CrossWorks IoT Library

Version: 3.0

Contents

CrossWorks IoT Library

CrossWorks IoT Library

About the CrossWorks IoT Library

The *CrossWorks IoT Library* is designed to reduce the development time for customers wishing to send data to an IoT service. This library uses the facilities of other CrossWorks Technology Libraries:

- *CrossWorks Tools Library*: provides add-ons for CTL such as read-write locks and ring buffers.
- *CrossWorks Device Library*: provides drivers for common digital sensors, such as accelerometers, gyroscopes, magnetometers, and so on.
- *CrossWorks TCP/IP Library*: provides TCP/IP networking for integrated and external network controllers on memory-constrained microcontrollers.

Architecture

The *CrossWorks IoT Library* is one part of the *CrossWorks Technology Libraries*. Many of the low-level functions provided by the target library are built using features of the *CrossWorks Tasking Library* for multi-threaded operation.

Delivery format

The *CrossWorks IoT Library* is delivered in source form.

Feedback

This facility is a work in progress and may undergo rapid change. If you have comments, observations, suggestions, or problems, please feel free to air them on the **CrossWorks Target and Platform API** discussion forum.

CrossWorks IoT Library

License

The following terms apply to the Rowley Associates IoT Library.

Introduction

About the CrossWorks IoT Library

The *CrossWorks IoT Library* is a standard API that runs on a collection of popular microprocessors and evaluation boards. Using the library, you can push data to IoT services over a network connection.

The IoT Library requires the *CrossWorks Tasking Library* for operation. Because the IoT Library, and facilities built on top of it, use interrupts and background processing, we made the decision to use the CrossWorks Tasking Library as a foundation stone for all CrossWorks Technology Libraries. We have not abstracted the IoT Library to use a generic RTOS as this adds more complexity to the design.

Why use the IoT Library?

The IoT Library is designed to work well on memory-constrained microcontrollers that take measurements or record data and, in the background, push that data to the IoT service.

We intend to support a number of IoT services, but we're launching this library with support for Xively.

What the IoT Library isn't

The IoT Library is not a general-purpose API supporting every feature offered by the Xively service, nor does it cater for all types of data your may wish to push to Xively. The IoT Library is tested on the microprocessors and evaluation boards that Rowley Associates deliver examples for. Certainly, you can use it with little or no modification on boards that have other processors in the families we support, because it is highly portable and delivered in source code—but, you will need to test, and assure yourself, that the software you ported does indeed function correctly.

What the IoT Library runs on

The IoT Library runs on, and has examples for, the boards that can support the TCP/IP Library. At present, the following microprocessor families support the IoT Library:

- LPC1700
- LM3S
- STM32F1
- STM32F2
- STM32F4

The range of boards and microprocessors that run the IoT Library continues to expand. Please check the CrossWorks web site for the latest information.

<ctl_xively.h>

Overview

This is an interface to push data to the Xively cloud-based service.

The initial implementation of `<ctl_xively.h>` is somewhat inspired by, and takes some direction from, the existing C and mbed APIs offered by Xively. However, we believe that the existing API implementations are not sufficiently balanced for small systems. The existing mbed API has a number of shortcomings:

- It uses static buffers and a configuration header file to size them. It builds the static buffers into the implementation, where necessary, and repurposes the C API, based on dynamic allocation, using them.
- Datastreams and datapoint arrays are statically allocated as a matrix and very large. Every datastream contains the same number of datapoints, irrespective of the datapoint production rate.
- There is no capability to queue data to be sent to Xively and the API is not thread safe.

The CrossWorks Xively API is more flexible, uses memory in a more efficient manner, and supports posting datapoints to queues from interrupt service routines or from threads.

API Summary

API	
CTL_XIVELY_API_t	Xively API context
ctl_xively_api_initialize_http_csv	Initialize API context
Feeds	
CTL_XIVELY_FEED_ID_t	Feed identifier
CTL_XIVELY_FEED_t	Feed controller
ctl_xively_feed_has_pending_data	Outstanding data?
ctl_xively_feed_register_datastream	Register datastream
ctl_xively_feed_update	Push feed data to Xively
Datastreams	
CTL_XIVELY_DATASTREAM_t	Datastream buffer
ctl_xively_datastream_create	Create datastream
ctl_xively_datastream_delete	Delete datastream from server
ctl_xively_datastream_initialize	Initialize datastream
ctl_xively_datastream_post	Post datapoint to a datastream
ctl_xively_datastream_update	Single-point datastream update
Datapoints	
CTL_XIVELY_DATAPOINT_VALUE_t	Datapoint value

CTL_XIVELY_DATAPOINT_t	Datapoint
CTL_XIVELY_VALUE_TYPE_t	Datapoint value type
ctl_xively_datapoint_delete	Delete datapoint from server
ctl_xively_datapoint_delete_range	Delete datapoints from server
ctl_xively_datapoint_write_float	Initialize floating datapoint
ctl_xively_datapoint_write_int	Initialize integer datapoint
ctl_xively_datapoint_write_stamped_float	Initialize timestamped floating datapoint
ctl_xively_datapoint_write_stamped_int	Initialize timestamped integer datapoint
Utility	
ctl_xively_register_error_decoder	Register Xively error decoder

CTL_XIVELY_API_t

Synopsis

```
typedef struct {
  const char *api_key;
  CTL_XIVELY_FEED_t *feed;
  const CTL_XIVELY_TRANSPORT_t *transport;
  const CTL_XIVELY_DATA_ENCODER_t *data_encoder;
  const CTL_XIVELY_COMMS_t *comms;
  CTL_STRBUF_t strbuf;
} CTL_XIVELY_API_t;
```

Description

CTL_XIVELY_API_t defines the Xively API context that the library uses to communicate with and present data to the Xively REST API.

Structure

`api_key`

The API key that's presented to Xively when accessing the Xively REST API.

`feed`

Private pointer to the feed managed by this API context.

`transport`

Private pointer to the transport layer implementation. At present, this library only implements an HTTP transport.

`data_encoder`

Private pointer to the data encoder for datapoints and datastreams.

`comms`

Private pointer to the communications layer to access the Xively API. At present, only the TCP/IP protocol is implemented using the CrossWorks TCP/IP Library.

CTL_XIVELY_DATAPOINT_VALUE_t

Synopsis

```
typedef struct {
  int i;
  float f;
} CTL_XIVELY_DATAPOINT_VALUE_t;
```

Description

CTL_XIVELY_DATAPOINT_VALUE_t defines the data held by the datapoint according to the discriminant **CTL_XIVELY_VALUE_TYPE_t**.

See Also

CTL_XIVELY_VALUE_TYPE_t

CTL_XIVELY_DATAPOINT_t

Synopsis

```
typedef struct {
  CTL_XIVELY_VALUE_TYPE_t type;
  CTL_XIVELY_DATAPOINT_VALUE_t value;
  CTL_XIVELY_TIMESTAMP_t stamp;
} CTL_XIVELY_DATAPOINT_t;
```

Description

CTL_XIVELY_DATAPOINT_t defines a single datapoint that has data in a specified format and an optional timestamp.

Structure

`type`

The type of data held in the `value` member.

`value`

The value of the datapoint, discriminated by the `type` member.

`stamp`

The timestamp of the datapoint. An unstamped datapoint has a null stamp (indicated by a zero `tv_sec` member).

CTL_XIVELY_DATASTREAM_t

Synopsis

```
typedef struct {
  const char *id;
  size_t capacity;
  volatile unsigned count;
  volatile unsigned w;
  volatile unsigned r;
  unsigned u;
  CTL_XIVELY_DATAPOINT_t *data;
  CTL_XIVELY_DATASTREAM_t *__next;
  CTL_XIVELY_FEED_t *__feed;
} CTL_XIVELY_DATASTREAM_t;
```

Description

CTL_XIVELY_DATASTREAM_t defines a queue of datapoints that can be pushed to Xively. The queue is implemented as a ring buffer of datapoints such that datapoints can be posted to the datastream from an interrupt service routine.

Structure

`id`

The datastream identifier.

`capacity`

The maximum number of datapoints that this datastream buffer can contain.

`count`

The current number of datapoints held in the datastream buffer.

`r`

Private read index.

`w`

Private write index.

`u`

Private update count.

`data`

Private pointer to the memory that implements the ring buffer.

`data`

Private pointer to the next datastream in the list of feed datastreams.

CTL_XIVELY_FEED_ID_t

Synopsis

```
typedef unsigned CTL_XIVELY_FEED_ID_t;
```

Description

CTL_XIVELY_FEED_ID_t defines the Xively-assigned feed ID for the device, which is a 32-bit unsigned integer.

CTL_XIVELY_FEED_t

Synopsis

```
typedef struct {
  CTL_XIVELY_FEED_ID_t id;
  CTL_XIVELY_DATASTREAM_t *__head;
  CTL_XIVELY_API_t *__api;
} CTL_XIVELY_FEED_t;
```

Description

CTL_XIVELY_FEED_t defines a Xively feed. Multiple datastreams can be registered with a feed and must be registered before pushing data to Xively.

Structure

id

The feed identifier.

__head

Private pointer to the first datastream registered with the the feed. Datastreams are threaded using a simple linked list.

__api

Private pointer to the API context where the feed is registered.

CTL_XIVELY_VALUE_TYPE_t

Synopsis

```
typedef enum {
  CTL_XIVELY_VALUE_TYPE_INT,
  CTL_XIVELY_VALUE_TYPE_FLOAT
} CTL_XIVELY_VALUE_TYPE_t;
```

Description

CTL_XIVELY_VALUE_TYPE_t defines the type of data held in the datapoint. The encoders use this to present data to the Xively service in the correct format.

See Also

ctl_xively_datapoint_write_int, ctl_xively_datapoint_write_float

ctl_xively_api_initialize_http_csv

Synopsis

```
void ctl_xively_api_initialize_http_csv(CTL_XIVELY_API_t *self,
                                        const char *api_key,
                                        CTL_XIVELY_FEED_t *feed);
```

Description

ctl_xively_api_initialize_http_csv initializes the Xively API context **self** with an HTTP transport and CSV data encoding for the feed **feed** using API key **api_key**.

ctl_xively_datapoint_delete

Synopsis

```
CTL_STATUS_t ctl_xively_datapoint_delete(CTL_XIVELY_DATASTREAM_t *stream,
                                         const CTL_XIVELY_TIMESTAMP_t *stamp);
```

Description

ctl_xively_datapoint_delete sends a request to the Xively API to delete the datapoint with timestamp **stamp** from the datastream **self** in the feed that the datastream is registered to. This function does not remove any data from the local datastream buffer: it is a direct request using a live connection to the Xively server.

Return Value

ctl_xively_datapoint_delete returns a standard status code.

ctl_xively_datapoint_delete_range

Synopsis

```
CTL_STATUS_t ctl_xively_datapoint_delete_range(CTL_XIVELY_DATASTREAM_t *self,
                                               const CTL_XIVELY_TIMESTAMP_t *start,
                                               const CTL_XIVELY_TIMESTAMP_t *end);
```

Description

ctl_xively_datapoint_delete_range sends a request to the Xively API to delete the datapoints with timestamps **start** through **end** from the datastream **self** in the feed that the datastream is registered to. This function does not remove data from the local datastream buffer: it is a direct request using a live connection to the Xively server.

Return Value

ctl_xively_datapoint_delete_range returns a standard status code.

ctl_xively_datapoint_write_float

Synopsis

```
void ctl_xively_datapoint_write_float(CTL_XIVELY_DATAPOINT_t *self,
                                      float value);
```

Description

ctl_xively_datapoint_write_float initializes the datapoint **self** to contain the floating-point value **value** with a null timestamp.

ctl_xively_datapoint_write_int

Synopsis

```
void ctl_xively_datapoint_write_int(CTL_XIVELY_DATAPOINT_t *self,
                                    int value);
```

Description

ctl_xively_datapoint_write_int initializes the datapoint **self** to contain the integer value **value** with a null timestamp.

ctl_xively_datapoint_write_stamped_float

Synopsis

```
void ctl_xively_datapoint_write_stamped_float(CTL_XIVELY_DATAPOINT_t *self,
                                              CTL_XIVELY_TIMESTAMP_t *stamp,
                                              float value);
```

Description

ctl_xively_datapoint_write_stamped_float initializes the datapoint **self** to contain the floating-point value **value** with the timestamp **stamp**. If **stamp** is zero, the datapoint is initialized with a null timestamp.

ctl_xively_datapoint_write_stamped_int

Synopsis

```
void ctl_xively_datapoint_write_stamped_int(CTL_XIVELY_DATAPOINT_t *self,
                                            CTL_XIVELY_TIMESTAMP_t *stamp,
                                            int value);
```

Description

ctl_xively_datapoint_write_stamped_int initializes the datapoint **self** to contain the integer value **value** with the timestamp **stamp**. If **stamp** is zero, the datapoint is initialized with a null timestamp.

ctl_xively_datastream_create

Synopsis

```
CTL_STATUS_t ctl_xively_datastream_create(CTL_XIVELY_DATASTREAM_t *self,
                                          const CTL_XIVELY_DATAPOINT_t *point);
```

Description

ctl_xively_datastream_create sends a request to the Xively API to create the datastream **self** with a single datapoint **point** in the feed that the datastream is registered to. This function does not store any data in the local datastream buffer: it is a direct request using a live connection to the Xively server.

Return Value

ctl_xively_datastream_create returns a standard status code.

See Also

ctl_xively_feed_update

ctl_xively_datastream_delete

Synopsis

```
CTL_STATUS_t ctl_xively_datastream_delete(CTL_XIVELY_DATASTREAM_t *self);
```

Description

ctl_xively_datastream_delete sends a request to the Xively API to delete the datastream **self** in the feed that the datastream is registered to. This function does remove any data in the local datastream buffer: it is a direct request using a live connection to the Xively server.

Return Value

ctl_xively_datastream_delete returns a standard status code.

ctl_xively_datastream_initialize

Synopsis

```
CTL_STATUS_t ctl_xively_datastream_initialize(CTL_XIVELY_DATASTREAM_t *self,
                                              const char *id,
                                              size_t capacity,
                                              CTL_XIVELY_DATAPOINT_t *data);
```

Description

ctl_xively_datastream_initialize initializes the local datastream **self** using the datastream ID **id** to contain at most **capacity** datapoints in the array pointed to by **data**.

Once the datastream is initialized, it must be registered with a feed using **ctl_xively_feed_register_datastream**.

Return Value

ctl_xively_datastream_initialize returns a standard status code.

See Also

ctl_xively_feed_register_datastream

ctl_xively_datastream_post

Synopsis

```
CTL_STATUS_t ctl_xively_datastream_post(CTL_XIVELY_DATASTREAM_t *self,
                                        CTL_XIVELY_DATAPOINT_t *point);
```

Description

ctl_xively_datastream_post posts the datapoint **point** to the datastream **self**.

Return Value

ctl_xively_datastream_post returns a standard status code. If there is not enough space in the stream to post the point, **ctl_xively_datastream_post** return **CTL_OUT_OF_MEMORY**.

Thread Safety

This is safe to call from a thread or CTL interrupt handler.

ctl_xively_datastream_update

Synopsis

```
CTL_STATUS_t ctl_xively_datastream_update(CTL_XIVELY_DATASTREAM_t *self,
                                          const CTL_XIVELY_DATAPOINT_t *point);
```

Description

ctl_xively_datastream_update sends a request to the Xively API to write the datapoint **point** to the datastream **self** in the feed that the datastream is registered to. This function does not store any data in the local datastream buffer: it is a direct request using a live connection to the Xively server.

Return Value

ctl_xively_datastream_update returns a standard status code.

See Also

ctl_xively_feed_update

ctl_xively_feed_has_pending_data

Synopsis

```
int ctl_xively_feed_has_pending_data(CTL_XIVELY_FEED_t *self);
```

Description

ctl_xively_feed_has_pending_data returns non-zero if any of the datastreams registered with the feed **self** have outstanding data that has not been pushed to Xively using **ctl_xively_feed_update**.

Note that a feed may well have data outstanding even if **ctl_xively_feed_update** completes successfully as only a subset of outstanding data in the feed may have been sent to Xively.

Return Value

ctl_xively_feed_has_pending_data returns non-zero if there is outstanding data to push to Xively and zero if there is none.

See Also

ctl_xively_feed_update

ctl_xively_feed_register_datastream

Synopsis

```
CTL_STATUS_t ctl_xively_feed_register_datastream(CTL_XIVELY_FEED_t *self,
                                                 CTL_XIVELY_DATASTREAM_t *stream);
```

Description

ctl_xively_feed_register_datastream registers an initialized datastream **stream** with the feed **self**. You must register datastreams with a feed before posting datapoints to the datastream and before pushing the datastreams to Xively.

Return Value

ctl_xively_feed_register_datastream returns a standard status code.

See Also

ctl_xively_datastream_initialize

ctl_xively_feed_update

Synopsis

```
CTL_STATUS_t ctl_xively_feed_update(CTL_XIVELY_FEED_t *self);
```

Description

ctl_xively_feed_update pushes as much data as it can from **feed** to the Xively service.

A feed may well have data outstanding even if **ctl_xively_feed_update** completes successfully as only a subset of outstanding data in the feed may have been sent to Xively. **ctl_xively_feed_update** uses the string buffer in the feed's context to encode the data to send to Xively, so smaller string buffers will mean that fewer datapoints can be encoded without overflowing the buffer and, hence, there is a possibility that not all outstanding datapoints in the feed will be pushed to Xively.

After encoding the datapoints, the datapoints are sent to Xively using the selected transport—only the HTTP transport is implemented at present. If there is an error sending the encoded datapoints to Xively, the datapoints buffered in the feed **self** are not discarded, they remain in the datastream buffers such that they can be pushed to Xively by invoking **ctl_xively_feed_update** again.

The way to ensure that all datapoints are sent to Xively is to inquire whether there are more outstanding datapoints after successfully pushing to Xively, for instance:

```
while (ctl_xively_feed_has_pending_data(&feed))
  {
    // Send as much data to Xively as we can.
    ctl_xively_feed_update(&feed);

    // Wait a little before sending more.
    ctl_delay(1000);
  }
```

See Also

ctl_xively_feed_has_pending_data

ctl_xively_register_error_decoder

Synopsis

```
void ctl_xively_register_error_decoder(void);
```

Description

ctl_xively_register_error_decoder registers an error decoder with the CrossWorks runtime to decode errors generated by Xively API calls.