Timers

# Introduction

This application note provides a brief about the Timer and Callout functionalities available in the SDK, the benefit of Callouts as compared to Timers, and their corresponding APIs.

Example codes are accompanied with this application note, for showcasing various Timer and Callout-related APIs available in the Talaria TWO SDK, explaining how to create, start, handle, and stop timers and callouts.

# Timers in Talaria TWO

Talaria TWO device has four 64-bit hardware timer counters.

|  |
| --- |
| enum timer\_base {  TIMER\_BASE\_US, /\*\*< System clock (1 MHz) \*/  TIMER\_BASE\_BT, /\*\*< Bluetooth clock (1.6384 MHz) \*/  TIMER\_BASE\_APP, /\*\*< Application use \*/  TIMER\_BASE\_HFC /\*\*< Hardware clock (80 MHz) \*/  }; |

Time counter TIMER\_BASE\_US is dedicated as a system timer and is configured to count at a frequency of 1MHz. The counter TIMER\_BASE\_BT is set up to count at a frequency suitable for the Bluetooth protocol stack (using 1.6384 MHz).

The third counter TIMER\_BASE\_APP is available for application use with a configurable counter frequency. The fourth counter TIMER\_BASE\_HFC is fixed at the core frequency; 80MHz.

In addition to the four counters, there are 16 timer match units. Each of these units can be set up to generate an interrupt when one specific counter reaches a certain value. The match-unit values are only 32-bit wide, so it is only the lower 32-bits of each counter that are considered when generating the match.

The kernel provides API functions for allocating any of these match units for application use. The caller can choose which available counter and time unit to use for the allocated timer.

The calling code can specify a pointer to a callback function, to be invoked when the match-unit signals an interrupt. The callback will be invoked in the hardware interrupt context. No blocking calls may be performed in this callback.

Then this timer can be started, stopped, can be made capable to wake up the device from suspend, the lower 32 bits or full 64 bits value of the hardware linked to the timer can be read, can be set, reset, or released through various Timer APIs available. Few APIs are demonstrated in the example code.

But instead of using this low-level Timer interface, it is recommended to use the callout interface which is explained in the next section.

# Callouts in Talaria TWO

Callouts are objects representing a callback function to be invoked at some point in the future when a specified timeout expires.

Callouts are a better-managed way to use OS Timers as they internally work in a less power-consuming manner and have features, such as they can be differed to be invoked in a way that we can minimize the number of wakeups from suspend state of the system. The expiration time of a particular callout at init (or expiration time of all the pending callouts at once), can be moved to the closest non-soft timer, later than requested, and this could lead to savings in power consumption.

Invocation to the callback function can be scheduled either after a specified number of microseconds elapsed or at an absolute time.

The initialization function used for callout decides it is a soft or hard deadline callout.

The callbacks will be invoked in an interrupt context. All the Callouts interrupt the device from the suspended system state.

Various operations are possible for managing the callouts, for example, querying the status of the callout if it’s scheduled or not, stopping the callout timer and removing it from the list of pending callouts, and so on.

Few of these callout APIs are demonstrated in the example code.

# Timer and Callout APIs

## Timer APIs

### os\_timer\_allocate()

Used to allocate one of the sixteen hardware match-units.

|  |
| --- |
| os\_timer\_id\_t  os\_timer\_allocate(enum timer\_base base, unsigned idx, timer\_callback\_t callback, void \*userdata) |

Following parameters are passed as arguments to this API:

1. The hardware counter to use from the four available counters in enum timer\_base as listed earlier in the document
2. The timer ID to allocate (or TIMER\_ANY for any free timer available)
3. The Function to be called when a timer match occurs
4. The userdata passed to this callback

API returns an ID representing the timer, or TIMER\_INVALID if one of the arguments is incorrect or if there is no free timer.

### os\_timer\_set()

Sets the timer match value in absolute time ticks. The timer will match when the counter reaches the value specified.

|  |
| --- |
| void os\_timer\_set(os\_timer\_id\_t timer, uint32\_t ticks) |

### os\_timer\_reset()

Clears the timer match and prevents the timer from generating an interrupt.

|  |
| --- |
| void os\_timer\_reset(os\_timer\_id\_t timer) |

### os\_timer\_read ()

Reads the lower 32-bit value of the hardware counter linked to the timer.

|  |
| --- |
| uint32\_t os\_timer\_read(os\_timer\_id\_t timer) |

### os\_timer\_ read64()

Reads the full 64-bit value of the hardware counter linked to the timer.

|  |
| --- |
| uint32\_t os\_timer\_read64(os\_timer\_id\_t timer) |

### os\_timer\_release()

Frees a previously allocated match unit.

|  |
| --- |
| void os\_timer\_release(os\_timer\_id\_t timer) |

### os\_timer\_wakeup\_enable()

Makes a specified timer capable of waking up the device from suspension when expired.

|  |
| --- |
| void os\_timer\_wakeup\_enable(os\_timer\_id\_t timer) |

### os\_timer\_wakeup\_disable()

Restores the specified timer’s capability to the default state that it will not cause the device to wake-up from suspension.

|  |
| --- |
| void os\_timer\_wakeup\_disable(os\_timer\_id\_t timer) |

## Callout APIs

### callout\_init()

Takes a pointer to the callout object and a pointer to the callback function to be invoked on this callout as input parameters and initializes the callout object.

|  |
| --- |
| static inline void callout\_init(struct callout \*co, void (\*fn)(struct callout \*)) |

### callout\_schedule()

Schedules callback function to be invoked after the specified number of microseconds.

|  |
| --- |
| int callout\_schedule(struct callout \*co, uint32\_t time) |

Returns True if the callout was previously pending, false otherwise.

### callout\_schedule\_at()

Schedules callback function to be invoked at a specified absolute time.

|  |
| --- |
| int callout\_schedule\_at(struct callout \*co, uint32\_t time) |

Returns True if the callout was previously pending, false otherwise. Passed parameter time is matched against the lower 32-bits of the system time.

### callout\_pending()

Checks the state of a callout object if this callout is scheduled or not.

|  |
| --- |
| static inline int callout\_pending(const struct callout \*co) |

Returns True if the specified callout object is scheduled.

### callout\_init\_soft()

Initializes callout object with a soft deadline.

|  |
| --- |
| static inline void callout\_init\_soft(struct callout \*co, void(\*fn)(struct callout \*)) |

A soft callout timeout accepts the timeout to be invoked later than requested if this could lead to savings in power consumption.

A soft callout expiration will be aligned to the closest non-soft one when entering suspend.

### callout\_stop()

Used to stop the callout timer and remove it from the list of pending callouts.

|  |
| --- |
| int callout\_stop(struct callout \*co) |

Returns True if the callout object was stopped while pending, false otherwise.

# Code Walkthrough

**Note**: All the applicable ELFs are available in the following location of the SDK release package: sdk\_x.y\examples\ innoos\_timers\_callouts\bin).

x and y in sdk\_x.y refer to the SDK release version. For example: *sdk\_2.4\examples\* *innoos\_timers\_callouts\bin*.

## Timer Example

### Overview

The sample code in the path: examples/innoos\_timers\_callouts/timer/main.c showcases:

1. Functionality of allocating and filling timer user data
2. Setting the timer callback
3. Setting user data
4. Starting the timer for the required timeout
5. Creating a timer for demonstrating of canceling the timer
6. Clearing the timer or setting it again from the timer callback

In this sample, one-timer is created for 2 seconds, and a callback is received after the timer elapses. A second timer is created and stopped before it elapses.

### Sample Code Walkthrough

Header file for the timer functions is kernel/timer.h

|  |
| --- |
| #include <kernel/timer.h> |

#### Defining Timer Context Data

Upon timer elapse, to understand the context of the timer, the user can set a context or user data. As a first step, a data structure for the user data must be defined (optional).

|  |
| --- |
| /\*timer context data structure\*/  typedefstruct timer\_user\_data\_t  {  unsignedint timer\_created\_at;  unsignedint timeout;  os\_timer\_id\_t timer\_id;  }timer\_user\_data; |

#### Defining the Timer Callback function

After the timer’s timeout elapses, the timer callback function will be invoked. Timer and user context data must be released in this callback.

|  |
| --- |
| static void on\_timer\_event\_callback(void \*user\_data);  /\*creating a timer\*/  /\*allocating and filling timer user data\*/  timer\_user\_data \*ptimer\_user\_data;  ptimer\_user\_data = os\_alloc(sizeof(timer\_user\_data));  ptimer\_user\_data->timer\_created\_at = os\_systime();  ptimer\_user\_data->timeout = os\_systime()+(TIMER\_SAMPLE\_TIMEOUT\_2\_SEC\_IN\_MICRO); |

The Timer can be repeated from the callback. To do that, instead of calling os\_timer\_release() to free up the user data, use os\_timer\_set() with a new timeout.

#### Creating and Starting Timer

Here, the user data is allocated and filled. Then the timer is created using os\_timer\_allocate(). This will associate the timer callback and user data for the timer. Using os\_timer\_set(), timer is set for the required timeout value. The timeout value is in microseconds.

|  |
| --- |
| /\*allocating and filling timer user data\*/  timer\_user\_data \*ptimer\_user\_data;  ptimer\_user\_data = os\_alloc(sizeof(timer\_user\_data));  ptimer\_user\_data->timer\_created\_at = os\_systime();  ptimer\_user\_data->timeout = os\_systime()+(TIMER\_SAMPLE\_TIMEOUT\_2\_SEC\_IN\_MICRO);    /\*setting the timer callback and user data\*/  ptimer\_user\_data->timer\_id = os\_timer\_allocate(TIMER\_BASE\_US,  TIMER\_ANY, on\_timer\_event\_callback, ptimer\_user\_data);  /\*starting the timer for required timeout. timeout time is in microseconds\*/  os\_timer\_set(ptimer\_user\_data->timer\_id, ptimer\_user\_data->timeout); |

#### Creating another timer, checking the remaining time, and canceling the timer

A timer is created with 5 seconds as the timeout and we intend to cancel it before the timeout. Timer callback and timer user data are not set since even if timer call back is set, it will not be called in case the timer has been stopped before the timeout.

A timer’s time to timeout can be checked using os\_timer\_read(). Timer can be reset by os\_timer\_reset(). Release the timer when the timer is no longer required using os\_timer\_release(). After calling os\_timer\_release(), timer is no longer valid.

os\_timer\_read64() can be used to get 64-bit resolution timer reading.

|  |
| --- |
| /\*create a timer for demonstrating cancelling a timer\*/  os\_timer\_id\_t timer\_id;  timer\_id = os\_timer\_allocate(TIMER\_BASE\_US,  TIMER\_ANY, NULL, NULL);  os\_timer\_set(timer\_id, os\_systime()+(TIMER\_SAMPLE\_TIMEOUT\_5\_SEC\_IN\_MICRO));    /\* at timeout, if the device in suspend mode,  \* below function will wakeup the device\*/  os\_timer\_wakeup\_enable(timer\_id);  int nTimerReader = 3;  while (nTimerReader--)  {  os\_printf ("\nTimer read current ticks:%u", os\_timer\_read(timer\_id));  os\_msleep(1000);  }  /\*clearing the timer.\*/  os\_timer\_reset(timer\_id);  os\_timer\_release(timer\_id);    os\_printf ("\nTimer canceled. Timer call back will not be called"); |

### Running the Application

Program timer.elf using the Download tool:

1. Launch the Download tool provided with InnoPhase Talaria TWO SDK.
2. In the GUI window:
   1. Boot Target: Select the appropriate EVK from the drop-down.
   2. ELF Input: Load the timer.elf by clicking on Select ELF File.
   3. Programming: Prog RAM or Prog Flash as per requirement.

For more details on using the Download tool, refer to the document: UG\_Download\_Tool.pdf (path: *sdk\_x.y/pc\_tools/Download\_Tool/doc*).

**Note**: x and y refer to the SDK release version. For example: sdk\_2.4/doc.

### Expected Output

timer.elf is created when compiling the code mentioned in section 6.1.2 and gives the following console output when programmed to Talaria TWO.

|  |
| --- |
| UART:NWWWWWAE4 DWT comparators, range 0x8000  Build $Id: git-7e2fd6a94 $  app=gordon  flash: Gordon ready!  Y-BOOT 208ef13 2019-07-22 12:26:54 -0500 790da1-b-7  ROM yoda-h0-rom-16-0-gd5a8e586  FLASH:PNWWWAEBuild $Id: git-65f6c1f46 $  $App:git-46e2bea7  SDK Ver: sdk\_2.4  Innos Timer Demo App  b2ee0:Timer started. current time:[84929]  Timer read current ticks:34651392  Timer read current ticks:34651392  be768:First Timer: Timer Event occured. Current time[2084930] timer created at[84927]  Timer read current ticks:34651392  Timer canceled. Timer call back will not be called |

## Callout Basics Example 1

### Overview

The sample code in the path /examples/innoos\_timers\_callouts/src/ callout\_basics\_1.c showcases:

1. Creating and forming the application context for the callout
2. Initializing the callout with a callback function
3. Scheduling the callout after the given time
4. Checking the callout status
5. Confirming that the callout has occurred using waitq
6. Scheduling a callout at a specific absolute time.

In this example, callouts are initiated and scheduled one after another. The first one is scheduled relatively while the second one is scheduled in an absolute way.

### Sample Code Walkthrough

Header file for the callout functions is kernel/callout.h

|  |
| --- |
| #include <kernel/callout.h> |

#### Initializing an os\_waitq and a static flag

To programmatically confirm that the callout call-back has occurred, a waitq is used and the program flow blocks on it after resetting a flag gf\_callout\_occurred.

Later from the callback, this flag is set and waitq API os\_wakeup\_all(&waitq) is called for waitq to unblock the flow.

|  |
| --- |
| static struct os\_waitq waitq = OS\_WAITQ\_INITIALIZER(waitq);  static int gf\_callout\_occurred = 0; |

#### Defining the Callout Context Data

The callback is invoked with a pointer to the callout objects for the function. This pointer can be used by the callback function to obtain its context. This is achieved by embedding the struct callout object within another struct, that represents the context or state on which the callback should operate. Data structure for this is defined.

|  |
| --- |
| typedef struct co\_msg\_info\_t  {  struct callout timeout;  /\*application specific info\*/  char callout\_reason\_str[64];  int index;  unsigned int created\_at;  }callout\_msg\_info; |

#### Defining the Callout Call-back Function

After the callout timeout elapses, the call-back function will be invoked. The callout pointer passed to this callback is stored in member timeout. This pointer is used by the callback function to obtain its context.

This is achieved by using the macro container\_of(ptr, type,member) which takes three arguments:

1. A pointer
2. Type of the container
3. Name of the member the pointer refers to.

The macro will then expand to a new address pointing to the container which accommodates the respective member.

|  |
| --- |
| static void on\_callout\_callback\_fn(struct callout \*co)  {  struct co\_msg\_info\_t \*co\_msg;  co\_msg = container\_of(co, struct co\_msg\_info\_t, timeout);  /\*apply the business logic\*/  os\_printf("\n%x:%u:Timeout occured [%s] index[%d] created at[%u]",  CALLOUT\_SAMPLE\_CURRENT\_THREAD\_ID,  os\_systime(),co\_msg->callout\_reason\_str,  co\_msg->index,  co\_msg->created\_at);  gf\_callout\_occurred = 1;  os\_wakeup\_all(&waitq);  } |

This way, we can access all the members of the struct created and get the context or state on which the callback should operate. This structure was originally initialized in /examples/innoos\_timers\_callouts/timer/main.c as shown in section 6.2.2.

#### Creating, Initializing and Scheduling the Callout with Relative Time

Here, the co\_msg\_info\_t struct is allocated and filled. A callout reason string, an index, and the time it got created are filled in as the context. Then the callout is initialized with the function callout\_init()and a callback is associated through this API.

Before a callout object is used, it must be initialized via a call to either callout\_init() or callout\_init\_soft(). The flag gf\_callout\_occurred is reset and callout is scheduled with callout\_schedule(). Its status is checked using callout\_pending() before and after the sleep gave equivalent to the timeout scheduled.

struct co\_msg\_info\_t \*co\_msg creates and formats the application context for the callout.

|  |
| --- |
| struct co\_msg\_info\_t \*co\_msg = NULL;  co\_msg = os\_alloc(sizeof(struct co\_msg\_info\_t));  snprintf(co\_msg->callout\_reason\_str,sizeof(co\_msg->callout\_reason\_str),  "Callout sample data. Created at[%u]", os\_systime());  co\_msg->index = 1;  co\_msg->created\_at = os\_systime(); |

callout\_init()initiates the callout with a callback function.

|  |
| --- |
| callout\_init(&co\_msg->timeout, on\_callout\_callback\_fn); |

callout\_schedule()schedules the callout for the given time.

|  |
| --- |
| gf\_callout\_occurred = 0;  callout\_schedule(&co\_msg->timeout, CALLOUT\_SAMPLE\_2SEC\_IN\_MICRO\_SECONDS);  os\_printf("\n%x:%u:callout created for [%d]",  CALLOUT\_SAMPLE\_CURRENT\_THREAD\_ID,os\_systime(),  CALLOUT\_SAMPLE\_2SEC\_IN\_MICRO\_SECONDS); |

callout\_pending()checks the status of the callout.

|  |
| --- |
| os\_printf("\n%x:%u:callout pending status [%s]",CALLOUT\_SAMPLE\_CURRENT\_THREAD\_ID,  os\_systime(),  callout\_pending(&co\_msg->timeout)?"Pending":"Completed"); |

os\_wait\_event()function ensures the callout has occurred.

|  |
| --- |
| os\_wait\_event(&waitq, gf\_callout\_occurred); |

os\_free()releases the context data.

|  |
| --- |
| os\_free(co\_msg);  co\_msg = NULL;  ...  ...  ...  } |

By using waitq, the occurrence of the callback is doubly confirmed, and co\_msg is freed.

#### Creating, Initializing and Scheduling the Callout with Absolute Time

Sequence similar to the one mentioned in section 6.2.2.4 is followed except callout\_schedule\_at()is used instead of the previously used callout\_schedule().

### Running the Application

Program callout\_basics\_1.elf using the Download tool:

1. Launch the Download tool provided with InnoPhase Talaria TWO SDK.
2. In the GUI window:
   1. Boot Target: Select the appropriate EVK from the drop-down.
   2. ELF Input: Load the callout\_basics\_1.elf by clicking on Select ELF File.
   3. Programming: Prog RAM or Prog Flash as per requirement.

For more details on using the Download tool, refer to the document: UG\_Download\_Tool.pdf (path: *sdk\_x.y/pc\_tools/Download\_Tool/doc*).

**Note**: x and y refer to the SDK release version. For example: sdk\_2.4/doc.

### Expected Output

callout\_basics\_1.elf is created when compiling which provides the following console output when programmed to Talaria TWO.

|  |
| --- |
| UART:NWWWWWAE4 DWT comparators, range 0x8000  Build $Id: git-7e2fd6a94 $  app=gordon  flash: Gordon ready!  Y-BOOT 208ef13 2019-07-22 12:26:54 -0500 790da1-b-7  ROM yoda-h0-rom-16-0-gd5a8e586  FLASH:PNWWWAEBuild $Id: git-65f6c1f46 $  $App:git-46e2bea7  SDK Ver: sdk\_2.4  Callout Basic App 1  b2ee0:84999:callout created for [2000000]  b2ee0:85045:callout pending status [Pending]  be768:2085000:Timeout occured [Callout sample data. Created at[84865]] index[1] created at[84995]  b2ee0:2085100:callout pending status [Completed]  b2ee0:2085173:callout schedule at [6085173]  be768:6085175:Timeout occured [This is a scheduled timer created at [2085154]] index[999] created at[2085170]  Callout example completed |

## Callout Basics Example 2

### Overview

The sample code in the path /examples/innoos\_timers\_callouts/src/ callout\_basics\_2.c showcases callout functionality of repeat and cancel.

### Sample Code Walkthrough

In this example, a callout is programmed to be scheduled at 1 second and the callback rescheduled for every 1 second. A global flag is toggled in every callout call-back.

|  |
| --- |
| static void on\_callout\_callback\_fn(struct callout \*co)  {  callout\_sample\_msg\_info \*co\_msg;  co\_msg = container\_of(co, callout\_sample\_msg\_info, timeout);  os\_printf("\n%x:%u:callout event occured",  CALLOUT\_SAMPLE\_CURRENT\_THREAD\_ID,os\_systime());  /\*toggle the value\*/  gbToggleValue = !gbToggleValue;  /\*repeating or resetting the callout\*/  callout\_schedule(&co\_msg->timeout, CALLOUT\_SAMPLE\_1SEC\_IN\_MICRO\_SECONDS);  } |

The main thread sleeps for 10 seconds after which the callout is stopped. While main sleep, callouts still occur as they are called from the other context. One thread is created which wakes up every 100 milliseconds, checks, and prints the toggle flag.

Hence, ten such prints can be passed before a toggle event occurs through the callout which is periodic at 1 second.

This continues until another flag called terminate is toggled from main, in which case this thread stops printing and returns. This termination happens after 20 seconds.

|  |
| --- |
| static void\* gh\_application\_logic\_thread(void \*p)  {  while (!gfTerminate)  {  os\_printf("\n%x:%u:Current state of toggle value[%s]", CALLOUT\_SAMPLE\_CURRENT\_THREAD\_ID,os\_systime(),  gbToggleValue==true?"ON":"OFF");  os\_msleep(100);  }  os\_printf("\n%x:%u:Current state of toggle value[%s]",  CALLOUT\_SAMPLE\_CURRENT\_THREAD\_ID,os\_systime(),  gbToggleValue==true?"ON":"OFF");  return NULL;  } |

Overall output in console is ten consecutive toggles ON or OFF prints per 100 milliseconds until toggle event happens per second. These toggle events happen ten times and then stop. The other thread keeps printing the fixed last toggle flag value until it terminated after another 10 seconds.

### Running the Application

Program callout\_basics\_2.elf using the Download tool:

1. Launch the Download tool provided with InnoPhase Talaria TWO SDK.
2. In the GUI window:
   1. Boot Target: Select the appropriate EVK from the drop-down.
   2. ELF Input: Load the call\_out\_basics\_2.elf by clicking on Select ELF File.
   3. Programming: Prog RAM or Prog Flash as per requirement.

For more details on using the Download tool, refer to the document: UG\_Download\_Tool.pdf (path: *sdk\_x.y/pc\_tools/Download\_Tool/doc*).

**Note**: x and y refer to the SDK release version. For example: sdk\_2.4/doc.

### Expected Output

callout\_basics\_2.elf is created when compiling which gives a console output as follows when programmed to Talaria TWO.

|  |
| --- |
| WWWWAE4 DWT comparators, range 0x8000  Build $Id: git-7e2fd6a94 $  app=gordon  flash: Gordon ready!  Y-BOOT 208ef13 2019-07-22 12:26:54 -0500 790da1-b-7  ROM yoda-h0-rom-16-0-gd5a8e586  FLASH:PNWWWAEBuild $Id: git-65f6c1f46 $  $App:git-46e2bea7  SDK Ver: sdk\_2.4  Callout Basic App 2  b2de0:85021:callout created for [1000000]  bf780:85074:Current state of toggle value[OFF]  bf780:185130:Current state of toggle value[OFF]  bf780:285187:Current state of toggle value[OFF]  bf780:385244:Current state of toggle value[OFF]  bf780:485301:Current state of toggle value[OFF]  bf780:585358:Current state of toggle value[OFF]  bf780:685415:Current state of toggle value[OFF]  bf780:785472:Current state of toggle value[OFF]  bf780:885529:Current state of toggle value[OFF]  bf780:985586:Current state of toggle value[OFF]  bfe68:1085022:callout event occured  bf780:1085643:Current state of toggle value[ON]  bf780:1185700:Current state of toggle value[ON]  bf780:1285757:Current state of toggle value[ON]  bf780:1385814:Current state of toggle value[ON]  bf780:1485871:Current state of toggle value[ON]  bf780:1585928:Current state of toggle value[ON]  bf780:1685985:Current state of toggle value[ON]  bf780:1786042:Current state of toggle value[ON]  bf780:1886099:Current state of toggle value[ON]  bf780:1986156:Current state of toggle value[ON]  bfe68:2085064:callout event occured  .  .  .  bf780:16894649:Current state of toggle value[ON]  bf780:16994706:Current state of toggle value[ON]  bf780:17094763:Current state of toggle value[ON]  bf780:17194820:Current state of toggle value[ON]  bf780:17294877:Current state of toggle value[ON]  bf780:17394934:Current state of toggle value[ON]  bf780:17494991:Current state of toggle value[ON]  bf780:17595048:Current state of toggle value[ON]  bf780:17695105:Current state of toggle value[ON]  bf780:17795162:Current state of toggle value[ON]  bf780:17895219:Current state of toggle value[ON]  bf780:17995276:Current state of toggle value[ON]  bf780:18095333:Current state of toggle value[ON]  bf780:18195390:Current state of toggle value[ON]  bf780:18295447:Current state of toggle value[ON]  bf780:18395504:Current state of toggle value[ON]  bf780:18495561:Current state of toggle value[ON]  bf780:18595618:Current state of toggle value[ON]  bf780:18695675:Current state of toggle value[ON]  bf780:18795732:Current state of toggle value[ON]  bf780:18895789:Current state of toggle value[ON]  bf780:18995846:Current state of toggle value[ON]  bf780:19095903:Current state of toggle value[ON]  bf780:19195960:Current state of toggle value[ON]  bf780:19296017:Current state of toggle value[ON]  bf780:19396074:Current state of toggle value[ON]  bf780:19496131:Current state of toggle value[ON]  bf780:19596188:Current state of toggle value[ON]  bf780:19696245:Current state of toggle value[ON]  bf780:19796302:Current state of toggle value[ON]  bf780:19896359:Current state of toggle value[ON]  bf780:19996416:Current state of toggle value[ON]  Callout example completed  bf780:20096472:Current state of toggle value[ON] |