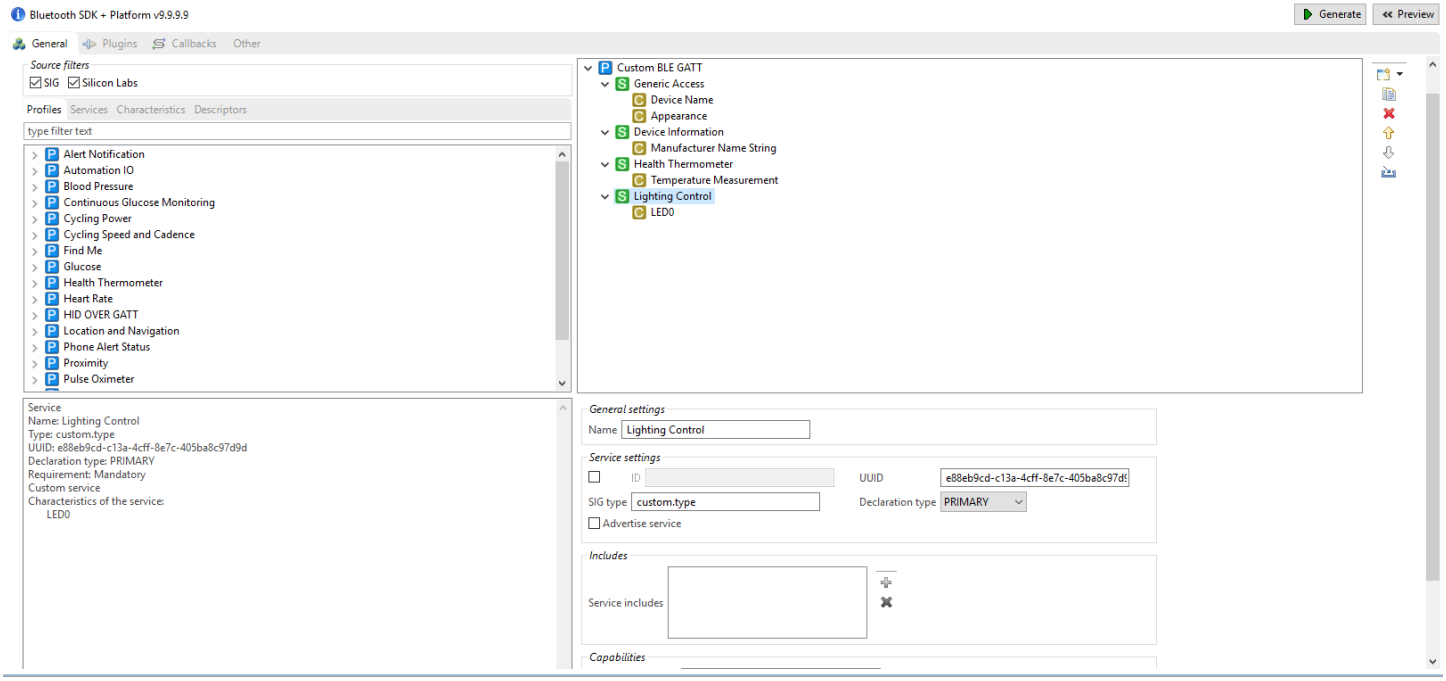
This section describes some common tasks such as customizing GATT attributes, adding event handlers, and adding support for other peripherals.

3.3.1 GATT Services and Characteristics

This section describes how to add a service and characteristic to control an LED on the wireless starter kit. One of the tools provided with Simplicity Studio is the Visual GATT Editor (VGE). This tool provides a graphical interface for creating and editing the GATT database. To open the VGE, double-click the *.isc file in the Simplicity Studio project. A window such as the following is displayed.



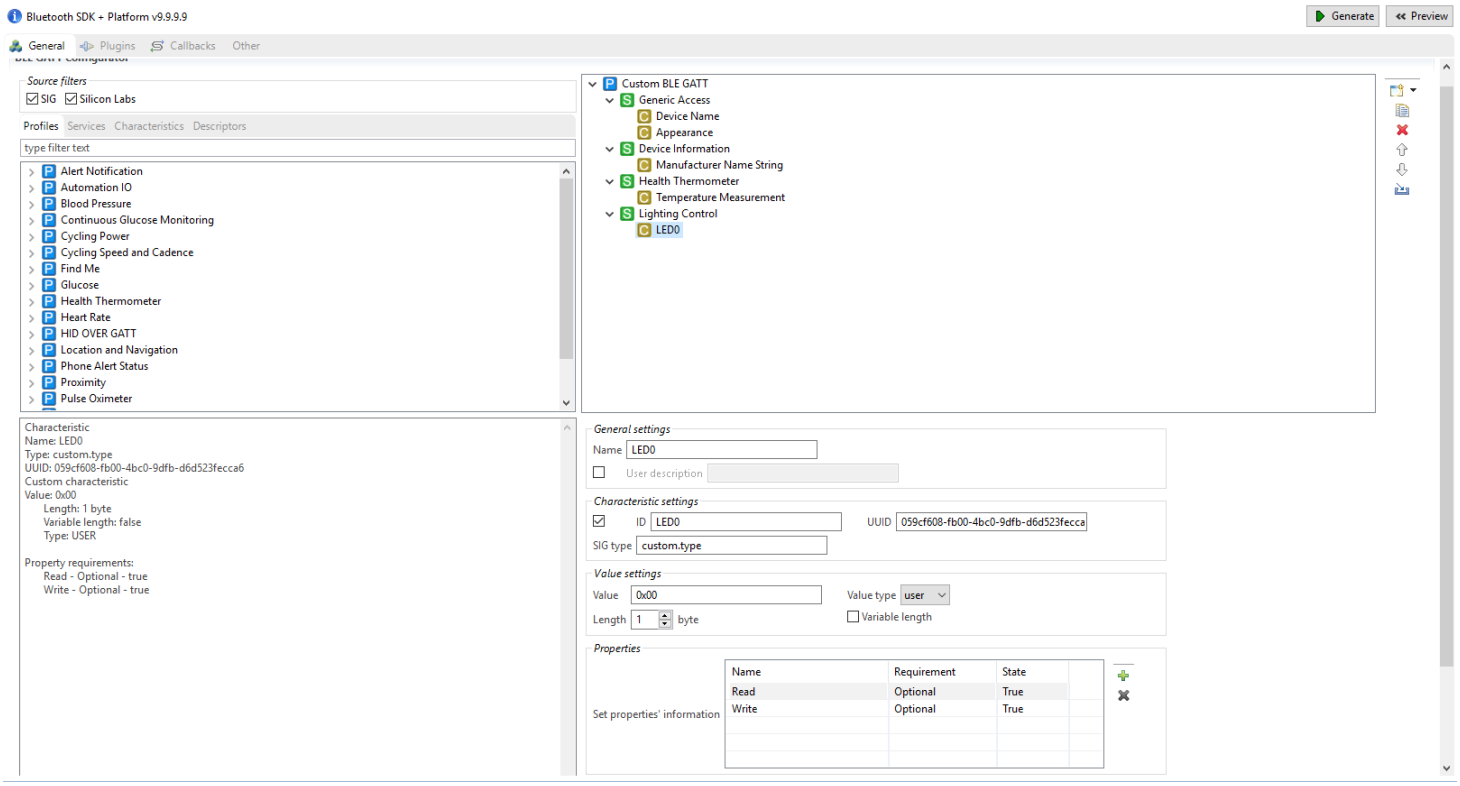
Create a new service by highlighting the “Custom BLE GATT” item at the top and click the new item dropdown in the top right corner, then select **New service**. Name the service **Lighting Control** as shown in the following figure.



Now add a characteristic:

1. Click the new item dropdown and select **New characteristic**.
2. Name the characteristic **LED0**.
3. Check the **id** checkbox and enter **LED0** for the ID.
4. Change **Value type** to **user**.
5. In the Properties section, click + to add the read and write properties to this characteristic.

The characteristic should now look like the following figure.



Click **Generate** to update the generated source code.

3.3.2 Event Handlers

This section discusses how to add an event handler for reading from and writing to the GATT characteristic added in the previous section. The characteristic has the write and read properties and is user type so the application needs to handle the following events:

- evt_gatt_server_user_write_request
- evt_gatt_server_user_read_request

Define a static variable at the global level to keep track of the state of the LED.

```
static uint8 led0State;
```

Include the following header file, which has declarations of functions to set and clear LEDs.

```
#include “bsp.h”
```

Add the following code to implement the user write request handler

```
case gecko_evt_gatt_server_user_write_request_id:
  if (evt->data.evt_gatt_server_user_write_request.characteristic == gattdb_LED0){
    if (evt->data.evt_gatt_server_user_write_request.value.data[0]){
      BSP_LedSet(0);
      }
    else {
      BSP_LedClear(0);
      }

gecko_cmd_gatt_server_send_user_write_response(evt->data.evt_gatt_server_user_write_request.connection,evt->
data.evt_gatt_server_user_write_request.characteristic,0);
  }
break;
```

This event handler verifies that the characteristic to be written is the LED0 characteristic, then turns the LED either on or off depending on the data written. Finally, it sends a response to the remote GATT client to indicate that the write has been performed.

It is also necessary to add the following code to the `system_boot` event handler to turn on the LED driver on the development board.

```
BSP_LedsInit()
```

Implement the handler for the user_read_request event by adding the following code to `BluetoothEventHandler()`.

```
case gecko_evt_gatt_server_user_read_request_id:
  if(evt->data.evt_gatt_server_user_read_request.characteristic == gattdb_LED0){
    led0State = BSP_LedGet(0);

gecko_cmd_gatt_server_send_user_read_response(evt->data.evt_gatt_server_user_read_request.connection,evt->data.
evt_gatt_server_user_read_request.characteristic,0,1,&led0State);
  }
break;
```

This event handler sends the state of the LED to the client. You can add similar handlers for other events in this way, as your application requires them. The API call used here requires more stack space so it is necessary to increase the amount of stack allocated to the Bluetooth application task as shown here.

```
//event handler task
#define APPLICATION_STACK_SIZE (1500 / sizeof(CPU_STK))
```

3.3.3 Adding Support for Other Peripherals

The easiest way to add support for other peripherals is through the use of Silicon Labs' emlib/emdrv peripheral libraries. These libraries contain APIs for initializing and controlling the EFR32 family's peripherals. A link to the documentation for these libraries is found in the additional reading section below.

A good place to set up peripherals is in the `system_boot` event, described in section 3.3.2 Event Handlers. This allows the application to wait until the Bluetooth stack is ready to run before initializing a peripheral. The Silicon Labs knowledgebase has several examples of adding support for other peripherals.

The board support package (BSP) includes functions for supporting commonly available peripherals such as LEDs and USART serial I/O.

USART

The USART can be used to direct STDIN and STDOUT to a serial console simply by adding a few source files (bsp_bcc.c, bsp_stk.c, retargetserial.c and retargetio.c) to the project and then calling an initialization function. These files are found in the following locations:

C:\SiliconLabs\SimplicityStudio\v4\developer\sdks\gecko_sdk_suite\<version>\hardware\kit\common\bsp\

C:\SiliconLabs\SimplicityStudio\v4\developer\sdks\gecko_sdk_suite\<version>\hardware\kit\common\drivers\

The bspconfig.h header file determines which USART is used and defines the pins used for transmit, receive and flow control if needed. The following call is made to initialize the USART

```
BSP_Init(BSP_INIT_BCC);
```

LEDs

The BSP also includes functions to initialize, set, clear, toggle and get the state of the LEDS. These are documented in the BSP section of the Gecko HAL & Driver API Reference Guide mentioned in section 5. Additional Reading.

Push Buttons

BGAPI functions cannot be called from an interrupt handler. Therefore, an interrupt-safe function called `gecko_external_signal()` is provided. This function can be called from an interrupt handler to indicate to the stack that an external interrupt has occurred. The signal parameter is a 32-bit integer treated as individual bit masks. The signal parameter is cleared when the `system_external_signal` event is raised by the stack. Sample initialization and interrupt handlers are shown in the following figure.

```
#include "em_gpio.h"
#include "em_int.h"
#include "gpiointerrupt.h"
#include "em_assert.h"
#include "em_common.h"
#include "native_gecko.h"
#include "em_system.h"
#include <stdio.h>

void ButtonHandler( uint8_t pin)
{
  int level;
  static int rise_count = 0, fall_count=0;
  if(pin == BSP_GPIO_PB0_PIN )  {
      gecko_external_signal(1);

  }
  else if(pin == BSP_GPIO_PB1_PIN ){
      gecko_external_signal(2);
  }
}
```

```
void setup_ext_interrupts(void)
{
   SYSTEM_ChipRevision_TypeDef *revision;
  /* config Button0 as input*/
  GPIO_PinModeSet(BSP_GPIO_PB0_PORT, BSP_GPIO_PB0_PIN, gpioModeInput, 0);

  GPIO_PinModeSet(BSP_GPIO_PB1_PORT, BSP_GPIO_PB1_PIN, gpioModeInput, 0);


  /* set up button 0 to generate an interrupt when it is either pressed
     or released */

  SYSTEM_ChipRevisionGet(revision);

  GPIO_IntConfig(BSP_GPIO_PB0_PORT,
          BSP_GPIO_PB0_PIN,
          false, //rising edge trigger
          true, //falling edge trigger
          true);
  GPIO_IntConfig(BSP_GPIO_PB1_PORT,
            BSP_GPIO_PB1_PIN,
            false, //rising edge trigger
            true, //falling edge trigger
            true);

  GPIOINT_Init();
  GPIOINT_CallbackRegister(BSP_GPIO_PB0_PIN,ButtonHandler);
  GPIO_IntEnable(1<<BSP_GPIO_PB0_PIN);
  GPIOINT_CallbackRegister(BSP_GPIO_PB1_PIN,ButtonHandler);
  GPIO_IntEnable(1<<BSP_GPIO_PB1_PIN);
}
```

An event handler for the `system_external_signal` event that sets a LED if a specific signal is set resembles the following figure.

```
case gecko_evt_system_external_signal_id:
    /* check to see if PB1 was pressed */
    if(evt->data.evt_system_external_signal.extsignals == 2 ){
       BSP_LedToggle(1);
    }
 break;
```

4. Analyzing with µC/Probe

4.1 What is µC/Probe

Micrium's µC/Probe is a Windows application that allows you to read and write the memory of any embedded target processor during run-time, and map those values to a set of virtual controls and indicators placed on a graphical dashboard. No programming is required. Simply drag and drop the graphic components into place.

4.2 How to use µC/Probe

This section describes how to watch a variable using a graphical control. The example uses the semicircle3 angular gauge on the temperature_counter variable in the **SOC-Thermometer-RTOS** example project. Note that the µC/Probe tool must be separately installed through the Simplicity Studio Package Manager.

1. Create a new **SOC-Thermometer-RTOS** application.

   In the Launcher perspective, click **New Project** or, if you are in the Simplicity IDE perspective, select **Project > New > Silicon Labs AppBuilder Project**.

   Select the Bluetooth SDK and click **Next**.

   If you have more than one SDK version installed, select version 2.6.0.0 (or higher) and click **Next**.

   Select SOC- Thermometer – RTOS and click **Next**.

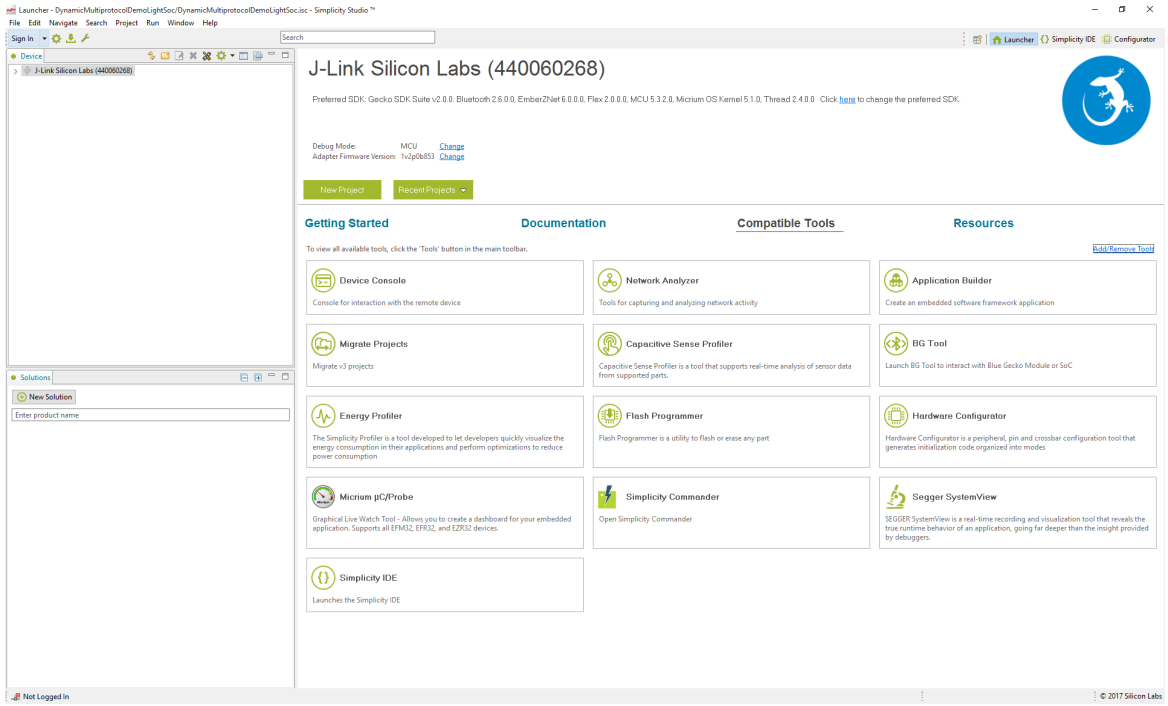
   Optionally edit the default name and click **Next**.

   Choose your board from the dropdown list. If you have both IAR-EWARM and GCC installed, deselect the one you want to use. Click **Finish**.
2. µC/Probe can watch global variable, so move the static keyword to the definition of the temperature_counter variable in main.c outside of the App_TaskThermometer function..

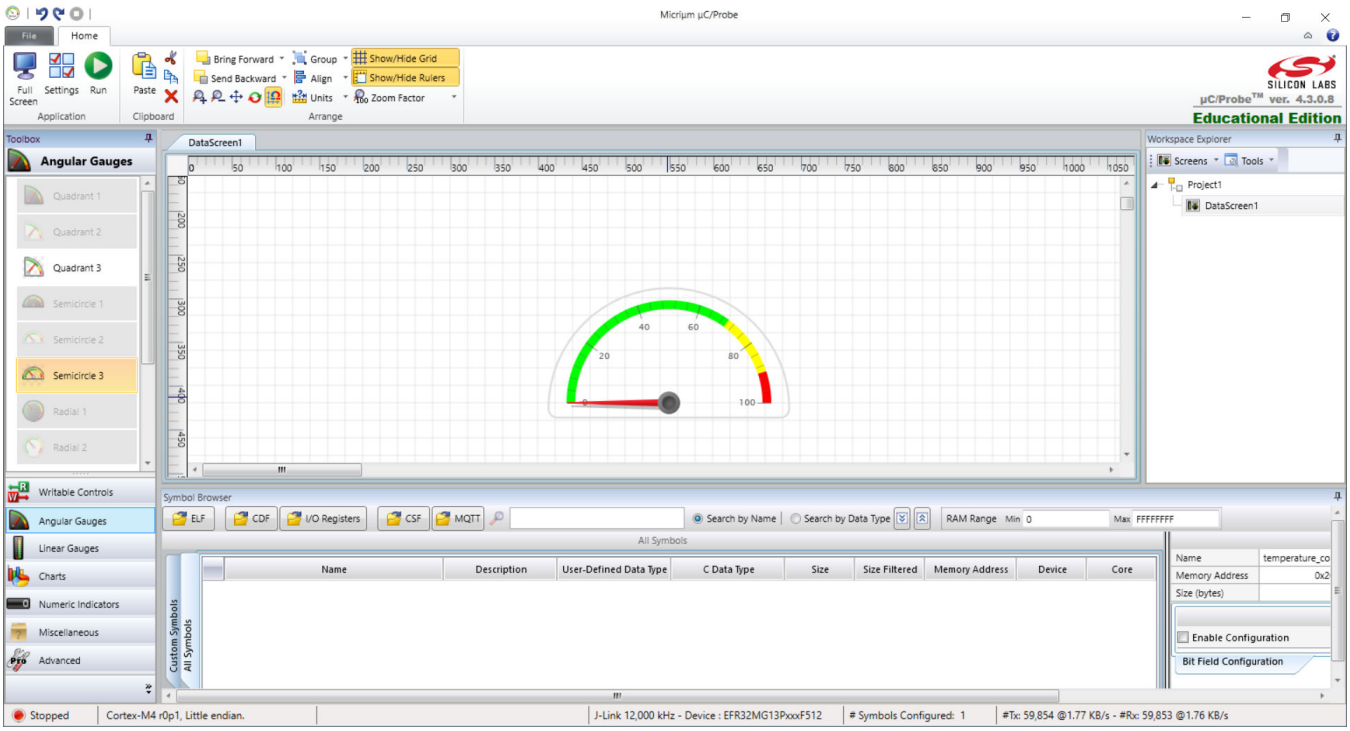
```
static int temperature_counter = 20;
static  void  App_TaskThermometer(void *p_arg)
{
  RTOS_ERR  err;
```

3. In the Simplicity IDE, select Project > Build Project. If the Build Project option isn't enabled click the project in the Project Explorer view to select it. When build is complete, In the binaries folder you will now find a file named soc-thermometer-rtos.axf or soc-thermometer-rtos.out, depending whether you used GCC or IAR-EWARM. You will use this symbol file with the µC/Probe application.
4. Flash the application onto the target board.

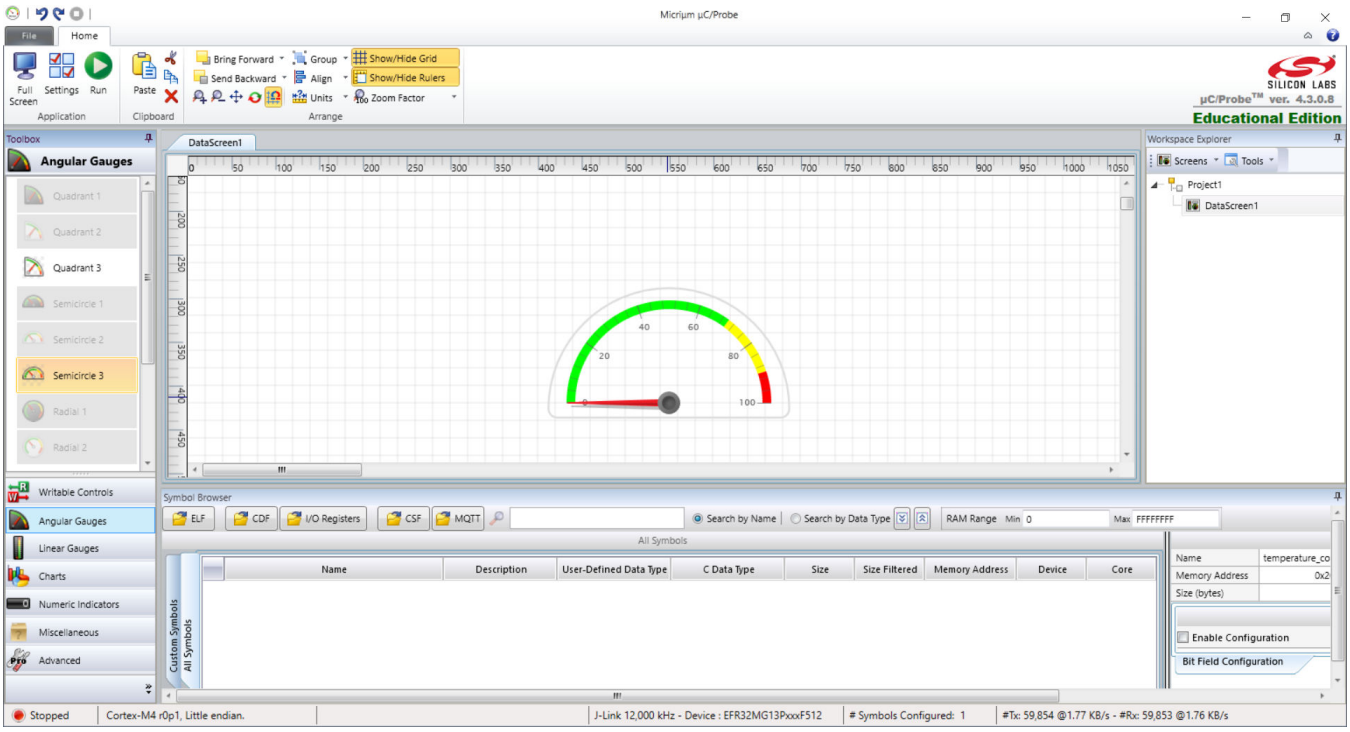
To get started with µC/Probe, launch the application from the Compatible Tools section in Simplicity Studio's Launcher perspective or Tools drop down.



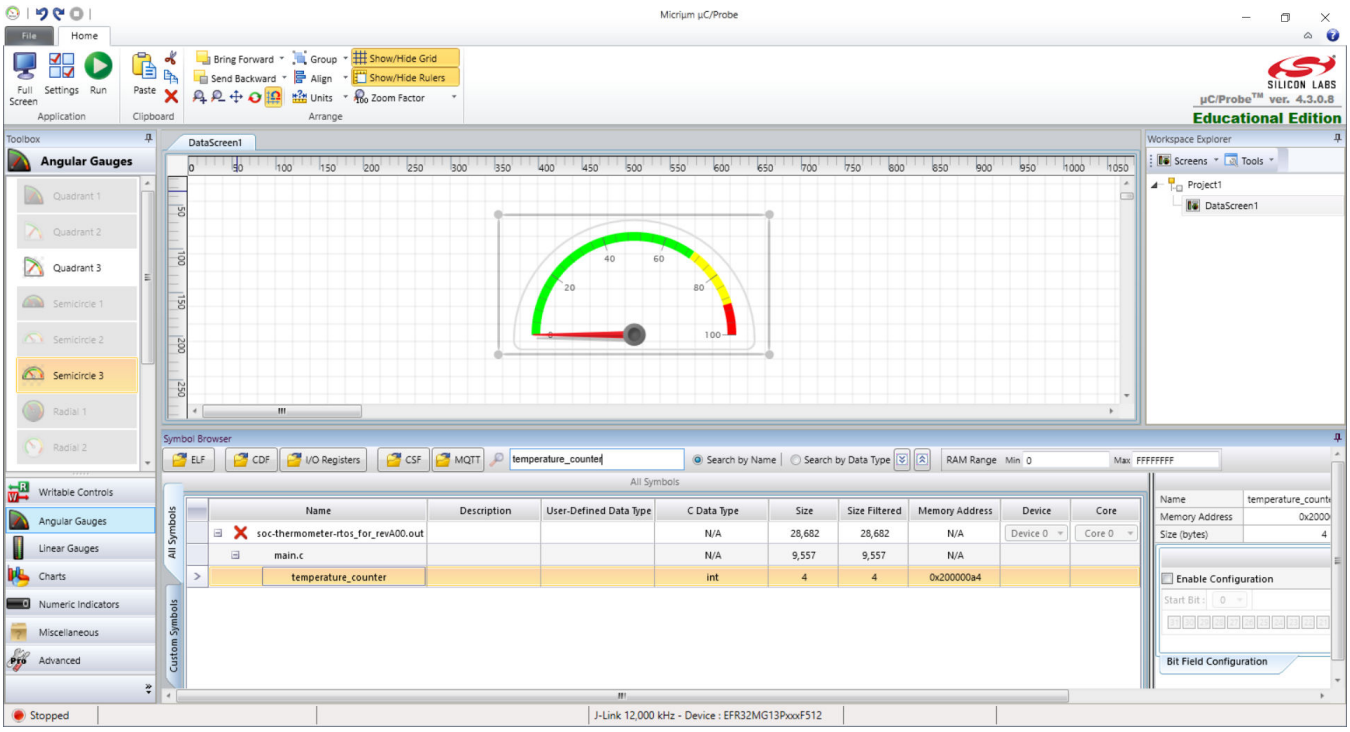
1. Select Angular Gauges and drag the Semicircle3 gauge onto the DataScreen as shown.



2. In Symbol Browser, click **ELF**. Browse to the soc-thermometer-rtos.axf or soc-thermometer-rtos.out file containing the symbol information for your project.

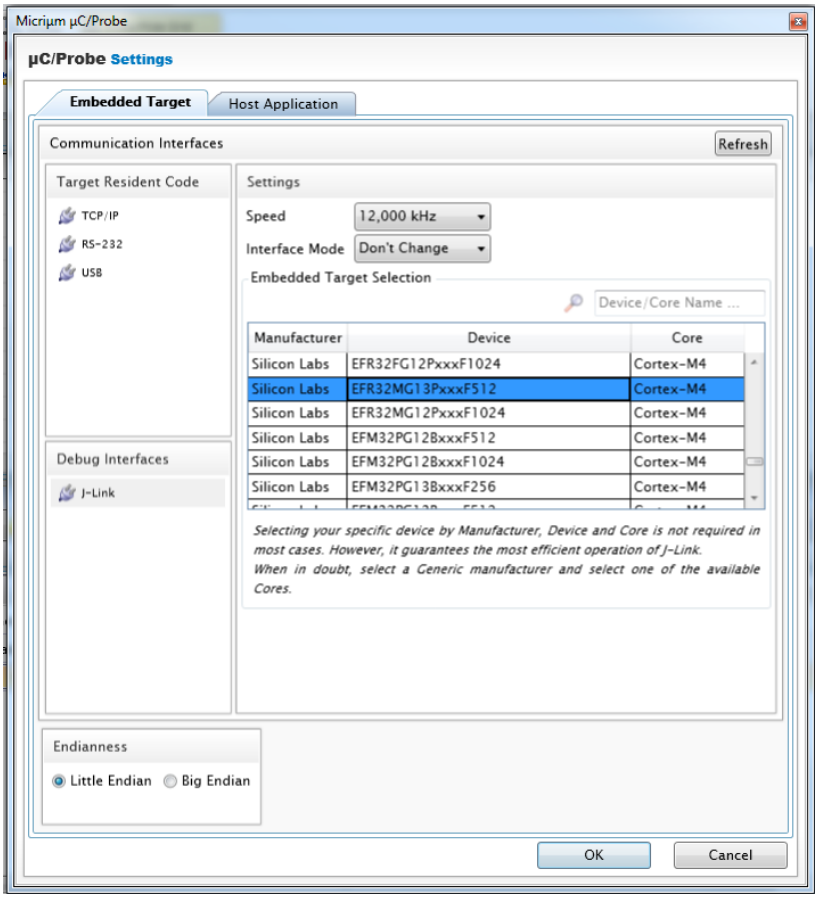


3. Once the symbol file has been loaded, search for the temperature_counter variable using the search field as shown below and drag it on top of the semicircle3 control.



4. In the top toolbar, click **Run** to start the session. The gauge gradually increments from 20 to 40 and then resets.

   Note: The first time you run, µC/Probe may show you a dialog asking you to select your microcontroller variant. Open the settings dialog, select your device as shown, and click **OK**.



4.3 Limitations

- The µC/Probe application delivered is the educational edition. The full list of limitations of this edition can be found here:

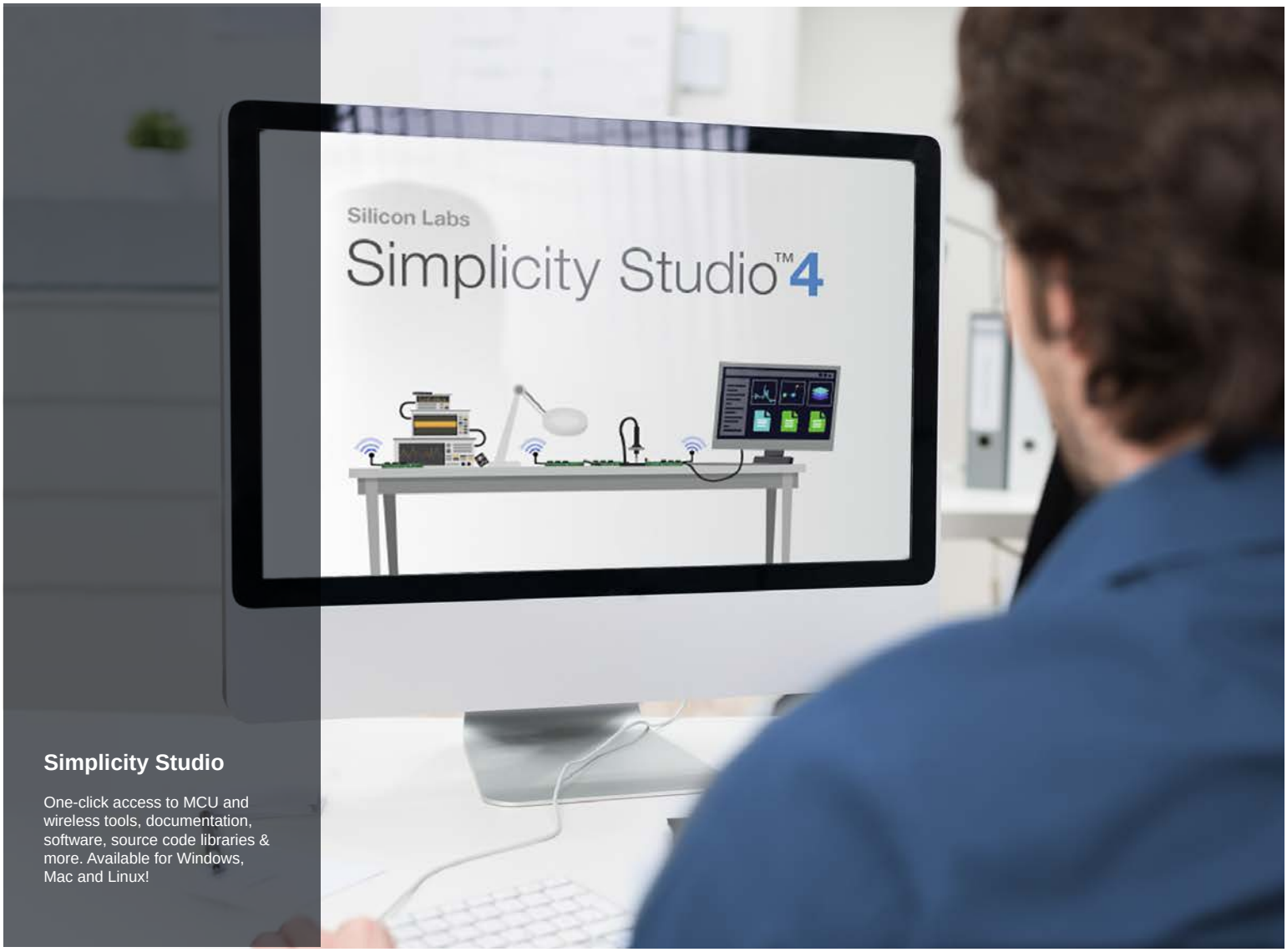
https://www.micrium.com/ucprobe/about/

- µC/Probe prevents the EFR32 from entering EM2 sleep mode. Once the µC/Probe session has finished, it is necessary to cycle power on the WSTK to return to EM2.

5. Additional Reading

See the following resources for additional information.

- Bluetooth Smart API reference
- *UG136: Bluetooth C Application Developers Guide*
- Micrium OS User Manual V5.00.00
- Gecko HAL & Driver API Reference Guide



Simplicity Studio

One-click access to MCU and wireless tools, documentation, software, source code libraries & more. Available for Windows, Mac and Linux!

IoT Portfolio
www.silabs.com/IoT

SW/HW
www.silabs.com/simplicity

Quality
www.silabs.com/quality

Support and Community
community.silabs.com

SILICON LABS

Silicon Laboratories Inc.
400 West Cesar Chavez
Austin, TX 78701
USA

http://www.silabs.com