ThingWorx C SDK Developer's Guide

Version 1.3.1

January 2016

PTC Inc., 140 Kendrick Street, Needham, MA 02494 USA

Contents

About this Guide

ThingWorx has introduced Software Development Kits (SDKs) for Edge devices, machines, and systems in several languages. These SDKs allow companies to incorporate connectivity functionality into their products, and to easily connect those products to a ThingWorx Platform. These SDKs can either be implemented as a gateway to several connected products, or be embedded directly within a product on a one-to-one basis.

All ThingWorx Edge SDKs share a common reference implementation and provide a secure communication channel to a ThingWorx Platform, allowing a machine/device to be a full participant in a ThingWorx IoT solution.

This document describes how to use the ThingWorx Edge C SDK. The complete API reference is available in the C SDK bundle.

Note

This document is accurate as of this release and is subject to change. For the latest documentation, see the Help Center available at PTC ThingWorx eSupport, https://support.ptc.com/appserver/cs/portal/.

Pre-requisites

This document assumes that you have a solid background in the C/C++ programming language. Further, it assumes that you have had at least basic training in ThingWorx. For example, you know how to use the ThingWorx Composer and understand the main concepts of things, data shapes, properties, events, and services.

To develop an application using the C SDK, you need to have a C/C++ development environment. No specific compiler version is required, but the compiler must be C99 (the C language spec) compatible.

To get started, it is recommended that you review the sample projects provided in the SDK. To use these examples, you need Microsoft Visual Studio

Technical Support

Contact PTC Technical Support via the PTC Web site, phone, fax, or e-mail if you encounter problems using your product or the product documentation.

For complete details, refer to Contacting Technical Support in the *PTC Customer Service Guide*. This guide can be found under the Related Resources section of the PTC Web site at:

http://www.ptc.com/support/

The PTC Web site also provides a search facility for technical documentation of particular interest. To access this search facility, use the URL above and search the knowledge base.

You must have a Service Contract Number (SCN) before you can receive technical support. If you do not have an SCN, contact PTC Maintenance Department using the instructions found in your *PTC Customer Service Guide* under Contacting Your Maintenance Support Representative.

Documentation for PTC ThingWorx Products

You can access PTC ThingWorx documentation, using the following resources:

- PTC ThingWorx Help Centers — The PTC ThingWorx Platform Help Center provides documentation for the PTC ThingWorx Platform, including ThingWorx Composer. The PTC ThingWorx Edge Help Center provides documentation for the ThingWorx Edge WebSocket-based MicroServer (WS EMS) and for the Edge SDKs. You can browse the entire documentation set in a Help Center, or use the search capability to perform a keyword search. To access the PTC ThingWorx Help Centers, visit ThingWorx Help Center.
- PTC ThingWorx Reference Documentation — The Reference Documents page on the PTC Support site provides access to the PDF documents available for the PTC ThingWorx Edge SDKs and the PTC ThingWorx Platform at http://support.ptc.com/appserver/cs/doc/refdoc.jsp, These PDF documents include documents that explain system requirements. For ThingWorx Edge products, this document is entitled *ThingWorx Edge Requirements and Compatibility Matrix*.

  A Service Contract Number (SCN) is required to access the PTC documentation from the Reference Documents website. If you do not know your SCN, see “Preparing to contact TS” on the Processes tab of the PTC

Customer Support Guide for information about how to locate it: http://support.ptc.com/appserver/support/csguide/csguide.jsp. When you enter a keyword in the Search Our Knowledge field on the PTC eSupport portal, your search results include both knowledge base articles and PDF guides.

Comments

PTC welcomes your suggestions and comments on our documentation. To submit your feedback, you can:

- Send an e-mail to documentation@ptc.com. To help us more quickly address your concern, include the name of the PTC product and its release number with your comments. If your comments are about a specific help topic or book, include the title.
- Click the feedback icon in any PTC ThingWorx Help Center toolbar and complete the feedback form. The title of the help topic you were viewing when you clicked the icon is automatically included with your feedback.

1

ThingWorx Edge C SDK

Introducing the ThingWorx Edge C SDK

This section provides an introduction to the ThingWorx Edge C SDK, explains its purpose, requirements for using it, and main features. It then explains how to install the SDK and provides a table that shows the directories and files in the installation. Finally, this section provides a Getting Started section, which contains an overview of the process for creating an application using the C SDK. This process references later sections of this document where you can find more details.

About the C SDK

The ThingWorx Edge C SDK is a lightweight, but fully functional implementation of the ThingWorx AlwaysOn™ protocol. It is designed to minimize memoroy footprint while making it easy to integrate applications into the ThingWorx distributed computing environment of the Internet of Things (IoT). The goal of the C SDK is to make creating applications that use it simple, but to also give developers enough flexibility to create very sophisticated applications. For example, the SDK contains a simple "tasker" framework that you can use to call functions repeatedly at a set interval. You can use the tasker framework to drive not only the connectivity layer of your application, but also the functionality of your application. However, it is not required to use the tasker at all. The API is thread safe and can be used in a complex, multi-threaded environment as well. Other examples of this flexibility are highlighted in this document.

Purpose

The three primary functions of the C SDK are as follows:

- Establish and manage a secure AlwaysOn connection with the ThingWorx Platform. This includes SSL negotiation, duty-cycle modulation, and connection maintenance such as re-establishing a connection after network connectivity is lost and restored.
- Enable easy programmatic interaction with the properties, services, and events that are exposed by entities running on the Platform.
- Implement a callback infrastructure that makes it easy to expose a set of properties and services to the Platform. These properties and services can be surfaced from multiple entities. When a request is made from the Platform for a registered property or service, a callback is made to a function that you supply during the registration process.

As previously mentioned, the C SDK also includes a simple tasker that you can use to drive data collection routines and other application-level functionality.

The C SDK uses callback functions to notify your application of requests for property reads and writes as well as requests to execute a service. The callback function signatures are defined in the `twApi.h` file. Your application can register properties and services along with their metadata with the API. The metadata is used when browsing remote entities from ThingWorx Composer, making it simple to import functionality created in your application as a thing or thing template into your application model.

The properties, services, and events for Platform-side things are easily accessed through appropriate API calls: `twApi_ReadProperty/twApi_WriteProperty`, `twAPI_InvokeService`, and `twApi_FireEvent`, respectively.

Features

The C SDK supports the following functionality that allows your machine, device, or application to work with the ThingWorx Platform:

- Data Shapes — Allow you to create definitions that model types of metadata for a remote machine/device.
- File transfer — Used for remote directory/file browsing with the Platform, and to permit bidirectional file transfer between a machine/device and the Platform.
- Tunneling — allows you to establish secure, firewall transparent application tunnels for applications that use TCP, such as VNC and SSH.
- Proxy settings — used to enable connections to the server through a proxy server.
- Client/server certificate validation — allows you to enable/disable various levels of TLS certificate validation.
- Offline message storage — enabled by default, queues outgoing messages if the network is down or if the duty cycle is in the “off” state.
- Subscribed properties — allows events to subscribe to changes in property values and other aspects of properties.

Installing and Navigating the Directories of the C SDK

Installation

To install the C SDK, go to the PTC Support site (http://www.ptc.com/support/) and download the C SDK bundle to your computer, and extract the files.

Directories and Files

The installation includes the following directories and files:

This Directory	Contains	See Also
tw-c-sdk	The file, version.txt, which provides the version number for the SDK. This directory also contains the subdirectories, build, doc, examples, and src. The rest of this table describes the contents of these main directories.	
tw-c-sdk/build	Dosyfile, a supporting file for the Doxygen API documentation. Make.settings and Makefile for building an application.	
tw-c-sdk/build/platforms	Subdirectories for all supported platforms. Each subdirectory contains the Makefile appropriate to the platform (Linux platforms) or sln, vcproj, and vcproj.filters files needed to build your application for the target platform.	Building an Application on page 65
tw-c-sdk/doc	The PDF file for this document and the mainpage.md file for the API documentation.	The Help Center, available at PTC ThingWorx eSupport, https://support.ptc.com/appserver/cs/portal/.
tw-c-sdk/doc/html	All the files and the search subdirectory for the API documentation.	
tw-c-sdk/examples	Subdirectories for the various SteamSensor examples. Each SteamSensor subdirectory contains subdirectories for the source file (/src/main.c) and for the target platforms (Linux and Win32). The win32 subdirectory contains	For information about building and running these examples, refer to the document, SDK Steam Sensor Example, which is available in the Help Center, at https://support.ptc.com/appserver/cs/portal/.

This Directory	Contains	See Also
	the files needed to open the project in Visual Studio. The `linux` and `osx` subdirectories contain the `Make.CommonSettings` and `Makefile` for the SteamSensor example.	
`src/api`	The following API source (*.c) and header (*.h) files for the C SDK: • `twApi.c, twApi.h` • `twDefinitions.c, twDefinitions.h` contains the enumerated message types, message codes (status, errors), as well as type definitions (characteristic, BaseType, entityType). • `twErrors.h` contains definitions for different types of errors, including websocket, messaging, primitive/infotable, api, tasker, logger, utils, system socket, file transfer, tunneling, and managed property. It also contains the `#defines` for the `msgCodeEnum` errors. • `twProperties.c, twProperties.h` • `twServices.c, twServices.h` • `twVersion.h`	Initialize the API Singleton on page 42 Register Properties and Services on page 43 twPrimitiveStructure on page 50 Base Types on page 50 twInfoTable on page 52 Property Access Callbacks on page 54 Service Callbacks on page 56 SDK Application-Initiated Interaction on page 58 Error Codes on page 85 (`twErrors.h`) CallBack Function Return Codes on page 99 (`twDefinitions.h`)
`src/config`	Two configuration files, `twConfig.h` and `twDefaultSettings.h`.	Configuring Components for an Application on page 22

This Directory	Contains	See Also
	As its name implies, the `twDefaultSettings.h` file contains default settings for the C SDK. The `twConfig.h` file is provided should you need to override common settings provided in the Windows Solution (sln) or gcc Makefiles. Use this file only if you are not using one of these provided files to do per project configuration. Note that the settings here apply to ALL of your projects that use the SDK.	Building an Application on page 65
`src/fileTransfer`	This directory contains the source and header files for the file transfer functionality of the C SDK, `twFileManager.c`, `twFileManager.h`, `twFileTransferCallbacks.c`, and `twFileTransferCallbacks.h`. For information about this functionality, refer to	Create a File Transfer Event Handler (Optional) on page 38
`src/messaging`	The following source and header files: • `twBaseTypes.c`, `twBaseTypes.h` contain the definitions of the Base Types of the SDK. • `twInfoTable.c`, `twInfoTables.h` contain the definitions of functions related to creating an InfoTable with the SDK.	twPrimitiveStructure on page 50 Base Types on page 50 twInfoTable on page 52 Handling Offline Messaging on page 23

This Directory	Contains	See Also
	- twMessages.c, twMessages.h - twMessaging.c, twMessaging.h	
src/porting	Source and header files that contain wrappers for OS-specific functionality (twLinux.c, twLinux.h, twLinux-openssl.h, twMarvell.c, twMarvell.h, twOSPort.c, twOSPort.h, twPThreads.c, twThreads.h, twTiSimplelink.c, twTiSimplelink.h, twWin32Threads.c, twWindows.c, twWindows.h, twWindows-openssl.h)	Porting to Another Platform on page 69
src/subscribedProps	Source and header files (subscribedProps.c and subscribedProps.h) that contain the functionality to support subscribed properties.	Defining Properties on page 28
src/thirdParty	Third-party libraries for the C SDK, including ax-TLS, cJSON, joyent-http-parser, ntlm, tomcrypt, and wildcard.	For ax-TLS: TLS Support on page 71 TLS Provider Plugins on page 79
src/tls	The files needed to use TLS with the SDK.	TLS Support on page 71
src/tunneling	The source (twTunnelManager.c) and header (twTunnelManager.h) files for the Tunneling	Configuring Application Tunneling on page 23 Create a Tunnel Event Handler (Optional) on page 38

This Directory	Contains	See Also
	component.	
src/utils	The source and header files for utilities used by the SDK: • cryptoWrapper.c, cryptoWrapper.h • jsonUtils.c, jsonUtils.h • list.c, list.h (for doubly-linked list utilities • stringUtils.c, stringUtils.h • twHttpProxy.c, twHttpProxy.h • twLogger.c, twLogger.h • twNtlm.c, twNtlm.h • twTasker.c, twTasker.h	Configuring the Tasker on page 22 Connect to the Server and Initiate Any Defined Tasks on page 47 Error Codes on page 85
src/websocket	The source (twWebsocket.c) and header (twWebsocket.h) files for the Websocket Client (abstraction layer).	Interacting with the ThingWorx Platform on page 49 Error Codes on page 85

Getting Started

The best place to start is by examining the examples provided in the tw-c-sdk/examples directory, compiling and running them. Refer to the document for the examples, SDK Steam Sensor Example, which is available in the Help Center, at https://support.ptc.com/appserver/cs/portal/.

ThingWorx Configuration

The SDK requires that a RemoteThing be created in ThingWorx in order to communicate. Creating a RemoteThing is as simple as creating a Thing with a ThingTemplate of RemoteThing and optionally an Identifier. If an Identifier is supplied, the SDK must use the same identifier as well. Without an Identifier, the RemoteThing is referenced by name. The Identifier may be used if a device has access to its serial number via firmware, for instance.

If many Things are to be created with the same properties, services, and events, it is recommended that a ThingTemplate based on the RemoteThing template be created. It will be much easier to maintain the Things and will require less memory on the ThingWorx Platform. One way to do this is to create a Thing with a RemoteThing template and then browse the client application created with the SDK for its properties, services, and events. Once this work has been completed, create a template based on this Thing. Then use the template instead of recreating all the properties, services, and events on each Thing.

Application Development

This section provides an overview of the main steps for developing an application using the C SDK.

1. Configure the components that your application will use. The components may include the following:
   - Tasker (Configuring the Tasker)
   - File Transfer (Configuring File Transfer)
   - Application Tunneling (Configuring Application Tunneling on page 23)
   - Handling of Offline Messages (Handling Offline Messages on page 23)
   - Any additional settings (Additional Settings on page 24)
2. If you need to minimize code footprint, follow the instructions in the section, Minimizing Code Footprint on page 25
3. Define the properties, events, and services that you want to expose to the server and create the required callback functions. Callback functions can be created to handle individual properties and services, or a single property or service callback can be created to handle all of those types of entities. Refer to the following sections:
   - Defining Properties on page 28
   - Defining Events on page 33
   - Define Property Callback Functions on page 33
   - Defining Service Callback Functions on page 35
4. If your application requires, set up the following:
   - Tasks — Refer toCreate Your Tasks (Optional) on page 36.
   - Bind Event Handler — Refer to Create a Bind Event Handler (Optional) on page 37.
   - Event Handler for File Transfers — Create a File Transfer Event Handler (Optional) on page 38.

- Event Handler for Tunneling — Create a Tunnel Event Handler (Optional) on page 38.
- TLS for secure communications — Refer to TLS Support on page 71 and then to TLS Provider Plugins on page 79.

5. Initialize the API Singleton on page 42.

Note

This initialization function initializes the Subscribed Properties Manager automatically.

6. Register Properties and Services on page 43.

   Register Events on page 44
7. Bind Your Entities on page 44 (Things).
8. If your application requires it, initialize the following components:
   - File Manager — Refer to Initialize the File Manager (Optional) on page 45x.
   - Tunnel Manager — Refer to Initialize the Tunnel Manager (Optional) on page 45.
9. Connect to the Server and Initiate Tasks on page 47.
10. Once your connection is alive and active, any requests made on the server for registered properties and services are automatically forwarded to your application, and the appropriate callback function is called. For information about server-initiated actions and callback functions, refer to Server-Initiated Interaction on page 54.

    Helper functions are available to push properties to the server, execute a service on another entity in the system, or trigger an event on the server. Refer to SDK Application-Initiated Interaction on page 58.
11. Build your application. Follow the instructions in Building an Application on page 65.

If you need to port your application to a platform for which the SDK does not provide a makefile or solution file, refer to Porting to Another Platform on page 69.

2

Configuring the C SDK

Configuring Components for an Application

Once you have decided which components your application requires, you must define the components as explained in this section.

Configure the desired components to include and verify that the SDK supports your platform/OS. If not, refer to the chapters on building (Building an Application on page 65) and porting (Porting to Another Platform on page 69), which describe the requirements and process for porting the SDK.

Provided within the SDK examples directory are example applications that demonstrate various capabilities of the SDK. Within each of those directories are a `win32`, `osx`, and a `linux` subdirectory, each with their own `.sln` file or `Makefile`, respectively. It is HIGHLY RECOMMENDED that you use one of these build files as a template or at least gain an understanding of what source files and configuration settings need to be included in your build environment.

In each of the build directories mentioned, there is a file named `CommonSettings` (Linux) or `CommonSettings.targets` (win32) that contains the configuration settings for building the SDK for your application. A full description of each of these settings is provided in this chapter. To change these settings or use something other than Visual Studio or Make to build the SDK and your application, edit the `src/config/twConfigOverrides.h` file to add your preferred options.

Configuring the Tasker

The built-in tasker is a simple round-robin execution engine that will call all registered functions at a rate defined when those functions are registered. If using a multitasking or multi-threaded environment you may want to disable the tasker and use the native environment. If you choose to disable the tasker, you must call `twApi_TaskerFunction()` and `twMessageHandler_msgHandlerTask()` on a regular basis (every 5 milliseconds or so). Undefine this setting if you are using your own threads to drive the API, as you do not want the tasker running in parallel with another thread running the API.

To properly initialize the Tasker, you must define `ENABLE_TASKER`:

```
/*********************************/
/*  Tasker Configuration        */
/*********************************/
#define ENABLE_TASKER 1
```

Configuring File Transfer

The C SDK has full support for all the remote directory/file browsing capabilities of the ThingWorx platform as well as bidirectional file transfer. To use this functionality, you must define ENABLE_FILE_XFER. This module will add ~15KB of code space to your application, so severely constrained environments may want to omit this functionality.

```
/********************************/
/* File Transfer Configuration   */
/********************************/
#define ENABLE_FILE_XFER 1
```

Configuring Application Tunneling

The C SDK has full support for application tunneling. Application tunnels allow for secure, firewall transparent tunneling of TCP client server applications such as VNC and SSH. To use this functionality, you must define ENABLE_TUNNELING. This module will add ~5KB of code space to your application, and upwards of 100KB RAM, depending on usage, so severely constrained environments may want to omit this functionality.

```
a/********************************/
/*    Tunneling Configuration    */
/*If defined, the tunneling system will be enabled.
*/
#define ENABLE_TUNNELING 1
```

Handling Offline Messages

The C SDK has multiple options for offline message storage. Offline message storage will queue up outgoing request messages for later delivery if the network is down or the duty cycle modulation component of the AlwaysOn protocol happens to be in the “off” state. If OFFLINE_MSG_STORE is not defined or set to 0, outbound messages are not queued at all. If OFFLINE_MSG_STORE is set to 1, messages will be queued up in RAM, up to a limit of OFFLINE_MSG_QUEUE_SIZE as defined in src/config/twDefaultSettings.h. When connectivity is re-established, all the messages in this queue will be sent out to the server.

Note

These messages are in RAM only. If there is a power outage or the system is shut down for any reason, these messages will be lost and not delivered to the server.

If OFFLINE_MSG_STORE is set to 2, messages will be persisted to a file in the directory OFFLINE_MSG_STORE_DIR, which is defined in src/config/twDefaultSettings.h, up to a limit of OFFLINE_MSG_QUEUE_SIZE. In both the RAM-based, and file-based offline message stores, when connectivity is re-established all the messages in this queue will be sent out to the server. Note that it is quite likely that all of these original messages will time out waiting for a response from the server, so you will not receive any indication or confirmation that these messages were successfully processed by the server. Also in either case, if the total size of the queued messages exceeds the limit defined in OFFLINE_MSG_QUEUE_SIZE, any subsequent attempt to queue more messages will fail and those new messages will be lost.

Here is an example of configuring offline message handling:

```
/*********************************/
/*   Offline Message Handling    */
/*********************************/
/*
The following settings define how to handle outgoing
Request messages that occur when offline
0 or undefined - Do nothing
1 - store in memory up to a limit of OFFLINE_MSG_QUEUE_SIZE
2 - persist to the directory OFFLINE_MSG_STORE_DIR
*/
#define OFFLINE_MSG_STORE 1
```

Additional Settings

The C SDK has several settings that you can modify, based on the needs of your application for things such as minimizing RAM usage or improving performance. The defaults for these settings are found in the file src/config/twDefaultSettings.h. In most cases you do not need to change these settings. If you must change them, exercise caution when making the changes.

With the exception of TW_MAX_TASKS, all of the settings can be modified at runtime by changing the appropriate setting in the global **twcfg** structure. The structure definition can be found in src/config/twDefaultSettings.h.

Note

As of release 1.2 of the C SDK, the default setting for DEFAULT_SOCKET_READ_TIMEOUT in twDefaultSettings.h is 500 ms. If you are using AxTLS and a web socket read times out in the middle of reading a record, the SSL state is lost. As a result, the SDK tries to start read the record header again, and the operation fails. To detect this situation, check the log for the SDK for the error, twTlsClient_Read: Timed out after X milliseconds, and consider increasing the value of the DEFAULT_SOCKET_READ_TIMEOUT. You can change the setting at runtime by modifying the value of twcfg.socket_read_timeout.

Minimizing Code Footprint

To attempt to create the smallest possible code footprint, define TW_LEAN_AND_MEAN. Using TW_LEAN_AND_MEAN disables optional, resource-consuming entities, such as offline message storage, tunneling, and file transfer. The default behavior is to remove all logging from the system.

Another way to minimize code footprint is to disable the resource-consuming entities you do not require.

The following code example shows the definition for TW_LEAN_AND_MEAN:

```
/*********************************/
/*   Minimize Code Footprint     */
/*********************************/
/*
Attempts to minimize the code footprint at the
expense of functionality.  Check your OS port
 header file to see what is disabled.
*/
#define TW_LEAN_AND_MEAN
```

Tips for Minimizing Footprint and Maximizing Performance

The C SDK has several settings that can significantly impact code footprint and performance. For performance, key among them is disabling verbose logging mode. Verbose logging parses every message sent between your application and the ThingWorx server. While extremely valuable for debugging, it can have a significant impact on performance. It is recommended that you disable verbose logging by calling twLogger_SetIsVerbose(FALSE);

Several areas impact code footprint. Support for connecting through HTTP Proxies adds ~5KB to your final code size. If not needed, follow this example: Suppose you are connecting over a cellular connection. To disable the support for HTTP Proxies, use #undef ENABLE_HTTP_PROXY_SUPPORT.

In addition, support for NTLM proxies adds ~45KB of code. To disable this support, use #undef USE_NTLM_PROXY.

File Transfer and Tunneling add ~15KB and 5KB respectively. You can disable them, using #undef ENABLE_FILE_XFER and #undef ENABLE_TUNNELING.

Finally logging itself adds ~20KB of code. Logging can be disabled with macros in parts by defining the log functions as empty as follows:

```
#define TW_LOG(level, fmt, ...)
#define TW_LOG_HEX(msg, preamble, length)
#define TW_LOG_MSG(msg, preamble)
```

The twWindows.h or twLinux.h files provide examples of using TW_LEAN_AND_MEAN to minimize the code footprint

.

3

Setting Up an Application

Overview

What do you need to do to set up the application using the C SDK? You must define the properties and services that you want to expose to the server and create the required callback functions. Callback functions can be created to handle individual properties and services or a single property or service callback can be created to handle all of those types of entities. This decision is left to the application developer

Optionally, you may need to set up tasks as well as event handlers:

- Bind event handler, so the application can determine which entities are bound to the ThingWorx Platform),
- File transfer event handler for file transfers to and from the ThingWorx Platform.
- Tunneling event handler for open and close events.

The C SDK uses a callback mechanism to handle server-initiated requests to read or write properties and invoke services. The signatures of the callback functions and the registration functions themselves are found in the file, `src/api/twApi.h`.

Defining Properties

In the ThingWorx environment, a property represents a data point, which has a name, a value, a timestamp, and optionally, a quality. In the ThingWorx Platform, properties can also have *aspects*, which provide additional details about the property. Once a client application binds an entity to a corresponding RemoteThing on the ThingWorx Platform, you can associate properties with the RemoteThing, using ThingWorx Composer.

The C SDK supports two types of properties, properties with do not have Remote Binding Information ”aspects” and so-called *subscribed* properties that have Remote Binding Information aspects that are displayed in ThingWorx Composer. These aspects are described in the *Property Definitions* section below.

Two types of structures are used by the C SDK to define properties:

- Property Definitions (`twPropertyDef`) to describe the basic information for the properties that are going to be available to ThingWorx and can be added to a client application.
- Property Values (`twProperty`) to associate the property name with a value, timestamp, and quality.

The structures are defined in the file, `twProperties.h`. The functions that support the creation and deletion of PropertyDef and Property structures are also defined in this file. The following table lists operations you may want to perform and the functions to use:

To	Use this Function
Create/allocate a Property Definition structure.	twPropertyDef_Create()
Free all memory associated with a Property Definition structure and all its owned substructures.	twPropertyDef_Delete()
Create/allocate a new Property value structure (name, value, and timestamp).	twProperty_Create()
Create/allocate a new Property VTQ structure (name, Value, Timestamp, and Quality).	twPropertyVTQ_Create()
Create a new Property VTQ structure from stream.	twProperty_CreateFromStream()
Free all memory associated with a Property value structure and all its owned substructures.	twProperty_Delete()

The following example of a simple property structure from the Steam Sensor example shows how the declaration of properties works:

```
/*****************
A simple structure to handle
properties.
******************/
struct  {
double TotalFlow;
char FaultStatus;
char InletValve;
double Pressure;
double Temperature;
double TemperatureLimit;
twLocation Location;
char * BigGiantString;
} properties;
```

To store the values sent by the Platform, you must use a callback method to either allocate a new variable or set the memory in an already allocated variable. For information about registering callbacks for properties, refer to Registering Properties and Services on page 43. For additional information, see also Property Access Callbacks on page 54 and the sections on reading, writing, and pushing properties in the section,, SDK Application-Initiated Interaction on page 58.

Property Definitions

The basic information that you provide for a Property Definition includes the following attributes:

- name — Specifies the name of the property that will appear in ThingWorx when users browse the related Thing when binding to it.
- description — Provides a description of the property that gives further understanding of the meaning of the property.
- baseType — Specifies the type of the property. For a list of base types supported by the SDK, refer to Base Types on page 50.
- aspects — Define the ways to interact with a property. All properties have the following aspects:
  - isPersistent — Set to TRUE for the ThingWorx server to persist the value even if it restarts. It is extremely expensive to have persistent values, so it is recommended to set this value to FALSE unless absolutely necessary.
  - isReadOnly — Set to TRUE to inform the ThingWorx server that this value is only readable and cannot be changed by a request from the server.
  - dataChangeType — Describes how the ThingWorx server responds when the value changes in the client application. Subscriptions to these value changes can be modeled in the ThingWorx server. If nothing needs to react to the property change, set this value to NEVER. The possible values are:

Select	To
ALWAYS	Always notify of the value change even if the new value is the same as the last reported value.
VALUE	Only notify of a change when a newly reported value is different than its previous value.
ON	For BOOLEAN types, notify only when the value is true.
OFF	For BOOLEAN types only, notify when the value is false.
NEVER	Ignore all changes to this value.

  - dataChangeThreshold — Defines how much the value must change to trigger a change event. For example 0 (zero) indicates that any change triggers an event. A value of 10 (ten) for example would not trigger an update unless the value changed by an amount greater than or equal to 10.
  - defaultValue — The default value is the value that the ThingWorx server uses when the RemoteThing connected to the device first starts up and has not received an update from the device. The value is different based on the different value for each base type.
- Only properties defined as *subscribed* properties have the following Remote Binding aspects:
  - cacheTime — The amount of time that the ThingWorx server caches the value before reading it again. A value of -1 informs the server that the

client application always sends its value and the server should never go and get it. A value of 0 (zero) indicates that every time the server uses the value, it should go and get it from the client application. Any other positive value indicates that the server caches the value for that many seconds and then retrieves it from the client application only after that time expired.

Note

For the client application to set the value every time it changes, set this value to -1.

- pushType — Informs the ThingWorx server how the client application pushes its values to the server. The possible values are as follows:

Select	For the Client to
ALWAYS	Send updates even if the value has not changed. It is common to use a `cacheTime` setting of `-1` in this case.
NEVER	Never send the value, which indicates that ThingWorx server only writes to this value.It is common to use a `cacheTime` setting of 0 or greater in this case.
VALUE	Send updates only when the value changes. It is common to use a `cacheTime` setting of `-1` in this case.

Properties need to be registered so that the ThingWorx server can browse them. Refer to Registering Properties and Services on page 43.

Property Values

You can define the property value in two ways – one with specific settings for timestamp and quality and one with the default quality.

Note

Updating a property value does not send the value to the ThingWorx server. To send the value to the server, the `twSubscribedPropsMgr_PushSubscribedProperties` function must be called.

Helper functions for creating property values include:

- setPropertyVTQ — Sets a property’s value using a VTQ (value, time, and quality) structure.
  - name — The name of the property.
  - value — The VTQ (value, time, and quality) for the property’s value.
  - forceChange — Set this value to true to force the value to be sent to the ThingWorx server even if it hasn’t changed. This option is a good option for sending the first value or sending a value immediately after reconnect.
- setPropertyValue — Sets a property’s value using a Primitive type.
  - name — The name of the property.
  - value — The Primitive type for the value.
- setProperty — Sets a property’s value from an object.
  - name — The name of the property.
  - value — The value to set. The value will be cast to the type of property if possible; otherwise an exception will be thrown.

Setting Up Subscribed Properties

The *subscribed properties* have a separate manager, called Subscribed Properties Manager.

Note

The Subscribed Properties Manager is initialized automatically when you call `twAPI_initialize()`. You do not need to initialize it separately.

You can set values for each subscribed property individually, using `twSetSubscribedPropery`, and then push them all at once to the ThingWorx server. To push subscribed properties, use `twSubscribedPropsMgr_PushSubscribedProperties`

The functions you can use for subscribed properties are listed in the table below.

To	Use
Define subscribed properties	The structure, `twSubscribedProperty`
Create a subscribed property	The function, `twSubscribedProperty_Create()`
Create a subscribed property from a stream	The function, `twSubscribedProperty_CreateFromStream()`

To	Use
Delete a subscribed property	The function, `twSubscribedProperty_Delete()`
Write a subscribed property to a stream	The function, `twSubscribedProperty_ToStream()`
Get the length of a subscribed property	The function, `twSubscribedProperty_GetLength()`
Send (“push”) subscribed properties to the ThingWorx server	The function, `twSubscribedPropsMgr_PushSubscribedProperties()`

Defining Events

Event definitions describe interrupts that ThingWorx Platform users can subscribe to if they want to be notified when something happens.

Events require that a data shape for event data be defined in code. Events can be defined in code or by using the following attributes:

- ThingWorxEventDefinition — Defines the event.
- name — Name of the event.
- description — A description for the event.
- dataShape — The name of the data shape for the event data.

Events must be registered. Refer to Register Events on page 44 for details. The registered event is reported back to the server when it is browsing. Note that Events do not have callbacks since they cannot be invoked from the ThingWorx Platform to the Edge. You can add aspects to an Event that is already registered, using `twApi_AddAspectToService`.

Define Property Callback Functions

The property callback function is registered to be called when a request for a specific property is received from the ThingWorx server; for example, if a service or a mashup references a property.

```
typedef enum msgCodeEnum (*property_cb)
(const char * entityName, const char * propertyName,
twInfoTable ** value, char isWrite, void * userdata)
```

The following parameters are passed to this function:

- entityName — the name of the entity this request is for
- propertyName — the name of the property the request is for
- twInfoTable ** value — a pointer to an twInfoTable that will contain the new property value if this is a write or will be populated with the current property value if this is a read. (For information on InfoTables, see the section, twInfoTable on page 52.)
- isWrite — a Boolean indicator saying whether this is a read or a write
- userdata — any user data value that was passed in when the callback was registered.

The return value of the function should be a message code enumeration as defined in src/api/twDefinitions.h. These message codes reflect the overall success or failure of your read or write operation locally. For more information about the return values, refer to the appendix, Callback Function Return Codes on page 99.

Example

```
/*****************
Property Handler Callbacks
******************/
enum msgCodeEnum propertyHandler(const char * entityName,
const char * propertyName,  twInfoTable ** value,
char isWrite, void * userdata) {
  TW_LOG(TW_TRACE,"propertyHandler - Function called for Entity %s,
        Property %s", entityName, propertyName);
  if (value) {
    if (isWrite && *value) {
        /* Property Writes */
        if (strcmp(propertyName, "InletValve") == 0)
           twInfoTable_GetBoolean(*value, propertyName, 0,
           &properties.InletValve);
        else if (strcmp(propertyName, "FaultStatus") == 0)
           twInfoTable_GetBoolean(*value, propertyName, 0,
           &properties.FaultStatus);
        else if (strcmp(propertyName, "TemperatureLimit") == 0)
           twInfoTable_GetNumber(*value, propertyName,
           0, &properties.TemperatureLimit);
        else return NOT_FOUND;
        return SUCCESS;
 } else {
        /* Property Reads */
        if (strcmp(propertyName, "InletValve") == 0)
          *value = twInfoTable_CreateFromBoolean(propertyName,
                    properties.InletValve);
        else if (strcmp(propertyName, "Temperature") == 0)
             *value = twInfoTable_CreateFromNumber(propertyName,
                        properties.Temperature);
        else if (strcmp(propertyName, "TemperatureLimit") == 0)
```

```
             *value = twInfoTable_CreateFromNumber(propertyName,
                      properties.TemperatureLimit);
        else if (strcmp(propertyName, "Location") == 0)
             *value = twInfoTable_CreateFromLocation(propertyName,
                      &properties.Location);
        else if (strcmp(propertyName, "BigGiantString") == 0)
             *value = twInfoTable_CreateFromString(propertyName,
                      properties.BigGiantString, TRUE);
        else return NOT_FOUND;
        }
        return SUCCESS;
 } else {
     TW_LOG(TW_ERROR,"propertyHandler - NULL pointer for value");
     return BAD_REQUEST;
     }
}
```

Define Service Callback Functions

The service callback function is registered to be called when a request for a specific service is received from the ThingWorx server.

```
typedef enum msgCodeEnum (*service_cb)
(const char * entityName, const char * serviceName,
twInfoTable * params,twInfoTable ** content, void * userdata)
```

The following parameters are passed to this callback function:

- `entityName` — the name of the entity this request is for (Thing, Resource, for example). Guaranteed to not be `NULL`.
- `serviceName` — the name of the service being requested
- `twInfoTable *params` — a pointer to an `twInfoTable` that contains all the parameters for the service. May be NULL if service has no parameters. (For information on InfoTables, see the section,twInfoTable on page 52)
- `twInfoTable ** content` — a pointer to a pointer to a `twInfoTable`. `content` is guaranteed to not be `NULL`. `*content` is not.

Note

A new instance of a `twInfoTable` should be created on the heap and a pointer to it returned.

- `userdata` — any user data value that was passed in when the callback was registered.

The return value of the function is TWX_SUCCESS if the request completes successfully or an appropriate error code if not (should be a message code enumeration as defined in twDefinitions.h).

Example

Here is an example of hanadling a single service in a callback:

```
/*****************
Service Callbacks
******************/
/* Example of handling a single service in a callback */
enum msgCodeEnum addNumbersService(const char * entityName,
const char * serviceName, twInfoTable * params,
twInfoTable ** content, void * userdata) {
        double a, b, res;
        TW_LOG(TW_TRACE,"addNumbersService - Function called");
        if (!params || !content) {
                TW_LOG(TW_ERROR,"addNumbersService -
                  NULL params or content pointer");
                return BAD_REQUEST;
        }

        twInfoTable_GetNumber(params, "a", 0, &a);
        twInfoTable_GetNumber(params, "b", 0, &b);
        res = a + b;
        *content = twInfoTable_CreateFromNumber("result", res);
        if (*content) return SUCCESS;
        else return INTERNAL_SERVER_ERROR;
}
```

Create Your Tasks (Optional)

If using the built-in tasker to drive data collection or other types of repetitive or periodic activities, create a function for the task. Task functions are registered with the Tasker and then called at the rate specified after they are registered. The Tasker is a very simple, cooperative multitasker, so these functions should not take long to return and most certainly must not go into an infinite loop.

The signature for a task function is found in src/utils/twTasker.h. The function is passed a DATETIME value with the current time and a void pointer that is passed into the Tasker when the task is registered.

Here is an example of a data collection task:

```
/***************
Data Collection Task
****************/
/*
This function gets called at the rate defined in the task creation.
The SDK has a simple cooperative multitasker, so the function
cannot infinitely loop.
```

```
Use of a task like this is optional and not required in a multithreaded
environment where this functionality could be provided in a separate thread.
*/
#define DATA_COLLECTION_RATE_MSEC 2000
void dataCollectionTask(DATETIME now, void * params) {
  /* TW_LOG(TW_TRACE,"dataCollectionTask: Executing"); */
      properties.TotalFlow = rand()/(RAND_MAX/10.0);
      properties.Pressure = 18 + rand()/(RAND_MAX/5.0);
      properties.Location.latitude = properties.Location.latitude +
                       ((double)(rand() - RAND_MAX))/RAND_MAX/5;
        properties.Location.longitude = properties.Location.longitude +
                       ((double)(rand() - RAND_MAX))/RAND_MAX/5;
        properties.Temperature  = 400 + rand()/(RAND_MAX/40);
        /* Check for a fault.  Only do something if we haven't already */
        if (properties.Temperature > properties.TemperatureLimit &&
                                     properties.FaultStatus == FALSE) {
                twInfoTable * faultData = 0;
                char msg[140];
                properties.FaultStatus = TRUE;
                properties.InletValve = TRUE;
                sprintf(msg,"%s Temperature %2f exceeds threshold of %2f",
                        thingName, properties.Temperature,
                        properties.TemperatureLimit);
                faultData = twInfoTable_CreateFromString("msg", msg, TRUE);
                twApi_FireEvent(TW_THING, thingName,
                                "SteamSensorFault", faultData, -1, TRUE);
                twInfoTable_Delete(faultData);
        }
        /* Update the properties on the server */
        sendPropertyUpdate();
}
```

Creating a Bind Event Handler (Optional)

You may want to track exactly when your edge entities are successfully bound to or unbound from the server. The reason for this is that only bound items should be interacting with the ThingWorx Platform and the ThingWorx Platform will never send any requests targeted at an entity that is not bound.

```
/* Register a bind event handler */
/* Callbacks only when thingName is bound/unbound */
    twApi_RegisterBindEventCallback(thingName, BindEventHandler, NULL);

/* First NULL says "tell me about all things that are bound */

/* twApi_RegisterBindEventCallback(NULL, BindEventHandler, NULL
```

Create a File Transfer Event Handler (Optional)

If you are using the File Transfer capability of the C SDK, you may want to create an event handler for any file transfer events. This handler will be called whenever a new file is successfully sent from the server to your application, and when an asynchronous file transfer from your device to the service has completed either successfully or unsuccessfully.

The signature for a file transfer event callback is as follows:

```
typedef void (*file_cb) (char fileRcvd, twFileTransferInfo * info);
```

The input parameters for this callback function are as follows:

- `fileRcvd` — a Boolean. `TRUE` is the file was received, `FALSE` if it was being sent
- `info` — a pointer to the file transfer info structure. The called function retains ownership of this pointer and must delete it with `twFileTransferInfo_Delete()` when it has finished using it

Return:

- None

The structure definition of `twFileTransferInfo` can be found in the file `src/fileTransfer/twFileManager.h`.

Create a Tunnel Event Handler (Optional)

If you are using the Tunneling capability of the C SDK, you may want to create an event handler for any tunneling events. This handler will be called whenever a new tunnel is established or when a tunnel closes. The `twTunnelManager` also provides functions to list active tunnels as well as to force a shutdown of an active tunnel.

The signature for a tunnel event callback is as follows:

```
typedef void (*tunnel_cb) (char started, const char * tid,
const char * thingName, const char * peerName,
const char * host, int16_t port, DATETIME startTime,
  DATETIME endTime, uint64_t bytesSent, uint64_t bytesRcvd,
const char * type, const char * msg, void * userdata);
```

The Input parameters for this callback function are as follows:

- `started` — Boolean. TRUE is the tunnel is started, FALSE if tunnel has ended.
- `tid` — the unique id of the tunnel
- `thingName` — the name of the thing this tunnel is targeted at
- `peerName` — the name of the peer user of the tunnel

- `host` — the hostname of the local connection that is tunneled to
- `port` — the port number of the local connection that is tunneled to
- `startTime` — the time the tunnel started (0 if it never started)
- `endTime` — the time the tunnel ended (0 if it hasn't ended yet)
- `bytesSent` — the total number of bytes that were sent to the peer
- `bytesRcvd` — the total number of bytes that were received from the peer
- `type` — the type of the tunnel (tcp, udp, or serial)
- `userdata` — an opaque pointer that was passed in during registration

Return:

- None

The definition of the `twTunnelManager` singleton's functions can be found in the file `src/tunneling/twTunnelManager.h`.

4

Running the C SDK

After developing the callback handler functions, it is now time to do something with them. Continue here to learn what you should typically do in your ‘main’ function (or in a function called by main).

Initializing the API Singleton

Initializing the API singleton configures the connection to the server, but does NOT establish the connection. Typically, only the Host and the apiKey need to be modified, all other defaults can be used. For security purposes, the API defaults to rejecting self-signed certificates. If you choose to override this behavior, you can tell the API to allow them.

To initialize the API:

```
/* Initialize the API */
api = twApi_Initialize("localhost", 443,
         TW_URI, "1724be81-fa15-4485-a966-287bf8f6683c",
         NULL, MESSAGE_CHUNK_SIZE, MESSAGE_CHUNK_SIZE, TRUE);
```

The signature for this function and definitions of its parameters can be found in the file, twApi.h.

Note

This function initializes the Subscribed Properties Manager. You do not need to initialize this manager separately.

By default the API is set up to ensure the most secure connection possible. For the most secure connection, set the issuer and subject fields of your server certificates before starting the connection by using the twApi_SetX509Fields() function. These settings mean that it will attempt to validate certificates and reject self-signed certificates. Many settings are available to modify the default behavior and may provide some level of convenience during development, such as allowing self-signed certificates. However, modifying from the most secure settings possible for production is NOT recommended. These functions can be found in the file, twApi.h, and are as follows:

```
int twApi_SetProxyInfo(char * proxyHost, uint16_t proxyPort,
  char * proxyUser, char * proxyPass);
void twApi_SetSelfSignedOk();
int twApi_EnableFipsMode();
void twApi_DisableCertValidation();
void twApi_DisableEncryption();
int twApi_SetX509Fields(char * subject_cn,
  char * subject_o, char * subject_ou, char * issuer_cn,
  char * issuer_o, char * issuer_ou);
int twApi_LoadCACert(const char *file, int type);
int twApi_LoadClientCert(char *file);
int twApi_SetClientKey(const char *file, char * passphrase, int type);
```

Cipher Suites Supported

The C SDK supports the following cipher suites by default:

- TLS_RSA_WITH_AES_256_CBC_SHA
- TLS_RSA_WITH_AES_128_CBC_SHA
- TLS_RSA_WITH_RC4_128_SHA
- TLS_RSA_WITH_RC4_128_MD5.

The standard builds of the EMS, .NET SDK, iOS SDK all use the C SDK, and these same ciphers. When you set up a ThingWorx Platform make sure you enable TLS v1.1 and disable SSL in your `server.xml` file.

FIPS Mode

Your application can use an embedded FIPS-140-2-validated cryptographic module (Certificate #1747; OpenSSL FIPS module version 2.0.2) running on all supported platforms per FIPS 140-2 Implementation Guidance section G.5 guidelines. The C SDK uses the OpenSSL toolkit in conjunction with the OpenSSL FIPS Object Module 2.0.2. .

Note

Not all hardware platforms where applications written using the C SDK can run support FIPS-140-2-validated cryptography. For example, on platforms based on IA32 architecture, the processor must support the SSE2 instruction set. The SSE2 instruction set is available in Intel x86 CPUs, starting with Pentium 4. The application log will have a message that FIPS-140-2-validated cryptography is enabled. If you enable it, be sure that your certificates include only FIPS approved encryption algorithms. The FIPS approved algorithms are AES, Triple-DES, RSA, DSA, DH, SHA1, and SHA2.

If the FIPS module is enabled and the application directly communicates with a Java-based SSL/TLS server (such as the ThingWorx Platform), the cipher suite list should include `!kEDH` (as shown below). Otherwise, ephemeral Diffie-Hellman (EDH) key exchange may fail:

```
<CipherSuites>DEFAULT:!kEDH</CipherSuites>
```

Registering Properties and Services

Registering properties and services with the API accomplishes two things:

1. Tells the API what callback function to invoke when a request for that property or service comes in froRegistreing Properties and Servicesm the server.
2. Gives the API information about the property or service so that when the ThingWorx Composer browses the Edge device, it can be informed about the availability and the definition of that property or service.

To register services and properties, follow these examples:

```
/* Register our services */
ds = twDataShape_Create(twDataShapeEntry_Create("a",NULL,TW_NUMBER));
twDataShape_AddEntry(ds, twDataShapeEntry_Create("b",NULL,TW_NUMBER));
twApi_RegisterService(TW_THING, thingName,
    "AddNumbers", NULL, ds, TW_NUMBER, NULL, addNumbersService, NULL);


/* Register our properties */
twApi_RegisterProperty(TW_THING, thingName,
    "InletValve", TW_BOOLEAN, NULL, "ALWAYS", 0, propertyHandler, NULL);
twApi_RegisterProperty(TW_THING, thingName,
    "Pressure", TW_NUMBER, NULL, "ALWAYS", 0, propertyHandler, NULL);
twApi_RegisterProperty(TW_THING, thingName,
    "BigGiantString", TW_STRING, NULL, "ALWAYS", 0, propertyHandler, NULL);
```

For more information about using the callbacks, refer to the section, Server-Initiated Interactions on page 54.

Registering Events

Events do not have callbacks because they cannot be invoked from the ThingWorx Platform as a request to the edge device running your application. For your application to report events back to the ThingWorx Platform, use the `twApi_RegisterEvent` function to register the events. For more information about the function, refer to the Doxygen documentation that accompanies the C SDK.

Binding Your Entities

Bind each entity (Thing) so that when the API connects (and reconnects) to the server, it will announce that your entity is connected and available for interaction. The API can be used as a gateway, where multiple entities can be bound at the same time. In addition, the API supports unbinding entities so transient “Things” are supported.

To bind an entity, use its `thingName`, as shown here:

```
/* Bind our thing */
twApi_BindThing(thingName);
```

Initializing the File Manager (Optional)

If using the directory browsing and file transfer capability of the SDK, perform the following steps:

1. Set the staging directory — You must set the staging directory before initializing the FileManager. The default directory of the FileManager is most likely owned by root and will require a change to either the location of the staging directory and the ownership of the staging directory, or running the application as a user with the correct permissions.
2. Initialize the FileManager singleton.
3. Define any virtual directories — Virtual directories allow you to expose only a subset of the entire file system of the device to the server for browsing and file transfer. This restriction is for both performance and security reasons.

   Registering a virtual directory with the FileManager consists of mapping a unique name to an absolute path of a directory in your file system. Note that all subdirectories of the specified directory in the file system will be exposed to the server. Multiple virtual directories can be defined and there is no requirement that they be contiguous.
4. Register the `FileCallback` function that was previously defined so that the FileManager will call that function when any file transfer events occur. You can provide a wildcard filter so that only file transfer events of files that match the filter call the callback function. In addition, callbacks can be set up as "one-shots" such that the callback is unregistered automatically after it is invoked the first time

Here are examples for each of these steps:

```
/* Staging Directory Variable */, must be set before initializing file manager
twcfg.file_xfer_staging_dir="/home/user/stagingdir";

/* Initialize the FileTransfer Manager */
twFileManager_Create();

/* Create our virtual directories */
twFileManager_AddVirtualDir(thingName, "tw", "/opt/thingworx");
twFileManager_AddVirtualDir(thingName, "tw2", "/twFile_tmp");

/* Register the file transfer callback function */
twFileManager_RegisterFileCallback(fileCallbackFunc, NULL, FALSE, NULL);
```

Initializing the Tunnel Manager (Optional)

If using the tunneling capability of the C SDK you must create `#define ENABLE_TUNNELING`. A tunnel manager singleton is automatically created for you when you initialize the API. If you wish to disable tunneling for any reason you may call twTunnelManager_Delete(). The tunnel manager may be started up again by calling twTunnelManager_Create(). Once the tunnel manager is running you may register any callback functions. Passing a NULL for the id parameter when registering a callback will result in callbacks for all tunnel events.

```
/* Register the tunnel callback function */
twTunnelManager_RegisterTunnelCallback(tunnelCallbackFunc, NULL, NULL);
```

When new tunnels are requested by the server, the tunnel manager creates a new tunnel. These tunnels establish an independent websocket back to the server. By default these websockets connect back to the same host/port that the API's websocket uses as well as the same TLS certificate validation criteria. You can override these defaults by using the built-in tunnel manager functions as found in the file, twTunnelManager.h:

```
int twTunnelManager_UpdateTunnelServerInfo(char * host,
                                        uint16_t port, char * appkey);
void twTunnelManager_SetProxyInfo(char * proxyHost, uint16_t proxyPort,
    char * proxyUser, char * proxyPass);
void twTunnelManager_SetSelfSignedOk(char state);
void twTunnelManager_EnableFipsMode(char state);
void twTunnelManager_DisableCertValidation(char state);
void twTunnelManager_DisableEncryption(char state);
void twTunnelManager_SetX509Fields(char * subject_cn, char * subject_o,
   char * subject_ou, char * issuer_cn,
   char * issuer_o, char * issuer_ou);
void twTunnelManager_LoadCACert(const char *file, int type);
void twTunnelManager_LoadClientCert(char *file);
void twTunnelManager_SetClientKey(const char *file, char * passphrase, int type);
```

Note

If you are not using the built-in tasker, you must call the function twTunnelManager_TaskerFunction on a very frequent basis (every 5 msec or so).

Creating a Bind Event Handler (Optional)

You may want to track exactly when your edge entities are successfully bound to or unbound from the server. The reason for this is that only bound items should be interacting with the ThingWorx Platform and the ThingWorx Platform will never send any requests targeted at an entity that is not bound.

```
/* Register a bind event handler */
/* Callbacks only when thingName is bound/unbound */
    twApi_RegisterBindEventCallback(thingName, BindEventHandler, NULL);

/* First NULL says "tell me about all things that are bound */
/* twApi_RegisterBindEventCallback(NULL, BindEventHandler, NULL
```

Connecting to the Server and Initiating Defined Tasks

Connecting to the server first and then initiating tasks is the preferable order, especially if your tasks will be pushing data to the server. If you start the tasks earlier, they may attempt to send property updates or invoke services on the server before the connection has been established. While reversing the order will not cause any lasting problems, it will tend to keep the system very busy with retries before the connection is established.

The connection to the server will be attempted and retried with the parameters specified to the `twApi_Connect()` function. By default, the API will automatically reconnect using the same parameters if the connection is subsequently lost. This behavior can be overridden when the API is initialized by setting the `autoreconnect` parameter to `FALSE`.

Note

As of release 1.2 of the C SDK, the default setting for `DEFAULT_SOCKET_READ_TIMEOUT` in `twDefaultSettings.h` is 500 ms. If you are using AxTLS and a web socket read times out in the middle of reading a record, the SSL state is lost. As a result, the SDK tries to start read the record header again, and the operation fails. To detect this situation, check the log for the SDK for the error, `twTlsClient_Read: Timed out after X milliseconds`, and consider increasing the value of the `DEFAULT_SOCKET_READ_TIMEOUT`. You can change the setting at runtime by modifying the value of `twcfg.socket_read_timeout`.

In release 1.2, the inputs to the `twApi_Connect()` function calls were changed in the `main.c` source for the Steam Sensor example files. The retries parameter was changed from `CONNECT_RETRIES` to the globally defined `twcfg.connect_retries`.

The API also supports callback notifications when a connection is successfully made and when a connection is lost. The signature for “event callback” functions can be found in the file, `src/messaging/twMessaging.h`, and the task registration functions are found in the file, `twApi.h`.

```
/* Connect to server */
if (!twApi_Connect(CONNECT_TIMEOUT, twcfg.connect_retries)) {
/* Register our "Data collection Task" with the tasker */
twApi_CreateTask(DATA_COLLECTION_RATE_MSEC, dataCollectionTask);
}
```

5

Interacting with the ThingWorx Platform

This section discusses how to interact with the ThingWorx Platform.

Basic Data Structures

Once your connection is alive and active, any requests made to the server for registered properties and services will automatically be forwarded to your application, and the appropriate callback function will be called. To push properties to the server, execute a service on another entity in the system, or trigger an event on the server. Helper functions are available for these actions. These functions are described in the section, SDK Application-Initiated Interaction on page 58.

Data in the C SDK are represented in the form of a twPrimitive structure. Collections of data values are represented in the form of a twInfoTable structure. Each of these structures is defined below and the API functions to access them are found in src/messaging/twBaseTypes.h and twInfoTable.h, respectively.

twPrimitiveStructure

The twPrimitiveStructure is a form of a variant that can represent any of the base types supported in the ThingWorx platform. The structure is defined in src/messaging/twBaseTypes.h as follows:

```
typedef struct twPrimitive {
  enum BaseType type;
  enum BaseType typeFamily;
  uint32_t length;
  union {
         int32_t integer;
         double number;
         DATETIME datetime;
         twLocation location;
         char boolean;
         struct {
               char * data;
               uint32_t len;
         } bytes;
         struct twInfoTable * infotable;
         struct twPrimitive * variant;
  } val;
} twPrimitive;
```

The key fields are the type enumeration and the val union. The fields typeFamily and length are for internal API use and are typically not used by an application.

Base Types

The supported base types are defined in src/api/twDefinitions.h and consist of the following:

Base Types

Base Type	Description
TW_NOTHING	An empty val.
TW_STRING	A modified UTF8 encoded string. Data and length are stored in val.bytes and val.len, respectively. The twPrimitive owns the data pointer and will free it when deleted. TW_STRING types are null terminated.
TW_NUMBER	A C double value, stored in val.double.
TW_BOOLEAN	Represented as a single char, stored in val.boolean.
TW_DATETIME	A DATETIME value, which is an unsigned 64 bit value representing milliseconds since the epoch 1/1/1970. Data is stored in val.datetime.
TW_INFOTABLE	A pointer to a complex structure (defined in the next section) and stored in val.infotable. The twPrimitive owns this pointer and will free up the memory pointed to when the twPrimitive is deleted.
TW_LOCATION	A structure consisting of three double floating point values – longitude, latitude, and elevation. Stored as val.location.
TW_BLOB	A pointer to a character array. Data and length are stored in val.bytes and val.len, respectively. Differs from TW_STRING in that the array may contain nulls. The twPrimitive owns the data pointer and will free it when deleted.
TW_IMAGE	Identical to TW_BLOB except for the type difference.
TW_INTEGER	Assigned 4 by integral value. Stored as val.integer.
TW_VARIANT	Pointer to a structure that contain a type enum and a twPrimitive value. The pointer is stored as val.variant. The twPrimitive owns the pointer and will free the structure when deleted.
TW_XML,TW_JSON, TW_QUERY, TW_HYPERLINK, TW_IMAGELINK, TW_PASSWORD, TW_HTML, TW_TEXT, TW_TAGS, TW_GUID,TW_THINGNAME, TW_THINGSHAPENAME, TW_THINGTEMPLATENAME, TW_DATASHAPENAME, TW_MASHUPNAME, TW_MENUNAME, TW_BASETYPENAME, TW_USERNAME, TW_GROUPNAME, TW_CATEGORYNAME, TW_STATEDEFINITIONNAME, TW_STYLEDEFINITIONNAME, TW_MODELTAGVOCABULARYNAME, TW_DATATAGVOCABULARYNAME, TW_NETWORKNAME, TW_MEDIAENTITYNAME, TW_APPLICATIONKEYNAME, TW_LOCALIZATIONTABLENAME, TW_ORGANIZATIONNAME	These base types are all of the TW_STRING family and are stored similarly.

There are many helper functions for creating twPrimitive structures from base types so that you will rarely have to create one manually. These function definitions can be found in src/messaging/twBaseTypes.h.

twInfoTable

The **twInfoTable** is the primary mechanism for sending data to and from the ThingWorx server. A **twInfoTable**is essentially a self-describing collection of twPrimitive values.

Structure of an InfoTable

The structure of a **twInfoTable** follows:

```
typedef struct twInfoTable {
  twDataShape * ds;
  twList * rows;
  uint32_t length;
  TW_MUTEX mtx;
} twInfoTable;
```

The ds element is a pointer to a twDatashape structure that describes what each field (column) of the table is – its name, description, and the base type of that field. The base type of a field can be any one of the base types described in the table on page 50, including a **twInfoTable**, as the SDK and platform allow nesting of these tables.

The rows element is a pointer to a list of values. Each entry in the list is a pointer to a twInfoTableRow structure. The twInfoTableRow structure contains values for each of the fields described in the data shape and must contain the values in the same order as in the data shape. The number of rows in an **twInfoTable** is a 32-bit value and therefore only practically limited to how much memory you wish to allow the **twInfoTable** to consume.

The length and mtx elements of the **twInfoTable** structure are for internal use and are typically not accessed directly. All the pointer elements of an InfoTable are owned and managed by the **twInfoTable** and should not be deleted or freed on their own.

Creating an InfoTable

Creating an **InfoTable** is a three step process, as follows:

1. Create your data shape and add any necessary entries (fields) to the data shape.

```
twDataShapeEntry * twDataShapeEntry_Create(const char * name,
    const char description, enum BaseType type);

twDataShape * twDataShape_Create(twDataShapeEntry * firstEntry);

int twDataShape_AddEntry(struct twDataShape * ds,
    struct twDataShapeEntry * entry);
```

Caution

You must create a data shape to hold the schema for the **twInfoTable** BEFORE creating the table. Once the table is created, data is added one row at a time. When a row is created, data must be added to the row in the same order that it is in data shape. If the data is not added in the correct order, the table does not form correctly. There is no warning about this, and it becomes evident only when a user attempts to view the data in ThingWorx Composer or a mashup that the data is being added incorrectly.

2. Create the **twInfoTable**, which requires its data shape to be passed in as a parameter.

   ```
   twInfoTable * twInfoTable_Create(twDataShape * shape)
   ```

3. Add data to the **twInfoTable** by individually creating the rows and adding them to the it.

   ```
   twInfoTableRow * twInfoTableRow_Create(twPrimitive * firstEntry)
   int twInfoTableRow_AddEntry(twInfoTableRow * row, twPrimitive * entry)
   int twInfoTable_AddRow(twInfoTable * it, twInfoTableRow * row)
   ```

Helper Functions for InfoTables

One very common pattern is a **twInfoTable** that contains a single field and a single row, for example the current value of a single property. The API provides several helper functions that make it easy to create these simple tables, using just a single function call.

```
twInfoTable * twInfoTable_CreateFromString(const char * name, char * value,
     char duplicate);
twInfoTable * twInfoTable_CreateFromNumber(const char * name, double value);
twInfoTable * twInfoTable_CreateFromInteger(const char * name, int32_t value);
twInfoTable * twInfoTable_CreateFromLocation(const char * name, twLocation * value
twInfoTable * twInfoTable_CreateFromDatetime(const char * name, DATETIME value);
twInfoTable * twInfoTable_CreateFromBoolean(const char * name, char value);
twInfoTable * twInfoTable_CreateFromPrimitive(const char * name, twPrimitive * val
twInfoTable * twInfoTable_CreateFromBlob(const char * name, char * value,
     int32_t length, char isImage, char duplicate);
```

Accessing data contained in a **twInfoTable** is also easy with several helper functions defined to assist with the common usage patterns. You simply pass in the name of the field and which row you wish to retrieve the value from.

```
int twInfoTable_GetString(twInfoTable * it, const char * name,
    int32_t row, char ** value);
int twInfoTable_GetNumber(twInfoTable * it, const char * name,
    int32_t row, double * value);
int twInfoTable_GetInteger(twInfoTable * it, const char * name,
    int32_t row,int32_t * value);
int twInfoTable_GetLocation(twInfoTable * it, const char * name,
    int32_t row, twLocation * value);
int twInfoTable_GetBlob(twInfoTable * it, const char * name,
```

```
    int32_t row, char ** value, int32_t * length);
int twInfoTable_GetDatetime(twInfoTable * it, const char * name,
    int32_t row, DATETIME * value);
int twInfoTable_GetBoolean(twInfoTable * it, const char * name,
    int32_t row, char * value);
int twInfoTable_GetPrimitive(twInfoTable * it, const char * name,
    int32_t row, twPrimitive ** value);
```

Server-Initiated Interaction

To respond to requests for properties and services from the server, the API provides the property access and service access callbacks. The next two sections describe these callbacks, their parameters, and return values, and provide examples of using these callbacks.

Property Access Callbacks

Property access callbacks are the functions that are called when a request comes from the server to either read or write a specific property. These functions have the following signature:

```
enum msgCodeEnum myPropCallback (
    const char * entityName,
    const char * propertyName,
    twInfoTable ** value,
    char isWrite,
    void * userdata
)
```

The following table lists and describes the parameters:

Parameter	Type	Description
entityName	Input	Pointer to a character array. The name is represented as a modified UTF-8 string with the name of the entity targeted in this request. This parameter is guaranteed not to be null.
propertyName	Input	Pointer to a character array. This is the name of the property, represented in modified UTF-8. This value may be null or '*" which means the request is to return the value of all properties registered for this entity.
value	Input/Output	Pointer to a pointer to a twInfoTable. If this is a request to read the value of a property a new twInfoTable structure should be created and it pointer should assigned to value. If this is a write, the

Parameter	Type	Description
		value will contain a pointer to the infotable that contains the data to be written. This pointer is guaranteed to be non-NULL. In either case, the calling function will assume ownership of the pointer in `*value`, so the callback function does not need to worry about memory management of any infotables passed in or created and returned as values.
`isWrite`	Input	A Boolean value describing whether this is a read (`FALSE`) or write (`TRUE`) request for the property.
`userdata`	Input	The same pointer value that was passed in when this property was registered. This pointer can be used for anything. A typical use is to specify the `this` pointer when using C++ class wrappers.

The return value of the callback is an indicator of the success or failure of the function. You are free to choose any of the return codes defined in the `msgCodeEnum` enumeration type, defined in `src/api/twDefinitions.h`, starting with `SUCCESS` or any applicable larger value.

Below is a simple example of a property handler callback function.

```
enum msgCodeEnum propertyHandler(const char * entityName,
                                 const char * propertyName,
                                 twInfoTable ** value,
                                 char isWrite,
void * userdata) {
        char * asterisk = “*”;
   if (!propertyName) propertyName = asterisk;
   TW_LOG(TW_TRACE,"propertyHandler - Function called for Entity %s,
     Property %s", entityName, propertyName);
        if (value) {
                if (isWrite && *value) {
                        /* Property Writes */
                        if (strcmp(propertyName, "TemperatureLimit") == 0) {
               twInfoTable_GetNumber(*value, propertyName, 0,
                                    &properties.TemperatureLimit);
                        } else return NOT_FOUND;
                        return SUCCESS;
                } else {
                        /* Property Reads */
                        if (strcmp(propertyName, "TemperatureLimit") == 0)
                      {*value = twInfoTable_CreateFromNumber(propertyName,
                            properties.TemperatureLimit);
                        } else return NOT_FOUND;
                }
                return SUCCESS;
        } else {
                TW_LOG(TW_ERROR,"propertyHandler - NULL pointer for value");
                return BAD_REQUEST;
        }
}
```

Service Callbacks

Service callbacks are the functions that are called when a request comes from the ThingWorx Platform to execute a service on a particular entity. These functions have the following signature:

```
typedef enum msgCodeEnum (*service_cb) (
                                const char * entityName,
                                const char * serviceName,
                                twInfoTable * params,
                                twInfoTable ** content);
```

The following table defines the parameters:

Parameters for msgCodeEnum()

Parameter	Type	Description
`entityName`	Input	Pointer to a character array. The name is represented as a modified UTF-8 string with the name of the entity targeted in this request. This parameter is guaranteed not to be NULL.
`serviceName`	Input	Pointer to a character array. This is the name of the service to be executed, represented in modified UTF-8. This parameter is guaranteed not to be NULL.
`params`	Input	Pointer to a `twInfoTable`. This is a pointer to an infotable that contains all of the parameters specified for this invocation of the service. This pointer may be NULL if the service in question has no input parameters. The API owns this pointer and will manage any memory associated with it.

Parameters for msgCodeEnum() (continued)

Parameter	Type	Description
content	Output	Pointer to a pointer to a twInfoTable. This is used to return any data the service returns back to the server. The callback function should create a twInfoTable as described previously and pass a pointer to that structure to *content. If the service does not return any data it is OK to set *content to NULL. The API will assume ownership of the pointer in *value, so the callback function does not need to worry about memory management of any infotables passed in or created and returned as values.
userdata	Input	The same pointer value that was passed in when this property was registered. This pointer can be used for anything, a typical use is to specify the 'this' pointer when using C++ class wrappers.

The return value of the callback is an indicator of the success or failure of the service call. You are free to choose any of the return codes defined in the `msgCodeEnum` enumeration type, defined in `src/api/twDefinitions.h`, starting with `SUCCESS` or any applicable larger value. Here is an example of a service handler callback:

```
enum msgCodeEnum addNumbersService(const char * entityName,
                                   const char * serviceName,
                                   twInfoTable * params,
  twInfoTable ** content,
  void * userdata) {
        double a, b, res;
        TW_LOG(TW_TRACE,"addNumbersService - Function called");
        if (!params || !content) {
                TW_LOG(TW_ERROR,"addNumbersService - NULL params or content pointe
                return BAD_REQUEST;
        }
        if (twInfoTable_GetNumber(params, "a", 0, &a) ||
            twInfoTable_GetNumber(params, "b", 0, &b)) {
                TW_LOG(TW_ERROR,"addNumbersService – Missing parameter data");
                return BAD_REQUEST;
    }
        res = a + b;
        *content = twInfoTable_CreateFromNumber("result", res);
        if (*content) return SUCCESS;
        else return INTERNAL_SERVER_ERROR;
}
```

SDK Application-Initiated Interaction

The SDK provides functions to make it easy for an application to initiate interaction with the ThingWorx Platform. Assuming all the proper visibility, permissions, and other security aspects are correct, an entity built using the C SDK can read or write properties, create a list of subscribed properties, set values of subscribed properties, invoke services, and trigger events on itself or other entities in the system. The following sections describe the helper functions

Read a Property

This helper function retrieves the current value of a property of a specific entity on the ThingWorx Platform.

```
enum msgCodeEnum twApi_ReadProperty(enum entityTypeEnum entityType,
                                    char * entityName, char * propertyName,
                                    twPrimitive ** result, int32_t timeout,
                                    char forceConnect)
```

The following table lists and describes the parameters for this helper function:

Parameter	Type	Description
`entityType`	Input	The type of entity that the property belongs to. Enumeration values can be found in `twDefinitions.h`
`entityName`	Input	The name of the entity that the property belongs to.
`propertyName`	Input	The name of the property to retrieve.
`result`	Input/Ouput	A pointer to a `twPrimitive` pointer. In a successful request, this parameter will end up with a valid pointer to a `twPrimitive` value. The caller is responsible for deleting the returned primitive using `twPrimitive_Delete`. It is possible for the returned pointer be a NULL if an error occurred.

Parameter	Type	Description
timeout	Input	The time (in milliseconds) to wait for a response from the server. A value of -1 uses the DEFAULT_MESSAGE_TIMEOUT as defined in twDefaultSettings.h
forceConnect	Input	A Boolean value. If TRUE and the API is in the disconnected state of the duty cycle, the API will force a reconnect to send the request.

Return:

- msgCodeEnum — the result of the call. See twDefinitions.h for the enumeration definition.

Write a Property

This helper function writes a new value for a property of a specific entity on the ThingWorx Platform.

```
enum msgCodeEnum twApi_WriteProperty(enum entityTypeEnum entityType,
char * entityName, char * propertyName,
twPrimitive * value, int32_t timeout, char forceConnect)
```

The following table lists and describes the parameters for this helper function:

Parameter	Type	Description
entityType	Input	The type of entity that the property belongs to. Enumeration values can be found in twDefinitions.h.
entityName	Input	The name of the entity that the property belongs to.
propertyName	Input	The name of the property to retrieve.
value	Input	A pointer to a twPrimitive that contains the value to set for the property. Once called, the calling function will retain ownership of this pointer and must manage the memory lifecycle. NOTE: The called function WILL alter the contents of this primitive, so the original contents cannot be relied upon after the function returns..

Parameter	Type	Description
timeout	Input	The time (in milliseconds) to wait for a response from the server. A value of -1 uses the DEFAULT_MESSAGE_TIMEOUT as defined in twDefaultSettings.h.
forceConnect	Input	A Boolean value. If TRUE and the API is in the disconnected state of the duty cycle, the API will force a reconnect to send the request.

Return:

- msgCodeEnum — the result of the call. See twDefinitions.h for the enumeration definition.

Push Properties

Use this function to update one or more properties with a single message to the ThingWorx Platform. You can also use it to send multiple values of the same property to the ThingWorx Platform in a single message.

```
enum msgCodeEnum twApi_PushProperties(enum entityTypeEnum entityType,
char * entityName, propertyList * properties, int32_t timeout,
      char forceConnect)
```

The following table lists and describes the parameters for this helper function:

Parameter	Type	Description
entityType	Input	The type of entity that the properties belong to. Enumeration values can be found in the file, twDefinitions.h
entityName	Input	The name of the entity that the properties belong to.
properties	Input	A pointer to a list of twPrimitives. The calling function will retain ownership of this pointer and is responsible for cleaning up the memory after the call is complete.

Parameter	Type	Description
timeout	Input	The time (in milliseconds) to wait for a response from the server. A value of -1 uses the DEFAULT_MESSAGE_TIMEOUT as defined in twDefaultSettings.h
forceConnect	Input	A Boolean value. If TRUE and the API is in the disconnected state of the duty cycle, the API will force a reconnect to send the request.

Return:

- msgCodeEnum — the result of the call. See twDefinitions.h for the enumeration definition.

An example usage of the twApi_PushProperties function is as follows:

```
void sendPropertyUpdate() {propertyList * proplist =
      twApi_CreatePropertyList("FaultStatus",
      twPrimitive_CreateFromBoolean(properties.FaultStatus), 0);
if (!proplist) {
 TW_LOG(TW_ERROR,"sendPropertyUpdate: Error allocating property
list");
   return;
   }
  twApi_AddPropertyToList(proplist,"InletValve",
          twPrimitive_CreateFromBoolean(properties.InletValve), 0);
twApi_AddPropertyToList(proplist,"Temperature",
          twPrimitive_CreateFromNumber(properties.Temperature), 0);
  twApi_AddPropertyToList(proplist,"TotalFlow",
          twPrimitive_CreateFromNumber(properties.TotalFlow), 0);
  twApi_AddPropertyToList(proplist,"Pressure",
          twPrimitive_CreateFromNumber(properties.Pressure), 0);
  twApi_AddPropertyToList(proplist,"Location",
          twPrimitive_CreateFromLocation(&properties.Location), 0);
  twApi_PushProperties(TW_THING, thingName, proplist, -1, FALSE);
  twApi_DeletePropertyList(proplist);
}
```

Execute a Service

This helper function executes a service on a named entity on the ThingWorx Platform.

```
enum msgCodeEnum twApi_InvokeService(enum entityTypeEnum entityType,
     char * entityName, char * serviceName,
twInfoTable * params, twInfoTable ** result, int32_t timeout,
     char forceConnect)
```

The following table lists and describes the parameters for this helper function:

Parameter	Type	Description
entityType	Input	The type of entity that the service belongs to. Enumeration values can be found in twDefinitions.h.
entityName	Input	The name of the entity that the service belongs to.
serviceName	Input	The name of the service to execute.
params	Input	A pointer to an infotable containing the parameters to be passed in to the service. The calling function will retain ownership of this pointer and is responsible for cleaning up the memory after the call is complete.
result	Input/Ouput	A pointer to a twInfoTable pointer. In a successful request, this parameter will end up with a valid pointer to a twInfoTable that is the result of the service invocation. The caller is responsible for deleting the returned primitive using twInfoTable_Delete. It is possible for the returned pointer be a NULL if an error occurred or no data is returned.
timeout	Input	The time (in milliseconds) to wait for a response from the server. A value of -1 uses the DEFAULT_MESSAGE_TIMEOUT as defined in twDefaultSettings.h.
forceConnect	Input	A Boolean value. If TRUE and the API is in the disconnected state of the duty cycle, the API will force a reconnect to send the request.

Return:

- msgCodeEnum — the result of the call. See twDefinitions.h for the enumeration definition.

Trigger an Event

This helper function triggers a specific event on a named entity on the ThingWorx Platform.

```
enum msgCodeEnum twApi_FireEvent(enum entityTypeEnum entityType,
char * entityName, char * eventName,
twInfoTable * params, int32_t timeout, char forceConnect)
```

The following table lists and describes the parameters for this helper function:

Parameter	Type	Description
`entityType`	Input	The type of entity that the service belongs to. Enumeration values can be found in `twDefinitions.h`.
`entityName`	Input	The name of the entity that the service belongs to.
`eventName`	Input	The name of the event to trigger.
`params`	Input	A pointer to an infotable containing the parameters to be passed to the event. The calling function will retain ownership of this pointer and is responsible for cleaning up the memory after the call is complete.
`timeout`	Input	The time (in milliseconds) to wait for a response from the server. A value of `-1` uses the `DEFAULT_MESSAGE_TIMEOUT` as defined in `twDefaultSettings.h`.
`forceConnect`	Input	A Boolean value. If `TRUE` and the API is in the disconnected state of the duty cycle, the API will force a reconnect to send the request.

Return:

- `msgCodeEnum` — the result of the call. See `src/api/twDefinitions.h` for the enumeration definition.

6

Building an Application

Use the information presented here to build the example applications as well as your own applications. As applicable, you can reuse build files from the examples or modify a build file to support a new platform.

Introduction

The C SDK is a set of ANSI C header and source files that can be easily integrated into any build environment. There is are example applications in the `examples` directory with build files for both a Make-based and Microsoft Visual Studio environments. Both the Visual Studio solution and the Makefile-based build have separate 'projects' for independently building a statically linkable library that can be reused for any other applications. The Visual Studio project for building the library is found in `build/platforms/win32`.

The Makefile structure is designed to aid in porting and cross compilation and should be used as a starting point for non-Windows based ports. There is a generic Makefile used to build the library that is found in the `build` directory. This Makefile in turn includes a platform-specific makefile that contains all the compiler-specific and processor-specific settings for that particular platform.

To build the library, run the following command in the build directory:

```
make PLATFORM=<your platform> BUILD=<debug|release>
```

Where:

- The `PLATFORM` specified must match the name of a subdirectory of the `build/platforms` directory and follow the pattern `compiler-os-processor` (for example, `gcc-linux-x86`). If a `PLATFORM` is not specified, the default is `gcc-linux-x86-32`, which builds a 32-bit library for Linux on the x86 processor using the gcc compiler.
- The `BUILD` specifier lets you create either a debug or release build. The release build optimizes for code size and strips all symbols (and potentially logging) to create a significantly smaller executable, on the order of 5 times smaller than the debug version.

Building Your Applications

In each of the example application build directories there is a file named `CommonSettings` (linux) or `CommonSettings.targets` (win32) that contains the configuration settings for how the SDK should be built for your application. If using any of the preprocessor definitions, incorporate these settings into your project file (win32) or Makefile (linux). The C SDK examples should be used as a template for how to do this. For Windows, your project can inherit the settings that are defined in your overall solution. However, to do this, you must hand edit your `*.vcxproj` file and add the following somewhere within the `<Project>` XML element:

```
<PropertyGroup Condition="'$(SolutionDir)' == '' or
     '$(SolutionDir)' == '*undefined*'">
   <SolutionDir>.\</SolutionDir>
</PropertyGroup>
<Import Project="$(SolutionDir)CommonSettings.targets"
```

```
  Condition="exists('$(SolutionDir)CommonSettings.targets')" />
```

If you are using a Make system, your Makefile should include the following lines:

```
include ./Make.CommonSettings
include $(TW_SDK_ROOT)/build/Make.settings
```

In all cases, it is STRONGLY recommended that you use one of the provided examples as a starting point for your customization.

Supporting New Platforms

If you are using a platform that is different than the provided options, modify a platform-specific Makefile to support your processor and toolchain. The Makefile should be named `Makefile.<compiler-os-processor>` and placed in a sub-directory called `build/platforms/< compiler-os-processor>`. As an example, below is the platform and application specific portion of the Makefile for a native Linux build on a 32-bit X86 platform.

The C SDK is designed for portability and can be ported to most any OS, RTOS, or even simple taskers. In your Makefile you must specify what OS you will be using. You do this by defining `TW_OS_INCLUDE` to point to the required include file for your OS, as shown here:

```
# Set up your compiler and link options here.

TOOLROOT = /usr   # Points to the root directory of the toolchain

TOOL_PREFIX =   # Typically defined if you are cross compiling
                #  e.g. arm-linux-gnueabi-

CCDIR = ${TOOLROOT}/bin

LIBDIR = ${TOOLROOT}/lib

INCDIR = ${TOOLROOT}/include

export LD_LIBRARY_PATH=$(TOOLROOT)/lib/gcc

OS_INCLUDE = "twLinux.h"



CC = ${CCDIR}/$(TOOL_PREFIX)gcc

CC_OPTS = -DTW_OS_INCLUDE='$(OS_INCLUDE)' \

        -Wall -D_GNU_SOURCE -D_FILE_OFFSET_BITS=64 -DCC_GNU \


             -pthread -fPIC -ffunction-sections -fdata-sections
```

```
LINKER          = $(CC)
LIBOPTS         = -pthread -fPIC -Wl,--gc-sections -L${LIBDIR}
STATIC_LIBS     =
AR              = ${CCDIR}/$(TOOL_PREFIX)ar cru
RANLIB          = ${CCDIR}/$(TOOL_PREFIX)ranlib

# OS FILES
OS_INC_DIR   =
OS_SRCS      = $(TW_SDK_ROOT)/src/porting/twLinux.c
OS_DEFS      =
OS_LIBS      =
OS_LIB_DIR   =
```

7

Porting to Another Platform

To port to a platform other than those that the SDK currently supports (with files specifically for the platforms), you'll need the information presented here. Included here is information about defining the OS, TLS support, and the various types of functions (logging, memory management, date/time, synchronization, socket).

Requirements for Platforms

The ThingWorx C SDK is designed for easy porting to even the most basic of platforms. The key requirements for the platform are as follows:

- ANSI C compiler and run time support
- TCP/IP stack
- Dynamic memory allocation (malloc, calloc, free)
- Millisecond granularity timer, preferably with a Real Time Clock
- Some form of Mutual Exclusion capability (Mutex, Critical Section, Spinlock, etc.)
- Tick Timer Interrupt/Callback capability (if using the built-in tasker)
- File System functions if using the File Transfer capability of the SDK
- Threads (optional)

All custom configurations for a platform are typically encapsulated in a single C source and header file pair. For example, the SDK comes with example ports for Windows and Linux (or any POSIX environment). The files are located in the porting directory and are `twWindows.h/twWindows.c` and `twLinux.h/twLinux.c` respectively. It is strongly recommended that you start with one of these files as the basis for your porting efforts. The Linux port will be used as an example in the sections that follow.

For the platforms supported by this release of the ThingWorx C SDK, refer to the *ThingWorx Edge Requirements and Compatibility Matrix*, which is available on the PTC Support site, Reference Documents page for ThingWorx products.

Defining the Chosen OS

When building the library or an application, you must tell the compiler to include the appropriate header file for your port. The definition of which OS to use is done in your platform specific Makefile. In the supplied Makefiles for building on Linux it is:

```
OS_INCLUDE  = "twLinux.h"
```

And then in the compiler options as:

```
DTW_OS_INCLUDE='$(OS_INCLUDE)'
```

The actual inclusion of the appropriate header file based on the above mentioned preprocessor definition is done in the file `porting/twOSPort.h` as follows:

```
#include TW_OS_INCLUDE
```

If you are not using Make as your build environment, you must ensure that the following conditions are met:

1. Define the preprocessor macro `TW_OS_INCLUDE` to point to your platform-specific header file as described below, and
2. Ensure that only the source files listed in `/build/Makefile` are included in your build. This step is important as the C SDK includes several third party open source libraries that may have test applications and extraneous source files that are not part of the C SDK. However, to maintain the integrity of the open source library, those files may be included in the C SDK distribution.

The platform-specific include file mentioned above must define certain entities, which are listed and categorized in the sections that follow.

TLS Support

The C SDK has a pluggable security layer, but it defaults to using the built-in AxTLS library for full TLS 1.1 compliant certificate-based authentication and 128-bit AES encryption. The AxTLS is an extremely lightweight (~50KB) TLS client implementation, but there may be good reasons for using other security/encryption layers, such as HW-based acceleration or a need for a FIPS compliant implementation based on OpenSSL. If you choose to use a different TLS library you should point `TW_TLS_INCLUDE` to the required header file for your implementation. Refer to the TLS Provider Plugins on page 79 section for further information. The `No_TLS` option will result in clear-text communications between your application and the ThingWorx Platform. If you choose to use that setting you must also `#define NO_TLS`.

Note

The `NO_TLS` option is provided as a convenience for development purposes, but is NOT recommended for any production implementations.

```
/********************************/
/*      Which TLS Library?      */
/********************************/
/*
Define which pluggable TLS library is used.  Default is AxTLS.
The NO_TLS option turns off encryption altogether.  This is
useful for debugging but IS NOT RECOMMENDED
FOR PRODUCTION ENVIRONMENTS.  Refer to the documentation
on how to add another TLS library.
*/

#define TW_TLS_INCLUDE "twAxTls.h"
/*#define TW_TLS_INCLUDE "twOpenSSL.h" */
/***
#define TW_TLS_INCLUDE "twNoTls.h"
#define NO_TLS
```

For details, see also the Doxygen documentation provided with the SDK bundle.

Logging Functions

The C SDK has a pluggable logging provider that defaults to simple `printf` statements. The function definition is in the `utils/twLogger.c` file. Your platform/OS specific header file also defines some macros for logging, as shown below.

```
/* Logging */
#ifdef _DEBUG
#ifndef DBG_LOGGING
#define DBG_LOGGING
#endif
#endif
#ifdef DBG_LOGGING
#define TW_LOGGER_BUF_SIZE 4096 /* Max size of log buffer */
#define TW_LOG(level, fmt, ...)  twLog(level, fmt, ##__VA_ARGS__)
#define TW_LOG_HEX(msg, preamble, length)   twLogHexString(msg, preamble, length)
#define TW_LOG_MSG(msg, preamble)   twLogMessage(msg, preamble)
#else
#define TW_LOGGER_BUF_SIZE 1
#define TW_LOG(level, fmt, ...)
#define TW_LOG_HEX(msg, preamble, length)
#define TW_LOG_MSG(msg, preamble)
#endif
```

To minimize the code footprint of a released application, the default for logging is that it is enabled for debug builds and entirely disabled for release builds. Both the logging functions and buffer size need to be defined if logging is enabled. The macros `TW_LOG_HEX` and `TW_LOG_MSG` are used to display the hex bytes moving over the wire and the actual message content, respectively. These functions tend to have a serious impact on performance and are not recommended for use in a released system.

The logging system also provides a convenient way for you to define you own logging function without changing these macros. This function is

```
int twLogger_SetFunction(log_function f);
```

For details about this function, refer to the Doxygen documentation provided with the SDK bundle.

Memory Management Functions

The SDK uses dynamic memory allocation and de-allocation. In all but the most basic of platforms, this means the use of the standard C malloc, calloc, and free functions. The SDK does not use realloc itself, but any underlying TLS library may. To create an abstraction layer, the SDK uses `#defines` to give you the flexibility of creating your own implementations of these functions. These definitions, which are required, and their most basic implementations are as follows:

```
#define TW_MALLOC(a) malloc(a)
#define TW_CALLOC(a, b) calloc(a,b)
#define TW_REALLOC(a, b) realloc(a, b)
#define TW_FREE(a) free(a)
```

Date/Time Functions

The SDK requires a timer with millisecond granularity for things such as messaging timeouts and task scheduling. In addition, some form of real-time clock may be required if using DATETIME base types or the standard logging plugin. The DATETIME base type uses the standard javascript representation of milliseconds since the epoch of midnight 1/1/1970. In the Linux environment this is represented as an unsigned 64-bit integer with a direct correlation to the number of milliseconds, but the SDK makes no requirement that a DATETIME must be a simple element.

```
/* Time */
typedef uint64_t DATETIME;  /* AS DEFINED IN LINUX.H */
```

To support potentially complex DATETIME structures, a port of the SDK must provide a few DATETIME manipulation and comparison functions. The function definitions are in the file, twOSPort.h, but the implementations are typically in your OS-specific C file, or in the file, twLinux.c for a Linux port. The required functions are listed and described in the table that follows. For the signature and parameter definitions for the functions, refer to the Doxygen documentation provided with the SDK bundle.

To	Use this function
Compare two DATETIME entities and returns a value of TRUE if t1 > t2 or FALSE if not.	twTimeGreaterThan
Compare two DATETIME entities and returns a TRUE value if t1 < t2 or FALSE if not.	twTimeLessThan
Add a number (msec) of milliseconds to the value in t1.	twAddMilliseconds
Get the current millisecond count since the system started (or since the epoch if the system time has millisecond granularity). On systems where the real-time clock has a millisecond granularity, it is recommended that this value be the same as the current system time, representing the current date/time.	twGetSystemMillisecondCount
Get the current system time, representing milliseconds since the epoch. If utc is TRUE (the default for the SDK), the time is corrected to	twGetSystemTime

To	Use this function
Universal Coordinate Time (UTC).	
Get the current system time and converts it to a string using `strftime` formatting.	twGetSystemTimeString
Convert a `DATETIME` to a string using `strftime` formatting.	twGetTimeString
Delay execution. In a single-threaded, single-processor system, this may be a blocking call.	twSleepMsec

Synchronization Functions

The SDK may run in a multithreaded or multitasking environment. Therefore, it is important to protect access to certain data structures. The functions described in the following table provide such access protection. While they may be stubbed out in a single-tasking environment, it is highly recommended that these functions be fully implemented with whatever facility your OS provides. Note that functions using the `TW_MUTEX typedef` assume that this will be a pointer to whatever structure or synchronization mechanism you wish to use.

To	Use this function
Create a synchronization entity.	twMutex_Create
Delete a synchronization entity and free up its memory.	twMutex_Delete
Lock the synchronization entity.	twMutex_Lock
Unlock the synchronization entity.	twMutex_Unlock

For more information about these functions, refer to the Doxygen documentation provided with the SDK bundle.

Socket Functions

The C SDK does not include a TCP/IP stack. Rather, it assumes that the underlying platform provides that functionality. To that end, the SDK has defined a series of wrapper functions to mask the underlying native socket functions. The function definitions use an underlying `twSocket` structure that abstracts away some of the differences in how certain platforms deal with socket descriptors – for example, Linux uses an `int` while Windows uses a `HANDLE`. The structure is defined in the file, `src/porting/twOSPort.h`, as follows:

```
typedef struct twSocket {
        TW_SOCKET_TYPE sock;  /* socket descriptor  */
        TW_ADDR_INFO addr; /* address to use */
        TW_ADDR_INFO * addrInfo; /* Addr Info struct head - use to free */
        char state;
} twSocket;
```

The actual definition of and TW_ADDR_INFO and the implementation of the functions above should be done in your platform-specific C file. The following table lists and describes the socket functions that must be provided by a port. For signatures, parameter details, and return information, refer to the Doxygen documentation provided with the SDK.

To	Use this function
Allocate and initialize a socket structure.	twSocket_Create
Establish a connection to the specified host/port pair.	twSocket_Connect
Re-establish a connection to the specified host/port pair. The underlying socket will be torn down and recreated, but all other twSocket parameters should remain intact.	twSocket_Reconnect
Close a previously opened connection.	twSocket_Close
Check to see if data is available on a socket. Use this function to prevent a twSocket_Read call from blocking permanently if no data is available. This function is especially important if using the built-in tasker, which cannot have tasks that block.	twSocket_WaitFor
Read data from a socket.	twSocket_Read
Write data to a socket.	twSocket_Write
Delete a twSocket structure. This function should close the socket if it is still open before deleting the structure.	twSocket_Delete
Get the error code of the last error that occurred while using a socket. Note that this is typically a system-wide call and not a call to a specific socket.	twSocket_GetLastError

Tasker Functions

The C SDK has a simple built-in tasker that can be used in conjunction with or in place of an underlying OS. The key requirement for the underlying architecture is to provide some sort of tick-timer that allows the execution of what could be a relatively long running callback function at one millisecond intervals. The callback function is twTaskerStart. This function initializes the tasker by setting up a mechanism to call the tickTimerCallback function every millisecond. This function call is blocking, so it is best to use some separate thread of execution, or at least re-enable priority interrupts before making this call. This function is called only once when a process using the API starts.

To shut down the tickTimerCallback mechansim, use the twTaskerStop function. Call this function only once when a process using the API ends.

For signatures, parameter details, and return information for these functions, refer to the Doxygen documentation provided with the SDK.

File System Functions

To use the file transfer or directory browsing capability of the C SDK, implement the functions listed in the following table. For signatures, parameter details, and return information, refer to the Doxygen documentation provided with the SDK.

To	Use this function
Retrieve information about a directory entry (file or subdirectory).	twDirectory_GetFileInfo
Check if a directory entry (file or directory) exists.	twDirectory_FileExists
Create a file.	twDirectory_CreateFile
Move a file.	twDirectory_MoveFile
Delete a file.	twDirectory_DeleteFile
Create a directory.	twDirectory_CreateDirectory
Delete the specified directory (and all its contents).	twDirectory_DeleteDirectory
Iterate through a directory, retrieving the information of the next file or subdirectory.	twDirectory_IterateEntries
Retrieve the last error that occurred as a result of a file system activity.	twDirectory_GetLastError

Native Threads

With the built-in tasker, the C SDK has does not depend on a threading OS. However, if one is present, there are advantages to using native threads. Therefore, the C SDK provides a wrapper layer around native threads that maps tasks as defined for the built-in tasker to native threads. Porting the wrapper to a native threading model is straightforward and requires the implementation of only a few functions. These functions are defined in the file `src/porting/twThreads.h`.

The `twThread` structure follows:

```
typedef struct twThread {
     TW_THREAD_ID id;
     twTaskFunction func;
     uint32_t rate;
     char isRunning;
     char isPaused;
     char shutdownRequested;
     char hasStopped;
     void * opaquePtr;
}twThread;
```

The following table explains which function to use to perform an operation. For details, refer to the Doxygen documentation provided with the SDK.

To	Use this function
Create a new thread and optionally start it.	twThread_Create
Stop a thread and free up the thread structure memory.	twThread_Delete
Start a thread.	twThread_Start
Stop a thread and optionally specify a number of milliseconds to wait for the thread to exit before forcefully killingit.	twThread_Stop
Pause the execution of a thread.	twThread_Pause
Resume the execution of a thread.	twThread_Resume
Check if a specified thread is running.	twThread_IsRunning
Check if a specified thread is paused.	twThread_IsPaused
Check if a specified thread is stopped.	twThread_IsStopped

8

TLS Provider Plugins

This section discusses how to implement TLS Provider Plugins.

TLS Implementation (AxTLS)

The ThingWorx C SDK has a built-in TLS implementation which is based on the open source AxTLS library. This library is a lightweight portable implementation of TLS and is sufficient for most implementations. However, if working with a platform that already has another TLS implementation, such as OpenSSL, or has built-in hardware acceleration, it may be desirable to use that functionality instead of AxTLS. To that end, the SDK has been designed with wrapper functions that closely follow the OpenSSL API in order to make it easy to plug in your own TLS implementation. Selection of which TLS implementation to use is done in the main `src/config/YourPlatformName.h` file as follows:

```
/********************************/
/*      Which TLS Library?      */
/********************************/
/*
Define which pluggable TLS library is used.  Default is AxTLS.
The NO_TLS option turns off encryption altogether.  This is
useful for debugging but IS NOT RECOMMENDED FOR PRODUCTION
ENVIRONMENTS.  Refer to the documentation on how to add
another TLS library.
*/
#define TW_TLS_INCLUDE "twAxTls.h"
/* #define TW_TLS_INCLUDE "twOpenSSL.h" */
```

If using your own TLS implementation, add your own definition and point `TW_TLS_INCLUDE` to your header file.

The functions defined in `twTLS.h` can be used for any TLS connections that your application needs to make. These functions are the abstracted interface that sit on top of the underlying TLS implementation.

Consistent with both the OpenSSL and AxTLS APIs, the SDK uses a structure for an SSL context that manages all the SSL sessions, and a structure for an SSL session itself. In addition, the APIs expose several functions for operations. The definitions and functions are exposed with preprocessor definitions. For these details, refer to the Doxygen documentation provided with the SDK.

Item	Description
`TW_SSL_CTX`	The SSL context structure as defined by the implementation. Mapped directly to `SSL_CTX` in AxTLS.
`TW_SSL`	The SSL session structure as defined by the implementation. Mapped directly to SSL in AxTLS.
`TW_SSL_SESSION_ID_SIZE`	The SSL session structure as defined by the implementation. Mapped directly to SSL in AxTLS.
`TW_SSL_SESSION_ID_SIZE`	The size of an SSL session ID as defined by the implementation. This ID is used for session resumption. Mapped directly to `SSL_SESSION_ID_SIZE` in AxTLS.
`TW_GET_CERT_SIZE`	Returns the maximum number of certificates allowed by the implementation. Mapped directly to `ssl_get_config(SSL_MAX_CERT_CFG_OFFSET)` in AxTLS.
`TW_GET_CA_CERT_SIZE`	Returns the maximum number of CA certificates allowed by the implementation. Mapped directly to `ssl_get_config(SSL_MAX_CA_CERT_CFG_OFFSET)` in AxTLSL.

Item	Description
TW_NEW_SSL_CTX	Creates and initializes new instance of an SSL_CTX. Maps directly to ssl_ctx_new in AxTLS.
TW_NEW_SSL_CLIENT(a,b,c,d)	Creates and initializes a new instance of an SSL structure within the provided SSL_CTX. Parameters: • a — pointer to a TW_SSL_CTX structure. • b — a TW_SOCKET_TYPE value that is the descriptor of the socket to be used. The underlying socket should not be opened before calling this function. • c — session id. The session ID if session resumption is being used. The SDK does not use session resumption and sets this to NULL. • d — size of the session ID that was passed in. Maps directly to ssl_client_new(a,((twSocket *)b)->sock,c,d) in AxTLS.
TW_HANDSHAKE_SUCCEEDED	Returns a Boolean (char) value, TRUE if the SSL handshake succeeded and data can be securely exchanged, FALSE if otherwise. Maps to ((ssl_handshake_status(ssl)) == SSL_OK) in AxTLS.
TW_SSL_FREE(a)	Close any socket and free up any memory associated with an SSL session. Parameters: • a — pointer to the TW_SSL structure to free. Maps directly to ssl_free(a) in AxTLS.
TW_SSL_CTX_FREE(a)	Free up any memory associated with an SSL context. Parameters: • a — pointer to the TW_SSL_CTX structure to free. Maps directly to ssl_ctx_free(a) in AxTLS.
TW_SSL_WRITE(a,b,c)	Writes data out the secure connection. Parametes: • a — pointer to the TW_SSL structure to write to. • b — pointer to the buffer containing the data to write. • c — the amount of data to write. This result of this macro should contain the number of bytes sent, or a negative number if an error occurred. Maps to ssl_write in AxTLS.
TW_SSL_READ(a, b, c, d)	Reads data from the secure connection. Parameters: • a — pointer to the TW_SSL structure to read from. • b — pointer to the buffer that the data should be placed in.

Item	Description
	• c — the amount of data to read. • d — the number of milliseconds to wait while trying to read the desired amount of data. This result of this macro should contain the number of bytes read, or a negative number if an error occurred. Maps to `ssl_read` in AxTLS.
`TW_USE_CERT_FILE(a,b,c)`	Loads an X509 certificate in PEM or DER format from the file specified. Parameters: • a — pointer to the TW_SSL_CTX structure load the certificate into. • b — name of the file containing the certificate. • c — a password to access the certificate (if required). Maps to `ssl_obj_load(a, SSL_OBJ_X509_CERT, b, NULL)` in AxTLS.
`TW_USE_KEY_FILE(a,b,c,d)`	Loads an encrypted key in PEM or DER format from the file specified. Parameters: • a — pointer to the `TW_SSL_CTX` structure to read from • b — name of the file containing the key • c — the type of key • d — a password to access the key. Maps to `ssl_obj_load(a, SSL_OBJ_RSA_KEY, b, d)` in AxTLS.
`TW_USE_CERT_CHAIN_FILE(a,b,c)`	Loads a certificate chain in PEM or DER format from the file specified. Parameters: • a — pointer to the `TW_SSL_CTX` structure load the certificate into. • b — name of the file containing the certificate chain. • c — a password to access the certificate (if required). Maps to `ssl_obj_load(a, SSL_OBJ_X509_CERT, b, NULL)` in AxTLS.
`TW_SET_CLIENT_CA_LIST(a,b)`	Sets the list of supported CAs from the file specified. Parameters: • a — pointer to the `TW_SSL_CTX` structure load the certificate into. • b — pointer to the CA list. Maps to ssl_obj_load(a, SSL_OBJ_X509_CERT, (const char *)b, NULL)) in AxTLS.
`TW_VALIDATE_CERT(TW_SSL * ssl, char selfSignedOk)`	Inline function that validates the received certificate. Parameters:

Item	Description
	• ssl — pointer to the TW_SSL structure that has received the certificate • selfSignedOk — boolean, set to TRUE if self-signed certificates are allowed, FALSE if not. Default is FALSE. Returns zero if the certificate is valid, non-zero if not.
TW_ENABLE_FIPS_MODE(a)	Enables FIPS mode. Parameters: • a – pointer to the TW_SSL_CTX structure Returns zero if successful or an error code if FIPS is supported but enabling failed or TW_FIPS_MODE_NOT_SUPPORTED if the TLS layer does not support FIPS
TW_GET_X509_FIELD(TW_SSL * ssl, char field)	Inline function that gets the value of a field in the certificate. Parameters: • ssl — pointer to the TW_SSL structure that has received the certificate • field – char, the field to retrieve. Fields supported must be SUBJECT_CN, SUBJECT_O, SUBJECT_OU, ISSUER CN, ISSUER_O, ISSUER_OU Returns the value of the field, or NULL if the field is not found.

Note

As of release 1.2 of the C SDK, the default setting for DEFAULT_SOCKET_READ_TIMEOUT in twDefaultSettings.h is 500 ms. If you are using AxTLS and a web socket read times out in the middle of reading a record, the SSL state is lost. As a result, the SDK tries to start read the record header again, and the operation fails. To detect this situation, check the log for the SDK for the error, twTlsClient_Read: Timed out after X milliseconds, and consider increasing the value of the DEFAULT_SOCKET_READ_TIMEOUT. You can change the setting at runtime by modifying the value of twcfg.socket_read_timeout.

Supported Ciphers

The C SDK supports the following cipher suites by default:

- TLS_RSA_WITH_AES_256_CBC_SHA
- TLS_RSA_WITH_AES_128_CBC_SHA

- TLS_RSA_WITH_RC4_128_SHA
- TLS_RSA_WITH_RC4_128_MD5

When you set up a ThingWorx server make sure you enable TLS v1.1 and disable SSL in your `server.xml` file.

A

Error Codes

This appendix lists and categorizes the error messages (and their codes) that are returned by the C SDK. You can find all of the definitions for these messages in the `twErrors.h` file, located in the `api` subdirectory of the SDK installation.

The types of messages include:

Note

The error codes are presented in the same order as they appear in `twErrors.h`. The 5xx series is for errors for the List component. None are defined for this component at this time, so the sequence jumps from the 4xx to 6xx series.

General Errors

The following table lists general errors and their corresponding codes:

Code	Message	Troubleshooting
100	`TW_UNKNOWN_ERROR`	An error occurred, but it was not recognized by the SDK. You should not see this error
101	`TW_INVALID_PARAM`	The parameter value is not allowed. Typically indicative of a NULL pointer being passed in where a NULL pointer is not allowed.
102	`TW_ERROR_ALLOCATING_MEMORY`	The specified amount of memory could not be allocated. Make sure that components free memory when they exit. Make sure you free up memory when finished using data structures. This error is very serious, and your application will usually terminate soon after.
103	`TW_ERROR_CREATING_MTX`	An error occurred while creating a mutex.

Websocket Errors

A Websocket connection is run using a system socket; a system socket sits one layer lower in the networking stack. All Websocket errors indicate some general issue communicating with the ThingWorx Platform. The following table lists websocket errors, their corresponding codes, and an explanation of the issue.

Note

As of release 1.2 of the C SDK, the default setting for `DEFAULT_SOCKET_READ_TIMEOUT` in `twDefaultSettings.h` is 500 ms. If you are using AxTLS and a web socket read times out in the middle of reading a record, the SSL state is lost. As a result, the SDK tries to start read the record header again, and the operation fails. To detect this situation, check the log for the SDK for the error, `twTlsClient_Read: Timed out after X milliseconds`, and consider increasing the value of the `DEFAULT_SOCKET_READ_TIMEOUT`. You can change the setting at runtime by modifying the value of `twcfg.socket_read_timeout`.

Code	Message	Troubleshooting
200	`TW_UNKNOWN_WEBSOCKET_ERROR`	An unknown error occurred on the websocket. You should not see this error.
201	`TW_ERROR_INITIALIZING_WEBSOCKET`	An error occurred while initializing the websocket. Check your websocket configuration parameters for validity.
202	`TW_TIMEOUT_INITIALIZING_WEBSOCKET`	A timeout occurred while initializing the websocket. Check the status of the connection to the ThingWorx Platform.
203	`TW_WEBSOCKET_NOT_CONNECTED`	The websocket is not connected to the ThingWorx Platform. The requested operation cannot be performed.
204	`TW_ERROR_PARSING_WEBSOCKET_DATA`	An error occurred while parsing websocket data. The parser could not break down the data from the websocket.
205	`TW_ERROR_READING_FROM_WEBSOCKET`	An error occurred while reading data from the websocket. Retry the read operation. If necessary, resend the data.
206	`TW_WEBSOCKET_FRAME_TOO_LARGE`	The SDK is attempting to send a websocket frame that is too large. The Maximum Frame Size is set when calling `twAPI_`

Code	Message	Troubleshooting
		Initialize and should always be set to the Message Chunk Size (twcfg.message_chunk_size).
207	TW_INVALID_WEBSOCKET_FRAME_TYPE	The type of the frame coming in over the websocket is invalid.
208	TW_WEBSOCKET_MSG_TOO_LARGE	The application is attempting to send a message that has been broken up in to chunks that are too large to fit in a frame. You should not see this error.
209	TW_ERROR_WRITING_TO_WEBSOCKET	An error occurred while writing to the Web socket.
210	TW_INVALID_ACCEPT_KEY	The Accept key sent earlier from the ThingWorx Platform is not valid.

Messaging Errors

The following table lists the error codes and messages for Messaging errors and provides some troubleshooting information.

Code	Message	Troubleshooting
300	TW_NULL_OR_INVALID_MSG_HANDLER	The message handler singleton has not been initialized.
301	TW_INVALID_CALLBACK_STRUCT	The callback structure was not valid. Check that your application properly implements the callback.
302	TW_ERROR_CALLBACK_NOT_FOUND	The specified callback was not found. Check the callback parameters passed to the function.
303	TW_INVALID_MSG_CODE	An attempt to set an invalid message code was made. Valid message codes are defined in twDefinitions.h. You should not see this internal error in your code.
304	TW_INVALID_MSG_TYPE	A function was called with

Code	Message	Troubleshooting
		an invalid message code. Valid message codes are defined in twDefinitions.h. You should not see this internal error.
305	TW_ERROR_SENDING_MSG	An error occurred while sending the message. Check the network connections and the destination host. If network connections and the destination host are working properly, check the configuration of the destination host to be sure it is correct.
306	TW_ERROR_WRITING_OFFLINE_MSG_STORE	An error occurred while writing to the offline message store.
307	TW_ERROR_MESSAGE_TOO_LARGE	The message was too large. Check that the size you configured for messages is adequate for all expected traffic. Consider increasing the size.
308	TW_WROTE_TO_OFFLINE_MSG_STORE	The message was not sent to the ThingWorx Platform, but was stored in the offline message store. The message will be delivered next time the websocket is connected.
309	TW_INVALID_MSG_STORE_DIR	The directory for the message store was not correct. Make sure the path is valid and that you have write permission.
310	TW_MSG_STORE_FILE_NOT_EMPTY	The on-disk file that is uses to store offline messages contains some messages that have not been sent yet. The file name cannot be changed.

Primitive and InfoTable Errors

The following table lists the errors related to the data structures, twPrimitive and twInfoTable, and their supporting functions. It also provides suggestions for troubleshooting. For more information about these data structures, refer to twPrimitiveStructure on page 50 and twInfoTable on page 52.

Note

When creating an InfoTable, keep in mind that the twInfoTableRow structure must contain the field values of the datashape **in the same order** as in the datashape.

Code	Message	Troubleshooting
400	TW_ERROR_ADDING_DATASHAPE_ENTRY	An error occurred while attempting to add an entry (field) to the DataShape.
401	TW_INDEX_NOT_FOUND	Attempted to access a non-existent field from a row in an InfoTable. The index value must be less than the number of fields defined in the DataShape.
402	TW_ERROR_GETTING_PRIMITIVE	The function twInfoTable_GetPrimitive failed to retrieve the requested primitive from the InfoTable.
403	TW_INVALID_BASE_TYPE	The specified base type is not valid. Check the spelling in your code, or select a different base type. For a table of the available base types, refer to Base Types on page 50

List Errors

The following table lists the error related to lists (for example, subscribed properties):

Code	Message	Troubleshooting
500	TW_LIST_ENTRY_NOT_FOUND	The entry was not found in the list. For example, the requested property was not found in the list of subscribed properties.

API Errors

The following table lists the errors related to the API:

Code	Message	Troubleshooting
600	TW_NULL_OR_INVALID_API_SINGLETON	The API singleton is either null or invalid. This error occurs if the API was not initialized properly. Check the parameters that you are passing to the initialize function. Check the log.
601	TW_ERROR_SENDING_RESP	An error occurred while sending a response message to the ThingWorx Platform.
602	TW_INVALID_MSG_BODY	A message was received from the ThingWorx Platform that had an invalid or malformed message body.
603	TW_INVALID_MSG_PARAMS	A Property PUT was received from the ThingWorx Platform with an empty parameters InfoTable. The property value will not be changed.
604	TW_INVALID_RESP_MSG	The response message was not valid. You should not see this internal error.
605	TW_NULL_API_SINGLETON	The API singleton was null. This message indicates that the API was not initialized properly. Check the parameters that you are passing to the initialize function. Check the log.
606	TW_ERROR_CREATING_MSG	An error occurred while creating the message. This error typically indicates an out-of-memory condition.
607	TW_ERROR_INITIALIZING_API	An error occurred while initializing the API. Check the parameters that you are passing to the initialize function. Check the log.

Tasker Errors

The following table lists the errors related to the Tasker:

Code	Message	Troubleshooting
700	TW_MAX_TASKS_EXCEEDED	You have attempted to create more tasks than are allowed for the built-in tasker. The maximum number of tasks allowed is set at compile time with the constant TW_MAX_TASKS which is defined in twDefinitions.h. If you have many tasks running you may wish to consider using native threads if your platform supports them.
701	TW_TASK_NOT_FOUND	The specified task ID was not found. Make sure the task ID passed to this function is correct. The task ID is returned from the function call twTasker_CreateTask.

Logging Errors

The following table lists the error related to logging:

Code	Message	Troubleshooting
800	TW_NULL_OR_INVALID_LOGGER_SINGLETON	The logger singleton was not initialized properly. This error indicates a memory allocation error. Check your TW_LOGGER_BUF_SIZE setting in your platform-specific header file in the src/porting directory.

Utils Errors

The SDK uses Base64 encoding/decoding. The following table lists the related errors. At this time, the code does not use them.

Code	Message
900	TW_BASE64_ENCODE_OVERRUN
901	TW_BASE64_DECODE_OVERRUN

System Socket Errors

System Sockets are Operating System-provided networking APIs. The TW_ERROR_WRITING_TO_SOCKET error in the System Socket category is a general socket write error. All errors in this category are in the context of a connection to ThingWorx Platform.

As appropriate, first check the network connection between the Thing where your application is running and the ThingWorx Platform to resolve the problem. If a proxy server is used between your Thing and the Platform, check that the proxy server is operating properly. If so, check the configuration for the connection to the proxy server.

Code	Message	Troubleshooting
1000	TW_ERROR_WRITING_TO_SOCKET	General socket write error encountered while writing to the ThingWorx Platform.
1001	TW_SOCKET_INIT_ERROR	An error occurred while initializing the socket. The network connection may have dropped.
1002	TW_INVALID_SSL_CERT	The SSL certificate provided by the server was not valid or was self-signed. Check your certificate settings.
1003	TW_SOCKET_NOT_FOUND	The socket was not found. The network connection may have dropped.
1004	TW_HOST_NOT_FOUND	The specified ThingWorx Platform was not found. Check network connections and make sure that your application configuration specifies a valid host address.
1005	TW_ERROR_CREATING_SSL_CTX	An error occurred creating the SSL context.
1006	TW_ERROR_CONNECTING_TO_PROXY	An error occurred connecting to the specified proxy server. Make sure the proxy server address is correctly specified. Check network connections.
1007	TW_TIMEOUT_READING_FROM_SOCKET	An attempt to read from a socket timed out with no data available.

Code	Message	Troubleshooting
1008	`TW_ERROR_READING_RESPONSE`	An error occurred while reading the response from the proxy server. Check your proxy configuration in your application.
1009	`TW_INVALID_PROXY_CREDENTIALS`	The credentials presented to the proxy server were not valid. Check with the administrator for the proxy server and re-enter the credentials for the proxy server. NOTE: While the connection to the proxy server is not encrypted, the credentials are obfuscated using standard HTTP Basic, Digest, or NTLM encoding.
1010	`TW_UNSUPPORTED_PROXY_AUTH_TYPE`	The specified authentication type for the proxy server is not supported. Make sure that the authentication type is correctly specified in your application.
1011	`TW_ENABLE_FIPS_MODE_FAILED`	FIPS Mode could not be enabled. Ensure that you are using an OpenSSL library with FIPS validated cryptographic algorithms.
1012	`TW_FIPS_MODE_NOT_SUPPORTED`	FIPS Mode is not supported. Ensure that you are using an OpenSSL library with FIPS validated cryptographic algorithms.

Message Code Errors

The message code errors can be returned when the SDK makes a request to the ThingWorx Platform. They can also be the return values for property/service requests executed by the application using the SDK. For example, if the server queried the SDK application for the property ‘temperature’, but the application did not have that property, it could return TW_NOT_FOUND. The server could also return the same code if the application asked the server for a property that it did not have defined.

Most of these are standard HTTP error codes. You can see more information about them at http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html.

Code	Message	Troubleshooting
1100	TW_BAD_REQUEST	The HTTP request contained syntax errors, so the server did not understand it. Modify the request before attempting it again..
1101	TW_UNAUTHORIZED	The request requires authentication. This error results from a failed login attempt — whether from credentials that were not valid or from the request being sent before authentication occurred.
1102	TW_ERROR_BAD_OPTION	An option or a parameter for a function has a value that is not valid or is not spelled correctly (and so is not recognized).
1103	TW_FORBIDDEN	The ThingWorx Platform is denying you access to the requested resource. Check your permission settings on the Platform.
1104	TW_NOT_FOUND	This message is returned for anything that was not found — a property, a service, a thing, a datashape, and so on.
1105	TW_METHOD_NOT_ALLOWED	The specified method is not allowed. Check the spelling and syntax of your code.
1106	TW_NOT_ACCEPTABLE	Not acceptable.
1107	TW_PRECONDITION_FAILED	The precondition for the operation was not met.
1108	TW_ENTITY_TOO_LARGE	This error occurs if you attempt to send a Property, or

Code	Message	Troubleshooting
		Service or Event parameters that are too large for the ThingWorx Platform to handle.
1109	`TW_UNSUPPORTED_CONTENT_FORMAT`	This error occurs if you attempt to send a Property, or Service or Event parameter that has the wrong baseType as defined on the ThingWorx Platform.
1110	`TW_INTERNAL_SERVER_ERROR`	An error occurred on the ThingWorx Platform while processing this request.
1111	`TW_NOT_IMPLEMENTED`	The ThingWorx Platform may return this error if you attempt a function that is not implemented.
1112	`TW_BAD_GATEWAY`	A gateway could be bad if it cannot communicate to the next component in the chain.
1113	`TW_SERVICE_UNAVAILABLE`	The requested service is not defined. You could also use the `TW_NOT_FOUND` error code, but this one is more specific.
1114	`TW_GATEWAY_TIMEOUT`	If the application sends a request to the ThingWorx Platform and does not get a response within some amount of time, the service call results in this error. The amount of time is configurable.

Subscribed Properties Errors

The following table lists the errors related to subscribed properties:

Code	Message	Troubleshooting
1200	TW_SUBSCRIBEDPROP_MGR_NOT_INTIALIZED	The Subscribed Properties Manager is initialized by twApi_Initialize automatically. For this error to occur, it is most likely that other, more serious errors have occurred. Investigate the other errors first.
1201	TW_SUBSCRIBED_PROPERTY_NOT_FOUND	The requested subscribed property was not found.

File Transfer Errors

The following table lists the errors for the File Transfer component:

Code	Message	Troubleshooting
1300	TW_FILE_XFER_MANAGER_NOT_INITIALIZED	The File Transfer Manager has not been initialized. The File Transfer Manager is initialized when twApi_Initialize is called only if ENABLE_FILE_XFER is defined. If you wish to use file transfer functionality make sure ENABLE_FILE_XFER is defined.
1301	TW_ERROR_CREATING_STAGING_DIR	An error occurred while creating the staging directory. The error happens if there is an invalid path or if you do not have the proper permissions to create the directory specified.

Code	Message	Troubleshooting
1302	TW_FILE_NOT_FOUND	The specified file for the transfer was not found. Check the name of the file specified. If it is correct, check for the presence of the file in the file system at the specified location.
1303	FILE_TRANSFER_FAILED	The file transfer operation failed. The network connection may have dropped during the transfer, the destination for the transfer may be unavailable (down for maintenance or power outage), or the MD5 checksum of the file indicated invalid file content.

Tunneling Errors

The following table lists the errors related to the Tunneling Manager

Code	Message	Troubleshooting
1400	TW_TUNNEL_MANAGER_NOT_INITIALIZED	The Tunnel Manager has not been initialized. The Tunnel Manager is initialized when twApi_Initialize is called only if ENABLE_TUNNELING is defined. If you wish to use tunneling functionality make sure ENABLE_TUNNELING is defined.
1401	TW_TUNNEL_CREATION_FAILED	The tunnel was not created. This error could be because of an out-of-memory condition.

B

Callback Function Return Codes

The following table contains the acceptable return codes (msgCodeEnum) for all Property and Service callback functions. These codes are defined in src/api/twDefinitions.h. The callback functions are invoked as a result incoming requests from the ThingWorx Platform. The property and service callback function signatures are defined in src/api/twApi.h.

Return Code	Returned When
HTTP Client Error Status Codes	
TWX_SUCCESS = 0x40	0x40 (2.00) Success. The request completes successfully.
TWX_BAD_REQUEST = 0x80	0x80 (4.00) Bad request. The HTTP request contains syntax errors, so the server cannot understand it. Modify the request before attempting it again.
TWX_UNAUTHORIZED	0x81 (4.01) Unauthorized. The request requires authentication. This error results from a failed login attempt — whether from credentials that were not valid or from the request being sent before authentication occurred.
TWX_BAD_OPTION	0x82 (4.02) Bad option. An option or a parameter for a function has a value that is not valid or is not spelled correctly (and so is not recognized).

Return Code	Returned When
`TWX_FORBIDDEN`	0x83 (4.03) Forbidden. The ThingWorx Platform is denying you access to the requested resource. Check your permission settings on the Platform.
`TWX_NOT_FOUND`	0x84 (4.04) Not found. Anything is not found — a property, a service, a thing, a datashape, and so on.
`TWX_METHOD_NOT_ALLOWED`	0x85 (4.05) Method not allowed. The specified method is not allowed. Check the spelling and syntax of your code.
`TWX_NOT_ACCEPTABLE`	0x86 (4.06) Not acceptable.
`TWX_PRECONDITION_FAILED = 0x8C`	0x8C (4.12) Precondition failed. The precondition for the operation is not met.
`TWX_ENTITY_TOO_LARGE`	0x8D (4.13) Entity too large. An attempt is made to send a Property, or Service or Event parameter that is too large for the ThingWorx Platform to handle.
`TWX_UNSUPPORTED_CONTENT_FORMAT = 0x8F`	0x8F (4.15) Unsupported content format. An attempt is made to send a Property, or Service or Event parameter that has the wrong baseType as defined on the ThingWorx Platform.
HTTP Server Error Status Codes	
`TWX_INTERNAL_SERVER_ERROR = 0xA0`	0xA0 (5.00) Internal server error. An error occurs on the ThingWorx Platform while processing this request.
`TWX_NOT_IMPLEMENTED`	0xA1 (5.01) Not implemented. The ThingWorx Platform may return this error if you attempt a function that is not implemented.
`TWX_BAD_GATEWAY`	0xA2 (5.02) Bad gateway. A gateway could be bad if it cannot communicate to the next component in the chain.
`TWX_SERVICE_UNAVAILABLE`	0xA3 (5.03) Service unavailable. The requested service is not defined. You could also use the `TW_NOT_FOUND` error code, but this one is more specific.

Return Code	Returned When
TWX_GATEWAY_TIMEOUT	0xA4 (5.04) Gateway timeout. If the application sends a request to the ThingWorx Platform and does not get a response within some amount of time, the service call results in this error. The amount of time is configurable.
TWX_WROTE_TO_OFFLINE_MSG_STORE	Wrote to offline message store. The message is not sent to the ThingWorx Platform, but instead is stored in the offline message store. The message will be delivered next time the websocket is connected.