CARETAKER LIBRARY API 1.6.5 REFERENCE MANUAL

Caretaker Medical

October 11, 2018

Contents

Chapter 1

Introduction

The Caretaker library is a cross-platform library to link with Android, Linux, and Windows applications.The library provides an interface to the Caretaker device wireless vitals signs monitoring device.

Scope

This version of the manual covers the Caretaker Library Linux API. The Caretaker library Android and Windows APIs are covered in separate documents. Please contact customer service for copies of the other APIs documentation as needed.

Getting Started

Read the following sections to integrate the library with an application to monitor the Caretaker device.

Chapter 2

Library Integration

This section provides an overview of the library integration with Linux applications.

Library Package

Linux versions of the library are released in zip format *ctlibrary-linux-ARCH-VERSION.zip* for use with Linux applications, where *ARCH* is the target platform architecture, and *VERSION* is the library version bundling the following components.

Component	Description
caretaker.h	Header file exposing the Caretaker APIs to the application.
libcaretaker.a	Caretaker library image to use for static linking with the application.
libcaretaker.so	Caretaker library image to use for dynamic linking with the application.
libbluetooth.a	Precompiled version of the BlueZ Linux Bluetooth library for static linking provided for convenience.
libbluetooth.so	Precompiled version of the BLueZ Linux Bluetooth library for dynamic linking provided for conveniece.
manual	The library API documentation. Documentation is available in pdf and html formats. The pdf version is supplied for completeness though the formatting is optimized for viewing in html format. For viewing the html version main page open *html/index.html*.
examples	Examples illustrating library use.

Supported Platforms

This library version links with Ubuntu and Debian applications running on x86 and ARM based platforms.

The library was tested on the following Linux systems.

- Linux systems with Ubuntu 14.04 or later
- Raspberry Pi with Raspbian 9.1 or later

Software Configuration

Ubuntu 14.04, Linux kernel 4.4.0-72-generic or later must be installed on the computer that will execute the library targeting Ubuntu application.

Raspbian 9.1, Linux kernel 4.9.59-v7 or late must be installed on the computer that will execute the library targeting Raspberry Pi application.

Hardware Configuration

If the Linux computer that will run the application does not have Bluetooth support or the Bluetooth hardware is not current, you will need the CSR8510 A10 BLE dongle or similar to enable the computer to communicate with the Caretaker device. If so, plug the dongle into a USB port on the computer and then run the following commands to configure it. Replace *hcin* with the actual host control interface name printed in the hcitool output for the dongle.

```
hcitool dev
sudo hciconfig <hcin> reset
```

Linking with the Library

Linux applications can link the with the library using either static or dynamic linking.

Static Linking

For static linking, the static library image *libcaretaker.a* and the supplied *libbluetooth.a* for BLE communication will need to be linked to the application. Copy these two library images to the application project library directory and link them as follows if compiling with *gcc*.

```
$(PROG): $(OBJS)
    gcc -static -o $@ $^ $(LIBS)/libcaretaker-VERSION.a $(LIBS)/libbluetooth.a -lpthread
```

Where *PROG* is the application program name, *OBJS* are the application object files, and *LIBS* is the path to the project library directory. Note the caretaker library uses POSIX threads so *-lpthread* is required to link the pthread library.

Dynamic Linking

For dynamic linking, the dynamic library image *libcaretaker.so* and the supplied *libbluetooth.so* for BLE communication will need to be linked to the application. Copy these library images to the application project library directory and link them as follows if compiling with *gcc*.

```
$(PROG): $(OBJS)
    gcc -o $@ $^ -lcaretaker-linux-VERSION -lbluetooth -lpthread -L$(LIBS)
```

Where *PROG* is the application program name, *OBJS* are the application object files, and *LIBS* is the path to the project library directory. Note the caretaker library uses POSIX threads so *-lpthread* is required to link the pthread library.

Debugging

The library supports generating log messages with the following levels of verbosity.

Log Level	Description
0	Show all log messages. This is the most verbose level.
1	Show informational, warning, and error messages only.
2	Show warning and error messages only.
3	Show error mesages only. This is the least verbose level.

You can call libct_set_log_level() from the application code to set the log level.

Log messages are written to standard output (stdout) and will be printed to the terminal window where the application was started by default. So to view the log messages, the application must be started from the terminal window explicilty. The logs can be viewed and saved as follows assuming *appExe* is the application executable.

```
appExe | tee libcaretaker.log
```

Log messages from the library code will have the following format.

```
DATE       TIME         LEVEL/FILTER   THREAD  MESSAGE
---------- ------------ -------------- ------- -------------------------------------------------
2018-08-01 14:47:34.632 I/libcaretaker (10592) libct_init : Version: 0.0.0.f8b97f70.x64.debug
2018-08-01 14:47:45.566 D/libcaretaker (06040) glue_rcv_thread : started.
```

The log levels are defined as follows.

- D/ –> Debug
- T/ –> Trace
- I/ –> Informational
- W/ –> Warning
- E/ –> Error

Chapter 3

Monitoring the Device

This section documents monitoring Caretaker data after integratng the application with the library.

Overview

Two groups of APIs are defined to simplify getting started: Primary and Secondary APIs. The primary API is the core interface required to connect to the Caretaker device to monitor numeric and waveform data, and the secondary API is an auxiliary interface to parse, read and write additional Caretaker data.

The sequence diagram below illustrates how a user application will interact with the primary functions to connect and start monitoring Caretaker data, which can be summarized as the following six steps.

- **Step 1:** Initialize a library context to associate with a Caretaker device.
- **Step 2:** Discover the device.
- **Step 3:** Connect to the device.
- **Step 4:** Start monitoring device data.
- **Step 5:** Calibrate and start measurements.
- **Step 6:** Handle numeric and waveform data updates.

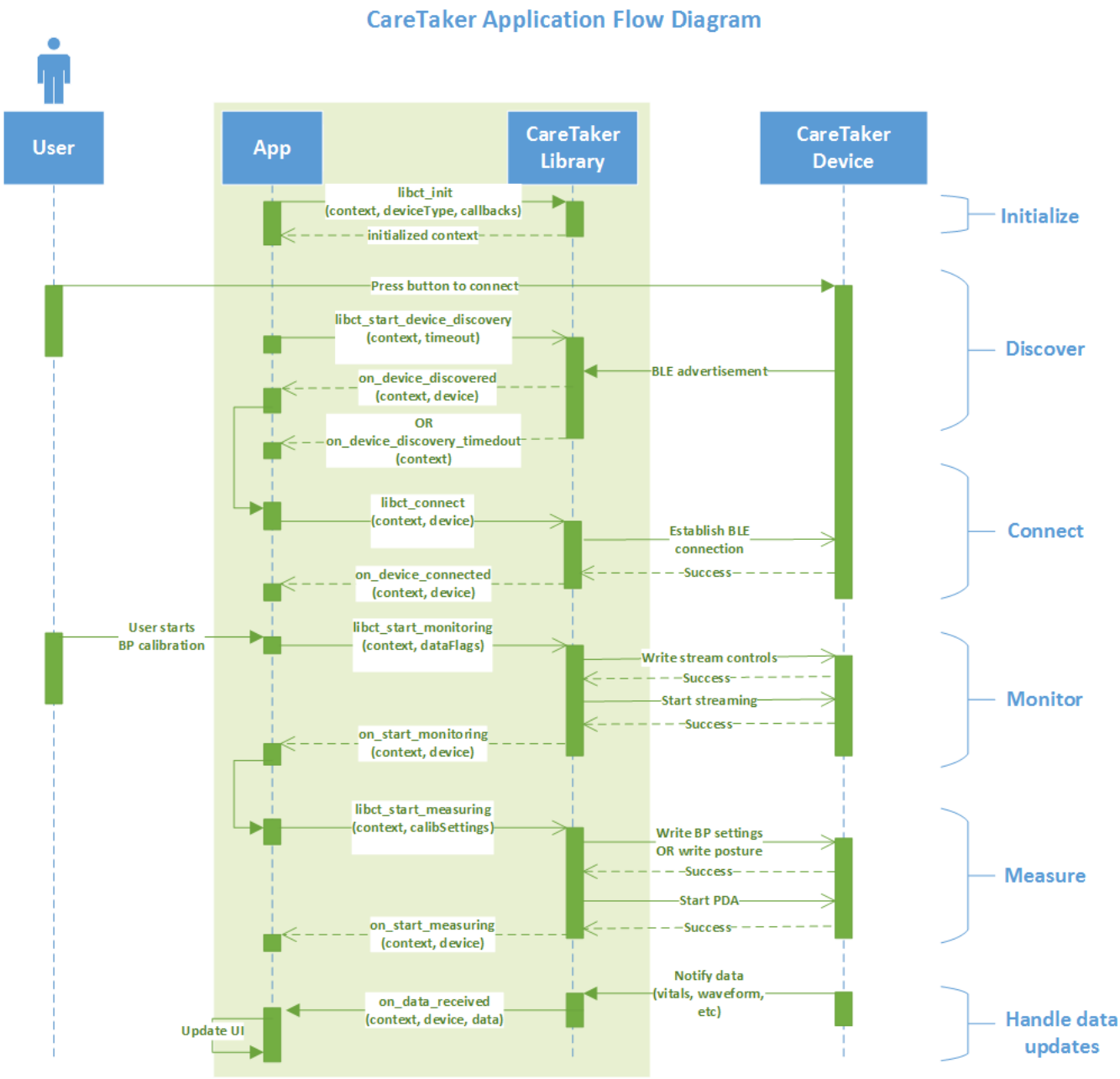

Figure 3.1 Sequence diagram to connect and monitor data.

height=8in

Note: For simplification the code snippets used throughout this manual omit error handling. It is recommended that any application handle errors returned from library functions appropriately.

Step 1: Initialize a library context

Start by initializing a library context (or library instance) by calling libct_init(). Specify the class of device to associate with this library context in the init data. Set the appropriate application callback functions, and set any unused callbacks to null. At a mimimum, the following callback functions should be implemented.

- on_device_discovered()
- on_device_connected_ready()
- on_device_disconnected()
- on_data_received()

Next call libct_init() with the context pointer, init data, and callback variables. Note the context pointer must be initialized to null prior to being passed to libct_init() to indicate it is not in use, otherwise libct_init() will return error.

```
libct_init_data_t init_data;
memset(&init_data, 0, sizeof(init_data));
init_data.device_class = LIBCT_DEVICE_CLASS_BLE_CARETAKER4;

libct_app_callbacks_t callbacks = { ... }

libct_context_t* context = NULL;
int status = libct_init(&context, &init_data, &callbacks);
if ( LIBCT_FAILED(status) ) {
    // Handle error
}
```

Optionally, after initializing the context, the application specific data can be saved in the context for retrieval and use later in the application callbacks. For example, a C++ main application instance can be set as app specific data for access in the callbacks. See libct_set_app_specific_data() and libct_get_app_specific_data() for more information.

```
libct_set_app_specific_data(context, this);
```

Step 2: Discover a device

If libct_init() returned success, a device context has been created and initialized to start device discovery. Call libct_start_discovery() to scan for nearby devices. It will scan for some specified timeout (20 seconds in the code example below) and automatically stop if the time out has been reached or if libct_stop_discovery() has been explicitly called to stop device discovery.

```
libct_start_discovery(context, 20000);
```

The application will receive notifications later from devices matching the device class specified in the init_data passed to libct_init(). These notifications will be signalled to the application with the following callback functions.

- on_device_discovered()
- on_discovery_timedout()
- on_discovery_failed()

Note: The on_device_discovered() callback must be implemented to receive notification when a matching device is found.

Step 3: Connect to the device

The following implementation illustrates connecting to the first device found. However, by implementing a device whitelist to check discovered devices against a known acceptable list, a specific device can be searched for and automatically connected or all discovered devices can be displayed on the application GUI allowing the user to select the appropriate device.

```
void LIBCTAPI on_device_discovered_cb(libct_context_t* context,
      libct_device_t* device) {
    libct_stop_discovery(context);
    libct_connect(context, device);
}
```

After calling libct_connect(), the application will later receive notfications signalling the connection status with the following callbacks.

- on_device_connected_not_ready()
- on_device_connected_ready()
- on_connect_error()
- on_connect_timedout()

Note: The on_device_connected_ready() callback must be implemented to receive notification when the connection is established and the device is ready to receive requests.

Step 4: Monitor device data

After the connection has been established, monitoring vitals from the device can be started. The following is a sample implementation illustrating this in the connection ready callback, but note monitoring the device data can be defered until other application events are received, such as user input from the application GUI.

```
 void LIBCTAPI on_device_connected_ready_cb(libct_context_t* context,
      libct_device_t* device) {
     int flags = (LIBCT_MONITOR_INT_PULSE |
                  LIBCT_MONITOR_PARAM_PULSE |
                  LIBCT_MONITOR_VITALS |
                  LIBCT_MONITOR_CUFF_PRESSURE |
                  LIBCT_MONITOR_DEVICE_STATUS |
                  LIBCT_MONITOR_BATTERY_INFO);

     libct_start_monitoring(context, flags);
}
```

Note: The application will receive monitor status via theon_start_monitoring() callback. The callback will be invoked only once in response to each libct_start_monitoring() call and is thus a oneshot callback.

Step 5: Calibrate and start measurements

After calling libct_start_monitoring(), the application will start receiving data from the device via the on_data_received() callback, however, the application will not receive valid vitals and waveform data until the blood pressure measurements are calibrated. Again, starting calibration can be deferred until other application events are received, such as user input from the application GUI.

The following code illustrates starting automatic calibration. Note the patient posture must be retrieved elsewhere, such as from the application GUI.

```
libct_cal_t cal;
cal.type = LIBCT_AUTO_CAL;
cal.config.auto_cal.posture = posture;
libct_start_measuring(context, &cal);
```

And the following code illustrates starting manual calibration. Again, the systolic and diastolic initial values must be retrieved elsewhere, such as from the application GUI.

```
libct_cal_t cal;
cal.type = LIBCT_MANUAL_CAL;
cal.config.manual_cal.settings.systolic = systolic;
cal.config.manual_cal.settings.diastolic = diastolic;
libct_start_measuring(context, &cal);
```

Note: The application will receive measurement status via the on_start_measuring() callback, which is a oneshot callback, i.e., the callback will be invoked only once in response to each libct_start_measuring() call.

Step 6: Handle numeric and waveform data updates

If monitoring and measurments were started successfully, the application will start receiving numeric and waveform data updates. The application on_data_received() callback will be notified continously while data is received from the device.

The following code snippet illustrates processing data received from the device in the application on_data_received() callback. See the stream data structure for data format details.

```
void LIBCTAPI on_data_received_cb(libct_context_t* context,
      libct_device_t* device, libct_stream_data_t* data) {
   // Obtain the application instance set earlier with libct_set_app_specific_data().
   // Note libct_get_app_specific_data() returns null if libct_set_app_specific_data() was not
   // called earlier to set the application instance.
   MainWindow* window = (MainWindow*) libct_get_app_specific_data(context);

   // Update device status
   if ( data->device_status->valid ) {
        // ... check device status flags
   }

   // Update vitals
   libct_vitals_t* vitals = libct_get_last_dp(data, vitals);
   if ( vitals && vitals->valid ) {
        window->setHr(vitals->heart_rate);
        window->setRes(vitals->respiration);
        window->setMap(vitals->map);
        window->setBp(vitals->systolic, vitals->diastolic);
   }

   // Update the pulse rate waveform
   unsigned int idx;
   libct_pulse_t* pulse;
   for_each_dp(data, idx, pulse, raw_pulse) {
        if ( pulse && pulse->valid ) {
             window->rawPulseWaveform->add(pulse->timestamp, pulse->
      value);
        }
```

```
    }

    // Update the pulse pressure waveform
    for_each_dp(data, idx, pulse, int_pulse) {
         if ( pulse && pulse->valid ) {
              if ( pulse && pulse->valid ) {
                   window->intPulseWaveform->add(pulse->timestamp, pulse->
       value);
              }
         }
    }
}
```

Chapter 4

Module Index

4.1 Modules

Here is a list of all modules:

Chapter 5

Data Structure Index

5.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 6

Module Documentation

6.1 Device Information

This modules describes the interface to retrieve general information about the Caretaker device.

Data Structures

- class libct_device_t

  Handle used to identify a connected device the application is monitoring.

Macros

- #define libct_device_get_state(dev) (dev)->get_state(dev)

  Convenience macro to device->get_state(device).
- #define libct_device_uninitialized(dev) (((dev)->get_state(dev)) & LIBCT_STATE_UNITIALIZED)

  Returns non-zero (true) if the device is not initialized, and zero (false) otherwise.
- #define libct_device_intialized(dev) (!libct_device_uninitialized(dev))

  Returns non-zero (true) if the device is initialized, and zero (false) otherwise .
- #define libct_device_discovering(dev) (((dev)->get_state(dev)) & LIBCT_STATE_DISCOVERING)

  Returns non-zero (true) if discovering the device, and zero (false) otherwise.
- #define libct_device_connecting(dev) (((dev)->get_state(dev)) & LIBCT_STATE_CONNECTING)

  Returns non-zero (true) if connecting to the device, and zero (false) otherwise.
- #define libct_device_connected(dev) (((dev)->get_state(dev)) & LIBCT_STATE_CONNECTED)

  Returns non-zero (true) if connected to the device, and zero (false) otherwise.
- #define libct_device_disconnecting(dev) (((dev)->get_state(dev)) & LIBCT_STATE_DISCONNECTING)

  Returns non-zero (true) if disconnecting from the device, and zero (false) otherwise.
- #define libct_device_disconnected(dev) (((dev)->get_state(dev)) & LIBCT_STATE_DISCONNECTED)

  Returns non-zero (true) if disconnected from the device, and zero (false) otherwise.
- #define libct_device_monitoring(dev) (((dev)->get_state(dev)) & LIBCT_STATE_MONITIORING)

  Returns non-zero (true) if receiving data from the device, and zero (false) otherwise.
- #define libct_device_measuring(dev) (((dev)->get_state(dev)) & LIBCT_STATE_MEASURING)

  Returns non-zero (true) if taking blood pressure measurements, and zero (false) otherwise.
- #define libct_device_get_class(dev) (dev)->get_class(dev)

  Convenience macro to device->get_class(device).

- #define libct_device_get_name(dev) (dev)->get_name(dev)

  Convenience macro to device->get_name(device).
- #define libct_device_get_address(dev) (dev)->get_address(dev)

  Convenience macro to device->get_address(device).
- #define libct_device_get_serial_number(dev) (dev)->get_serial_number(dev)

  Convenience macro to device->get_serial_number(device).
- #define libct_device_get_hw_version(dev) (dev)->get_hw_version(dev)

  Convenience macro to device->get_hw_version(device).
- #define libct_device_get_fw_version(dev) (dev)->get_fw_version(dev)

  Convenience macro to device->get_fw_version(device).
- #define libct_device_get_context(dev) (dev)->get_context(dev)

  Convenience macro to device->get_context(device).

6.1.1 Detailed Description

This modules describes the interface to retrieve general information about the Caretaker device.

6.1.2 Macro Definition Documentation

6.1.2.1 #define libct_device_get_state(*dev*) (dev)->get_state(dev)

Convenience macro to device->get_state(device).

Parameters

dev	Pointer to the device instance.

6.1.2.2 #define libct_device_uninitialized(*dev*) (((dev)->get_state(dev)) & LIBCT_STATE_UNITIALIZED)

Returns non-zero (true) if the device is not initialized, and zero (false) otherwise.

Parameters

dev	Pointer to device instance.

6.1.2.3 #define libct_device_discovering(*dev*) (((dev)->get_state(dev)) & LIBCT_STATE_DISCOVERING)

Returns non-zero (true) if discovering the device, and zero (false) otherwise.

Parameters

dev	Pointer to device instance.

6.1.2.4 #define libct_device_connecting(*dev*) (((dev)->get_state(dev)) & LIBCT_STATE_CONNECTING)

Returns non-zero (true) if connecting to the device, and zero (false) otherwise.

Parameters

dev	Pointer to device instance.

6.1.2.5 #define libct_device_connected(*dev*) (((dev)->get_state(dev)) & LIBCT_STATE_CONNECTED)

Returns non-zero (true) if connected to the device, and zero (false) otherwise.

Parameters

dev	Pointer to device instance.

6.1.2.6 #define libct_device_disconnecting(*dev*) (((dev)->get_state(dev)) & LIBCT_STATE_DISCONNECTING)

Returns non-zero (true) if disconnecting from the device, and zero (false) otherwise.

Parameters

dev	Pointer to device instance.

6.1.2.7 #define libct_device_disconnected(*dev*) (((dev)->get_state(dev)) & LIBCT_STATE_DISCONNECTED)

Returns non-zero (true) if disconnected from the device, and zero (false) otherwise.

Parameters

dev	Pointer to device instance.

6.1.2.8 #define libct_device_monitoring(*dev*) (((dev)->get_state(dev)) & LIBCT_STATE_MONITIORING)

Returns non-zero (true) if receiving data from the device, and zero (false) otherwise.

Parameters

dev	Pointer to device instance.

6.1.2.9 #define libct_device_measuring(*dev*) (((dev)->get_state(dev)) & LIBCT_STATE_MEASURING)

Returns non-zero (true) if taking blood pressure measurements, and zero (false) otherwise.

Parameters

dev	Pointer to device instance.

6.1.2.10 #define libct_device_get_class(*dev*) (dev)->get_class(dev)

Convenience macro to device->get_class(device).

Parameters

dev	Pointer to the device instance.

6.1.2.11 #define libct_device_get_name(*dev*) (dev)->get_name(dev)

Convenience macro to device->get_name(device).

Parameters

dev	Pointer to the device instance.

6.1.2.12 #define libct_device_get_address(*dev*) (dev)->get_address(dev)

Convenience macro to device->get_address(device).

Parameters

dev	Pointer to the device instance.

6.1.2.13 #define libct_device_get_serial_number(*dev*) (dev)->get_serial_number(dev)

Convenience macro to device->get_serial_number(device).

Parameters

dev	Pointer to the device instance.

6.1.2.14 #define libct_device_get_hw_version(*dev*) (dev)->get_hw_version(dev)

Convenience macro to device->get_hw_version(device).

Parameters

dev	Pointer to the device instance.

6.1.2.15 #define libct_device_get_fw_version(*dev*) (dev)->get_fw_version(dev)

Convenience macro to device->get_fw_version(device).

Parameters

dev	Pointer to the device instance.

6.1.2.16 #define libct_device_get_context(*dev*) (dev)->get_context(dev)

Convenience macro to device->get_context(device).

Parameters

dev	Pointer to the device instance.

6.2 Primary API

The group of primary functions that are required to connect to a Caretaker device and monitor data.

Functions

- LIBCTEXPORT int libct_init (libct_context_t **context, libct_init_data_t *data, libct_app_callbacks_t *callbacks)

  Initializes device context.
- LIBCTEXPORT void libct_deinit (libct_context_t *context)

  De-initializes the context.
- LIBCTEXPORT int libct_start_discovery (libct_context_t *context, unsigned long timeout)

  Discover the device.
- LIBCTEXPORT int libct_stop_discovery (libct_context_t *context)

  Stop device discovery.
- LIBCTEXPORT int libct_connect (libct_context_t *context, libct_device_t *device)

  Connect to a device.
- LIBCTEXPORT int libct_disconnect (libct_context_t *context)

  Disconnect from a device.
- LIBCTEXPORT int libct_start_monitoring (libct_context_t *context, int flags)

  Start monitoring data at the remote caretaker device.
- LIBCTEXPORT int libct_stop_monitoring (libct_context_t *context)

  Stops monitoring.
- LIBCTEXPORT int libct_start_measuring (libct_context_t *context, libct_cal_t *cal)

  Start taking measurement.
- LIBCTEXPORT int libct_stop_measuring (libct_context_t *context)

  Stops measuring.

6.2.1 Detailed Description

The group of primary functions that are required to connect to a Caretaker device and monitor data.

6.2.2 Function Documentation

6.2.2.1 LIBCTEXPORT int libct_init (libct_context_t ** *context*, libct_init_data_t * *data*, libct_app_callbacks_t * *callbacks*)

Initializes device context.

Call this function to initialize a device context before calling any other library functions with the said context.

Note

You can initialize multiple contexts if you wish to connect to multiple devices simultaneously, but you must call libct_deinit() from the same thread to de-initialize each context when it is no longer needed.

Parameters

context	Address to store the created context. **IMPORTANT:** Initialize the context pointer to null before passing it. The internal library code depends on this to ensure the context is initialized only once.
data	Data to initialize the context.
callbacks	The application callback functions to receive asynchronous notifications. This pointer must not be null, or else your application will not receive notifications notifying connection and data events. However, you can set function pointers within this structure that you don't care about to null. **NOTE:** You can set application specific data to use inside your callbacks with libct_set_app_specific_data() after initialization, and later retrieve it with libct_get_app_specific_data() to get the application instance data to act upon inside the callbacks.

Returns

An appropriate status code indicating success or error.

6.2.2.2 LIBCTEXPORT void libct_deinit (libct_context_t * *context*)

De-initializes the context.

Call this function to release resources when you no longer need the context.

IMPORTANT: The application must call dibct_deinit() some time after calling libct_init() to prevent resource leaks. libct_deinit() must not be called from any library callback function. Library callbacks are called from internal library threads that this function attempts to kill. As such, it must only be called from an application thread.

Parameters

context	The context returned from libct_init().

6.2.2.3 LIBCTEXPORT int libct_start_discovery (libct_context_t * *context*, unsigned long *timeout*)

Discover the device.

This function start scanning for devices specify by the device class in the initialization data passed earlier to libct_init(). Scan results will be notified asynchronously via the application callbacks passed to libct_init(); specifically, these discovery callback functions will be invoked some time later with the results when devices are discovered or if scanning timed out or failed.

- on_device_discovered()
- on_discovery_timedout()
- on_discovery_failed()

Note

Devices must be advertising and be within range for this method to discover them. Press the button on the caretaker to start advertising. Note the caretaker only advertises for 20 seconds after pressing the button and then stops.

Parameters

context	The context returned from libct_init().
timeout	Scanning will be canceled after the number of milliseconds specified by this timeout and the application discovery timeout callback will be invoked.

Returns

An appropriate status code indicating success or error.

6.2.2.4 LIBCTEXPORT int libct_stop_discovery (libct_context_t * *context*)

Stop device discovery.

Call this function to stop device discovery previously started with libct_start_discovery().

Parameters

context	The context returned from libct_init().

Returns

An appropriate status code indicating success or error.

6.2.2.5 LIBCTEXPORT int libct_connect (libct_context_t * *context*, libct_device_t * *device*)

Connect to a device.

Call this method after device discovery to establish connection to the device. The results will be notified asynchronously via the application callbacks passed to libct_init(); specifically, one or more of the following callback functions will be invoked some time later with the results if the connection is established, timed out or failed.

- on_device_connected_not_ready()
- on_device_connected_ready()
- on_connect_error()
- on_connect_timedout()

Parameters

context	The context returned from libct_init().
device	A discovered device notified with your application callback on_device_discovered().

Returns

An appropriate status code indicating success or error.

6.2.2.6 LIBCTEXPORT int libct_disconnect (libct_context_t * *context*)

Disconnect from a device.

Call this method after calling libct_connect() to clean up resources that were allocated by the connect call.

IMPORTANT: The application must call libct_disconnect() to release connection resources before calling libct_connect() subsequently on the same context. Otherwise, the subsequent connect calls may fail. Also, the application must not call libct_disconnect() from any library callback function. Library callbacks are called from internal library threads and this function attempts to kill. As such, it must only be called from an application thread.

Parameters

context	The context returned from libct_init().

Returns

An appropriate status code indicating success or error.

6.2.2.7 LIBCTEXPORT int libct_start_monitoring (libct_context_t * *context*, int *flags*)

Start monitoring data at the remote caretaker device.

Call this function after the connection is established with the device to start monitoring data or to change the data being monitored.

Calling this function will trigger your application's on_start_monitoring() to be invoked some time later with results. Also, if monitoring was started successfully, data from the device will be notified to your application's on_data_received() continuously until stopped explicitly by calling to libct_stop_monitoring() or if the device becomes disconnected.

Parameters

context	The context returned from libct_init().
flags	Bitwise OR of monitor flags specifying the data to monitor. **Note:** The stream data packets notified to the application depends on these flags. So you can control the amount of data reported to the application by specifying only the monitoring flags corresponding to the data you care about.

Returns

An appropriate status code indicating success or error.

6.2.2.8 LIBCTEXPORT int libct_stop_monitoring (libct_context_t * *context*)

Stops monitoring.

Call this method to stop monitoring data after calling libct_start_monitoring() successfully.

Calling this function will trigger on_stop_monitoring() to be invoked sometime later with results.

Parameters

context	The context returned from libct_init().

Returns

An appropriate status code indicating success or error.

6.2.2.9 LIBCTEXPORT int libct_start_measuring (libct_context_t * *context*, libct_cal_t * *cal*)

Start taking measurement.

If monitoring was started successfully, call this function to initialize (calibrate) blood pressure settings with either auto or manual calibration then start taking vital sign measurements.

Calling this function will trigger on_start_measuring() to be invoked sometime later with results.

Parameters

context	The context returned from libct_init().
cal	Auto or manual calibration settings.

Returns

An appropriate status code indicating success or error

6.2.2.10 LIBCTEXPORT int libct_stop_measuring (libct_context_t * *context*)

Stops measuring.

Call this method to stop measuring data after calling libct_start_measuring() successfully.

Calling this function will trigger on_stop_measuring() to be invoked sometime later with results.

Parameters

context	The context returned from libct_init().

Returns

An appropriate status code indicating success or error.

6.3 Secondary API

The group of auxiliary functions and macros available to parse, read and write additional Caretaker device data.

Macros

- #define libct_dp_count(data, memb) (data)->memb.count

  Returns the count of data points of the specified member array contained in stream data received at the application.
- #define libct_get_dp(data, memb, pos)

  Extract a single data point from the specified member array contained in stream data received at the application.
- #define libct_get_last_dp(data, memb) libct_get_dp(data, memb, (data)->memb.count-1)

  Extract the newest data point from the specified member array contained in stream data received at the application.
- #define libct_get_first_dp(data, memb) libct_get_dp(data, memb, 0)

  Extract the oldest data point from the specified member array contained in stream data received at the application.
- #define for_each_dp(data, idx, dp, memb) for(idx=0; (idx<(data)->memb.count) && (dp=((data)->memb.datapoints) ? &(data)->memb.datapoints[idx] : NULL); idx++)

  Iterate over data points of the specified member array to extract from stream data received at the application.
- #define libct_inc_cuff_pressure(context) libct_adjust_cuff_pressure((context), 1)

  Increments the cuff pressure in 10 mmHg increment.
- #define libct_dec_cuff_pressure(context) libct_adjust_cuff_pressure((context), 0)

  Decrements the cuff pressure in 10 mmHg increment.

Functions

- LIBCTEXPORT libct_device_t * libct_get_device (libct_context_t *context)

  Returns the device handle.
- LIBCTEXPORT void libct_set_app_specific_data (libct_context_t *context, void *data)

  Sets application specific data that can be retrieved and used later in the callbacks.
- LIBCTEXPORT void * libct_get_app_specific_data (libct_context_t *context)

  Retrieve application specific data.
- LIBCTEXPORT const char * libct_get_version_string (void)

  Get the library version info.
- LIBCTEXPORT const char * libct_get_build_date_string (void)

  Get the library build date and time string.
- LIBCTEXPORT void libct_set_log_level (int level)

  Sets the library log level.
- LIBCTEXPORT int libct_recalibrate (libct_context_t *context)

  Re-calibrates the device.
- LIBCTEXPORT int libct_adjust_cuff_pressure (libct_context_t *context, int direction)

  Adjusts the cuff pressure in 10 mmHg increment/decrement.
- LIBCTEXPORT int libct_rd_cuff_pressure (libct_context_t *context)

  Reads the cuff pressure.
- LIBCTEXPORT int libct_vent_cuff (libct_context_t *context)

  Deflates the cuff pressure.
- LIBCTEXPORT int libct_clr_status (libct_context_t *context)

  Clears the device status.
- LIBCTEXPORT int libct_diag_flush (libct_context_t *context)

  Invoke the device diagnostic plumbing tree flush.
- LIBCTEXPORT int libct_wrt_snr_min (libct_context_t *context, int snr)

Writes the device noise filter parameter.

- LIBCTEXPORT int libct_rd_snr_min (libct_context_t *context)

  Reads the device noise filter parameter.
- LIBCTEXPORT int libct_wrt_display_state (libct_context_t *context, unsigned char state)

  Turns the device display on/off.
- LIBCTEXPORT int libct_rd_display_state (libct_context_t *context)

  Reads the device display state.
- LIBCTEXPORT int libct_wrt_recal_itvl (libct_context_t *context, unsigned int itvl)

  Writes the recalibration interval.
- LIBCTEXPORT int libct_rd_recal_itvl (libct_context_t *context)

  Reads the recalibration interval.
- LIBCTEXPORT int libct_wrt_waveform_clamping (libct_context_t *context, unsigned char value)

  Writes the device waveform clamping setting.
- LIBCTEXPORT int libct_rd_waveform_clamping (libct_context_t *context)

  Reads the device waveform clamping setting.
- LIBCTEXPORT int libct_rd_vitals_filter (libct_context_t *context)

  Reads the current filter settings to enable or disable filtering outlier vitals measurements.
- LIBCTEXPORT int libct_wrt_vitals_filter (libct_context_t *context, unsigned char value)

  Writes the filter settings to enable or disable filtering outlier vitals measurements.
- LIBCTEXPORT int libct_wrt_simulation_mode (libct_context_t *context, unsigned char mode)

  Writes the device simulation mode.
- LIBCTEXPORT int libct_wrt_motion_timeout (libct_context_t *context, int timeout)

  Writes the motion tolerance timeout parameter.
- LIBCTEXPORT int libct_rd_motion_timeout (libct_context_t *context)

  Reads the motion tolerance timeout parameter.
- LIBCTEXPORT int libct_rd_persistent_log (libct_context_t *context)

  Reads the device log messages.

6.3.1 Detailed Description

The group of auxiliary functions and macros available to parse, read and write additional Caretaker device data.

6.3.2 Macro Definition Documentation

6.3.2.1 #define libct_dp_count(*data, memb*) (data)->memb.count

Returns the count of data points of the specified member array contained in stream data received at the application.

Parameters

data	The stream data received in your on_data_received() application callback.
memb	The stream data member whose data point count is being queried.

Returns

The extracted data point on success, and null on failure.

6.3.2.2 #define libct_get_dp(*data, memb, pos*)

Value:

```
({\
    __typeof__((data)->memb.datapoints[0]) *dp = NULL; \
    if ( (data)->memb.count && (pos) < (data)->memb.count ) { \
        dp = &(data)->memb.datapoints[(pos)]; \
    } \
    (dp);\
})
```

Extract a single data point from the specified member array contained in stream data received at the application.

Parameters

data	The stream data received in your on_data_received() application callback.
memb	The stream data member name of the data point to extract.
pos	The position of the data point to extract.

Returns

The extracted data point on success, and null on failure.

6.3.2.3 #define libct_get_last_dp(*data, memb*) libct_get_dp(data, memb, (data)->memb.count-1)

Extract the newest data point from the specified member array contained in stream data received at the application.

Parameters

data	The stream data received in your on_data_received() application callback.
memb	The stream data member name of the data point to extract.

Returns

The extracted data point on success, and null on failure.

6.3.2.4 #define libct_get_first_dp(*data, memb*) libct_get_dp(data, memb, 0)

Extract the oldest data point from the specified member array contained in stream data received at the application.

Parameters

data	The stream data received in your on_data_received() application callback.
memb	The stream data member name of the data point to extract.

Returns

The extracted data point on success, and null on failure.

6.3.2.5 #define for_each_dp(*data, idx, dp, memb*) for(idx=0; (idx<(data)->memb.count) && (dp=((data)->memb.datapoints) ? &(data)->memb.datapoints[idx] : NULL); idx++)

Iterate over data points of the specified member array to extract from stream data received at the application.

Parameters

data	The stream data received in your on_data_received() application callback.
idx	Iterator variable of type unsigned integer.

dp	Pointer variable of type corresponding to the memb argument.
memb	The stream data member name of the data points to extract.

6.3.2.6 #define libct_inc_cuff_pressure(*context*) libct_adjust_cuff_pressure((context), 1)

Increments the cuff pressure in 10 mmHg increment.

Parameters

context	The context returned from libct_init().

Returns

Success if a request was queued to be sent to the device, and error otherwise.

6.3.2.7 #define libct_dec_cuff_pressure(*context*) libct_adjust_cuff_pressure((context), 0)

Decrements the cuff pressure in 10 mmHg increment.

Parameters

context	The context returned from libct_init().

Returns

Success if a request was queued to be sent to the device, and error otherwise.

6.3.3 Function Documentation

6.3.3.1 LIBCTEXPORT libct_device_t* libct_get_device (libct_context_t * *context*)

Returns the device handle.

Call this function to get a pointer to the device instance associated with the context.

Parameters

context	The context returned from libct_init().

Returns

Device object or NULL;

See Also

The device APIs

6.3.3.2 LIBCTEXPORT void libct_set_app_specific_data (libct_context_t * *context*, void * *data*)

Sets application specific data that can be retrieved and used later in the callbacks.

Basically, this function provides the means to bind your callback application code with the library context. For example, set application instance data after initializing the library, and then retrieve the instance data using libct_get_app_specific_data() inside the callbacks.

```
// QT main window initialization
MainWindow::MainWindow(QWidget *parent) :
   QMainWindow(parent),
   ui(new Ui::MainWindow)
{
     ui->setupUi(this);
     setWindowTitle(QString("SampleApp"));

     // initialize ui (code not shown)

     // initialize the library
     int status = libct_init(&context, &init_data, &callbacks);
     if ( LIBCT_FAILED(status) ) {
         // handle error
     }

     // set main window instance to act upon in the callbacks
     libct_set_app_specific_data(context, this);
}

// libcaretaker callback - called to notify data from the device
void on_data_received_cb(libct_context_t* context, libct_device_t* device,
      libct_stream_data_t* data) {
     MainWindow* window = (MainWindow*) libct_get_app_specific_data(context);

     // display the most recent vitals
     libct_vitals_t* vitals = libct_get_last_dp(data, vitals);
     if ( vitals && vitals->valid ) {
         window->setHr(vitals->heart_rate);
         window->setRes(vitals->respiration);
         window->setMap(vitals->map);
         window->setBp(vitals->systolic, vitals->diastolic);
     }
}
```

Parameters

context	The context returned from libct_init().
data	Generic pointer to the application specific data, or null to clear the existing pointer.

6.3.3.3 LIBCTEXPORT void* libct_get_app_specific_data (libct_context_t * *context*)

Retrieve application specific data.

Retrieves application specific data last set with libct_set_app_specific_data().

Parameters

context	The context returned from libct_init().

6.3.3.4 LIBCTEXPORT const char* libct_get_version_string (void)

Get the library version info.

Call this function to get the library version.

Returns

The library version string.

6.3.3.5 LIBCTEXPORT const char* libct_get_build_date_string (void)

Get the library build date and time string.

Call this function to get the library build date and time.

Returns

The library build date string.

6.3.3.6 LIBCTEXPORT void libct_set_log_level (int *level*)

Sets the library log level.

Call this function to set the log level to increase or decrease log messages verbosity.

Parameters

level	One of the following log levels. • 0 - shows all logs, most verbose • 1 - shows only info, warning, and error logs • 2 - shows only warning and error logs • 3 - shows only error logs, least verbose

6.3.3.7 LIBCTEXPORT int libct_recalibrate (libct_context_t * *context*)

Re-calibrates the device.

Call this function sometime later after calling libct_start_measuring() to force vital signs re-calibration at the device while taking measurements.

Parameters

context	The context returned from libct_init().

Returns

Success if a request was queued to be sent to the device, and error otherwise.

6.3.3.8 LIBCTEXPORT int libct_adjust_cuff_pressure (libct_context_t * *context*, int *direction*)

Adjusts the cuff pressure in 10 mmHg increment/decrement.

Note

The macros libct_inc_cuff_pressure() and libct_dec_cuff_pressure() simplify this function so you should use them instead.

Parameters

context	The context returned from libct_init().
direction	Zero - Decrement pressure. Nonzero - Increment pressures

See Also

libct_inc_cuff_pressure()
libct_dec_cuff_pressure()

Returns

Success if a request was queued to be sent to the device, and error otherwise.

6.3.3.9 LIBCTEXPORT int libct_rd_cuff_pressure (libct_context_t * *context*)

Reads the cuff pressure.

Results will be notified later with on_rd_cuff_pressure_rsp().

Parameters

context	The context returned from libct_init().

Returns

Success if a request was queued to be sent to the device, and error otherwise.

6.3.3.10 LIBCTEXPORT int libct_vent_cuff (libct_context_t * *context*)

Deflates the cuff pressure.

Results will be notified later with on_vent_cuff_rsp().

Parameters

context	The context returned from libct_init().

Returns

Success if a request was queued to be sent to the device, and error otherwise.

6.3.3.11 LIBCTEXPORT int libct_clr_status (libct_context_t * *context*)

Clears the device status.

Results will be notified later with on_clr_status_rsp().

Parameters

context	The context returned from libct_init().

Returns

Success if a request was queued to be sent to the device, and error otherwise.

6.3.3.12 LIBCTEXPORT int libct_diag_flush (libct_context_t * *context*)

Invoke the device diagnostic plumbing tree flush.

Results will be notified later with on_diag_flush_rsp().

Parameters

context	The context returned from libct_init().

Returns

Success if a request was queued to be sent to the device, and error otherwise.

6.3.3.13 LIBCTEXPORT int libct_wrt_snr_min (libct_context_t * *context*, int *snr*)

Writes the device noise filter parameter.

Results will be notified later with on_wrt_snr_min_rsp().

Parameters

context	The context returned from libct_init().
snr	The minimum signal-to-noise value. Valid range [0, 100].

Returns

Success if a request was queued to be sent to the device, and error otherwise.

6.3.3.14 LIBCTEXPORT int libct_rd_snr_min (libct_context_t * *context*)

Reads the device noise filter parameter.

Results will be notified later with on_rd_snr_min_rsp().

Parameters

context	The context returned from libct_init().

Returns

Success if a request was queued to be sent to the device, and error otherwise.

6.3.3.15 LIBCTEXPORT int libct_wrt_display_state (libct_context_t * *context*, unsigned char *state*)

Turns the device display on/off.

Results will be notified later with on_rd_snr_min_rsp().

Parameters

context	The context returned from libct_init().
state	Display state to write: 0 = off, 1 = on.

Returns

Success if a request was queued to be sent to the device, and error otherwise.

6.3.3.16 LIBCTEXPORT int libct_rd_display_state (libct_context_t * *context*)

Reads the device display state.

Results will be notified later with on_rd_snr_min_rsp().

Parameters

context	The context returned from libct_init().

Returns

Success if a request was queued to be sent to the device, and error otherwise.

6.3.3.17 LIBCTEXPORT int libct_wrt_recal_itvl (libct_context_t * *context,* unsigned int *itvl*)

Writes the recalibration interval.

Results will be notified later with on_wrt_recal_itvl_rsp().

Parameters

context	The context returned from libct_init().
itvl	The recalibration interval in minutes. The acceptable range is [30, 240].

Returns

Success if a request was queued to be sent to the device, and error otherwise.

6.3.3.18 LIBCTEXPORT int libct_rd_recal_itvl (libct_context_t * *context*)

Reads the recalibration interval.

Results will be notified later with on_rd_recal_itvl_rsp().

Parameters

context	The context returned from libct_init().

Returns

Success if a request was queued to be sent to the device, and error otherwise.

6.3.3.19 LIBCTEXPORT int libct_wrt_waveform_clamping (libct_context_t * *context*, unsigned char *value*)

Writes the device waveform clamping setting.

Status will be notified later with on_wrt_waveform_clamping().

Parameters

context	The context returned from libct_init().
value	Clamp setting: 1 = ON, 0 = OFF.

Returns

Success if a request was queued to be sent to the device, and error otherwise.

6.3.3.20 LIBCTEXPORT int libct_rd_waveform_clamping (libct_context_t * *context*)

Reads the device waveform clamping setting.

Status will be notified later with on_rd_waveform_clamping().

Parameters

context	The context returned from libct_init().

Returns

Success if a request was queued to be sent to the device, and error otherwise.

6.3.3.21 LIBCTEXPORT int libct_rd_vitals_filter (libct_context_t * *context*)

Reads the current filter settings to enable or disable filtering outlier vitals measurements.

Status will be notified later with on_rd_vitals_filter().

Parameters

context	The context returned from libct_init().

Returns

Success if a request was queued to be sent to the device, and error otherwise.

6.3.3.22 LIBCTEXPORT int libct_wrt_vitals_filter (libct_context_t * *context*, unsigned char *value*)

Writes the filter settings to enable or disable filtering outlier vitals measurements.

Status will be notified later with on_wrt_vitals_filter().

Parameters

context	The context returned from libct_init().
value	Filter setting value: 1 = Enable, 0 = Disable.

Returns

Success if a request was queued to be sent to the device, and error otherwise.

6.3.3.23 LIBCTEXPORT int libct_wrt_simulation_mode (libct_context_t * *context*, unsigned char *mode*)

Writes the device simulation mode.

Note

The device does not provide real-time data when simulation mode is enabled. Hard-coded numeric and waveform data (i.e., synthetic data) is provided. As such, simulation mode should be enabled for demonstration and test purposes only.

Parameters

context	The context returned from libct_init().
mode	Simulation mode: 1 = Enable, 0 = Disable.

Returns

Success if a request was queued to be sent to the device, and error otherwise.

6.3.3.24 LIBCTEXPORT int libct_wrt_motion_timeout (libct_context_t * *context*, int *timeout*)

Writes the motion tolerance timeout parameter.

Parameters

context	The context returned from libct_init().
timeout	Time out in seconds. Acceptable range [0, 30]

Returns

Success if a request was queued to be sent to the device, and error otherwise.

6.3.3.25 LIBCTEXPORT int libct_rd_motion_timeout (libct_context_t * *context*)

Reads the motion tolerance timeout parameter.

Status will be notified later with on_rd_motion_timeout().

Parameters

context	The context returned from libct_init().

Returns

Success if a request was queued to be sent to the device, and error otherwise.

6.3.3.26 LIBCTEXPORT int libct_rd_persistent_log (libct_context_t * *context*)

Reads the device log messages.

Status will be notified later with on_rd_persistent_log().

Note

Reading the device log is a slow request so the results will be delayed by many seconds.

Parameters

context	The context returned from libct_init().

Returns

Success if a request was queued to be sent to the device, and error otherwise.

Chapter 7

Data Structure Documentation

7.1 libct_app_callbacks_t Struct Reference

Structure used to provide asynchronous notifications to the application.

Data Fields

- void(∗ on_device_discovered)(libct_context_t ∗context, libct_device_t ∗device)

  Function pointer to the application callback to receive scan notifications in response to calling libct_start_discovery().
- void(∗ on_discovery_timedout)(libct_context_t ∗context)

  Function pointer to the application callback to receive timeout notification in response to calling libct_start_discovery().
- void(∗ on_discovery_failed)(libct_context_t ∗context, int error)

  Function pointer to the application callback to receive error notification in response to calling libct_start_discovery().
- void(∗ on_device_connected_not_ready)(libct_context_t ∗context, libct_device_t ∗device)

  Function pointer to the application callback to receive early connection notification in response to calling libct_connect().
- void(∗ on_device_connected_ready)(libct_context_t ∗context, libct_device_t ∗device)

  Function pointer to the application callback to receive connection notification in response to calling libct_connect().
- void(∗ on_connect_error)(libct_context_t ∗context, libct_device_t ∗device, const char ∗error)

  Function pointer to the application callback to receive error notification in response to calling libct_connect().
- void(∗ on_connect_timedout)(libct_context_t ∗context, libct_device_t ∗device)

  Function pointer to the application callback to receive timed out notification in response to calling libct_connect().
- void(∗ on_device_disconnected)(libct_context_t ∗context, libct_device_t ∗device)

  Function pointer to the application callback to receive disconnect notification.
- void(∗ on_start_monitoring)(libct_context_t ∗context, libct_device_t ∗device, int status)

  Function pointer to the application callback to receive notification in response to calling libct_start_monitoring().
- void(∗ on_stop_monitoring)(libct_context_t ∗context, libct_device_t ∗device, int status)

  Function pointer to the application callback to receive notification in response to calling libct_stop_monitoring().
- void(∗ on_start_measuring)(libct_context_t ∗context, libct_device_t ∗device, int status)

  Function pointer to the application callback to receive notification in response to calling libct_start_measuring().
- void(∗ on_stop_measuring)(libct_context_t ∗context, libct_device_t ∗device, int status)

  Function pointer to the application callback to receive notification in response to calling libct_stop_measuring().
- void(∗ on_data_received)(libct_context_t ∗context, libct_device_t ∗device, libct_stream_data_t ∗data)

  Function pointer to the application callback to receive data notifications.
- void(∗ on_data_error)(libct_context_t ∗context, libct_device_t ∗device, const char ∗error)

Function pointer to the application callback to receive data error notification.

- void(∗ on_rd_snr_min_rsp)(libct_context_t ∗context, libct_device_t ∗device, int snr, int status)

  Function pointer to the application callback to receive status in response to calling libct_rd_snr_min().
- void(∗ on_wrt_snr_min_rsp)(libct_context_t ∗context, libct_device_t ∗device, int status)

  Function pointer to the application callback to receive status in response to calling libct_wrt_snr_min().
- void(∗ on_rd_display_state_rsp)(libct_context_t ∗context, libct_device_t ∗device, unsigned char state, int status)

  Function pointer to the application callback to receive status in response to calling libct_rd_display_state().
- void(∗ on_wrt_display_state_rsp)(libct_context_t ∗context, libct_device_t ∗device, int status)

  Function pointer to the application callback to receive status in response to calling libct_wrt_display_state().
- void(∗ on_rd_recal_itvl_rsp)(libct_context_t ∗context, libct_device_t ∗device, unsigned int itvl, int status)

  Function pointer to the application callback to receive status in response to calling libct_rd_recal_itvl().
- void(∗ on_wrt_recal_itvl_rsp)(libct_context_t ∗context, libct_device_t ∗device, int status)

  Function pointer to the application callback to receive status in response to calling libct_wrt_recal_itvl().
- void(∗ on_rd_cuff_pressure_rsp)(libct_context_t ∗context, libct_device_t ∗device, int pressure, int status)

  Function pointer to the application callback to receive status in response to calling libct_rd_cuff_pressure().
- void(∗ on_vent_cuff_rsp)(libct_context_t ∗context, libct_device_t ∗device, int status)

  Function pointer to the application callback to receive status in response to calling libct_vent_cuff().
- void(∗ on_clr_status_rsp)(libct_context_t ∗context, libct_device_t ∗device, int status)

  Function pointer to the application callback to receive status in response to calling libct_clr_status().
- void(∗ on_diag_flush_rsp)(libct_context_t ∗context, libct_device_t ∗device, int status)

  Function pointer to the application callback to receive status in response to calling libct_diag_flush().
- void(∗ on_wrt_waveform_clamping)(libct_context_t ∗context, libct_device_t ∗device, int status)

  Function pointer to the application callback to receive status in response to calling libct_wrt_waveform_clamping().
- void(∗ on_rd_waveform_clamping)(libct_context_t ∗context, libct_device_t ∗device, unsigned char value, int status)

  Function pointer to the application callback to receive status in response to calling libct_rd_waveform_clamping().
- void(∗ on_rd_vitals_filter)(libct_context_t ∗context, libct_device_t ∗device, int value, int status)

  Function pointer to the application callback to receive status in response to calling libct_rd_vitals_filter().
- void(∗ on_wrt_vitals_filter)(libct_context_t ∗context, libct_device_t ∗device, int status)

  Function pointer to the application callback to receive status in response to calling libct_wrt_vitals_filter().
- void(∗ on_rd_motion_timeout)(libct_context_t ∗context, libct_device_t ∗device, int timeout, int status)

  Function pointer to the application callback to receive status in response to calling libct_rd_motion_timeout().
- void(∗ on_rd_persistent_log)(libct_context_t ∗context, libct_device_t ∗device, const char ∗log, unsigned int len, int status)

  Function pointer to the application callback to receive status in response to calling libct_rd_persistent_log().

7.1.1 Detailed Description

Structure used to provide asynchronous notifications to the application.

This structure is a container of function pointers to your application callback functions to receive asynchronous notifications. Note you are not required to implement all callback functions. Instead, initialize the libct_app_callbacks_t object to zeros and then set only the function pointers to the callback functions you care about. However, you must implement at least the following callbacks to connect and receive data from the device.

- on_device_discovered()
- on_device_connected_ready()
- on_device_disconnected()

- on_data_received()

IMPORTANT: When implementing a callback function, you must include **LIBCTAPI** in the function signature to specify the calling convention. This ensures the application and library are using the same calling convention to prevent corrupting the stack. Some platforms, such as Windows, support many calling conventions and **LIBCTAPI** will be set to the default one. If you don't specify **LIBCTAPI** in the callback implementation, the application source code may not compile or serious failures may occur at runtime due to stack corruption. See the sample implementations included with the member descriptions below for details.

7.1.2 Field Documentation

7.1.2.1 void(∗ libct_app_callbacks_t::on_device_discovered)(libct_context_t ∗context, libct_device_t ∗device)

Function pointer to the application callback to receive scan notifications in response to calling libct_start_discovery().

These notifications are sent to the application during device discovery to notify a discovered device when scanning for Caretaker devices.

The following example illustrates a sample implementation of the callback. Note you must specify **LIBCTAPI** in the function signature.

```
void LIBCTAPI on_device_discovered(libct_context_t* context,
      libct_device_t* device)
{
     const char* address = libct_device_get_address(device);
     const char* name = libct_device_get_name(device);

    // do something
}
```

And you can set the function pointer as follows.

```
libct_app_callbacks_t callbacks = {0};
callbacks.on_device_discovered = on_device_discovered;
```

Parameters

context	The context returned from libct_init().
device	The discovered device. **Note:** This device object will be recycled when the callback returns so do not save the device pointer. Instead, make a copy of the device info if needed. Also, since a connection is not established to the device at this point, device functions that require the connection to be established will not return anything useful. You can only call the following functions safely on the device object passed to this callback. • libct_device_t::get_address() • libct_device_t::get_name()

See Also

on_discovery_timedout()
on_discovery_failed()

7.1.2.2 void(* libct_app_callbacks_t::on_discovery_timedout)(libct_context_t *context)

Function pointer to the application callback to receive timeout notification in response to calling libct_start_discovery().

The following example illustrates a sample implementation of the callback. Note you must specify **LIBCTAPI** in the function signature.

```
void LIBCTAPI on_discovery_timedout(libct_context_t* context)
{
    // do something
}
```

And you can set the function pointer as follows.

```
libct_app_callbacks_t callbacks = {0};
callbacks.on_discovery_timedout = on_discovery_timedout;
```

Parameters

context	The context returned from libct_init().

See Also

on_device_discovered()
on_discovery_failed()

7.1.2.3 void(* libct_app_callbacks_t::on_discovery_failed)(libct_context_t *context, int error)

Function pointer to the application callback to receive error notification in response to calling libct_start_discovery().

The following example illustrates a sample implementation of the callback. Note you must specify **LIBCTAPI** in the function signature.

```
void LIBCTAPI on_discovery_failed(libct_context_t* context, int error)
{
    // do something
}
```

And you can set the function pointer as follows.

```
libct_app_callbacks_t callbacks = {0};
callbacks.on_discovery_failed = on_discovery_failed;
```

Parameters

context	The context returned from libct_init().
error	Generic error code describing the failure.

See Also

on_device_discovered()
on_discovery_timedout()

7.1.2.4 void(* libct_app_callbacks_t::on_device_connected_not_ready)(libct_context_t *context, libct_device_t *device)

Function pointer to the application callback to receive early connection notification in response to calling libct_connect().

This is an early notification to allow the application to update the device connection status, however, the device is not ready for IO at this stage of the connection sequence.

The following example illustrates a sample implementation of the callback. Note you must specify **LIBCTAPI** in the function signature.

```
void LIBCTAPI on_device_connected_not_ready(
      libct_context_t* context, libct_device_t* device)
{
    // update connection status
}
```

And you can set the function pointer as follows.

```
libct_app_callbacks_t callbacks = {0};
callbacks.on_device_connected_not_ready =
      on_device_connected_not_ready;
```

Parameters

context	The context returned from libct_init().
device	The connected device. **Note:** Since the connection is established to the device at this point there is no restriction on which device functions you can call to obtain information about the device.

See Also

on_connect_error()
on_connect_timedout()

7.1.2.5 void(* libct_app_callbacks_t::on_device_connected_ready)(libct_context_t *context, libct_device_t *device)

Function pointer to the application callback to receive connection notification in response to calling libct_connect().

At this stage, the device is ready for IO and the application can issue requests to the device.

The following example illustrates a sample implementation of the callback. Note you must specify **LIBCTAPI** in the function signature.

```
void LIBCTAPI on_device_connected_ready(libct_context_t* context,
      libct_device_t* device)
{
    // do something
}
```

And you can set the function pointer as follows.

```
libct_app_callbacks_t callbacks = {0};
callbacks.on_device_connected_ready =
      on_device_connected_ready;
```

Parameters

context	The context returned from libct_init().
device	The connected device. **Note:** Since the connection is established to the device at this point there is no restriction on which device functions you can call to obtain information about the device.

See Also

on_connect_error()
on_connect_timedout()

7.1.2.6 void(* libct_app_callbacks_t::on_connect_error)(libct_context_t *context, libct_device_t *device, const char *error)

Function pointer to the application callback to receive error notification in response to calling libct_connect().

The following example illustrates a sample implementation of the callback. Note you must specify **LIBCTAPI** in the function signature..

```
void LIBCTAPI on_connect_error(libct_context_t* context,
      libct_device_t* device, const char* error)
{
    // do something
}
```

And you can set the function pointer as follows.

```
libct_app_callbacks_t callbacks = {0};
callbacks.on_connect_error = on_connect_error;
```

Parameters

context	The context returned from libct_init().
device	The affected device. **Note:** Since there is no connection to the device the device functions requiring a connection will not return anything useful.
error	String describing error.

See Also

on_device_connected_not_ready()
on_device_connected_ready()
on_connect_timedout()

7.1.2.7 void(* libct_app_callbacks_t::on_connect_timedout)(libct_context_t *context, libct_device_t *device)

Function pointer to the application callback to receive timed out notification in response to calling libct_connect().

The following example illustrates a sample implementation of the callback. Note you must specify **LIBCTAPI** in the function signature.

```
void LIBCTAPI on_connect_timedout(libct_context_t* context,
      libct_device_t* device)
{
    // do something
}
```

And you can set the function pointer as follows.

```
libct_app_callbacks_t callbacks = {0};
callbacks.on_connect_timedout = on_connect_timedout;
```

Parameters

context	The context returned from libct_init().
device	The affected device. **Note:** Since there is no connection to the device the device functions requiring a connection will not return anything useful.

See Also

on_device_connected_not_ready()
on_device_connected_ready()
on_connect_error()

7.1.2.8 void(∗ libct_app_callbacks_t::on_device_disconnected)(libct_context_t ∗context, libct_device_t ∗device)

Function pointer to the application callback to receive disconnect notification.

The disconnect notification is notified after the connection is established with the device and the connection is lost such as when the device moves out of range and disconnects.

The disconnect notification is also notified when the application calls libct_disconnect() to disconnect explicitly. However, the notification to the application is not guaranteed to occur for this scenario, which should be okay since the application initiated the disconnect.

IMPORTANT: The application must not call libct_disconnect() or libct_deinit() from within this or any library callback function. Callbacks are called from internal library threads that these functions attempt to kill. As such, libct_disconnect() and libct_deinit() must only be called from an application thread.

The following example illustrates a sample implementation of the callback. Note you must specify **LIBCTAPI** in the function signature.

```
void LIBCTAPI on_device_disconnected(libct_context_t* context,
      libct_device_t* device)
{
    // do something
}
```

And you can set the function pointer as follows.

```
libct_app_callbacks_t callbacks = {0};
callbacks.on_device_disconnected = on_device_disconnected;
```

Parameters

context	The context returned from libct_init().
device	The disconnected device.

7.1.2.9 void(* libct_app_callbacks_t::on_start_monitoring)(libct_context_t *context, libct_device_t *device, int status)

Function pointer to the application callback to receive notification in response to calling libct_start_monitoring().

This notification will be sent only once (one-shot) to notify success or failure after calling libct_start_monitoring(). If success, the application on_data_received() will be notified repeatedly with data from the device until libct_stop_monitoring() is called subsequently or the device is disconnected.

The following example illustrates a sample implementation of the callback. Note you must specify **LIBCTAPI** in the function signature.

```
void LIBCTAPI on_start_monitoring(libct_context_t* context,
      libct_device_t* device, int status)
{
    // do something
}
```

And you can set the function pointer as follows.

```
libct_app_callbacks_t callbacks = {0};
callbacks.on_start_monitoring = on_start_monitoring;
```

Parameters

context	The context returned from libct_init().
device	The device being monitored.
status	Status indicating success or failure: zero on success and non-zero otherwise.

See Also

on_data_received()

7.1.2.10 void(* libct_app_callbacks_t::on_stop_monitoring)(libct_context_t *context, libct_device_t *device, int status)

Function pointer to the application callback to receive notification in response to calling libct_stop_monitoring().

This notification will be sent only once (one-shot) to notify success or failure after calling libct_stop_monitoring(). If success, the application on_data_received() will stop receiving data notifications.

The following example illustrates a sample implementation of the callback. Note you must specify **LIBCTAPI** in the function signature.

```
void LIBCTAPI on_stop_monitoring(libct_context_t* context,
      libct_device_t* device, int status)
{
    // do something
}
```

And you can set the function pointer as follows.

```
libct_app_callbacks_t callbacks = {0};
callbacks.on_stop_monitoring = on_stop_monitoring;
```

Parameters

context	The context returned from libct_init().
device	The device being monitored.
status	Status indicating success or failure: zero on success and non-zero otherwise.

7.1.2.11 void(* libct_app_callbacks_t::on_start_measuring)(libct_context_t *context, libct_device_t *device, int status)

Function pointer to the application callback to receive notification in response to calling libct_start_measuring().

This notification will be sent only once (one-shot) to notify success or failure after calling libct_start_measuring(). If success, the device will begin pulse decomposition analysis (PDA) and vital sign measurements (blood pressure, heart rate, etc) will be notified to the application on_data_received() callback.

The following example illustrates a sample implementation of the callback. Note you must specify **LIBCTAPI** in the function signature.

```
void LIBCTAPI on_start_measuring(libct_context_t* context,
      libct_device_t* device, int status)
{
    // do something
}
```

And you can set the function pointer as follows.

```
libct_app_callbacks_t callbacks = {0};
callbacks.on_start_measuring = on_start_measuring;
```

Parameters

context	The context returned from libct_init().
device	The device providing measurements.
status	Status indicating success or failure: zero on success and non-zero otherwise.

7.1.2.12 void(* libct_app_callbacks_t::on_stop_measuring)(libct_context_t *context, libct_device_t *device, int status)

Function pointer to the application callback to receive notification in response to calling libct_stop_measuring().

This notification will be sent only once (one-shot) to notify success or failure after calling libct_stop_measuring(). If success, the device will stop pulse decomposition analysis (PDA) and vital sign measurements (blood pressure, heart rate, etc) will stop.

The following example illustrates a sample implementation of the callback. Note you must specify **LIBCTAPI** in the function signature.

```
void LIBCTAPI on_stop_measuring(libct_context_t* context,
      libct_device_t* device, int status)
{
    // do something
}
```

And you can set the function pointer as follows.

```
libct_app_callbacks_t callbacks = {0};
callbacks.on_stop_measuring = on_stop_measuring;
```

Parameters

context	The context returned from libct_init().
device	The device providing measurements.
status	Status indicating success or failure: zero on success and non-zero otherwise.

7.1.2.13 void(* libct_app_callbacks_t::on_data_received)(libct_context_t *context, libct_device_t *device, libct_stream_data_t *data)

Function pointer to the application callback to receive data notifications.

These notifications are sent repeatedly to the application to hand-off data received from the device some time after calling libct_start_monitoring() successfully.

Data notified via this callback depends on the monitor flags passed to libct_start_monitoring() and whether or not libct_start_measuring() was called to start taking vital sign measurements. See libct_stream_data_t for data details.

The following example illustrates a sample implementation of the callback. Note you must specify **LIBCTAPI** in the function signature.

```
void LIBCTAPI on_data_received(libct_context_t* context,
      libct_device_t* device, libct_stream_data_t* data)
{
    // handle data
}
```

And you can set the function pointer as follows.

```
libct_app_callbacks_t callbacks = {0};
callbacks.on_data_received = on_data_received;
```

Parameters

context	The context returned from libct_init().
device	The device originating the data.
data	Stream packet containing the data received from the device. **NOTE:** The stream data packet is created with dynamic memory that will be freed after the callback returns. So you should not save pointer(s) to the data, instead copy individual fields into application memory as needed if you need to access it after on_data_received() returns. Do not copy the entire libct_stream_data_t structure as it is a structure of pointers and doing so will be saving pointers to freed memory after the callback returned.

See Also

on_data_error()

7.1.2.14 void(∗ libct_app_callbacks_t::on_data_error)(libct_context_t ∗context, libct_device_t ∗device, const char ∗error)

Function pointer to the application callback to receive data error notification.

This notification is sent if the library encounters error receiving or processing data.

The following example illustrates a sample implementation of the callback. Note you must specify **LIBCTAPI** in the function signature.

```
void LIBCTAPI on_data_error(libct_context_t* context,
      libct_device_t* device, const char* error)
{
    // handle error
}
```

And you can set the function pointer as follows.

```
libct_app_callbacks_t callbacks = {0};
callbacks.on_data_error = on_data_error;
```

Parameters

context	The context returned from libct_init().
device	The affected device.
error	String describing the error.

7.1.2.15 void(∗ libct_app_callbacks_t::on_rd_snr_min_rsp)(libct_context_t ∗context, libct_device_t ∗device, int snr, int status)

Function pointer to the application callback to receive status in response to calling libct_rd_snr_min().

This notification will be sent only once (one-shot) to notify success or failure after calling libct_rd_snr_min().

The following example illustrates a sample implementation of the callback. Note you must specify **LIBCTAPI** in the function signature.

```
void LIBCTAPI on_rd_snr_min_rsp(libct_context_t* context,
      libct_device_t* device, int snr, int status)
{
    // do something
}
```

And you can set the function pointer as follows.

```
libct_app_callbacks_t callbacks = {0};
callbacks.on_rd_snr_min_rsp = on_rd_snr_min_rsp;
```

Parameters

context	The context returned from libct_init().
device	The device associated with the context.
snr	Minimum signal-to-noise value on success.
status	Status indicating success or failure: zero on success and non-zero otherwise.

7.1.2.16 void(* libct_app_callbacks_t::on_wrt_snr_min_rsp)(libct_context_t *context, libct_device_t *device, int status)

Function pointer to the application callback to receive status in response to calling libct_wrt_snr_min().

This notification will be sent only once (one-shot) to notify success or failure after calling libct_wrt_snr_min().

The following example illustrates a sample implementation of the callback. Note you must specify **LIBCTAPI** in the function signature.

```
void LIBCTAPI on_wrt_snr_min_rsp(libct_context_t* context,
      libct_device_t* device, int status)
{
    // do something
}
```

And you can set the function pointer as follows.

```
libct_app_callbacks_t callbacks = {0};
callbacks.on_wrt_snr_min_rsp = on_wrt_snr_min_rsp;
```

Parameters

context	The context returned from libct_init().
device	The device associated with the context.
status	Status indicating success or failure: zero on success and non-zero otherwise.

7.1.2.17 void(* libct_app_callbacks_t::on_rd_display_state_rsp)(libct_context_t *context, libct_device_t *device, unsigned char state, int status)

Function pointer to the application callback to receive status in response to calling libct_rd_display_state().

This notification will be sent only once (one-shot) to notify success or failure after calling libct_rd_display_state().

The following example illustrates a sample implementation of the callback. Note you must specify **LIBCTAPI** in the function signature.

```
void LIBCTAPI on_rd_display_state_rsp(libct_context_t* context,
      libct_device_t* device, unsigned char state, int status)
{
    // do something
}
```

And you can set the function pointer as follows.

```
libct_app_callbacks_t callbacks = {0};
callbacks.on_rd_display_state_rsp =
      on_rd_display_state_rsp;
```

Parameters

context	The context returned from libct_init().
device	The device associated with the context.
state	The device display state on success: 0 = off, 1 = on.
status	Status indicating success or failure: zero on success and non-zero otherwise.

7.1.2.18 void(* libct_app_callbacks_t::on_wrt_display_state_rsp)(libct_context_t *context, libct_device_t *device, int status)

Function pointer to the application callback to receive status in response to calling libct_wrt_display_state().

This notification will be sent only once (one-shot) to notify success or failure after calling libct_wrt_display_state().

The following example illustrates a sample implementation of the callback. Note you must specify **LIBCTAPI** in the function signature.

```
void LIBCTAPI on_wrt_display_state_rsp(libct_context_t* context,
      libct_device_t* device, int status)
{
    // do something
}
```

And you can set the function pointer as follows.

```
libct_app_callbacks_t callbacks = {0};
callbacks.on_wrt_display_state_rsp =
      on_wrt_display_state_rsp;
```

Parameters

context	The context returned from libct_init().
device	The device associated with the context.
status	Status indicating success or failure: zero on success and non-zero otherwise.

7.1.2.19 void(* libct_app_callbacks_t::on_rd_recal_itvl_rsp)(libct_context_t *context, libct_device_t *device, unsigned int itvl, int status)

Function pointer to the application callback to receive status in response to calling libct_rd_recal_itvl().

This notification will be sent only once (one-shot) to notify success or failure after calling libct_rd_recal_itvl().

The following example illustrates a sample implementation of the callback. Note you must specify **LIBCTAPI** in the function signature.

```
void LIBCTAPI on_rd_recal_itvl_rsp(libct_context_t* context,
      libct_device_t* device, unsigned int itvl, int status)
{
    // do something
}
```

And you can set the function pointer as follows.

```
libct_app_callbacks_t callbacks = {0};
callbacks.on_rd_recal_itvl_rsp = on_rd_recal_itvl_rsp;
```

Parameters

context	The context returned from libct_init().
device	The device associated with the context.
itvl	The recalibration interval in minutes on success.
status	Status indicating success or failure: zero on success and non-zero otherwise.

7.1.2.20 void(* libct_app_callbacks_t::on_wrt_recal_itvl_rsp)(libct_context_t *context, libct_device_t *device, int status)

Function pointer to the application callback to receive status in response to calling libct_wrt_recal_itvl().

This notification will be sent only once (one-shot) to notify success or failure after calling libct_wrt_recal_itvl().

The following example illustrates a sample implementation of the callback. Note you must specify **LIBCTAPI** in the function signature.

```
void LIBCTAPI on_wrt_recal_itvl_rsp(libct_context_t* context,
      libct_device_t* device, int status)
{
    // do something
}
```

And you can set the function pointer as follows.

```
libct_app_callbacks_t callbacks = {0};
callbacks.on_wrt_recal_itvl_rsp = on_wrt_recal_itvl_rsp;
```

Parameters

context	The context returned from libct_init().
device	The device associated with the context.
status	Status indicating success or failure: zero on success and non-zero otherwise.

7.1.2.21 void(* libct_app_callbacks_t::on_rd_cuff_pressure_rsp)(libct_context_t *context, libct_device_t *device, int pressure, int status)

Function pointer to the application callback to receive status in response to calling libct_rd_cuff_pressure().

This notification will be sent only once (one-shot) to notify success or failure after calling libct_rd_cuff_pressure().

The following example illustrates a sample implementation of the callback. Note you must specify **LIBCTAPI** in the function signature.

```
void LIBCTAPI on_rd_cuff_pressure_rsp(libct_context_t* context,
      libct_device_t* device, int status)
{
    // do something
}
```

And you can set the function pointer as follows.

```
libct_app_callbacks_t callbacks = {0};
callbacks.on_rd_cuff_pressure_rsp =
      on_rd_cuff_pressure_rsp;
```

Parameters

context	The context returned from libct_init().
device	The device associated with the context.
pressure	The cuff pressure in mmHg on success.
status	Status indicating success or failure: zero on success and non-zero otherwise.

7.1.2.22 void(* libct_app_callbacks_t::on_vent_cuff_rsp)(libct_context_t *context, libct_device_t *device, int status)

Function pointer to the application callback to receive status in response to calling libct_vent_cuff().

This notification will be sent only once (one-shot) to notify success or failure after calling libct_vent_cuff().

The following example illustrates a sample implementation of the callback. Note you must specify **LIBCTAPI** in the function signature.

```
void LIBCTAPI on_vent_cuff_rsp(libct_context_t* context,
      libct_device_t* device, int status)
{
    // do something
}
```

And you can set the function pointer as follows.

```
libct_app_callbacks_t callbacks = {0};
callbacks.on_vent_cuff_rsp = on_vent_cuff_rsp;
```

Parameters

context	The context returned from libct_init().
device	The device associated with the context.
status	Status indicating success or failure: zero on success and non-zero otherwise.

7.1.2.23 void(* libct_app_callbacks_t::on_clr_status_rsp)(libct_context_t *context, libct_device_t *device, int status)

Function pointer to the application callback to receive status in response to calling libct_clr_status().

This notification will be sent only once (one-shot) to notify success or failure after calling libct_clr_status().

The following example illustrates a sample implementation of the callback. Note you must specify **LIBCTAPI** in the function signature.

```
void LIBCTAPI on_clr_status_rsp(libct_context_t* context,
      libct_device_t* device, int status)
{
    // do something
}
```

And you can set the function pointer as follows.

```
libct_app_callbacks_t callbacks = {0};
callbacks.on_clr_status_rsp = on_clr_status_rsp;
```

Parameters

context	The context returned from libct_init().
device	The device associated with the context.
status	Status indicating success or failure: zero on success and non-zero otherwise.

7.1.2.24 void(* libct_app_callbacks_t::on_diag_flush_rsp)(libct_context_t *context, libct_device_t *device, int status)

Function pointer to the application callback to receive status in response to calling libct_diag_flush().

This notification will be sent only once (one-shot) to notify success or failure after calling libct_diag_flush().

The following example illustrates a sample implementation of the callback. Note you must specify **LIBCTAPI** in the function signature.

```
void LIBCTAPI on_diag_flush_rsp(libct_context_t* context,
      libct_device_t* device, int status)
{
    // do something
}
```

And you can set the function pointer as follows.

```
libct_app_callbacks_t callbacks = {0};
callbacks.on_diag_flush_rsp = on_diag_flush_rsp;
```

Parameters

context	The context returned from libct_init().
device	The device associated with the context.
status	Status indicating success or failure: zero on success and non-zero otherwise.

7.1.2.25 void(* libct_app_callbacks_t::on_wrt_waveform_clamping)(libct_context_t *context, libct_device_t *device, int status)

Function pointer to the application callback to receive status in response to calling libct_wrt_waveform_clamping().

This notification will be sent only once (one-shot) to notify success or failure after calling libct_wrt_waveform_-clamping().

The following example illustrates a sample implementation of the callback. Note you must specify **LIBCTAPI** in the function signature.

```
void LIBCTAPI on_wrt_waveform_clamping(libct_context_t* context,
                                       libct_device_t* device
                                       int status)
{
    // do something
}
```

And you can set the function pointer as follows.

```
libct_app_callbacks_t callbacks = {0};
callbacks.on_wrt_waveform_clamping =
      on_wrt_waveform_clamping;
```

Parameters

context	The context returned from libct_init().
device	The device associated with the context.
status	Status indicating success or failure: zero on success and non-zero otherwise.

7.1.2.26 void(∗ libct_app_callbacks_t::on_rd_waveform_clamping)(libct_context_t ∗context, libct_device_t ∗device, unsigned char value, int status)

Function pointer to the application callback to receive status in response to calling libct_rd_waveform_clamping().

This notification will be sent only once (one-shot) to notify success or failure after calling libct_wrt_waveform_-clamping().

The following example illustrates a sample implementation of the callback. Note you must specify **LIBCTAPI** in the function signature.

```
void LIBCTAPI on_rd_waveform_clamping(libct_context_t* context,
                                      libct_device_t* device,
                                      unsigned char value
                                      int status)
{
    // do something
}
```

And you can set the function pointer as follows.

```
libct_app_callbacks_t callbacks = {0};
callbacks.on_rd_waveform_clamping =
      on_rd_waveform_clamping;
```

Parameters

context	The context returned from libct_init().
device	The device associated with the context.
value	Clamp setting: 1 = ON, 0 = OFF
status	Status indicating success or failure: zero on success and non-zero otherwise.

7.1.2.27 void(∗ libct_app_callbacks_t::on_rd_vitals_filter)(libct_context_t ∗context, libct_device_t ∗device, int value, int status)

Function pointer to the application callback to receive status in response to calling libct_rd_vitals_filter().

This notification will be sent only once (one-shot) to notify success or failure after calling libct_rd_vitals_filter().

The following example illustrates a sample implementation of the callback. Note you must specify **LIBCTAPI** in the function signature.

```
void LIBCTAPI on_rd_vitals_filter(libct_context_t* context,
      libct_device_t* device, int status)
{
    // do something
}
```

And you can set the function pointer as follows.

```
libct_app_callbacks_t callbacks = {0};
callbacks.on_rd_vitals_filter = on_rd_vitals_filter;
```

Parameters

context	The context returned from libct_init().
device	The device associated with the context.
value	The median filter value when success. 0 = Disabled, 1 = Enabled.
status	Status indicating success or failure: zero on success and nonzero otherwise.

7.1.2.28 void(* libct_app_callbacks_t::on_wrt_vitals_filter)(libct_context_t *context, libct_device_t *device, int status)

Function pointer to the application callback to receive status in response to calling libct_wrt_vitals_filter().

This notification will be sent only once (one-shot) to notify success or failure after calling libct_wrt_vitals_filter().

The following example illustrates a sample implementation of the callback. Note you must specify **LIBCTAPI** in the function signature.

```
void LIBCTAPI on_wrt_vitals_filter(libct_context_t* context,
      libct_device_t* device, int status)
{
    // do something
}
```

And you can set the function pointer as follows.

```
libct_app_callbacks_t callbacks = {0};
callbacks.on_wrt_vitals_filter = on_wrt_vitals_filter;
```

Parameters

context	The context returned from libct_init().
device	The device associated with the context.
status	Status indicating success or failure: zero on success and non-zero otherwise.

7.1.2.29 void(* libct_app_callbacks_t::on_rd_motion_timeout)(libct_context_t *context, libct_device_t *device, int timeout, int status)

Function pointer to the application callback to receive status in response to calling libct_rd_motion_timeout().

This notification will be sent only once (one-shot) to notify success or failure after calling libct_rd_motion_timeout().

The following example illustrates a sample implementation of the callback. Note you must specify **LIBCTAPI** in the function signature.

```
void LIBCTAPI on_rd_motion_timeout(libct_context_t* context,
      libct_device_t* device, int status, int timeout)
{
    // do something
}
```

And you can set the function pointer as follows.

```
libct_app_callbacks_t callbacks = {0};
callbacks.on_rd_motion_timeout = on_rd_motion_timeout;
```

Parameters

context	The context returned from libct_init().
device	The device associated with the context.
status	Status indicating success or failure: zero on success and non-zero otherwise.
timeout	Motion timeout value in seconds.

7.1.2.30 void(* libct_app_callbacks_t::on_rd_persistent_log)(libct_context_t *context, libct_device_t *device, const char *log, unsigned int len, int status)

Function pointer to the application callback to receive status in response to calling libct_rd_persistent_log().

This notification will be sent only once (one-shot) to notify success or failure after calling libct_rd_persistent_log().

The following example illustrates a sample implementation of the callback. Note you must specify **LIBCTAPI** in the function signature.

```
void LIBCTAPI on_rd_persistent_log(libct_context_t* context,
                                   libct_device_t* device,
                                   const char* log,
                                   unsigned int len,
                                   int timeout)
{
    // do something
}
```

And you can set the function pointer as follows.

```
libct_app_callbacks_t callbacks = {0};
callbacks.on_rd_persistent_log = on_rd_persistent_log;
```

Parameters

context	The context returned from libct_init().
device	The device associated with the context.
log	The device persistent log.
len	Log length.
status	Status indicating success or failure: zero on success and non-zero otherwise.

The documentation for this struct was generated from the following file:

- caretaker.h

7.2 libct_battery_info_t Class Reference

Battery info data point within the libct_stream_data_t packet.

Data Fields

- bool valid

  The other fields are valid when this field is non-zero (true) and invalid otherwise.
- int voltage

  The battery voltage in millivolts.
- unsigned int timestamp

  Time stamp from the device associated with the data.

7.2.1 Detailed Description

Battery info data point within the libct_stream_data_t packet.

7.2.2 Field Documentation

7.2.2.1 int libct_battery_info_t::voltage

The battery voltage in millivolts.

7.2.2.2 unsigned int libct_battery_info_t::timestamp

Time stamp from the device associated with the data.

The documentation for this class was generated from the following file:

- caretaker.h

7.3 libct_bp_settings_t Class Reference

Structure to write the caretaker manual blood pressure settings.

Data Fields

- unsigned short systolic
  Systolic pressure setting used for blood pressure calibration.
- unsigned short diastolic
  Diastolic pressure setting used for blood pressure calibration.

7.3.1 Detailed Description

Structure to write the caretaker manual blood pressure settings.

7.3.2 Field Documentation

7.3.2.1 unsigned short libct_bp_settings_t::systolic

Systolic pressure setting used for blood pressure calibration.

Acceptable range [30, 250].

7.3.2.2 unsigned short libct_bp_settings_t::diastolic

Diastolic pressure setting used for blood pressure calibration.

Acceptable range [10, 150].

The documentation for this class was generated from the following file:

- caretaker.h

7.4 libct_cal_curve_t Class Reference

Calibration curve data point within the libct_stream_data_t packet.

Data Fields

- bool valid
  The other fields are valid when this field is non-zero (true) and invalid otherwise.
- int data_id
  Data ID.
- float val1
  Value 1.
- float val2
  Value 2.
- float val3
  Value 3.
- char * alternateData
  Alternate Data - does not come from device.

7.4.1 Detailed Description

Calibration curve data point within the libct_stream_data_t packet.

Note

The cal curve data is for internal use or research only

7.4.2 Field Documentation

7.4.2.1 bool libct_cal_curve_t::valid

The other fields are valid when this field is non-zero (true) and invalid otherwise.

7.4.2.2 int libct_cal_curve_t::data_id

Data ID.

7.4.2.3 float libct_cal_curve_t::val1

Value 1.

7.4.2.4 float libct_cal_curve_t::val2

Value 2.

7.4.2.5 float libct_cal_curve_t::val3

Value 3.

The documentation for this class was generated from the following file:

- caretaker.h

7.5 libct_cal_t Struct Reference

Structure used to pass calibration data to libct_start_measuring().

Data Fields

- int type
  Calibration type.
- union {
  Calibration data.

7.5.1 Detailed Description

Structure used to pass calibration data to libct_start_measuring().

7.5.2 Field Documentation

7.5.2.1 int libct_cal_t::type

Calibration type.

Set to one of the calibration types.

7.5.2.2 short libct_cal_t::posture

Patient posture.

Set to one of the patient postures.

7.5.2.3 union { ... } libct_cal_t::config

Calibration data.

The documentation for this struct was generated from the following file:

- caretaker.h

7.6 libct_cal_type_t Class Reference

The Caretaker calibration types.

7.6.1 Detailed Description

The Caretaker calibration types.

The documentation for this class was generated from the following file:

- caretaker.h

7.7 libct_context_t Class Reference

An opaque type representing a library instance associated with (or bound to) a device the application is monitoring.

7.7.1 Detailed Description

An opaque type representing a library instance associated with (or bound to) a device the application is monitoring.

The context is used internally to manage the library instance so its data structure is not exposed to the application. As such, the application cannot create a library context explicitly. A library context can only be created by calling libct_init() to initialize a library instance, which sets the context pointer passed in the first argument. If the call succeeded, the application can use the context to call other library functions, but must call libct_deinit() to destroy the context when it is no longer needed. Destroying the context releases resources that were allocated when the context was initialized, so the application is required to call libct_deinit() to release the context, and not doing so will leak system resources.

```
// Initialize library instance, which returns a device context pointer.
libct_context_t* context = NULL;
int status = libct_init(&context, &init_data, &app_callbacks);
if ( LIBCT_FAILED(status) {
    // Handle error
    return status;
}

// Connect to a device and monitor data (code not shown)

// Destroy context
 libct_deinit(context);
```

The documentation for this class was generated from the following file:

- caretaker.h

7.8 libct_cuff_pressure_t Class Reference

Cuff pressure data point within the libct_stream_data_t packet.

Data Fields

- bool valid
  The other fields are valid when this field is non-zero (true) and invalid otherwise.
- int value
  cuff pressure actual value.
- int target
  cuff pressure target value.
- int snr
  signal to noise ratio.
- unsigned int timestamp
  Time stamp from the device associated with the data.

7.8.1 Detailed Description

Cuff pressure data point within the libct_stream_data_t packet.

7.8.2 Field Documentation

7.8.2.1 bool libct_cuff_pressure_t::valid

The other fields are valid when this field is non-zero (true) and invalid otherwise.

7.8.2.2 int libct_cuff_pressure_t::value

cuff pressure actual value.

7.8.2.3 int libct_cuff_pressure_t::target

cuff pressure target value.

7.8.2.4 int libct_cuff_pressure_t::snr

signal to noise ratio.

7.8.2.5 unsigned int libct_cuff_pressure_t::timestamp

Time stamp from the device associated with the data.

The documentation for this class was generated from the following file:

- caretaker.h

7.9 libct_device_class_t Class Reference

Classes of devices that can be monitored by this library.

7.9.1 Detailed Description

Classes of devices that can be monitored by this library.

The documentation for this class was generated from the following file:

- caretaker.h

7.10 libct_device_state_t Class Reference

The Caretaker device states.

7.10.1 Detailed Description

The Caretaker device states.

The documentation for this class was generated from the following file:

- caretaker.h

7.11 libct_device_status_t Class Reference

Device status data point within the libct_stream_data_t packet.

Data Fields

- bool valid

  The other fields are valid when this field is non-zero (true) and invalid otherwise.
- long long value

  Integer value representing logically OR of all status flags, which essentially is the raw value from the device.
- bool pda_enabled

  An indicator of whether the system PDA measurement system is enabled.
- bool simulation_enabled

  An indicator of whether the system is in simulation mode.
- bool pressure_control_indicator

  An indicator of whether the system is currently running closed loop pressure control.
- bool inflated_indicator

  An indicator of whether the system has been inflated to pressure.
- bool clock_wrap_around

  The system clock (time since reset) has wrapped around its index.

- bool battery_voltage_low

  The battery voltage sensor has indicated the battery is near drop-out.
- bool critical_temperature

  The on-board temperature sensor has detected critically high temperature.
- bool pump_overrun

  The pump has violated an overrun condition.
- bool ble_temperature_sensor_paired

  The BLE Temperature Sensor is paired and actively communicating with CareTaker.
- bool ble_handheld_paired

  The a BLE hand-held device is paired and actively communicating with CareTaker.
- bool ble_stream_control

  TThe current stream control status bit of the BLE stream.
- bool cellular_control

  The current stream control status bit of the cellular stream.
- bool serial_stream_control

  The current stream control status bit of the serial stream.
- bool auto_cal_mode

  The system has been started and running in auto-calibration mode.
- bool manual_cal_mode

  The system has been started and running in manual calibration mode.
- bool motion_event

  The system is having trouble getting a good reading due to too much motion.
- bool poor_signal

  The system failed to calibrate or timed out process signals so measurements were aborted.
- bool data_valid

  There are valid vital signs measurements.
- bool calibrating

  The system is currently calibrating the blood pressure system.
- bool calibrated

  The system has current valid calibration.
- bool beta_processing

  The system has finished finding the oscillometric curve and is processing the beta (offset) value.
- bool inflate_failed

  Cuff did not inflate to expected value within timeout.
- bool calibration_failed

  The calibration values were out of range or oscillometric curve had invalid shape.
- bool calibration_offset_failed

  Too much movement.
- bool no_pulse_timeout

  The systems has gone greater than 3 minutes without a valid heart beat.
- bool cuff_too_loose

  The calibration pump up identified the cuff was too loose.
- bool cuff_too_tight

  The calibration pump up identified the cuff was too tight.
- bool weak_signal

  Calibration oscillometric curve amplitude is too weak to verify reading.
- bool bad_cuff

  The cuff is not holding pressure as expected.
- bool ble_adv

  The Bluetooth module is advertising.
- bool recal_soon

An automatic recalibration will be occurring shortly.

- bool too_many_fails

  Auto-calibration failed too many consecutive times try manual calibration.
- short autocal_pct

  Auto-calibration percentage complete.
- bool charging

  The device is charging.
- bool chargeComplete

  Charging complete.
- short posture

  Posture.
- bool invalid_data_entry

  Invalid input received in the last command.

7.11.1 Detailed Description

Device status data point within the libct_stream_data_t packet.

7.11.2 Field Documentation

7.11.2.1 bool libct_device_status_t::simulation_enabled

An indicator of whether the system is in simulation mode.

7.11.2.2 bool libct_device_status_t::inflated_indicator

An indicator of whether the system has been inflated to pressure.

7.11.2.3 bool libct_device_status_t::clock_wrap_around

The system clock (time since reset) has wrapped around its index.

7.11.2.4 bool libct_device_status_t::battery_voltage_low

The battery voltage sensor has indicated the battery is near drop-out.

7.11.2.5 bool libct_device_status_t::pump_overrun

The pump has violated an overrun condition.

7.11.2.6 bool libct_device_status_t::ble_temperature_sensor_paired

The BLE Temperature Sensor is paired and actively communicating with CareTaker.

7.11.2.7 bool libct_device_status_t::ble_stream_control

TThe current stream control status bit of the BLE stream.

7.11.2.8 bool libct_device_status_t::manual_cal_mode

The system has been started and running in manual calibration mode.

7.11.2.9 bool libct_device_status_t::motion_event

The system is having trouble getting a good reading due to too much motion.

7.11.2.10 bool libct_device_status_t::poor_signal

The system failed to calibrate or timed out process signals so measurements were aborted.

7.11.2.11 bool libct_device_status_t::data_valid

There are valid vital signs measurements.

This is used to notify the GUI if data should be displayed or hidden.

7.11.2.12 bool libct_device_status_t::calibration_offset_failed

Too much movement.

The calibration offset calculation failed to identify pulses due to movement.

7.11.2.13 bool libct_device_status_t::weak_signal

Calibration oscillometric curve amplitude is too weak to verify reading.

7.11.2.14 bool libct_device_status_t::bad_cuff

The cuff is not holding pressure as expected.

7.11.2.15 bool libct_device_status_t::ble_adv

The Bluetooth module is advertising.

7.11.2.16 bool libct_device_status_t::recal_soon

An automatic recalibration will be occurring shortly.

7.11.2.17 bool libct_device_status_t::too_many_fails

Auto-calibration failed too many consecutive times try manual calibration.

7.11.2.18 bool libct_device_status_t::invalid_data_entry

Invalid input received in the last command.

The documentation for this class was generated from the following file:

- caretaker.h

7.12 libct_device_t Class Reference

Handle used to identify a connected device the application is monitoring.

Data Fields

- int(∗ get_state)(struct libct_device_t ∗thiz)

  Return a device state enumeration representing the current state of the library context that is associated with this device.
- int(∗ get_class)(struct libct_device_t ∗thiz)

  Return the device class that was set in the initialization data passed to libct_init().
- const char ∗(∗ get_name)(struct libct_device_t ∗thiz)

  Return the device manufacturer friendly name.
- const char ∗(∗ get_address)(struct libct_device_t ∗thiz)

  Return the device address.
- const char ∗(∗ get_serial_number)(struct libct_device_t ∗thiz)

  Return the device serial number.
- const libct_version_t ∗(∗ get_hw_version)(struct libct_device_t ∗thiz)

  Return the device hardware version.
- const libct_version_t ∗(∗ get_fw_version)(struct libct_device_t ∗thiz)

  Return the device firmware version.
- libct_context_t ∗(∗ get_context)(struct libct_device_t ∗thiz)

  Return the library context bound to this device.

7.12.1 Detailed Description

Handle used to identify a connected device the application is monitoring.

The device handle is used to identify and aggregate general information about a connected device, such as the device name, address, serial number, etc., that the application can query. Note each device handle is associated with a library context and you can retrieve it anytime with libct_get_device() passing the context as argument. As such, you should not hold on to device handles in your application code as they may change when the devices they are associated with become disconnected.

The device handle primary purpose is to identify data notified to your application callbacks.

7.12.2 Field Documentation

7.12.2.1 int(∗ libct_device_t::get_state)(struct libct_device_t ∗thiz)

Return a device state enumeration representing the current state of the library context that is associated with this device.

You would invoke the function as follows, but note a convenience macro with simpler interface than the function is available.

```
// These two calls are similar, but the second is simpler.
// (1) Use the device function.
int state = device->get_state(device);

// (2) Or use macro with simpler interface.
int state = libct_device_get_state(device);
```

Parameters

thiz	The device instance.

See Also

libct_device_get_state()
libct_device_uninitialized()
libct_device_intialized()
libct_device_discovering()
libct_device_connecting()
libct_device_connected()
libct_device_disconnecting()
libct_device_disconnected()
libct_device_monitoring()
libct_device_measuring()

7.12.2.2 int(∗ libct_device_t::get_class)(struct libct_device_t ∗thiz)

Return the device class that was set in the initialization data passed to libct_init().

You would invoke the function as follows, but note a convenience macro with simpler interface than the function is available.

```
// These two calls are similar, but the second is simpler.
// (1) Use the device function.
int class = device->get_class(device);

// (2) Or use macro with simpler interface.
int class = libct_device_get_class(device);
```

Parameters

thiz	The device instance.

See Also

libct_device_get_class()

7.12.2.3 const char*(* libct_device_t::get_name)(struct libct_device_t *thiz)

Return the device manufacturer friendly name.

You would invoke the function as follows, but note a convenience macro with simpler interface than the function is available.

```
// These two calls are similar, but the second is simpler.
// (1) Use the device function.
const char* name = device->get_name(device);

// (2) Or use macro with simpler interface.
const char* name = libct_device_get_name(device);
```

Parameters

thiz	The device instance.

See Also

libct_device_get_name()

7.12.2.4 const char*(* libct_device_t::get_address)(struct libct_device_t *thiz)

Return the device address.

You would invoke the function as follows, but note a convenience macro with simpler interface than the function is available.

```
// These two calls are similar, but the second is simpler.
// (1) Use the device function.
const char* address = device->get_address(device);

// (2) Or use macro with simpler interface.
const char* address = libct_device_get_address(device);
```

Parameters

thiz	The device instance.

See Also

libct_device_get_address()

7.12.2.5 const char*(* libct_device_t::get_serial_number)(struct libct_device_t *thiz)

Return the device serial number.

You would invoke the function as follows, but note a convenience macro with simpler interface than the function is available.

```
// These two calls are similar, but the second is simpler.
// (1) Use the device function.
const char* sn = device->get_serial_number(device);

// (2) Or use macro with simpler interface.
const char* sn = libct_device_get_serial_number(device);
```

Parameters

thiz	The device instance.

See Also

libct_device_get_serial_number()

7.12.2.6 const libct_version_t*(* libct_device_t::get_hw_version)(struct libct_device_t *thiz)

Return the device hardware version.

You would invoke the function as follows, but note a convenience macro with simpler interface than the function is available.

```
// These two calls are similar, but the second is simpler.
// (1) Use the device function.
const libct_version_t* version = device->get_hw_version(device);

// (2) Or use macro with simpler interface.
const libct_version_t* version = libct_device_get_hw_version(
      device);
```

Parameters

thiz	The device instance.

See Also

libct_device_get_hw_version()

7.12.2.7 const libct_version_t*(* libct_device_t::get_fw_version)(struct libct_device_t *thiz)

Return the device firmware version.

You would invoke the function as follows, but note a convenience macro with simpler interface than the function is available.

```
// These two calls are similar, but the second is simpler.
// (1) Use the device function.
const libct_version_t* version = device->get_fw_version(device);

// (2) Or use macro with simpler interface.
const libct_version_t* version = libct_device_get_fw_version(
      device);
```

Parameters

thiz	The device instance.

See Also

libct_device_get_fw_version()

7.12.2.8 libct_context_t*(* libct_device_t::get_context)(struct libct_device_t *thiz)

Return the library context bound to this device.

You would invoke the function as follows, but note a convenience macro with simpler interface than the function is available.

```
// These two calls are similar, but the second is simpler.
// (1) Use the device function.
libct_context_t* context = device->get_context(device);

// (2) Or use macro with simpler interface.
libct_context_t* context  = libct_device_get_context(device);
```

Parameters

thiz	The device instance.

The documentation for this class was generated from the following file:

- caretaker.h

7.13 libct_init_data_t Struct Reference

Structure defining initialization data passed to libct_init().

Data Fields

- int device_class
  The device class.

7.13.1 Detailed Description

Structure defining initialization data passed to libct_init().

7.13.2 Field Documentation

7.13.2.1 int libct_init_data_t::device_class

The device class.

The documentation for this struct was generated from the following file:

- caretaker.h

7.14 libct_monitor_flags_t Class Reference

Data monitor flags passed to libct_start_monitoring()

7.14.1 Detailed Description

Data monitor flags passed to libct_start_monitoring()

The documentation for this class was generated from the following file:

- caretaker.h

7.15 libct_param_pulse_t Class Reference

Parametrized pulse data within the libct_stream_data_t packet.

Data Fields

- bool valid

  The other fields are valid when this field is non-zero (true) and invalid otherwise.
- short protocol_header

  Date transfer control byte used internally to assemble the data.
- short t0

  Pulse onset time (index).
- short t1

  First pulse peak time (index).
- short t2

  Second pulse peak time (index).
- short t3

  Third pulse peak time (index).
- int p0

  Integrated pulse onset value.
- int p1

  First integrated pulse peak value.
- int p2

  Second integrated pulse peak value.
- int p3

  Third integrated pulse peak value.
- short ibi

  Inter-beat interval (1/HR) in samples @ 500Hz.
- short as

  Arterial stiffness.
- short sqe

  Signal quality estimate.
- short pressure

  The most recent raw ADC cuff pressure.
- unsigned int time

  Relative system time of occurrence.
- int waveform_len

  The number of signed int8 snapshot data points.
- char waveform [0]

  The pulse snapshot waveform data.

7.15.1 Detailed Description

Parametrized pulse data within the libct_stream_data_t packet.

The parametrized pulse data is an aggregate of the pulse parameters and pulse snapshot waveform data.

7.15.2 Field Documentation

7.15.2.1 bool libct_param_pulse_t::valid

The other fields are valid when this field is non-zero (true) and invalid otherwise.

7.15.2.2 short libct_param_pulse_t::protocol_header

Date transfer control byte used internally to assemble the data.

7.15.2.3 short libct_param_pulse_t::t0

Pulse onset time (index).

7.15.2.4 short libct_param_pulse_t::t1

First pulse peak time (index).

7.15.2.5 short libct_param_pulse_t::t2

Second pulse peak time (index).

7.15.2.6 short libct_param_pulse_t::t3

Third pulse peak time (index).

7.15.2.7 int libct_param_pulse_t::p0

Integrated pulse onset value.

7.15.2.8 int libct_param_pulse_t::p1

First integrated pulse peak value.

7.15.2.9 int libct_param_pulse_t::p2

Second integrated pulse peak value.

7.15.2.10 int libct_param_pulse_t::p3

Third integrated pulse peak value.

7.15.2.11 short libct_param_pulse_t::ibi

Inter-beat interval (1/HR) in samples @ 500Hz.

7.15.2.12 short libct_param_pulse_t::as

Arterial stiffness.

7.15.2.13 short libct_param_pulse_t::sqe

Signal quality estimate.

7.15.2.14 short libct_param_pulse_t::pressure

The most recent raw ADC cuff pressure.

7.15.2.15 unsigned int libct_param_pulse_t::time

Relative system time of occurrence.

7.15.2.16 int libct_param_pulse_t::waveform_len

The number of signed int8 snapshot data points.

The documentation for this class was generated from the following file:

- caretaker.h

7.16 libct_posture_t Class Reference

Patient postures.

7.16.1 Detailed Description

Patient postures.

The documentation for this class was generated from the following file:

- caretaker.h

7.17 libct_pulse_ox_t Class Reference

Pulse oximetry data point within the libct_stream_data_t packet.

Data Fields

- bool valid

  The other fields are valid when this field is non-zero (true) and invalid otherwise.
- int sao2

  Blood oxygen level (percentage).
- int pulse_rate

  Pulse rate in beats per minute (30-200BPM).
- unsigned int timestamp

  Time stamp from the device associated with the data.

7.17.1 Detailed Description

Pulse oximetry data point within the libct_stream_data_t packet.

Note

Reserved for future use.

7.17.2 Field Documentation

7.17.2.1 int libct_pulse_ox_t::sao2

Blood oxygen level (percentage).

7.17.2.2 int libct_pulse_ox_t::pulse_rate

Pulse rate in beats per minute (30-200BPM).

7.17.2.3 unsigned int libct_pulse_ox_t::timestamp

Time stamp from the device associated with the data.

The documentation for this class was generated from the following file:

- caretaker.h

7.18 libct_pulse_t Class Reference

Raw or integrated pulse data point within the libct_stream_data_t packet.

Data Fields

- bool valid

  The other fields are valid when this field is non-zero (true) and invalid otherwise.
- short value

  Pulse value.
- unsigned int timestamp

  Counter value associated with the pulse value.

7.18.1 Detailed Description

Raw or integrated pulse data point within the libct_stream_data_t packet.

7.18.2 Field Documentation

7.18.2.1 unsigned int libct_pulse_t::timestamp

Counter value associated with the pulse value.

The documentation for this class was generated from the following file:

- caretaker.h

7.19 libct_pulse_waveform_t Struct Reference

Pulse waveform data returned from the device as a result of a previous read request.

Data Fields

- *Array of integrated waveform pulse data points.*

Array of parameterize pulse waveform data points.

long receive_time

Value of local clock (in milliseconds) when this stream packet was received.

7.19.1 Detailed Description

Pulse waveform data returned from the device as a result of a previous read request.

Note this data is returned only after an explicit read of raw pulse waveform data.

7.19.2 Field Documentation

7.19.2.1 long libct_pulse_waveform_t::receive_time

Value of local clock (in milliseconds) when this stream packet was received.

This time stamp is used to measure processing latency and for history logging. It differs from time stamp found in each data point generated by the remote device.

The documentation for this struct was generated from the following file:

- caretaker.h

7.20 libct_status_t Class Reference

Function return status codes.

7.20.1 Detailed Description

Function return status codes.

The documentation for this class was generated from the following file:

- caretaker.h

7.21 libct_stream_data_t Class Reference

This structure is used to hand-off data received from the remote device to the application.

Data Fields

- libct_device_t * device
  Reference to the device that generated this data.
- libct_device_status_t device_status
  Device status information.
- libct_battery_info_t battery_info
  Battery information.
- struct {
  libct_vitals_t * datapoints
  Array of vital sign datapoints.
  unsigned int count
  The count of datapoints.
  } vitals

  Array of vital sign data points.

- struct {
  libct_cuff_pressure_t * datapoints
  Array of cuff pressure data points.
  unsigned int count
  The count of data points.
  } cuff_pressure

  Array of cuff pressure data points.
- struct {
  libct_temperature_t * datapoints
  Array of temperature data points.
  unsigned int count
  The count of data points.
  } temperature

  Array of temperature data points.
- struct {
  libct_pulse_ox_t * datapoints
  Array of spo2 data points.
  unsigned int count
  The count of data points.
  } pulse_ox

  Array of pulse oximetry data points.
- struct {
  libct_vitals2_t * datapoints
  Array of secondary vital sign data points.
  unsigned int count
  The count of data points.
  } vitals2

  Array of secondary vital sign data points.
- struct {
  libct_pulse_t * datapoints
  Array of raw pulse (pulse rate) data points.
  unsigned int count
  The count of data points.
  } raw_pulse

  Array of raw pulse (pulse rate) waveform data points.
- struct {
  libct_pulse_t * datapoints
  Array of integrated pulse (pulse pressure) data points.
  unsigned int count
  The count of data points.
  } int_pulse

  Array of integrated pulse (pulse pressure) waveform data points.
- struct {
  libct_param_pulse_t * datapoints
  Array of parameterize pulse (pulse snapshot) data points.
  unsigned int count
  The count of data points.
  } param_pulse

  Array of parameterized pulse data.

- struct {
  libct_cal_curve_t * datapoints
  Array of calibration curve data points.
  unsigned int count
  The count of data points.
  } cal_curve

  Array of calibration curve data points.
- long receive_time

  Value of local clock (in milliseconds) when this stream packet was received.

7.21.1 Detailed Description

This structure is used to hand-off data received from the remote device to the application.

Data from the device is sent automatically after calling libct_start_monitoring() successfully, and delivered to your application via the on_data_received() callback function. This data structure is a container of arrays grouping one or more records of the same data type at different time instances. The various array data types are not produced coherently at the device so not all fields will be populated in stream data packets delivered to the application. If no data is available for a given array, the array data points field will be set to null and the count set to zero to signal no data.

The stream data packets notified to the application depends on the monitor flags passed to libct_start_monitoring() and whether or not libct_start_measuring() was called to start taking vital sign measurements. So you can control the data reported to the application by specifying only the monitoring flags corresponding to the data you care about.

With the exception of the device_status and battery_info data members that are not array fields, the following convenience macros are available to access array entries within the stream data packet. More details about usage is provided in the description for each stream data member where the macros apply.

- libct_dp_count()
- libct_get_dp()
- libct_get_first_dp()
- libct_get_last_dp()
- for_each_dp()

7.21.2 Field Documentation

7.21.2.1 libct_device_t* libct_stream_data_t::device

Reference to the device that generated this data.

7.21.2.2 libct_device_status_t libct_stream_data_t::device_status

Device status information.

7.21.2.3 libct_battery_info_t libct_stream_data_t::battery_info

Battery information.

7.21.2.4 libct_vitals_t* libct_stream_data_t::datapoints

Array of vital sign datapoints.

7.21.2.5 unsigned int libct_stream_data_t::count

The count of datapoints.

The count of data points.

7.21.2.6 struct { ... } libct_stream_data_t::vitals

Array of vital sign data points.

For convenience, you can use the macros libct_get_last_dp(), libct_get_first_dp(), and libct_get_dp() to extract a single vital sign data point from the stream packet like so.

```
libct_vitals_t* dp = libct_get_last_dp(data, vitals);
if ( dp && dp->valid ) {
    // use most recent vitals data point
}
```

Alternatively, you could iterate over all vital sign data points with the for_each_dp() macro like so.

```
libct_vitals_t* dp;
unsigned int idx;
for_each_dp(data, idx, dp, vitals) {
    if ( dp && dp->valid ) {
        // use vitals data point
    }
}
```

7.21.2.7 libct_cuff_pressure_t* libct_stream_data_t::datapoints

Array of cuff pressure data points.

7.21.2.8 struct { ... } libct_stream_data_t::cuff_pressure

Array of cuff pressure data points.

For convenience, you can use the macros libct_get_last_dp(), libct_get_first_dp(), and libct_get_dp() to extract a single cuff pressure data point from the stream packet like so.

```
libct_cuff_pressure_t* dp = libct_get_last_dp(data,
      cuff_pressure);
if ( dp && dp->valid ) {
    // use most recent cuff pressure data point
}
```

Alternatively, you could iterate over all cuff pressure data points with the for_each_dp() macro like so.

```
libct_cuff_pressure_t* dp;
unsigned int idx;
for_each_dp(data, idx, dp, cuff_pressure) {
    if ( dp && dp->valid ) {
        // use cuff pressure data point
    }
}
```

7.21.2.9 libct_temperature_t* libct_stream_data_t::datapoints

Array of temperature data points.

7.21.2.10 struct { ... } libct_stream_data_t::temperature

Array of temperature data points.

For convenience, you can use the macros libct_get_last_dp(), libct_get_first_dp(), and libct_get_dp() to extract a single temperature data point from the stream packet like so.

```
libct_temperature_t* dp = libct_get_last_dp(data,
      temperature);
if ( dp && dp->valid ) {
    // use most recent temperature data point
}
```

Alternatively, you could iterate over all temperature data points with the for_each_dp() macro like so.

```
libct_temperature_t* dp;
unsigned int idx;
for_each_dp(data, idx, dp, temperature) {
    if ( dp && dp->valid ) {
        // use temperature data point
    }
}
```

7.21.2.11 libct_pulse_ox_t* libct_stream_data_t::datapoints

Array of spo2 data points.

7.21.2.12 struct { ... } libct_stream_data_t::pulse_ox

Array of pulse oximetry data points.

For convenience, you can use the macros libct_get_last_dp(), libct_get_first_dp(), and libct_get_dp() to extract a single spo2 data point from the stream packet like so.

```
libct_pulse_ox_t* dp = libct_get_last_dp(data,
      pulse_ox);
if ( dp && dp->valid ) {
    // use most recent spo2 data point
}
```

Alternatively, you could iterate over all spo2 data points with the for_each_dp() macro like so.

```
libct_pulse_ox_t* dp;
unsigned int idx;
for_each_dp(data, idx, dp, pulse_ox) {
    if ( dp && dp->valid ) {
        // use spo2 data point
    }
}
```

7.21.2.13 libct_vitals2_t* libct_stream_data_t::datapoints

Array of secondary vital sign data points.

7.21.2.14 struct { ... } libct_stream_data_t::vitals2

Array of secondary vital sign data points.

Note

The secondary vitals are for internal use or research only.

For convenience, you can use the macros libct_get_last_dp(), libct_get_first_dp(), and libct_get_dp() to extract a single secondary vital sign data point from the stream packet.

```
libct_vitals2_t* dp = libct_get_last_dp(data,
      vitals2);
if ( dp && dp->valid ) {
    // use most recent secondary vitals data point
}
```

Alternatively, you could iterate over all secondary vital sign data points with the for_each_dp() macro like so.

```
libct_vitals2_t* dp;
unsigned int idx;
for_each_dp(data, idx, dp, vitals) {
    if ( dp && dp->valid ) {
        // use secondary vitals data point
    }
}
```

7.21.2.15 libct_pulse_t* libct_stream_data_t::datapoints

Array of raw pulse (pulse rate) data points.

Array of integrated pulse (pulse pressure) data points.

7.21.2.16 struct { ... } libct_stream_data_t::raw_pulse

Array of raw pulse (pulse rate) waveform data points.

For convenience, you could iterate over all raw pulse data points with the for_each_dp() macro like so.

```
libct_pulse_t* dp;
unsigned int idx;
for_each_dp(data, idx, dp, raw_pulse) {
    if ( dp && dp->valid ) {
        // use pulse data point
    }
}
```

7.21.2.17 struct { ... } libct_stream_data_t::int_pulse

Array of integrated pulse (pulse pressure) waveform data points.

For convenience, you could iterate over all integrated pulse data points with the for_each_dp() macro like so.

```
libct_pulse_t* dp;
unsigned int idx;
for_each_dp(data, idx, dp, int_pulse) {
    if ( dp && dp->valid ) {
        // use pulse data point
    }
}
```

7.21.2.18 libct_param_pulse_t* libct_stream_data_t::datapoints

Array of parameterize pulse (pulse snapshot) data points.

7.21.2.19 struct { ... } libct_stream_data_t::param_pulse

Array of parameterized pulse data.

The data is an aggregate of the pulse parameters and pulse snapshot waveform data.

For convenience, you can use the macros libct_get_last_dp(), libct_get_first_dp(), and libct_get_dp() to extract a single pulse snapshot from the stream packet like so.

```
libct_param_pulse_t* dp = libct_get_last_dp(data,
      param_pulse);
if ( dp && dp->valid ) {
    // use most recent pulse snapshot
}
```

Alternatively, you could iterate over all pulse snapshot data points with the for_each_dp() macro like so.

```
libct_param_pulse_t* dp;
unsigned int idx;
for_each_dp(data, idx, dp, pulse_param) {
    if ( dp && dp->valid ) {
        // use pulse snapshot
    }
}
```

7.21.2.20 libct_cal_curve_t* libct_stream_data_t::datapoints

Array of calibration curve data points.

7.21.2.21 struct { ... } libct_stream_data_t::cal_curve

Array of calibration curve data points.

Note

The cal curve data is for internal use or research only

For convenience, you can use the macros libct_get_last_dp(), libct_get_first_dp(), and libct_get_dp() to extract a single calibration curve data point from the stream packet like so.

```
libct_cal_curve_t* dp = libct_get_last_dp(data,
      cal_curve);
if ( dp && dp->valid ) {
    // use most recent calibration curve data point
}
```

Alternatively, you could iterate over all calibration curve data points with the for_each_dp() macro like so.

```
libct_cal_curve_t* dp;
unsigned int idx;
for_each_dp(data, idx, dp, cal_curve) {
    if ( dp && dp->valid ) {
        // use calibration curve data point
    }
}
```

7.21.2.22 long libct_stream_data_t::receive_time

Value of local clock (in milliseconds) when this stream packet was received.

This time stamp is used to measure processing latency and for history logging. It differs from time stamp found in each data point generated by the remote device.

The documentation for this class was generated from the following file:

- caretaker.h

7.22 libct_temperature_t Class Reference

Temperature data point within the libct_stream_data_t packet.

Data Fields

- bool valid

  The other fields are valid when this field is non-zero (true) and invalid otherwise.
- int value

  The temperature value.
- unsigned int timestamp

  Time stamp from the device associated with the data.

7.22.1 Detailed Description

Temperature data point within the libct_stream_data_t packet.

Note

Reserved for future use.

7.22.2 Field Documentation

7.22.2.1 int libct_temperature_t::value

The temperature value.

7.22.2.2 unsigned int libct_temperature_t::timestamp

Time stamp from the device associated with the data.

The documentation for this class was generated from the following file:

- caretaker.h

7.23 libct_version_t Struct Reference

CareTaker version information.

Data Fields

- int major
  major version number
- int minor
  minor version number
- int revision
  revision number
- int build
  build number

7.23.1 Detailed Description

CareTaker version information.

The documentation for this struct was generated from the following file:

- caretaker.h

7.24 libct_vitals2_t Class Reference

Secondary Vitals data point within the libct_stream_data_t packet.

Data Fields

- bool valid

  The other fields are valid when this field is non-zero (true) and invalid otherwise.
- unsigned short blood_volume

  Blood volume in mS.
- unsigned char cardiac_output

  Cardiac output in L/min.
- unsigned short ibi

  Inter-beat Interval in mS.
- unsigned short lvet

  Left ventricular ejection time.
- float p2p1

  P ratio.
- float hrComp

  hrComp
- float pr

  pr
- int reserved [7]

  Reserved for future use.
- unsigned int timestamp

  Time stamp from the device associated with the data.

7.24.1 Detailed Description

Secondary Vitals data point within the libct_stream_data_t packet.

Note

The secondary vitals are for internal use or research only.

7.24.2 Field Documentation

7.24.2.1 unsigned short libct_vitals2_t::blood_volume

Blood volume in mS.

7.24.2.2 unsigned int libct_vitals2_t::timestamp

Time stamp from the device associated with the data.

The documentation for this class was generated from the following file:

- caretaker.h

7.25 libct_vitals_t Class Reference

Vitals data point within the libct_stream_data_t packet.

Data Fields

- bool valid

  The other fields are valid when this field is non-zero (true) and invalid otherwise.
- bool bp_status

  An indicator if a valid blood pressure was found or if the algorithm failed.
- bool map_status

  An indicator of if a valid MAP measurement has been integrated.
- bool hr_status

  An indicator if a valid HR has been determined.
- bool respiration_status

  An indicator if a valid respiration reading was found.
- bool integration_error

  General catchall for integration errors.
- bool differentiation_error

  A discontinuity was detected in the differentiation.
- bool p12_finder_error

  Unable to locate P1 P2 within the pulse.
- bool p3_finder_eError

  Unable to locate P3 within the pulse.
- bool min_index_out_of_range

  The onset of the pulse was not found in the allowable window, so the values are being discarded.
- bool max_index_out_of_range

  The index of the minimum point in the integral was out of range.
- bool slope_out_of_range

  The slope correction of the integrated pulse was out of range.
- short systolic

  Systolic measurement.
- short diastolic

  Diastolic measurement.
- short map

  Mean arterial pressure value.
- short heart_rate

  Heart rate measurement.
- short respiration

  Respiration measurement.
- short as

  AS factor.
- short sqe

  Signal quality estimate (sqe).
- unsigned int timestamp

  Time stamp from the device associated with the data.

7.25.1 Detailed Description

Vitals data point within the libct_stream_data_t packet.

7.25.2 Field Documentation

7.25.2.1 bool libct_vitals_t::valid

The other fields are valid when this field is non-zero (true) and invalid otherwise.

7.25.2.2 bool libct_vitals_t::bp_status

An indicator if a valid blood pressure was found or if the algorithm failed.

True indicates pulse information is valid.

7.25.2.3 short libct_vitals_t::systolic

Systolic measurement.

7.25.2.4 short libct_vitals_t::diastolic

Diastolic measurement.

7.25.2.5 short libct_vitals_t::map

Mean arterial pressure value.

7.25.2.6 short libct_vitals_t::heart_rate

Heart rate measurement.

7.25.2.7 short libct_vitals_t::respiration

Respiration measurement.

7.25.2.8 short libct_vitals_t::as

AS factor.

7.25.2.9 short libct_vitals_t::sqe

Signal quality estimate (sqe).

Values are in the range [0, 1000], so the sqe can be expressed relatively as a percentage by dividing by 10, .i.e. sqe/10 %.

7.25.2.10 unsigned int libct_vitals_t::timestamp

Time stamp from the device associated with the data.

The documentation for this class was generated from the following file:

- caretaker.h