

CARETAKER LIBRARY API 1.5.3 REFERENCE MANUAL

Caretaker Medical

May 25, 2018

Contents

1	Introduction	1
2	Library Integration	3
3	Monitoring from Managed Application	7
4	Monitoring from Unmanaged Application	9
5	Module Index	15
5.1	Modules	15
6	Data Structure Index	17
6.1	Data Structures	17
7	Module Documentation	19
7.1	Unmanaged Device Information	19
7.1.1	Detailed Description	20
7.1.2	Macro Definition Documentation	20
7.1.2.1	libct_device_get_state	20
7.1.2.2	libct_device_uninitialized	20
7.1.2.3	libct_device_discovering	21
7.1.2.4	libct_device_connecting	21
7.1.2.5	libct_device_connected	21
7.1.2.6	libct_device_disconnecting	21
7.1.2.7	libct_device_disconnected	22
7.1.2.8	libct_device_monitoring	22

7.1.2.9	libct_device_measuring	22
7.1.2.10	libct_device_get_class	23
7.1.2.11	libct_device_get_name	23
7.1.2.12	libct_device_get_address	23
7.1.2.13	libct_device_get_serial_number	23
7.1.2.14	libct_device_get_hw_version	24
7.1.2.15	libct_device_get_fw_version	24
7.1.2.16	libct_device_get_context	24
7.2	Unmanaged Primary API	25
7.2.1	Detailed Description	25
7.2.2	Function Documentation	25
7.2.2.1	libct_init()	25
7.2.2.2	libct_deinit()	26
7.2.2.3	libct_start_discovery()	26
7.2.2.4	libct_stop_discovery()	27
7.2.2.5	libct_connect()	27
7.2.2.6	libct_disconnect()	28
7.2.2.7	libct_start_monitoring()	29
7.2.2.8	libct_stop_monitoring()	29
7.2.2.9	libct_start_measuring()	30
7.2.2.10	libct_stop_measuring()	30
7.3	Unmanaged Secondary API	31
7.3.1	Detailed Description	32
7.3.2	Macro Definition Documentation	32
7.3.2.1	libct_dp_count	32
7.3.2.2	libct_get_dp	33
7.3.2.3	libct_get_last_dp	33
7.3.2.4	libct_get_first_dp	34
7.3.2.5	for_each_dp	34
7.3.2.6	libct_inc_cuff_pressure	34

7.3.2.7	libct_dec_cuff_pressure	35
7.3.3	Function Documentation	35
7.3.3.1	libct_get_device()	35
7.3.3.2	libct_set_app_specific_data()	36
7.3.3.3	libct_get_app_specific_data()	36
7.3.3.4	libct_get_version_string()	37
7.3.3.5	libct_get_build_date_string()	37
7.3.3.6	libct_set_log_level()	37
7.3.3.7	libct_recalibrate()	38
7.3.3.8	libct_adjust_cuff_pressure()	38
7.3.3.9	libct_rd_cuff_pressure()	39
7.3.3.10	libct_vent_cuff()	39
7.3.3.11	libct_clr_status()	40
7.3.3.12	libct_diag_flush()	40
7.3.3.13	libct_wrt_snr_min()	40
7.3.3.14	libct_rd_snr_min()	41
7.3.3.15	libct_wrt_display_state()	41
7.3.3.16	libct_rd_display_state()	42
7.3.3.17	libct_wrt_recal_itvl()	42
7.3.3.18	libct_rd_recal_itvl()	42
7.3.3.19	libct_wrt_waveform_clamping()	43
7.3.3.20	libct_rd_waveform_clamping()	43
7.3.3.21	libct_rd_vitals_filter()	44
7.3.3.22	libct_wrt_vitals_filter()	44
7.3.3.23	libct_wrt_simulation_mode()	44
7.3.3.24	libct_wrt_motion_timeout()	45
7.3.3.25	libct_rd_motion_timeout()	45
7.3.3.26	libct_rd_persistent_log()	46
7.4	Managed API for .NET Applications	47
7.4.1	Detailed Description	47
7.4.2	Enumeration Type Documentation	47
7.4.2.1	ConnectionStatus	48
7.4.2.2	StartStatus	49
7.4.2.3	StopStatus	49
7.4.2.4	PatientPosture	50
7.5	Unmanaged Explicit Linking Example	51

8 Data Structure Documentation	53
8.1 Caretaker::BatteryStatus Class Reference	53
8.1.1 Detailed Description	53
8.1.2 Field Documentation	53
8.1.2.1 voltage	53
8.1.2.2 timestamp	54
8.2 Caretaker::CuffStatus Class Reference	54
8.2.1 Detailed Description	54
8.2.2 Field Documentation	54
8.2.2.1 actualPressure	54
8.2.2.2 targetPressure	55
8.2.2.3 signalToNoise	55
8.2.2.4 timestamp	55
8.3 Caretaker::Device Class Reference	55
8.3.1 Detailed Description	58
8.3.2 Constructor & Destructor Documentation	58
8.3.2.1 Device()	58
8.3.2.2 ~Device()	59
8.3.3 Member Function Documentation	59
8.3.3.1 ReleaseResources()	59
8.3.3.2 StartScan()	59
8.3.3.3 ConnectToSerialNumber()	60
8.3.3.4 ConnectToMacAddress()	60
8.3.3.5 ConnectToAny()	61
8.3.3.6 Disconnect()	61
8.3.3.7 StartAutoCal()	62
8.3.3.8 StartManualCal()	62
8.3.3.9 Stop()	62
8.3.3.10 Calibrating()	63
8.3.3.11 Calibrated()	63

8.3.3.12	CalibrationFailed()	63
8.3.3.13	GetFirmwareVersion()	63
8.3.3.14	GetHardwareVersion()	64
8.3.3.15	GetDeviceStatus()	64
8.3.3.16	GetBatteryStatus()	64
8.3.3.17	GetCuffStatus()	64
8.3.3.18	GetPrimaryVitals()	64
8.3.3.19	GetSecondaryVitals()	65
8.3.3.20	GetPulseRateWaveformDataPoints()	65
8.3.3.21	GetPulsePressureWaveformDataPoints()	65
8.3.3.22	IncrementCuffPressure()	65
8.3.3.23	DecrementCuffPressure()	66
8.3.3.24	VentCuff()	66
8.3.3.25	PeformDiagnosticsFlush()	66
8.3.3.26	WriteSnrMinimum()	66
8.3.3.27	ReadSnrMinimum()	67
8.3.3.28	WriteMotionTolerance()	67
8.3.3.29	ReadMotionTolerance()	68
8.3.3.30	TurnDisplayOff()	68
8.3.3.31	TurnDisplayOn()	68
8.3.3.32	ReadDisplayState()	69
8.3.3.33	Recalibrate()	69
8.3.3.34	WriteRecalibrationInterval()	69
8.3.3.35	ReadRecalibrationInterval()	70
8.3.3.36	WriteWaveformClampSetting()	70
8.3.3.37	ReadWaveformClampSetting()	71
8.3.3.38	WriteVitalsFilterSetting()	71
8.3.3.39	ReadVitalsFilterSetting()	71
8.3.3.40	ReadPersistentLog()	72
8.3.3.41	SetLibraryLogLevel()	72

8.3.3.42	EnableSimulationMode()	72
8.3.3.43	ClearStatus()	73
8.4	Caretaker::DeviceObserver Class Reference	73
8.4.1	Detailed Description	74
8.4.2	Member Function Documentation	75
8.4.2.1	OnDeviceDiscovered()	75
8.4.2.2	OnConnectionStatus()	75
8.4.2.3	OnStartStatus()	75
8.4.2.4	OnStopStatus()	76
8.4.2.5	OnPulseRateWaveformDataPoints()	76
8.4.2.6	OnPulsePressureWaveformDataPoints()	76
8.4.2.7	OnParameterizedPulse()	77
8.4.2.8	OnDeviceStatus()	77
8.4.2.9	OnBatteryStatus()	78
8.4.2.10	OnCuffStatus()	78
8.4.2.11	OnPrimaryVitals()	78
8.4.2.12	OnSecondaryVitals()	78
8.4.2.13	OnReadSnrMinimum()	79
8.4.2.14	OnReadDisplayState()	79
8.4.2.15	OnReadRecalibrationInterval()	80
8.4.2.16	OnReadVitalsFilterSetting()	80
8.4.2.17	OnReadMotionTolerance()	80
8.4.2.18	OnReadPersistentLog()	81
8.4.2.19	OnReadWaveformClampSetting()	81
8.5	Caretaker::DeviceStatus Class Reference	81
8.5.1	Detailed Description	83
8.5.2	Field Documentation	83
8.5.2.1	dataValid	83
8.6	libct_app_callbacks_t Struct Reference	83
8.6.1	Detailed Description	85

8.6.2	Field Documentation	85
8.6.2.1	on_device_discovered	86
8.6.2.2	on_discovery_timeout	86
8.6.2.3	on_discovery_failed	87
8.6.2.4	on_device_connected_not_ready	88
8.6.2.5	on_device_connected_ready	88
8.6.2.6	on_connect_error	89
8.6.2.7	on_connect_timeout	90
8.6.2.8	on_device_disconnected	90
8.6.2.9	on_start_monitoring	91
8.6.2.10	on_stop_monitoring	92
8.6.2.11	on_start_measuring	92
8.6.2.12	on_stop_measuring	93
8.6.2.13	on_data_received	94
8.6.2.14	on_data_error	95
8.6.2.15	on_rd_snr_min_rsp	95
8.6.2.16	on_wrt_snr_min_rsp	96
8.6.2.17	on_rd_display_state_rsp	96
8.6.2.18	on_wrt_display_state_rsp	97
8.6.2.19	on_rd_recal_itvl_rsp	98
8.6.2.20	on_wrt_recal_itvl_rsp	98
8.6.2.21	on_rd_cuff_pressure_rsp	99
8.6.2.22	on_vent_cuff_rsp	99
8.6.2.23	on_clr_status_rsp	100
8.6.2.24	on_diag_flush_rsp	101
8.6.2.25	on_wrt_waveform_clamping	101
8.6.2.26	on_rd_waveform_clamping	102
8.6.2.27	on_rd_vitals_filter	102
8.6.2.28	on_wrt_vitals_filter	103
8.6.2.29	on_rd_motion_timeout	104

8.6.2.30	on_rd_persistent_log	104
8.7	libct_battery_info_t Class Reference	105
8.7.1	Detailed Description	105
8.7.2	Field Documentation	105
8.7.2.1	voltage	105
8.7.2.2	timestamp	106
8.8	libct_bp_settings_t Class Reference	106
8.8.1	Detailed Description	106
8.8.2	Field Documentation	106
8.8.2.1	systolic	106
8.8.2.2	diastolic	106
8.9	libct_cal_curve_t Class Reference	107
8.9.1	Detailed Description	107
8.9.2	Field Documentation	107
8.9.2.1	valid	107
8.9.2.2	data_id	107
8.9.2.3	val1	108
8.9.2.4	val2	108
8.9.2.5	val3	108
8.10	libct_cal_t Struct Reference	108
8.10.1	Detailed Description	109
8.10.2	Field Documentation	109
8.10.2.1	type	109
8.10.2.2	posture	109
8.10.2.3	config	109
8.11	libct_cal_type_t Class Reference	109
8.11.1	Detailed Description	109
8.12	libct_context_t Class Reference	110
8.12.1	Detailed Description	110
8.13	libct_cuff_pressure_t Class Reference	110

8.13.1 Detailed Description	111
8.13.2 Field Documentation	111
8.13.2.1 valid	111
8.13.2.2 value	111
8.13.2.3 target	111
8.13.2.4 snr	111
8.13.2.5 timestamp	111
8.14 libct_device_class_t Class Reference	112
8.14.1 Detailed Description	112
8.15 libct_device_state_t Class Reference	112
8.15.1 Detailed Description	112
8.16 libct_device_status_t Class Reference	112
8.16.1 Detailed Description	114
8.16.2 Field Documentation	114
8.16.2.1 simulation_enabled	114
8.16.2.2 inflated_indicator	114
8.16.2.3 clock_wrap_around	114
8.16.2.4 battery_voltage_low	115
8.16.2.5 pump_overrun	115
8.16.2.6 ble_temperature_sensor_paired	115
8.16.2.7 ble_stream_control	115
8.16.2.8 manual_cal_mode	115
8.16.2.9 motion_event	115
8.16.2.10 poor_signal	116
8.16.2.11 data_valid	116
8.16.2.12 calibration_offset_failed	116
8.16.2.13 weak_signal	116
8.16.2.14 bad_cuff	116
8.16.2.15 ble_adv	116
8.16.2.16 recal_soon	117

8.16.2.17 too_many_fails	117
8.17 libct_device_t Class Reference	117
8.17.1 Detailed Description	117
8.17.2 Field Documentation	118
8.17.2.1 get_state	118
8.17.2.2 get_class	118
8.17.2.3 get_name	119
8.17.2.4 get_address	119
8.17.2.5 get_serial_number	120
8.17.2.6 get_hw_version	120
8.17.2.7 get_fw_version	121
8.17.2.8 get_context	121
8.18 libct_init_data_t Struct Reference	122
8.18.1 Detailed Description	122
8.18.2 Field Documentation	122
8.18.2.1 device_class	122
8.19 libct_monitor_flags_t Class Reference	122
8.19.1 Detailed Description	123
8.20 libct_param_pulse_t Class Reference	123
8.20.1 Detailed Description	124
8.20.2 Field Documentation	124
8.20.2.1 valid	124
8.20.2.2 protocol_header	124
8.20.2.3 t0	124
8.20.2.4 t1	124
8.20.2.5 t2	124
8.20.2.6 t3	125
8.20.2.7 p0	125
8.20.2.8 p1	125
8.20.2.9 p2	125

8.20.2.10 p3	125
8.20.2.11 ibi	125
8.20.2.12 as	126
8.20.2.13 sqe	126
8.20.2.14 pressure	126
8.20.2.15 time	126
8.20.2.16 waveform_len	126
8.21 libct_posture_t Class Reference	126
8.21.1 Detailed Description	127
8.22 libct_pulse_ox_t Class Reference	127
8.22.1 Detailed Description	127
8.22.2 Field Documentation	127
8.22.2.1 sao2	127
8.22.2.2 pulse_rate	128
8.22.2.3 timestamp	128
8.23 libct_pulse_t Class Reference	128
8.23.1 Detailed Description	128
8.23.2 Field Documentation	128
8.23.2.1 timestamp	128
8.24 libct_pulse_waveform_t Struct Reference	129
8.24.1 Detailed Description	129
8.24.2 Field Documentation	129
8.24.2.1 receive_time	129
8.25 libct_status_t Class Reference	130
8.25.1 Detailed Description	130
8.26 libct_stream_data_t Class Reference	130
8.26.1 Detailed Description	131
8.26.2 Field Documentation	132
8.26.2.1 device	132
8.26.2.2 device_status	132

8.26.2.3	battery_info	132
8.26.2.4	datapoints [1/8]	133
8.26.2.5	count	133
8.26.2.6	vitals	133
8.26.2.7	datapoints [2/8]	133
8.26.2.8	cuff_pressure	134
8.26.2.9	datapoints [3/8]	134
8.26.2.10	temperature	134
8.26.2.11	datapoints [4/8]	135
8.26.2.12	pulse_ox	135
8.26.2.13	datapoints [5/8]	135
8.26.2.14	vitals2	136
8.26.2.15	datapoints [6/8]	136
8.26.2.16	raw_pulse	136
8.26.2.17	int_pulse	137
8.26.2.18	datapoints [7/8]	137
8.26.2.19	param_pulse	137
8.26.2.20	datapoints [8/8]	138
8.26.2.21	cal_curve	138
8.26.2.22	receive_time	138
8.27	libct_temperature_t Class Reference	139
8.27.1	Detailed Description	139
8.27.2	Field Documentation	139
8.27.2.1	value	139
8.27.2.2	timestamp	139
8.28	libct_version_t Struct Reference	139
8.28.1	Detailed Description	140
8.29	libct_vitals2_t Class Reference	140
8.29.1	Detailed Description	141
8.29.2	Field Documentation	141

8.29.2.1	blood_volume	141
8.29.2.2	timestamp	141
8.30	libct_vitals_t Class Reference	141
8.30.1	Detailed Description	142
8.30.2	Field Documentation	142
8.30.2.1	valid	143
8.30.2.2	bp_status	143
8.30.2.3	systolic	143
8.30.2.4	diastolic	143
8.30.2.5	map	143
8.30.2.6	heart_rate	143
8.30.2.7	respiration	144
8.30.2.8	as	144
8.30.2.9	sqe	144
8.30.2.10	timestamp	144
8.31	Caretaker::Device::LibraryCallback Class Reference	144
8.31.1	Detailed Description	144
8.32	Caretaker::PrimaryVitals Class Reference	145
8.32.1	Detailed Description	145
8.32.2	Field Documentation	145
8.32.2.1	systolic	145
8.32.2.2	diastolic	145
8.32.2.3	map	146
8.32.2.4	heartRate	146
8.32.2.5	respiration	146
8.32.2.6	asFactor	146
8.32.2.7	signalQualityEstimate	146
8.32.2.8	timestamp	147
8.33	Caretaker::SecondaryVitals Class Reference	147
8.33.1	Detailed Description	147
8.33.2	Field Documentation	147
8.33.2.1	bloodVolume	148
8.33.2.2	timestamp	148
8.34	Caretaker::WaveformDataPoints Class Reference	148
8.34.1	Detailed Description	148

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 4 wireless vitals signs monitoring device.

Scope

This version of the manual covers the Caretaker Library Windows API. The Caretaker library Android and Linux 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.

- [Library Integration](#)
- [Monitoring from Managed Application](#), or
- [Monitoring from Unmanaged Application](#)

Chapter 2

Library Integration

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

Library Package

Windows versions of the library are released in zip format *ctlibrary-windows-VERSION.zip*, where *VERSION* is the library version number.

The archive bundle contains the following components.

Component	Description
examples	Examples illustrating library usage.
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. Open <i>html/index.html</i> to view the html version main page..
Win32	If present, this directory stores the library 32-bit (x86) version.
Win64	If present, this directory stores the library 64-bit (x64) version.
caretaker_static.h	Header file to include when linking with the unmanaged libcaretaker_static.lib.
caretaker_dynamic.h	Header file to include when linking with the unmanaged libcaretaker_dynamic.dll.
libcaretaker_static.lib	The unmanaged C/C++ library image to use for static linking with native applications.
libcaretaker_dynamic.dll	The unmanaged C/C++ library image to use for dynamic linking with native applications.
libcaretaker_clr.dll	Managed CLR library image to use for linking with .NET framework 4.0 or later applications.

Supported Platforms

This library version tested successfully with applications designed for the following configurations.

OS Version

- Windows 7
- Windows 8.1
- Windows 10

Platform

- Win32 (x86)
- Win64 (x64)

Build Tools

- Visual Studio 2015
- Visual Studio 2017

Hardware Configuration

Prior to connecting the application to the CareTaker 4, Bluetooth will need to be setup on Windows. Note that native Windows Bluetooth is not supported on this release of the Caretaker library so an external BLE dongle such as the Bluegiga BLE112 or similar is required. Plug the dongle into a USB port on the PC and install the necessary drivers. After successful setup, the dongle will appear to the library as a virtual COM port that will be used to communicate with the Caretaker device. Note that the library will automatically detect the dongle after the drivers have been installed.

Linking with the Managed Library

The Common Language Runtime (CLR) wrapper library (libcaretaker_clr.dll) is available to integrate .NET 4.0 or later applications with the Caretaker device. To use it, copy libcaretaker_clr.dll (the managed library) and libcaretaker_dynamic.dll (the unmanaged library) to the application project directory then add a reference to libcaretaker_clr.dll. For Visual Studio projects, the reference can be added by right-clicking the application project then selecting Add Reference and then browsing to libcaretaker_clr.dll and selecting it.

Note both DLLs will be loaded at runtime when the application is executed and are expected to be located with the application executable (.exe) file. So they must be copied also to the target directory where the application executable file is saved. The copy can be done manually after building the application, or the build can be configured to copy them to the target directory with the executable. For Visual Studio, this can be done by right-clicking the project then selecting Properties and then Build Events and adding the following Post-build event commandline.

```
copy /B /Y "$(ProjectDir)libcaretaker_dynamic.dll" "$(TargetDir) "
```

Linking with the Unmanaged Library

Native Win32 applications can link the with the unmanaged library using either implicit or explicit linking.

Implicit Linking

Add the following files to the application project to link implicitly with the library.

- caretaker_static.h
- libcaretaker_static.lib

caretaker_static.h

This header file contains the exported library functions that are callable from within the application. Add it to the application source (or include) folder and then include it in each source file that will call library functions.

libcaretaker_static.lib

Add libcaretaker_static.lib to the application source (or lib) directory and include it in the library path where the linker can find it. If using an external makefile, add libcaretaker_static.lib where the object files (.obj) and other libraries are listed. If Visual Studio, add it to the project Linker Input configuration where the other libraries are listed.

Additionally, the following Windows system libraries must be added to the project to resolve libcaretaker dependencies.

- Setupapi.lib (available in the Windows SDK)

Explicit Linking

For explicit linking with the library, the application must explicitly load and unload libcaretaker_dynamic.lib.

- Call LoadLibrary (or a similar function) to load the DLL and obtain a module handle.
- Call GetProcAddress to obtain a function pointer to each exported function that the application will call.
- Call FreeLibrary when done with the DLL.

Sample code illustrating explicit linking is available in the [Unmanaged Explicit Linking Example](#) section.

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 messages only. This is the least verbose level.

Call the managed Device::SetLogLevel() API or the unmanaged libct_set_log_level() API from the application code to set the log level.

Log messages will be written to standard output (stdout) and will be printed to the console where the application was started. For example, if the application is a console application then the log messages will be printed to the console automatically. However, for non-console applications, the standard output must be redirected to view the logs. For example, native, non-console applications can redirect standard output to a file as follows.

```
FILE* debugStream = NULL;  
freopen_s(&debugStream, "libcaretaker.log", "w", stdout);
```


Chapter 3

Monitoring from Managed Application

This section provides an illustration to monitor Caretaker data from within a managed (.NET framework) application.

After adding the reference to the library as discussed in the [Library Integration](#) section, monitoring the Caretaker device can be done using the managed Device and supporting classes, which are available in the Caretaker namespace.

Asynchronous Monitoring

The [Caretaker.DeviceObserver](#) class provides an asynchronous interface to object-oriented application to receive Caretaker data and status information in realtime. Using this approach, the application calls [Caretaker.Device](#) class methods to read or write device data and receive the data and status information asynchronously in the application's DeviceObserver implementation.

The following code snippets illustrate C# code to create device and observer instances for asynchronous monitoring.

```
// Implement DeviceObserver to receive Caretaker data and status notifications.
public partial class Observer : Caretaker.DeviceObserver
{
    // Override the methods to receive the desired data and status information.
    // See Caretaker.DeviceObserver class documentation for details.
}

// Create the device and observer instances.
// Note the observer instance is passed as the first argument to the device constructor,
// and the second argument autoReconnect=true configures the device instance to automatically
// reconnect if the connection to the Caretaker device is lost, such as when the Caretaker device
// moves out of range.
Observer observer = new Observer();
Caretaker.Device device = new Caretaker.Device(observer, true);

// Establish connection to a device that is advertising and timeout after 20 seconds.
// Note you can specify the Caretaker serial number to connect a specific device, or call device.Scan()
// to scan for a device to connect to.
// The connection status will be notified later to the application's DeviceObserver.OnConnectionStatus()
// implementation with status=CONNECTED if the connection was established.
device.ConnectToSerialNumber(null, 20000);

...

// Sometime later after the connection is established, start a calibration to receive data.
// Note StartManualCal() typically is invoked from DeviceObserver.OnConnectionStatus() for the
// CONNECTED status.
device.StartManualCal(120, 75);

// After calibration, the various DeviceObserver methods will be notified when data is received.

...

// Finally, call ReleaseResources() to release unmanaged resources allocated for the device instance.
// Note the device reference is no longer valid after calling ReleaseResources().
// Also, note ReleaseResources() cleans up unmanaged resources such as native threads and may delay the
// application thread for a couple seconds as it waits for the native threads to exit.
device.ReleaseResources();
device = null;
```

Poll Monitoring

The `Caretaker.Device` class also provides methods (getters) for procedural applications to poll Caretaker data. However, the methods must be called periodically to receive data after the connection is established. Note methods to get waveform data must be called at least 4 times per second to not drop data, while methods to get numeric data must be called at least once per second.

The following code illustrates snippet of C# code to create the device instance for poll monitoring.

```
// Create the device instance passing null observer as argument.
Caretaker.Device device = Caretaker.Device(null, true);

// Establish connection to a device that is advertising and timeout after 20 seconds.
device.ConnectToSerialNumber(null, 20000);

// Poll for connected status.
Boolean timedout = false;
while(!device.IsConnected() && !timedout) {
    // Update timedout
}

// Start a calibration to receive data if connection was established.
if ( device.IsConnected() ) {
    device.StartManualCal(120, 75);

    // Poll numeric data, waveform data and status information.
    uint count = 0;
    while(!exit) {
        // poll device status and numeric data once per second
        if ( count % 4 == 0 ) {
            Caretaker.DeviceStatus deviceStatus = device.GetDeviceStatus();
            if (deviceStatus) {
                // display device status
            }

            Caretaker.PrimaryVitals vitals = device.GetPrimaryVitals();
            if ( vitals ) {
                // display vitals
            }
        }

        // poll waveform data at least 4 times per second
        Caretaker.WaveformDataPoints waveform = mCaretaker.
GetPulsePressureWaveformDataPoints();
        if (waveform != null)
        {
            // display waveform
        }

        // sleep for 250 milliseconds
        System.Threading.Thread.Sleep(250);
        count++;
    }
}

// Finally, call ReleaseResources() to release unmanaged resources allocated for the device.
// Note the device reference is no longer valid after calling ReleaseResources().
// Also, note ReleaseResources() cleans up unmanaged resources such as native threads and may delay the
// application thread for a couple seconds as it waits for the native threads to exit.
device.ReleaseResources();
device = null;
```

Chapter 4

Monitoring from Unmanaged Application

This section provides an illustration to monitor Caretaker data from within an unmanaged application.

Overview

Two groups of unmanaged 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 an unmanaged application interacts 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 notifications.

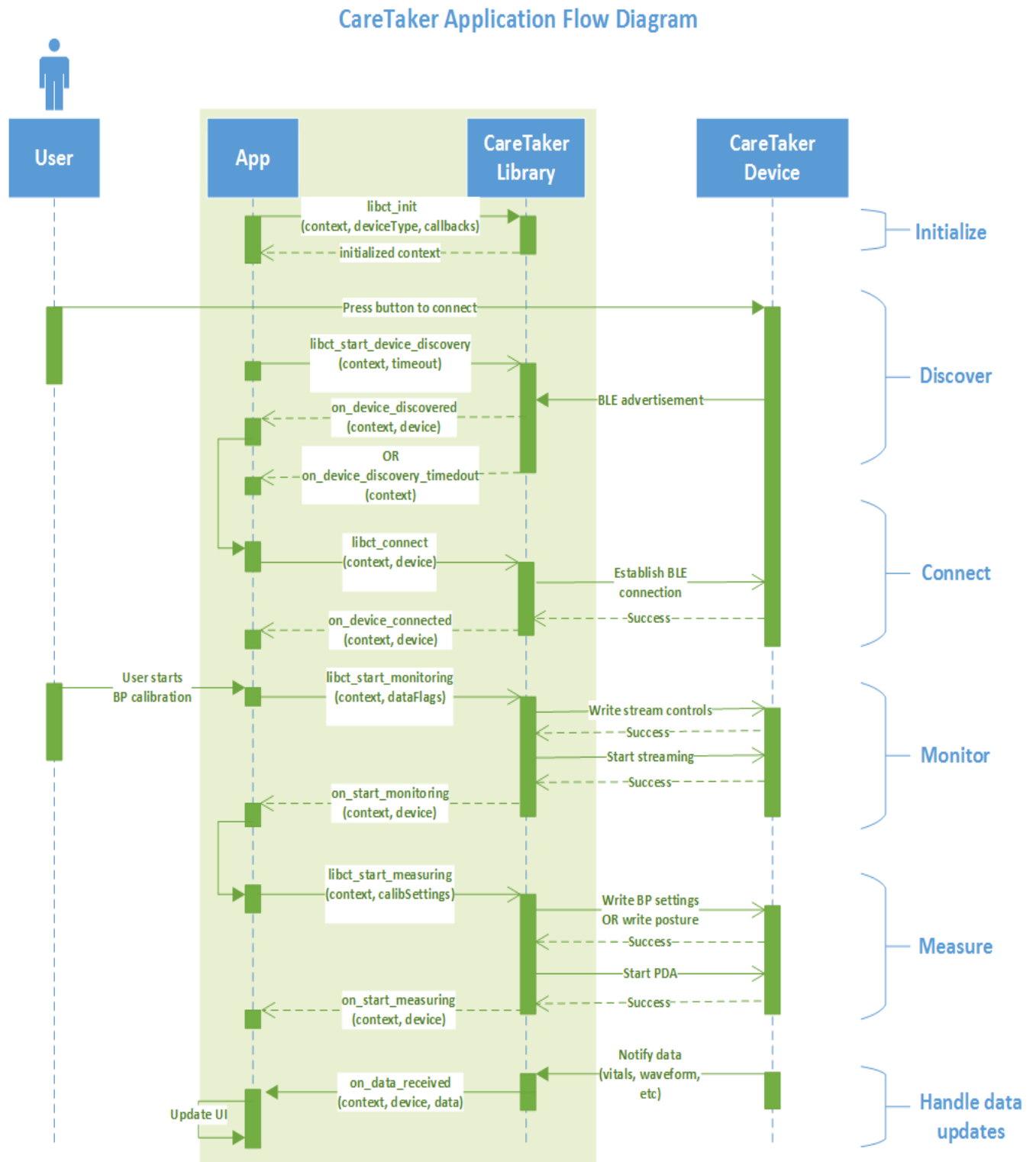


Figure 4.1 Sequence diagram to connect and monitor data.

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 minimum, 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 advertising 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 notifications 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 deferred 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 the `on_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 measurements were started successfully, the application will start receiving numeric and waveform data updates. The application `on_data_received()` callback will be notified continuously 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 5

Module Index

5.1 Modules

Here is a list of all modules:

Unmanaged Device Information	19
Unmanaged Primary API	25
Unmanaged Secondary API	31
Managed API for .NET Applications	47
Unmanaged Explicit Linking Example	51

Chapter 6

Data Structure Index

6.1 Data Structures

Here are the data structures with brief descriptions:

Caretaker::BatteryStatus	
Managed class defining battery status	53
Caretaker::CuffStatus	
Managed class defining cuff status	54
Caretaker::Device	
Managed class for .NET applications to monitor the Caretaker device	55
Caretaker::DeviceObserver	
Managed class defining the application observer interface to receive real-time notifications . . .	73
Caretaker::DeviceStatus	
Managed class defining device status	81
libct_app_callbacks_t	
Structure used to provide asynchronous notifications to the application	83
libct_battery_info_t	
Battery info data point within the libct_stream_data_t packet	105
libct_bp_settings_t	
Structure to write the caretaker manual blood pressure settings	106
libct_cal_curve_t	
Calibration curve data point within the libct_stream_data_t packet	107
libct_cal_t	
Structure used to pass calibration data to libct_start_measuring()	108
libct_cal_type_t	
The Caretaker calibration types	109
libct_context_t	
An opaque type representing a library instance associated with (or bound to) a device the application is monitoring	110
libct_cuff_pressure_t	
Cuff pressure data point within the libct_stream_data_t packet	110
libct_device_class_t	
Classes of devices that can be monitored by this library	112
libct_device_state_t	
The Caretaker device states	112
libct_device_status_t	
Device status data point within the libct_stream_data_t packet	112
libct_device_t	
Handle used to identify a connected device the application is monitoring	117

libct_init_data_t	Structure defining initialization data passed to libct_init()	122
libct_monitor_flags_t	Data monitor flags passed to libct_start_monitoring()	122
libct_param_pulse_t	Parametrized pulse data within the libct_stream_data_t packet	123
libct_posture_t	Patient postures	126
libct_pulse_ox_t	Pulse oximetry data point within the libct_stream_data_t packet	127
libct_pulse_t	Raw or integrated pulse data point within the libct_stream_data_t packet	128
libct_pulse_waveform_t	Pulse waveform data returned from the device as a result of a previous read request	129
libct_status_t	Function return status codes	130
libct_stream_data_t	This structure is used to hand-off data received from the remote device to the application	130
libct_temperature_t	Temperature data point within the libct_stream_data_t packet	139
libct_version_t	CareTaker version information	139
libct_vitals2_t	Secondary Vitals data point within the libct_stream_data_t packet	140
libct_vitals_t	Vitals data point within the libct_stream_data_t packet	141
Caretaker::Device::LibraryCallback	This is an internal class representing a callback into the unmanaged library code	144
Caretaker::PrimaryVitals	Managed class defining the primary vitals measured by the Caretaker	145
Caretaker::SecondaryVitals	Managed class defining the secondary vitals measured by the Caretaker	147
Caretaker::WaveformDataPoints	Pulse waveform data points Each data point is of format (t,v), where t is a monotonically incrementing counter representing time-stamp, and v is the pulse sample at time t	148

Chapter 7

Module Documentation

7.1 Unmanaged Device Information

This module 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_UNINITIALIZED)`
Returns non-zero (true) if the device is not initialized, and zero (false) otherwise.
- `#define libct_device_initialized(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_MONITORING)`
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)`.

7.1.1 Detailed Description

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

7.1.2 Macro Definition Documentation

7.1.2.1 libct_device_get_state

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

Convenience macro to `device->get_state(device)`.

Parameters

<i>dev</i>	Pointer to the device instance.
------------	---------------------------------

7.1.2.2 libct_device_uninitialized

```
#define libct_device_uninitialized(  
    dev ) (((dev)->get_state(dev)) & LIBCT_STATE_UNINITIALIZED)
```

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

Parameters

<i>dev</i>	Pointer to device instance.
------------	-----------------------------

7.1.2.3 libct_device_discovering

```
#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

<i>dev</i>	Pointer to device instance.
------------	-----------------------------

7.1.2.4 libct_device_connecting

```
#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

<i>dev</i>	Pointer to device instance.
------------	-----------------------------

7.1.2.5 libct_device_connected

```
#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

<i>dev</i>	Pointer to device instance.
------------	-----------------------------

7.1.2.6 libct_device_disconnecting

```
#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

<i>dev</i>	Pointer to device instance.
------------	-----------------------------

7.1.2.7 libct_device_disconnected

```
#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

<i>dev</i>	Pointer to device instance.
------------	-----------------------------

7.1.2.8 libct_device_monitoring

```
#define libct_device_monitoring(  
    dev ) (((dev)->get_state(dev)) & LIBCT_STATE_MONITORING)
```

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

Parameters

<i>dev</i>	Pointer to device instance.
------------	-----------------------------

7.1.2.9 libct_device_measuring

```
#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

<i>dev</i>	Pointer to device instance.
------------	-----------------------------

7.1.2.10 libct_device_get_class

```
#define libct_device_get_class(  
    dev ) (dev)->get_class(dev)
```

Convenience macro to [device->get_class\(device\)](#).

Parameters

<i>dev</i>	Pointer to the device instance.
------------	---------------------------------

7.1.2.11 libct_device_get_name

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

Convenience macro to [device->get_name\(device\)](#).

Parameters

<i>dev</i>	Pointer to the device instance.
------------	---------------------------------

7.1.2.12 libct_device_get_address

```
#define libct_device_get_address(  
    dev ) (dev)->get_address(dev)
```

Convenience macro to [device->get_address\(device\)](#).

Parameters

<i>dev</i>	Pointer to the device instance.
------------	---------------------------------

7.1.2.13 libct_device_get_serial_number

```
#define libct_device_get_serial_number(  
    dev ) (dev)->get_serial_number(dev)
```

Convenience macro to [device->get_serial_number\(device\)](#).

Parameters

<i>dev</i>	Pointer to the device instance.
------------	---------------------------------

7.1.2.14 libct_device_get_hw_version

```
#define libct_device_get_hw_version(  
    dev ) (dev)->get_hw_version(dev)
```

Convenience macro to [device->get_hw_version\(device\)](#).

Parameters

<i>dev</i>	Pointer to the device instance.
------------	---------------------------------

7.1.2.15 libct_device_get_fw_version

```
#define libct_device_get_fw_version(  
    dev ) (dev)->get_fw_version(dev)
```

Convenience macro to [device->get_fw_version\(device\)](#).

Parameters

<i>dev</i>	Pointer to the device instance.
------------	---------------------------------

7.1.2.16 libct_device_get_context

```
#define libct_device_get_context(  
    dev ) (dev)->get_context(dev)
```

Convenience macro to [device->get_context\(device\)](#).

Parameters

<i>dev</i>	Pointer to the device instance.
------------	---------------------------------

7.2 Unmanaged Primary API

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

Functions

- LIBCTEXTPORT int [libct_init](#) ([libct_context_t](#) **context, [libct_init_data_t](#) *data, [libct_app_callbacks_t](#) *callbacks)
Initializes device context.
- LIBCTEXTPORT void [libct_deinit](#) ([libct_context_t](#) *context)
De-initializes the context.
- LIBCTEXTPORT int [libct_start_discovery](#) ([libct_context_t](#) *context, unsigned long timeout)
Discover the device.
- LIBCTEXTPORT int [libct_stop_discovery](#) ([libct_context_t](#) *context)
Stop device discovery.
- LIBCTEXTPORT int [libct_connect](#) ([libct_context_t](#) *context, [libct_device_t](#) *device)
Connect to a device.
- LIBCTEXTPORT int [libct_disconnect](#) ([libct_context_t](#) *context)
Disconnect from a device.
- LIBCTEXTPORT int [libct_start_monitoring](#) ([libct_context_t](#) *context, int flags)
Start monitoring data at the remote caretaker device.
- LIBCTEXTPORT int [libct_stop_monitoring](#) ([libct_context_t](#) *context)
Stops monitoring.
- LIBCTEXTPORT int [libct_start_measuring](#) ([libct_context_t](#) *context, [libct_cal_t](#) *cal)
Start taking measurement.
- LIBCTEXTPORT int [libct_stop_measuring](#) ([libct_context_t](#) *context)
Stops measuring.

7.2.1 Detailed Description

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

7.2.2 Function Documentation

7.2.2.1 [libct_init\(\)](#)

```
LIBCTEXTPORT 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

<i>context</i>	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.
<i>data</i>	Data to initialize the context.
<i>callbacks</i>	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.

7.2.2.2 libct_deinit()

```
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 `libct_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

<i>context</i>	The context returned from libct_init() .
----------------	--

7.2.2.3 libct_start_discovery()

```
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

<i>context</i>	The context returned from libct_init() .
<i>timeout</i>	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.

7.2.2.4 libct_stop_discovery()

```
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

<i>context</i>	The context returned from libct_init() .
----------------	--

Returns

An appropriate [status code](#) indicating success or error.

7.2.2.5 libct_connect()

```
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

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

Returns

An appropriate [status code](#) indicating success or error.

7.2.2.6 libct_disconnect()

```
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

<i>context</i>	The context returned from libct_init() .
----------------	--

Returns

An appropriate [status code](#) indicating success or error.

7.2.2.7 libct_start_monitoring()

```
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

<i>context</i>	The context returned from libct_init() .
<i>flags</i>	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.

7.2.2.8 libct_stop_monitoring()

```
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

<i>context</i>	The context returned from libct_init() .
----------------	--

Returns

An appropriate [status code](#) indicating success or error.

7.2.2.9 libct_start_measuring()

```
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

<i>context</i>	The context returned from libct_init() .
<i>cal</i>	Auto or manual calibration settings.

Returns

An appropriate [status code](#) indicating success or error

7.2.2.10 libct_stop_measuring()

```
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

<i>context</i>	The context returned from libct_init() .
----------------	--

Returns

An appropriate [status code](#) indicating success or error.

7.3 Unmanaged 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.*

7.3.1 Detailed Description

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

7.3.2 Macro Definition Documentation

7.3.2.1 libct_dp_count

```
#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

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

Returns

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

7.3.2.2 libct_get_dp

```
#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

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

Returns

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

7.3.2.3 libct_get_last_dp

```
#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

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

Returns

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

7.3.2.4 libct_get_first_dp

```
#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

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

Returns

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

7.3.2.5 for_each_dp

```
#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

<i>data</i>	The stream data received in your on_data_received() application callback.
<i>idx</i>	Iterator variable of type unsigned integer.
<i>dp</i>	Pointer variable of type corresponding to the memb argument.
<i>memb</i>	The stream data member name of the data points to extract.

7.3.2.6 libct_inc_cuff_pressure

```
#define libct_inc_cuff_pressure(  
    context ) libct_adjust_cuff_pressure((context), 1)
```

Increments the cuff pressure in 10 mmHg increment.

Parameters

<i>context</i>	The context returned from libct_init() .
----------------	--

Returns

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

7.3.2.7 libct_dec_cuff_pressure

```
#define libct_dec_cuff_pressure(  
    context ) libct_adjust_cuff_pressure((context), 0)
```

Decrements the cuff pressure in 10 mmHg increment.

Parameters

<i>context</i>	The context returned from libct_init() .
----------------	--

Returns

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

7.3.3 Function Documentation

7.3.3.1 libct_get_device()

```
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

<i>context</i>	The context returned from libct_init() .
----------------	--

Returns

Device object or NULL;

See also

The device APIs

7.3.3.2 libct_set_app_specific_data()

```
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

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

7.3.3.3 libct_get_app_specific_data()

```
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

<i>context</i>	The context returned from libct_init() .
----------------	--

7.3.3.4 libct_get_version_string()

```
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.

7.3.3.5 libct_get_build_date_string()

```
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.

7.3.3.6 libct_set_log_level()

```
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

<i>level</i>	One of the following log levels. <ul style="list-style-type: none">• 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
--------------	---

7.3.3.7 libct_recalibrate()

```
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

<i>context</i>	The context returned from libct_init() .
----------------	--

Returns

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

7.3.3.8 libct_adjust_cuff_pressure()

```
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

<i>context</i>	The context returned from libct_init() .
<i>direction</i>	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.

7.3.3.9 libct_rd_cuff_pressure()

```
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

<i>context</i>	The context returned from libct_init() .
----------------	--

Returns

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

7.3.3.10 libct_vent_cuff()

```
LIBCTEXPORT int libct_vent_cuff (  
    libct_context_t * context )
```

Deflates the cuff pressure.

Results will be notified later with [on_vent_cuff_rsp\(\)](#).

Parameters

<i>context</i>	The context returned from libct_init() .
----------------	--

Returns

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

7.3.3.11 libct_clr_status()

```
LIBCTEXPORT int libct_clr_status (
    libct_context_t * context )
```

Clears the device status.

Results will be notified later with [on_clr_status_rsp\(\)](#).

Parameters

<i>context</i>	The context returned from libct_init() .
----------------	--

Returns

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

7.3.3.12 libct_diag_flush()

```
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

<i>context</i>	The context returned from libct_init() .
----------------	--

Returns

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

7.3.3.13 libct_wrt_snr_min()

```
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

<i>context</i>	The context returned from libct_init() .
<i>snr</i>	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.

7.3.3.14 libct_rd_snr_min()

```
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

<i>context</i>	The context returned from libct_init() .
----------------	--

Returns

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

7.3.3.15 libct_wrt_display_state()

```
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

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

Returns

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

7.3.3.16 libct_rd_display_state()

```
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

<i>context</i>	The context returned from libct_init() .
----------------	--

Returns

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

7.3.3.17 libct_wrt_recal_itvl()

```
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

<i>context</i>	The context returned from libct_init() .
<i>itvl</i>	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.

7.3.3.18 libct_rd_recal_itvl()

```
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

<i>context</i>	The context returned from libct_init() .
----------------	--

Returns

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

7.3.3.19 libct_wrt_waveform_clamping()

```
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

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

Returns

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

7.3.3.20 libct_rd_waveform_clamping()

```
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

<i>context</i>	The context returned from libct_init() .
----------------	--

Returns

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

7.3.3.21 libct_rd_vitals_filter()

```
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

<i>context</i>	The context returned from libct_init() .
----------------	--

Returns

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

7.3.3.22 libct_wrt_vitals_filter()

```
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

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

Returns

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

7.3.3.23 libct_wrt_simulation_mode()

```
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

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

Returns

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

7.3.3.24 libct_wrt_motion_timeout()

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

Writes the motion tolerance timeout parameter.

Parameters

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

Returns

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

7.3.3.25 libct_rd_motion_timeout()

```
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

<i>context</i>	The context returned from libct_init() .
----------------	--

Returns

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

7.3.3.26 libct_rd_persistent_log()

```
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

<i>context</i>	The context returned from libct_init() .
----------------	--

Returns

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

7.4 Managed API for .NET Applications

This module describes the Caretaker library managed APIs to be used with .NET applications.

Data Structures

- class [Caretaker::DeviceStatus](#)
Managed class defining device status.
- class [Caretaker::BatteryStatus](#)
Managed class defining battery status.
- class [Caretaker::CuffStatus](#)
Managed class defining cuff status.
- class [Caretaker::PrimaryVitals](#)
Managed class defining the primary vitals measured by the Caretaker.
- class [Caretaker::SecondaryVitals](#)
Managed class defining the secondary vitals measured by the Caretaker.
- class [Caretaker::DeviceObserver](#)
Managed class defining the application observer interface to receive real-time notifications.
- class [Caretaker::Device](#)
Managed class for .NET applications to monitor the Caretaker device.

Enumerations

- enum [Caretaker::ConnectionStatus](#) {
[Caretaker::ConnectionStatus::SCAN_ERROR](#) = 0, [Caretaker::ConnectionStatus::SCAN_TIMEOUT](#) = 1,
[Caretaker::ConnectionStatus::CONNECT_ERROR](#) = 2, [Caretaker::ConnectionStatus::CONNECT_TIMEOUT](#)
= 3,
[Caretaker::ConnectionStatus::CONNECTED](#) = 4, [Caretaker::ConnectionStatus::DISCONNECTED](#) = 5,
[Caretaker::ConnectionStatus::CONNECTION_LOST](#) = 6 }
Managed class defining connection status codes notified with DeviceObserver::OnConnectionStatus().
- enum [Caretaker::StartStatus](#) { [Caretaker::StartStatus::STARTED](#) = 0, [Caretaker::StartStatus::START_ERROR](#)
= 1 }
Windows application managed class defining start status.
- enum [Caretaker::StopStatus](#) { [Caretaker::StopStatus::STOPPED](#) = 0, [Caretaker::StopStatus::STOP_ERROR](#)
= 1 }
Managed class defining stop status.
- enum [Caretaker::PatientPosture](#) { [Caretaker::PatientPosture::SITTING](#) = 1, [Caretaker::PatientPosture::SUPINE](#)
= 3 }
Managed class defining patient postures.

7.4.1 Detailed Description

This module describes the Caretaker library managed APIs to be used with .NET applications.

7.4.2 Enumeration Type Documentation

7.4.2.1 ConnectionStatus

enum `Caretaker::ConnectionStatus` [strong]

Managed class defining connection status codes notified with `DeviceObserver::OnConnectionStatus()`.

Enumerator

SCAN_ERROR	Error while scanning for a device. Scan errors are typically due to protocol errors such as when the BLE dongle is not plugged.
SCAN_TIMEOUT	Scanning for a device timed out.
CONNECT_ERROR	Error while establishing connection to the device.
CONNECT_TIMEOUT	Connecting to the device timed out.
CONNECTED	The connection was established with the device successfully.
DISCONNECTED	The application initiated disconnect completed successfully.
CONNECTION_LOST	<p>The connection to the device was lost. Typically, the connection is lost after the Caretaker device moves outside the BLE range.</p> <p>Note</p> <p>The Device instance can be configured to reconnect automatically when the device moves back in range by setting the <code>autoReconnect</code> argument to true when invoking the Device constructor to create the instance.</p>

7.4.2.2 StartStatus

```
enum Caretaker::StartStatus [strong]
```

Windows application managed class defining start status.

Start status codes are notified with [DeviceObserver::OnStartStatus\(\)](#) so the application must implement this method to receive them.

Enumerator

STARTED	The previous call to Device::StartManualCal() or Device::StartAutoCal() completed successfully.
START_ERROR	The previous call to Device::StartManualCal() or Device::StartAutoCal() completed with error.

7.4.2.3 StopStatus

```
enum Caretaker::StopStatus [strong]
```

Managed class defining stop status.

Stop status codes are notified with [DeviceObserver::OnStopStatus\(\)](#) so the application must implement this method to receive them.

Enumerator

STOPPED	The previous call to Device::Stop() completed successfully.
STOP_ERROR	The previous call to Device::Stop() completed with error.

7.4.2.4 PatientPosture

enum `Caretaker::PatientPosture` [strong]

Managed class defining patient postures.

Pass these values as argument to `Device::StartAutoCal()` for automatic calibration.

Enumerator

SITTING	Medical body position for sitting.
SUPINE	Medical body position for supine.

7.5 Unmanaged Explicit Linking Example

This section provides an illustration of linking the libcaretaker library explicitly with unmanaged applications.

This section provides an illustration of linking the libcaretaker library explicitly with unmanaged applications.

To link explicitly to the Caretaker library, the application must explicitly load and unload libcaretaker_dynamic.lib.

- Call LoadLibrary (or a similar function) to load the DLL and obtain a module handle.
- Call GetProcAddress to obtain a function pointer to each exported function that the application will call.
- Call FreeLibrary when done with the DLL.

```
typedef UINT (CALLBACK* LIBCT_INIT_FUNC)(libct_context_t**,
    libct_init_data_t*, libct_app_callbacks_t*);
typedef UINT (CALLBACK* LIBCT_DEINIT_FUNC)(libct_context_t*);

...

HINSTANCE hDLL; // Handle to DLL
libct_context_t* context; // libcaretaker context
LP_LIBCT_INIT_FUNC LibctInit; // init function pointer
LP_LIBCT_DEINIT_FUNC LibctDeinit; // deinit function pointer

// Setup to connect to a Caretaker 4 device
libct_init_data_t initData;
memset(initData, 0, sizeof(initData));
initData.device_class = LIBCT_DEVICE_CLASS_BLE_CARETAKER4;

// Setup your application callbacks. (Only one is illustrated here but add more as needed.)
libct_app_callbacks_t appCallbacks;
memset(appCallbacks, 0, sizeof(appCallbacks));
appCallbacks.on_device_discovered = OnDeviceDiscovered;

hDLL = LoadLibrary("libcaretaker_dynamic.lib");
if (hDLL != NULL)
{
    LibctInit = (LP_LIBCT_INIT_FUNC)GetProcAddress(hDLL, "libct_init");
    if (!LibctInit)
    {
        // handle the error
        FreeLibrary(hDLL);
        return SOME_ERROR_CODE;
    }

    LibctDeinit = (LP_LIBCT_DEINIT_FUNC)GetProcAddress(hDLL, "libct_deinit");
    if (!LibctDeinit)
    {
        // handle the error
        FreeLibrary(hDLL);
        return SOME_ERROR_CODE;
    }

    // Initialize the library
    context = NULL
    int status = LibctInit(&context, &initData, &appCallbacks);
    if (LIBCT_FAILED(status))
    {
        // handle the error
        FreeLibrary(hDLL);
        return SOME_ERROR_CODE;
    }

    // Add your application code to connect and monitor data

    // Exit when done
    status = LibctDeinit(context);
    FreeLibrary(hDLL);
}
```


Chapter 8

Data Structure Documentation

8.1 Caretaker::BatteryStatus Class Reference

Managed class defining battery status.

Data Fields

- Int32 [voltage](#)
The battery voltage in millivolts.
- Int32 [timestamp](#)
Milliseconds time-stamp from the device associated with the data.

8.1.1 Detailed Description

Managed class defining battery status.

Battery status information is notified with [DeviceObserver::OnBatteryStatus\(\)](#) so the application must implement this method to receive it.

8.1.2 Field Documentation

8.1.2.1 voltage

```
Int32 Caretaker::BatteryStatus::voltage
```

The battery voltage in millivolts.

8.1.2.2 timestamp

```
Int32 Caretaker::BatteryStatus::timestamp
```

Milliseconds time-stamp from the device associated with the data.

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

- CaretakerDevice.h

8.2 Caretaker::CuffStatus Class Reference

Managed class defining cuff status.

Data Fields

- Int32 [actualPressure](#)
The actual cuff pressure in mmHg.
- Int32 [targetPressure](#)
The target cuff pressure in mmHg.
- Int32 [signalToNoise](#)
The signal to noise ratio Divide by 10 to convert to percentage.
- Int32 [timestamp](#)
Milliseconds time-stamp from the device associated with the data.

8.2.1 Detailed Description

Managed class defining cuff status.

Cuff status information is notified with [DeviceObserver::OnCuffStatus\(\)](#) so the application must implement this method to receive it.

8.2.2 Field Documentation

8.2.2.1 actualPressure

```
Int32 Caretaker::CuffStatus::actualPressure
```

The actual cuff pressure in mmHg.

8.2.2.2 targetPressure

```
Int32 Caretaker::CuffStatus::targetPressure
```

The target cuff pressure in mmHg.

8.2.2.3 signalToNoise

```
Int32 Caretaker::CuffStatus::signalToNoise
```

The signal to noise ratio Divide by 10 to convert to percentage.

```
displaySNR = MIN(100, status.signalToNoise / 10);  
displaySNR = MAX(0, displaySNR);
```

8.2.2.4 timestamp

```
Int32 Caretaker::CuffStatus::timestamp
```

Milliseconds time-stamp from the device associated with the data.

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

- CaretakerDevice.h

8.3 Caretaker::Device Class Reference

Managed class for .NET applications to monitor the Caretaker device.

Data Structures

- class [LibraryCallback](#)

This is an internal class representing a callback into the unmanaged library code.

Public Member Functions

- **Device** (**DeviceObserver**[^] observer, Boolean autoReconnect)
*Creates a **Device** instance.*
- **~Device** ()
Destructor.
- void **ReleaseResources** ()
*Release unmanaged library resources allocated for the **Device** instance.*
- Boolean **StartScan** (UInt32 timeout)
Initiates scanning for nearby Caretaker devices.
- Boolean **StopScan** ()
*Call this method to stop scanning after calling **StartScan()**.*
- Boolean **ConnectToSerialNumber** (String[^] sn, UInt32 timeout)
Initiates connecting to the Caretaker device with the specified serial number.
- Boolean **ConnectToMacAddress** (String[^] address, UInt32 timeout)
Initiates connecting to the Caretaker device with the specified MAC address.
- Boolean **ConnectToAny** (UInt32 timeout)
Initiates connecting to any Caretaker device that is advertising for a connection.
- Boolean **Disconnect** ()
Initiates disconnecting from the remote device if connected.
- Boolean **StartAutoCal** (**PatientPosture** posture)
Perform an automatic calibration and then start taking measurements if connected.
- Boolean **StartManualCal** (Int32 systolic, Int32 diastolic)
Perform a manual calibration and then start taking measurements if connected.
- Boolean **Stop** ()
*Call this method after calling **StartAutoCal()** or **StartManualCal()** to stop the device.*
- Boolean **IsConnected** ()
Returns true if the device is connected and false otherwise.
- Boolean **IsConnecting** ()
Returns true while attempting to establish connection to device and false otherwise.
- Boolean **Calibrating** ()
Returns true if the device is calibrating, and false otherwise.
- Boolean **Calibrated** ()
Returns true if the device is calibrated, and false otherwise.
- Boolean **CalibrationFailed** ()
Returns true if the device was calibrating and the calibration failed, and false otherwise.
- String **GetName** ()
Returns the device friendly name or null if the name is not available.
- String **GetMacAddress** ()
Returns the device MAC address or null if the MAC address is not available.
- String **GetSerialNumber** ()
Returns the device serial number or null if the serial number is not available.
- String **GetFirmwareVersion** ()
Returns the device firmware version or null if the firmware version is not available.
- String **GetHardwareVersion** ()
Returns the device hardware version or null if the hardware version is not available.
- String **GetLibraryVersion** ()
Returns the library version or null if the library version is not available.
- String **GetLibraryBuildDate** ()
Returns the library build date or null if the build date is not available.
- **DeviceStatus** **GetDeviceStatus** ()

- Method to poll device status.*
- [BatteryStatus GetBatteryStatus \(\)](#)
Method to poll battery status.
- [CuffStatus GetCuffStatus \(\)](#)
Method to poll cuff status.
- array< [PrimaryVitals](#) > [GetPrimaryVitals \(\)](#)
Method to poll primary vitals.
- array< [SecondaryVitals](#) > [GetSecondaryVitals \(\)](#)
Method to poll secondary vitals.
- [WaveformDataPoints GetPulseRateWaveformDataPoints \(\)](#)
Method to poll pulse rate waveform data points.
- [WaveformDataPoints GetPulsePressureWaveformDataPoints \(\)](#)
Method to poll pulse pressure waveform data points.
- Boolean [IncrementCuffPressure \(\)](#)
Increments the cuff pressure in 10 mmHg increment.
- Boolean [DecrementCuffPressure \(\)](#)
Decrements the cuff pressure in 10 mmHg increment.
- Boolean [VentCuff \(\)](#)
Deflates the cuff pressure.
- Boolean [PeformDiagnosticsFlush \(\)](#)
Flushes the device diagnostic plumbing tree.
- Boolean [WriteSnrMinimum \(Int32 snr\)](#)
Writes the device noise filter parameter.
- Boolean [ReadSnrMinimum \(\)](#)
Reads the device noise filter parameter.
- Boolean [WriteMotionTolerance \(Int32 value\)](#)
Writes the device motion tolerance parameter.
- Boolean [ReadMotionTolerance \(\)](#)
Reads the device motion tolerance parameter.
- Boolean [TurnDisplayOff \(\)](#)
Turns the device display screen off.
- Boolean [TurnDisplayOn \(\)](#)
Turns the device display screen on.
- Boolean [ReadDisplayState \(\)](#)
Reads the display state.
- Boolean [Recalibrate \(\)](#)
Recalibrate blood pressure measurements.
- Boolean [WriteRecalibrationInterval \(UInt32 interval\)](#)
Writes the calibration interval.
- Boolean [ReadRecalibrationInterval \(\)](#)
Reads the calibration interval.
- Boolean [WriteWaveformClampSetting \(Byte value\)](#)
Writes the setting to enable or disable clamping the waveforms.
- Boolean [ReadWaveformClampSetting \(\)](#)
Read the waveform clamping setting.
- Boolean [WriteVitalsFilterSetting \(Byte value\)](#)
Writes the filter settings to enable or disable filtering outlier vitals measurements.
- Boolean [ReadVitalsFilterSetting \(\)](#)
Reads the current filter settings to enable or disable filtering outlier vitals measurements.
- Boolean [ReadPersistentLog \(\)](#)
Reads the device logs.

- void [SetLibraryLogLevel](#) (Int32 level)
Sets the library log level.
- Boolean [EnableSimulationMode](#) (Boolean mode)
Writes the device simulation mode.
- Boolean [ClearStatus](#) ()
Clears the device status.

Protected Member Functions

- [!Device](#) ()
Finalizer Gets called when the object's memory is about to be reclaimed by the garbage collector.

8.3.1 Detailed Description

Managed class for .NET applications to monitor the Caretaker device.

This class is a wrapper to the unmanaged libcaretaker_dynamic.dll library to allow .NET applications to monitor the Caretaker device. The class is defined in the managed libcaretaker_clr.dll library so the application must reference libcaretaker_clr.dll to access this and supporting managed classes.

8.3.2 Constructor & Destructor Documentation

8.3.2.1 Device()

```
Caretaker::Device::Device (
    DeviceObserver^ observer,
    Boolean autoReconnect )
```

Creates a [Device](#) instance.

The device instance represents a proxy to the Caretaker device to which it has established a connection and provides an interface for the application to monitor the Caretaker device.

IMPORTANT The application must call [ReleaseResources\(\)](#) to explicitly release unmanaged library resources allocated for the device instance when it is no longer needed. Not calling [ReleaseResources\(\)](#) will leave unmanaged resources alive and unaccessible (i.e, zombie resources) that could still maintain the connection to the device after the device instance goes out of scope.

Parameters

<i>observer</i>	The application DeviceObserver implementation to receive Caretaker numeric and waveform data and event notifications in real-time. Passing null disables notifications and the application must call getters to poll for numeric data.
<i>autoReconnect</i>	Set this argument to true to reconnect automatically after the connection is lost. On reconnects, data streams will be re-enabled automatically if enabled prior to the disconnect.

8.3.2.2 ~Device()

```
Caretaker::Device::~~Device ( )
```

Destructor.

Gets called when the object is about to go out of scope, i.e, destroyed.

8.3.3 Member Function Documentation

8.3.3.1 ReleaseResources()

```
void Caretaker::Device::ReleaseResources ( )
```

Release unmanaged library resources allocated for the [Device](#) instance.

The application must call this method to clean things up in the unmanaged code when the [Device](#) instance is no longer needed. Note for C++ applications, the [Device](#) destructor [~Device\(\)](#) may not be called so [ReleaseResources\(\)](#) must be called explicitly to release library resources. Not calling [ReleaseResources\(\)](#) will leave unmanaged resources alive and unaccessible (i.e, zombie resources) that could still maintain the connection to the Caretaker device after the [Device](#) instance goes out of scope.

IMPORTANT: The application must not call [Disconnect\(\)](#) or [ReleaseResources\(\)](#) from any [DeviceObserver](#) methods. The [DeviceObserver](#) methods are called from unmanaged threads that the aforementioned methods attempt to kill. As such, [Disconnect\(\)](#) or [ReleaseResources\(\)](#) must only be called from an application thread, otherwise the application could deadlock.

Also, [ReleaseResources\(\)](#) invalidates the device instance so after calling it the application should set the [Device](#) reference to null to prevent further use.

Also, [ReleaseResources\(\)](#) may delay the calling application thread for a couple seconds as it waits for unmanaged threads to exit.

8.3.3.2 StartScan()

```
Boolean Caretaker::Device::StartScan (
    UInt32 timeout )
```

Initiates scanning for nearby Caretaker devices.

If the serial number or MAC address of the Caretaker device is known, call [ConnectToSerialNumber\(\)](#) or [ConnectToMacAddress\(\)](#) to establish connection with the device. Otherwise, call [StartScan\(\)](#) to scan for nearby Caretaker devices that are advertising. Each device discovered will be notified with [DeviceObserver::OnDeviceDiscovered\(\)](#) passing the discovered device name, serial number, and MAC address to the application. The application must then call [ConnectToSerialNumber\(\)](#) or [ConnectToMacAddress\(\)](#) to connect to the desired, discovered device.

Parameters

<i>timeout</i>	Milliseconds to timeout the scan operation.
----------------	---

Returns

True if scanning was initiated and false otherwise.

8.3.3.3 ConnectToSerialNumber()

```
Boolean Caretaker::Device::ConnectToSerialNumber (
    String^ sn,
    UInt32 timeout )
```

Initiates connecting to the Caretaker device with the specified serial number.

The Caretaker device must be advertising for this method to establish connection.

If the connection sequence was initiated, [DeviceObserver::OnConnectionStatus\(\)](#) will be notified later with the connection status.

Parameters

<i>sn</i>	Serial number of the Caretaker device to connect to. Passing a null or empty address returns failure.
<i>timeout</i>	Milliseconds to timeout connecting.

Returns

True if the connection sequence was initiated and false otherwise. This method also returns false if a previous connection sequence is executing or if the application is already connected to a device. To workaround this scenario, call [Disconnect\(\)](#) before calling this method.

8.3.3.4 ConnectToMacAddress()

```
Boolean Caretaker::Device::ConnectToMacAddress (
    String^ address,
    UInt32 timeout )
```

Initiates connecting to the Caretaker device with the specified MAC address.

The Caretaker device must be advertising for this method to establish connection.

If the connection sequence was initiated, [DeviceObserver::OnConnectionStatus\(\)](#) will be notified later with the connection status.

Parameters

<i>address</i>	MAC address of the Caretaker device to connect to. Passing a null or empty address returns failure.
<i>timeout</i>	Milliseconds to timeout connecting.

Returns

True if the connection sequence was initiated and false otherwise. This method also returns false if a previous connection sequence is executing or if the application is already connected to a device. To workaround this scenario, call [Disconnect\(\)](#) before calling this method.

8.3.3.5 ConnectToAny()

```
Boolean Caretaker::Device::ConnectToAny (
    UInt32 timeout )
```

Initiates connecting to any Caretaker device that is advertising for a connection.

If the connection sequence was initiated, [DeviceObserver::OnConnectionStatus\(\)](#) will be notified later with the connection status.

Parameters

<i>timeout</i>	Milliseconds to timeout connecting.
----------------	-------------------------------------

Returns

True if the connection sequence was initiated and false otherwise. This method also returns false if a previous connection sequence is executing or if the application is already connected to a device. To workaround this scenario, call [Disconnect\(\)](#) before calling this method.

8.3.3.6 Disconnect()

```
Boolean Caretaker::Device::Disconnect ( ) [inline]
```

Initiates disconnecting from the remote device if connected.

If the disconnection sequence was initiated, [DeviceObserver::OnConnectionStatus\(\)](#) will be notified later with connection status.

IMPORTANT: The application must not call [Disconnect\(\)](#) or [ReleaseResources\(\)](#) from any [DeviceObserver](#) methods. The [DeviceObserver](#) methods are called from unmanaged threads that the aforementioned methods attempt to kill. As such, [Disconnect\(\)](#) or [ReleaseResources\(\)](#) must only be called from an application thread, otherwise the application could deadlock.

Returns

Returns true if disconnecting from the device was initiated and false otherwise.

8.3.3.7 StartAutoCal()

```
Boolean Caretaker::Device::StartAutoCal (
    PatientPosture posture )
```

Perform an automatic calibration and then start taking measurements if connected.

If true is returned, [DeviceObserver::OnStartStatus\(\)](#) and [OnDeviceStatus\(\)](#) will be notified later with start and calibration status, respectively.

The application will start receiving data if the start process completed successfully.

Parameters

<i>posture</i>	The patient posture.
----------------	----------------------

Returns

Returns true if start was initiated and false otherwise.

8.3.3.8 StartManualCal()

```
Boolean Caretaker::Device::StartManualCal (
    Int32 systolic,
    Int32 diastolic )
```

Perform a manual calibration and then start taking measurements if connected.

If true is returned, [DeviceObserver::OnStartStatus\(\)](#) and [OnDeviceStatus\(\)](#) will be notified later with start and calibration status, respectively.

The application will start receiving data if the start process completed successfully.

Parameters

<i>systolic</i>	Initial systolic pressure. Acceptable range [80, 250].
<i>diastolic</i>	Initial diastolic pressure. Acceptable range [50, 150].

Returns

Returns true if start was initiated and false otherwise.

8.3.3.9 Stop()

```
Boolean Caretaker::Device::Stop ( )
```

Call this method after calling [StartAutoCal\(\)](#) or [StartManualCal\(\)](#) to stop the device.

This will also stop data and device status notifications.

If true is returned, [DeviceObserver::OnStopStatus\(\)](#) will be notified later with stop status.

Returns

Returns true if stop was initiated and false otherwise.

8.3.3.10 Calibrating()

```
Boolean Caretaker::Device::Calibrating ( ) [inline]
```

Returns true if the device is calibrating, and false otherwise.

Note this is the same as [GetDeviceStatus\(\)](#)->calibrating.

8.3.3.11 Calibrated()

```
Boolean Caretaker::Device::Calibrated ( ) [inline]
```

Returns true if the device is calibrated, and false otherwise.

Note this is the same as [GetDeviceStatus\(\)](#)->calibrated.

8.3.3.12 CalibrationFailed()

```
Boolean Caretaker::Device::CalibrationFailed ( ) [inline]
```

Returns true if the device was calibrating and the calibration failed, and false otherwise.

This method summarizes the various [DeviceStatus](#) flags reporting calibration failure.

8.3.3.13 GetFirmwareVersion()

```
String Caretaker::Device::GetFirmwareVersion ( )
```

Returns the device firmware version or null if the firmware version is not available.

Note

The firmware version is retrieved and cached locally after the connection is established so it will not be available if called from [DeviceObserver::OnConnectionStatus\(\)](#) for the CONNECTED status.

8.3.3.14 GetHardwareVersion()

```
String Caretaker::Device::GetHardwareVersion ( )
```

Returns the device hardware version or null if the hardware version is not available.

Note

The hardware version is retrieved and cached locally after the connection is established so it will not be available if called from [DeviceObserver::OnConnectionStatus\(\)](#) for the CONNECTED status.

8.3.3.15 GetDeviceStatus()

```
DeviceStatus Caretaker::Device::GetDeviceStatus ( ) [inline]
```

Method to poll device status.

Returns the last device status received from the device or null if it is not available.

8.3.3.16 GetBatteryStatus()

```
BatteryStatus Caretaker::Device::GetBatteryStatus ( ) [inline]
```

Method to poll battery status.

Returns the last battery status received from the device or null if it is not available.

8.3.3.17 GetCuffStatus()

```
CuffStatus Caretaker::Device::GetCuffStatus ( ) [inline]
```

Method to poll cuff status.

Returns the last cuff status received from the device or null if it is not available.

8.3.3.18 GetPrimaryVitals()

```
array<PrimaryVitals> Caretaker::Device::GetPrimaryVitals ( ) [inline]
```

Method to poll primary vitals.

Returns array of last primary vitals received from the device or null if they are not available.

8.3.3.19 GetSecondaryVitals()

```
array<SecondaryVitals> Caretaker::Device::GetSecondaryVitals ( ) [inline]
```

Method to poll secondary vitals.

Returns the last secondary vitals received from the device or null if they are not available.

8.3.3.20 GetPulseRateWaveformDataPoints()

```
WaveformDataPoints Caretaker::Device::GetPulseRateWaveformDataPoints ( ) [inline]
```

Method to poll pulse rate waveform data points.

If the application is polling, this method must be called at least 4 times per second to not drop data.

8.3.3.21 GetPulsePressureWaveformDataPoints()

```
WaveformDataPoints Caretaker::Device::GetPulsePressureWaveformDataPoints ( ) [inline]
```

Method to poll pulse pressure waveform data points.

If the application is polling, this method must be called at least 4 times per second to not drop data.

8.3.3.22 IncrementCuffPressure()

```
Boolean Caretaker::Device::IncrementCuffPressure ( )
```

Increments the cuff pressure in 10 mmHg increment.

This method issues a write request to the device to adjust the cuff pressure up 10 mmHg and may take a couple seconds to take effect.

Use [GetCuffStatus\(\)](#) to monitor the target and actual cuff pressure values.

See also

- [DecrementCuffPressure](#)
- [GetCuffStatus](#)
- [VentCuff](#)

Returns

Returns true if the request to the device was initiated and false otherwise.

8.3.3.23 DecrementCuffPressure()

```
Boolean Caretaker::Device::DecrementCuffPressure ( )
```

Decrements the cuff pressure in 10 mmHg increment.

This method issues a write request to the device to adjust the cuff pressure down 10 mmHg and may take a couple seconds to effect.

Use [GetCuffStatus\(\)](#) to monitor the target and actual cuff pressure values.

See also

[IncrementCuffPressure](#)
[GetCuffStatus](#)
[VentCuff](#)

Returns

Returns true if the request to the device was initiated and false otherwise.

8.3.3.24 VentCuff()

```
Boolean Caretaker::Device::VentCuff ( )
```

Deflates the cuff pressure.

This method issues a write request to the device to deflate cuff pressure.

Use [GetCuffStatus\(\)](#) to monitor the target and actual cuff pressure values.

See also

[DecrementCuffPressure](#)
[DecrementCuffPressure](#)
[GetCuffStatus](#)

Returns

Returns true if the request to the device was initiated and false otherwise.

8.3.3.25 PeformDiagnosticsFlush()

```
Boolean Caretaker::Device::PeformDiagnosticsFlush ( )
```

Flushes the device diagnostic plumbing tree.

Returns

Returns true if the request to the device was initiated and false otherwise.

8.3.3.26 WriteSnrMinimum()

```
Boolean Caretaker::Device::WriteSnrMinimum (
    Int32 snr )
```

Writes the device noise filter parameter.

Parameters

<i>snr</i>	The minimum signal-to-noise value. Acceptable range [0, 100].
------------	---

Returns

Returns true if the request to the device was initiated and false otherwise.

See also

[GetCuffStatus](#)

[ReadSnrMinimum](#)

8.3.3.27 ReadSnrMinimum()

```
Boolean Caretaker::Device::ReadSnrMinimum ( )
```

Reads the device noise filter parameter.

Use [GetCuffStatus\(\)](#) to monitor the signal-to-noise (snr) ratio at the device. Use [ReadSnrMinimum\(\)](#) to get an immediate reading of the current value.

On success, the result is notified later with [DeviceObserver::OnReadSnrMinimum\(\)](#).

Returns

Returns true if the request to the device was initiated and false otherwise.

8.3.3.28 WriteMotionTolerance()

```
Boolean Caretaker::Device::WriteMotionTolerance (
    Int32 value )
```

Writes the device motion tolerance parameter.

Parameters

<i>value</i>	Timeout in seconds. Acceptable range [0, 30].
--------------	---

Returns

Returns true if the request to the device was initiated and false otherwise.

See also

[ReadMotionTolerance](#)

8.3.3.29 ReadMotionTolerance()

```
Boolean Caretaker::Device::ReadMotionTolerance ( )
```

Reads the device motion tolerance parameter.

On success, the result is notified later with [DeviceObserver::OnReadMotionTolerance\(\)](#).

Returns

Returns true if the request to the device was initiated and false otherwise.

8.3.3.30 TurnDisplayOff()

```
Boolean Caretaker::Device::TurnDisplayOff ( )
```

Turns the device display screen off.

Returns

Returns true if the request to the device was initiated and false otherwise.

8.3.3.31 TurnDisplayOn()

```
Boolean Caretaker::Device::TurnDisplayOn ( )
```

Turns the device display screen on.

Returns

Returns true if the request to the device was initiated and false otherwise.

8.3.3.32 ReadDisplayState()

```
Boolean Caretaker::Device::ReadDisplayState ( )
```

Reads the display state.

On success, the result is notified later with [DeviceObserver::OnReadDisplayState\(\)](#).

Returns

Returns true if the request to the device was initiated and false otherwise.

8.3.3.33 Recalibrate()

```
Boolean Caretaker::Device::Recalibrate ( )
```

Recalibrate blood pressure measurements.

Returns

Returns true if the request to the device was initiated and false otherwise.

See also

[WriteRecalibrationInterval](#)
[ReadRecalibrationInterval](#)

8.3.3.34 WriteRecalibrationInterval()

```
Boolean Caretaker::Device::WriteRecalibrationInterval (
    UInt32 interval )
```

Writes the calibration interval.

Parameters

<i>interval</i>	The recalibration interval in minutes. Acceptable range [10, 1440].
-----------------	---

Returns

Returns true if the request to the device was initiated and false otherwise.

See also

[Recalibrate](#)
[ReadRecalibrationInterval](#)

8.3.3.35 ReadRecalibrationInterval()

```
Boolean Caretaker::Device::ReadRecalibrationInterval ( )
```

Reads the calibration interval.

On success, the result is notified later with [DeviceObserver::OnReadRecalibrationInterval\(\)](#).

Returns

Returns true if the request to the device was initiated and false otherwise.

See also

[Recalibrate](#)
[WriteRecalibrationInterval](#)

8.3.3.36 WriteWaveformClampSetting()

```
Boolean Caretaker::Device::WriteWaveformClampSetting (
    Byte value )
```

Writes the setting to enable or disable clamping the waveforms.

Parameters

<i>value</i>	Clamp setting: 0 = OFF, 1 = ON.
--------------	---------------------------------

Returns

Returns true if the request to the device was initiated and false otherwise.

See also

[ReadWaveformClampSetting](#)

8.3.3.37 ReadWaveformClampSetting()

```
Boolean Caretaker::Device::ReadWaveformClampSetting ( )
```

Read the waveform clamping setting.

On success, the result is notified later with [DeviceObserver::OnReadWaveformClampSetting\(\)](#).

Returns

Returns true if the request to the device was initiated and false otherwise.

See also

[WriteWaveformClampSetting](#)

8.3.3.38 WriteVitalsFilterSetting()

```
Boolean Caretaker::Device::WriteVitalsFilterSetting (
    Byte value )
```

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

Parameters

<i>value</i>	Filter setting: 0 = OFF, 1 = ON.
--------------	----------------------------------

Returns

Returns true if the request to the device was initiated and false otherwise.

See also

[ReadVitalsFiltersSettings](#)

8.3.3.39 ReadVitalsFilterSetting()

```
Boolean Caretaker::Device::ReadVitalsFilterSetting ( )
```

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

On success, the result is notified later with [DeviceObserver::OnReadVitalsFilterSetting\(\)](#).

Returns

Returns true if the request to the device was initiated and false otherwise.

See also

[WriteVitalsFilterSetting](#)

8.3.3.40 ReadPersistentLog()

```
Boolean Caretaker::Device::ReadPersistentLog ( )
```

Reads the device logs.

On success, the result is notified later with [DeviceObserver::OnReadPersistentLog\(\)](#).

Note

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

8.3.3.41 SetLibraryLogLevel()

```
void Caretaker::Device::SetLibraryLogLevel (
    Int32 level )
```

Sets the library log level.

Note

The library logs are written to standard out, which is the console for console application. For non-console application, you will need to redirect standard out to a file or otherwise to view library log messages.

Parameters

<i>level</i>	One of the following log levels. <ul style="list-style-type: none">• 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
--------------	---

8.3.3.42 EnableSimulationMode()

```
Boolean Caretaker::Device::EnableSimulationMode (
    Boolean 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

<i>mode</i>	Simulation mode: 1 = Enable, 0 = Disable.
-------------	---

Returns

Returns true if the request to the device was initiated and false otherwise.

8.3.3.43 ClearStatus()

```
Boolean Caretaker::Device::ClearStatus ( )
```

Clears the device status.

Returns

Returns true if the request to the device was initiated and false otherwise.

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

- CaretakerDevice.h

8.4 Caretaker::DeviceObserver Class Reference

Managed class defining the application observer interface to receive real-time notifications.

Public Member Functions

- virtual void [OnDeviceDiscovered](#) ([Device](#)[^] device, [String](#)[^] name, [String](#)[^] serialNumber, [String](#)[^] mac↔Address)
Notification sent to report a discovered device after the application called [Device::StartScan\(\)](#).
- virtual void [OnConnectionStatus](#) ([Device](#)[^] device, [ConnectionStatus](#) status)
Notification sent to report [Device::Connect\(\)](#) and [Device::Disconnect\(\)](#) status.
- virtual void [OnStartStatus](#) ([Device](#)[^] device, [StartStatus](#) status)
Notification sent to report [Device::StartAutoCal\(\)](#) and [Device::StartManualCal\(\)](#) status.
- virtual void [OnStopStatus](#) ([Device](#)[^] device, [StopStatus](#) status)
Notification sent to report [Device::Stop\(\)](#) status.
- virtual void [OnPulseRateWaveformDataPoints](#) ([Device](#)[^] device, [WaveformDataPoints](#)[^] dataPoints)
Notification sent to report real-time pulse rate (raw pulse) waveform data.
- virtual void [OnPulsePressureWaveformDataPoints](#) ([Device](#)[^] device, [WaveformDataPoints](#)[^] dataPoints)
Notification sent to report real-time pulse pressure (integrated pulse) waveform data.
- virtual void [OnParameterizedPulse](#) ([Device](#)[^] device, [WaveformDataPoints](#)[^] dataPoints, [Int16](#) t0, [Int16](#) t1, [Int16](#) t2, [Int16](#) t3, [Int32](#) p0, [Int32](#) p1, [Int32](#) p2, [Int32](#) p3)
Notification sent to report the most recent parameterized pulse including pulse snapshot waveform and pulse parameters.
- virtual void [OnDeviceStatus](#) ([Device](#)[^] device, [DeviceStatus](#)[^] status)

- Notification sent to report real-time device status information.*
- virtual void `OnBatteryStatus` (`Device`^ device, `BatteryStatus`^ status)
- Notification sent to report real-time battery status information.*
- virtual void `OnCuffStatus` (`Device`^ device, `CuffStatus`^ status)
- Notification sent to report real-time cuff status information.*
- virtual void `OnPrimaryVitals` (`Device`^ device, array< `PrimaryVitals` >^ vitals)
- Notification sent to report real-time primary vitals.*
- virtual void `OnSecondaryVitals` (`Device`^ device, array< `SecondaryVitals` >^ vitals)
- Notification sent to report real-time secondary vitals.*
- virtual void `OnReadSnrMinimum` (`Device`^ device, Boolean status, Int32 snr)
- Notification sent to report `Device::ReadSnrMinimum()` transaction result.*
- virtual void `OnReadDisplayState` (`Device`^ device, Boolean status, Boolean state)
- Notification sent to report `Device::ReadDisplayState()` transaction result.*
- virtual void `OnReadRecalibrationInterval` (`Device`^ device, Boolean status, UInt32 interval)
- Notification sent to report `Device::OnReadRecalibrationInterval()` transaction result.*
- virtual void `OnReadVitalsFilterSetting` (`Device`^ device, Boolean status, Byte value)
- Notification sent to report `Device::ReadVitalsFilterSetting()` transaction result.*
- virtual void `OnReadMotionTolerance` (`Device`^ device, Boolean status, int timeout)
- Notification sent to report `Device::ReadMotionTolerance()` transaction result.*
- virtual void `OnReadPersistentLog` (`Device`^ device, Boolean status, String^ log)
- Notification sent to report `Device::ReadPersistentTolerance()` transaction result.*
- virtual void `OnReadWaveformClampSetting` (`Device`^ device, Boolean status, Byte value)
- Notification sent to report `Device::ReadWaveformClampSetting()` transaction result.*

8.4.1 Detailed Description

Managed class defining the application observer interface to receive real-time notifications.

Extend this class in the application and override the methods to receive Caretaker data and event notifications.

IMPORTANT: The `DeviceObserver` methods are called from unmanaged threads within the Caretaker library so the application cannot access user interface (UI) elements from these methods directly. Also, the implementations of the methods should be light-weight (i.e., not much processing) since they are called from real-time threads that will degrade data reporting performance if delayed (blocked) for too long. Work must be dispatched to an application thread to process the data. The following C# snippet illustrates one way to dispatch work asynchronously to the application main UI thread.

```
public override void OnConnectionStatus(Device device, onnectionStatus status)
{
    if (status == ConnectionStatus.CONNECTED)
    {
        Application.Current.Dispatcher.BeginInvoke(new Action(() => {
            // add code to update status here
        }));
    }
}
```

Also, the application must not call `Disconnect()` or `ReleaseResources()` from any `DeviceObserver` methods. The `DeviceObserver` methods are called from unmanaged threads that the aforementioned methods attempt to kill. As such, `Disconnect()` or `ReleaseResources()` must only be called from an application thread, otherwise the application may deadlock.

8.4.2 Member Function Documentation

8.4.2.1 OnDeviceDiscovered()

```
virtual void Caretaker::DeviceObserver::OnDeviceDiscovered (
    Device^ device,
    String^ name,
    String^ serialNumber,
    String^ macAddress ) [inline], [virtual]
```

Notification sent to report a discovered device after the application called [Device::StartScan\(\)](#).

The application must call [Device::ConnectToSerialNumber\(\)](#) or [Device::ConnectToMacAddress\(\)](#) to connect to the desired, discovered device.

Parameters

<i>device</i>	The Device instance.
<i>name</i>	The discovered Caretaker device name.
<i>serialNumber</i>	The discovered Caretaker device serial number.
<i>macAddress</i>	The discovered Caretaker device MAC address.

8.4.2.2 OnConnectionStatus()

```
virtual void Caretaker::DeviceObserver::OnConnectionStatus (
    Device^ device,
    ConnectionStatus status ) [inline], [virtual]
```

Notification sent to report [Device::Connect\(\)](#) and [Device::Disconnect\(\)](#) status.

Parameters

<i>device</i>	The Device instance.
<i>status</i>	The connection status.

8.4.2.3 OnStartStatus()

```
virtual void Caretaker::DeviceObserver::OnStartStatus (
    Device^ device,
    StartStatus status ) [inline], [virtual]
```

Notification sent to report [Device::StartAutoCal\(\)](#) and [Device::StartManualCal\(\)](#) status.

Parameters

<i>device</i>	The Device instance.
<i>status</i>	The start status.

8.4.2.4 OnStopStatus()

```
virtual void Caretaker::DeviceObserver::OnStopStatus (
    Device^ device,
    StopStatus status ) [inline], [virtual]
```

Notification sent to report [Device::Stop\(\)](#) status.

Parameters

<i>device</i>	The Device instance.
<i>status</i>	The stop status.

8.4.2.5 OnPulseRateWaveformDataPoints()

```
virtual void Caretaker::DeviceObserver::OnPulseRateWaveformDataPoints (
    Device^ device,
    WaveformDataPoints^ dataPoints ) [inline], [virtual]
```

Notification sent to report real-time pulse rate (raw pulse) waveform data.

Parameters

<i>device</i>	The Device instance.
<i>dataPoints</i>	Pulse rate waveform data points.

8.4.2.6 OnPulsePressureWaveformDataPoints()

```
virtual void Caretaker::DeviceObserver::OnPulsePressureWaveformDataPoints (
    Device^ device,
    WaveformDataPoints^ dataPoints ) [inline], [virtual]
```

Notification sent to report real-time pulse pressure (integrated pulse) waveform data.

Parameters

<i>device</i>	The Device instance.
<i>dataPoints</i>	Pulse pressure waveform data points.

8.4.2.7 OnParameterizedPulse()

```
virtual void Caretaker::DeviceObserver::OnParameterizedPulse (
    Device^ device,
    WaveformDataPoints^ dataPoints,
    Int16 t0,
    Int16 t1,
    Int16 t2,
    Int16 t3,
    Int32 p0,
    Int32 p1,
    Int32 p2,
    Int32 p3 ) [inline], [virtual]
```

Notification sent to report the most recent parameterized pulse including pulse snapshot waveform and pulse parameters.

Parameters

<i>device</i>	The Device instance.
<i>dataPoints</i>	Parameterized pulse waveform data points.
<i>t0</i>	Pulse onset index.
<i>t1</i>	First pulse peak index.
<i>t2</i>	Second pulse peak index.
<i>t3</i>	Third pulse peak index.
<i>p0</i>	Integrated pulse onset value.
<i>p1</i>	First integrated pulse peak value.
<i>p2</i>	Second integrated pulse peak value.
<i>p3</i>	Third integrated pulse peak value.

8.4.2.8 OnDeviceStatus()

```
virtual void Caretaker::DeviceObserver::OnDeviceStatus (
    Device^ device,
    DeviceStatus^ status ) [inline], [virtual]
```

Notification sent to report real-time device status information.

Parameters

<i>device</i>	The Device instance.
<i>status</i>	The device status.

8.4.2.9 OnBatteryStatus()

```
virtual void Caretaker::DeviceObserver::OnBatteryStatus (
    Device^ device,
    BatteryStatus^ status ) [inline], [virtual]
```

Notification sent to report real-time battery status information.

Parameters

<i>device</i>	The Device instance.
<i>status</i>	The battery status.

8.4.2.10 OnCuffStatus()

```
virtual void Caretaker::DeviceObserver::OnCuffStatus (
    Device^ device,
    CuffStatus^ status ) [inline], [virtual]
```

Notification sent to report real-time cuff status information.

Parameters

<i>device</i>	The Device instance.
<i>status</i>	The cuff status.

8.4.2.11 OnPrimaryVitals()

```
virtual void Caretaker::DeviceObserver::OnPrimaryVitals (
    Device^ device,
    array< PrimaryVitals >^ vitals ) [inline], [virtual]
```

Notification sent to report real-time primary vitals.

Parameters

<i>device</i>	The Device instance.
<i>vitals</i>	Array of primary vitals.

8.4.2.12 OnSecondaryVitals()

```
virtual void Caretaker::DeviceObserver::OnSecondaryVitals (
```

```
Device^ device,  
array< SecondaryVitals >^ vitals ) [inline], [virtual]
```

Notification sent to report real-time secondary vitals.

For internal and research use only.

Parameters

<i>device</i>	The Device instance.
<i>vitals</i>	Array of secondary vitals.

8.4.2.13 OnReadSnrMinimum()

```
virtual void Caretaker::DeviceObserver::OnReadSnrMinimum (  
    Device^ device,  
    Boolean status,  
    Int32 snr ) [inline], [virtual]
```

Notification sent to report [Device::ReadSnrMinimum\(\)](#) transaction result.

Parameters

<i>device</i>	The Device instance.
<i>status</i>	Set to true if the transaction succeeded, and false otherwise.
<i>snr</i>	The SNR value if the transaction succeeded.

8.4.2.14 OnReadDisplayState()

```
virtual void Caretaker::DeviceObserver::OnReadDisplayState (  
    Device^ device,  
    Boolean status,  
    Boolean state ) [inline], [virtual]
```

Notification sent to report [Device::ReadDisplayState\(\)](#) transaction result.

Parameters

<i>device</i>	The Device instance.
<i>status</i>	Set to true if the transaction succeeded, and false otherwise.
<i>state</i>	The display state if the transaction succeeded: True if the display is turned on, and false otherwise.

8.4.2.15 OnReadRecalibrationInterval()

```
virtual void Caretaker::DeviceObserver::OnReadRecalibrationInterval (
    Device^ device,
    Boolean status,
    UInt32 interval ) [inline], [virtual]
```

Notification sent to report Device::OnReadRecalibrationInterval() transaction result.

Parameters

<i>device</i>	The Device instance.
<i>status</i>	Set to true if the transaction succeeded, and false otherwise.
<i>interval</i>	The recalibration interval in minutes if the transaction succeeded.

8.4.2.16 OnReadVitalsFilterSetting()

```
virtual void Caretaker::DeviceObserver::OnReadVitalsFilterSetting (
    Device^ device,
    Boolean status,
    Byte value ) [inline], [virtual]
```

Notification sent to report [Device::ReadVitalsFilterSetting\(\)](#) transaction result.

Parameters

<i>device</i>	The Device instance.
<i>status</i>	Set to true if the transaction succeeded, and false otherwise.
<i>value</i>	Filter setting on success: 0 = OFF, 1 = ON.

8.4.2.17 OnReadMotionTolerance()

```
virtual void Caretaker::DeviceObserver::OnReadMotionTolerance (
    Device^ device,
    Boolean status,
    int timeout ) [inline], [virtual]
```

Notification sent to report [Device::ReadMotionTolerance\(\)](#) transaction result.

Parameters

<i>device</i>	The Device instance.
<i>status</i>	Set to true if the transaction succeeded, and false otherwise.
<i>timeout</i>	Motion tolerance timeout in seconds.

8.4.2.18 OnReadPersistentLog()

```
virtual void Caretaker::DeviceObserver::OnReadPersistentLog (
    Device^ device,
    Boolean status,
    String^ log ) [inline], [virtual]
```

Notification sent to report Device::ReadPersistentTolerance() transaction result.

Parameters

<i>device</i>	The Device instance.
<i>status</i>	Set to true if the transaction succeeded, and false otherwise.
<i>log</i>	Device logs.

8.4.2.19 OnReadWaveformClampSetting()

```
virtual void Caretaker::DeviceObserver::OnReadWaveformClampSetting (
    Device^ device,
    Boolean status,
    Byte value ) [inline], [virtual]
```

Notification sent to report [Device::ReadWaveformClampSetting\(\)](#) transaction result.

Parameters

<i>device</i>	The Device instance.
<i>status</i>	Set to true if the transaction succeeded, and false otherwise.
<i>value</i>	Clamp setting: 0 = OFF, 1 = ON.

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

- CaretakerDevice.h

8.5 Caretaker::DeviceStatus Class Reference

Managed class defining device status.

Data Fields

- Boolean [pdaEnabled](#)
An indicator of whether the system PDA measurement system is enabled.

- Boolean [simulationEnabled](#)
An indicator of whether the system is in simulation mode.
- Boolean [pressureControlIndicator](#)
An indicator of whether the system is currently running closed loop pressure control.
- Boolean [inflatedIndicator](#)
An indicator of whether the system has been inflated to pressure.
- Boolean [clockWrapAround](#)
The system clock (time since reset) has wrapped around its index.
- Boolean [batteryVoltageLow](#)
The battery voltage sensor has indicated the battery is near drop-out.
- Boolean [criticalTemperature](#)
The on-board temperature sensor has detected critically high temperature.
- Boolean [pumpOverrun](#)
The pump has violated an overrun condition.
- Boolean [betaProcessing](#)
The system has finished finding the oscillometric curve and is processing the beta (offset) value.
- Boolean [autoCalMode](#)
The system has been started and running in auto-calibration mode.
- Boolean [manualCalMode](#)
The system has been started and running in manual calibration mode.
- Boolean [motionEvent](#)
The system is having trouble getting a good reading due to too much motion.
- Boolean [dataValid](#)
There are valid vital signs measurements.
- Boolean [calibrating](#)
The system is currently calibrating the blood pressure system.
- Boolean [calibrated](#)
The system has current valid calibration.
- Boolean [inflateFailed](#)
Cuff did not inflate to expected value within timeout.
- Boolean [calibrationFailed](#)
Calibration Failure: The calibration values were out of range or oscillometric curve had invalid shape.
- Boolean [poorSignal](#)
Calibration Failure: The system failed to calibrate or timed out process signals so measurements were aborted.
- Boolean [calibrationOffsetFailed](#)
Calibration Failure: The calibration offset calculation failed to identify pulses due to movement.
- Boolean [noPulseTimeout](#)
Calibration Failure: The systems has gone greater than 15 minutes without a valid heart beat.
- Boolean [cuffTooLoose](#)
Calibration Failure: The calibration pump up identified the cuff was too loose.
- Boolean [cuffTooTight](#)
Calibration Failure: The calibration pump up identified the cuff was too tight.
- Boolean [weakSignal](#)
Calibration Failure: Calibration oscillometric curve amplitude is too weak to verify reading.
- Boolean [badCuff](#)
Calibration Failure: The cuff is not holding pressure as expected.
- Boolean [recalSoon](#)
An automatic recalibration will be occurring shortly.
- Boolean [tooManyFails](#)
Calibration Failure: Auto calibration failed too many consecutive times try manual calibration.
- Int16 [calibrationPercentage](#)

Calibration percentage complete.

- Boolean [charging](#)

The device is charging.

- Boolean [chargeComplete](#)

Charging complete.

- Int16 [posture](#)

The last posture used for auto-calibration.

8.5.1 Detailed Description

Managed class defining device status.

[Device](#) status information is notified with [DeviceObserver::OnDeviceStatus\(\)](#) so the application must implement this method to receive it.

8.5.2 Field Documentation

8.5.2.1 dataValid

```
Boolean Caretaker::DeviceStatus::dataValid
```

There are valid vital signs measurements.

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

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

- CaretakerDevice.h

8.6 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_timeout](#))(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_timeout](#))(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()`.

8.6.1 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.

8.6.2 Field Documentation

8.6.2.1 on_device_discovered

```
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

<i>context</i>	The context returned from libct_init() .
<i>device</i>	<p>The discovered device.</p> <p>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.</p> <ul style="list-style-type: none">• libct_device_t::get_address()• libct_device_t::get_name()

See also

[on_discovery_timedout\(\)](#)
[on_discovery_failed\(\)](#)

8.6.2.2 on_discovery_timedout

```
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_timeout(libct_context_t* context)
{
    // do something
}
```

And you can set the function pointer as follows.

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

Parameters

<i>context</i>	The context returned from libct_init() .
----------------	--

See also

[on_device_discovered\(\)](#)
[on_discovery_failed\(\)](#)

8.6.2.3 on_discovery_failed

```
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

<i>context</i>	The context returned from libct_init() .
<i>error</i>	Generic error code describing the failure.

See also

[on_device_discovered\(\)](#)
[on_discovery_timeout\(\)](#)

8.6.2.4 on_device_connected_not_ready

```
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

<i>context</i>	The context returned from libct_init() .
<i>device</i>	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_timeout\(\)](#)

8.6.2.5 on_device_connected_ready

```
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

<i>context</i>	The context returned from libct_init() .
<i>device</i>	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\(\)](#)

8.6.2.6 on_connect_error

```
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

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

See also

[on_device_connected_not_ready\(\)](#)
[on_device_connected_ready\(\)](#)
[on_connect_timeout\(\)](#)

8.6.2.7 on_connect_timeout

```
void( * libct_app_callbacks_t::on_connect_timeout) (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_timeout(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_timeout = on_connect_timeout;
```

Parameters

<i>context</i>	The context returned from libct_init() .
<i>device</i>	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\(\)](#)

8.6.2.8 on_device_disconnected

```
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

<i>context</i>	The context returned from <code>libct_init()</code> .
<i>device</i>	The disconnected device.

8.6.2.9 on_start_monitoring

```
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

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

See also

[on_data_received\(\)](#)

8.6.2.10 on_stop_monitoring

```
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

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

8.6.2.11 on_start_measuring

```
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

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

8.6.2.12 on_stop_measuring

```
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

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

8.6.2.13 on_data_received

```
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

<i>context</i>	The context returned from libct_init() .
<i>device</i>	The device originating the data.
<i>data</i>	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\(\)](#)

8.6.2.14 on_data_error

```
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

<i>context</i>	The context returned from libct_init() .
<i>device</i>	The affected device.
<i>error</i>	String describing the error.

8.6.2.15 on_rd_snr_min_rsp

```
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

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

8.6.2.16 on_wrt_snr_min_rsp

```
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

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

8.6.2.17 on_rd_display_state_rsp

```
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

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

8.6.2.18 on_wrt_display_state_rsp

```
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

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

8.6.2.19 on_rd_recal_itvl_rsp

```
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

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

8.6.2.20 on_wrt_recal_itvl_rsp

```
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

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

8.6.2.21 on_rd_cuff_pressure_rsp

```
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

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

8.6.2.22 on_vent_cuff_rsp

```
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

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

8.6.2.23 on_clr_status_rsp

```
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

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

8.6.2.24 on_diag_flush_rsp

```
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

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

8.6.2.25 on_wrt_waveform_clamping

```
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

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

8.6.2.26 on_rd_waveform_clamping

```
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

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

8.6.2.27 on_rd_vitals_filter

```
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

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

8.6.2.28 on_wrt_vitals_filter

```
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

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

8.6.2.29 on_rd_motion_timeout

```
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

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

8.6.2.30 on_rd_persistent_log

```
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

<i>context</i>	The context returned from libct_init() .
<i>device</i>	The device associated with the context.
<i>log</i>	The device persistent log.
<i>len</i>	Log length.
<i>status</i>	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

8.7 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.

8.7.1 Detailed Description

Battery info data point within the [libct_stream_data_t](#) packet.

8.7.2 Field Documentation

8.7.2.1 voltage

```
int libct_battery_info_t::voltage
```

The battery voltage in millivolts.

8.7.2.2 timestamp

```
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

8.8 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.

8.8.1 Detailed Description

Structure to write the caretaker manual blood pressure settings.

8.8.2 Field Documentation

8.8.2.1 systolic

```
unsigned short libct_bp_settings_t::systolic
```

Systolic pressure setting used for blood pressure calibration.

Acceptable range [80, 250].

8.8.2.2 diastolic

```
unsigned short libct_bp_settings_t::diastolic
```

Diastolic pressure setting used for blood pressure calibration.

Acceptable range [50, 150].

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

- caretaker.h

8.9 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.

8.9.1 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

8.9.2 Field Documentation

8.9.2.1 valid

```
bool libct_cal_curve_t::valid
```

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

8.9.2.2 data_id

```
int libct_cal_curve_t::data_id
```

Data ID.

8.9.2.3 val1

```
float libct_cal_curve_t::val1
```

Value 1.

8.9.2.4 val2

```
float libct_cal_curve_t::val2
```

Value 2.

8.9.2.5 val3

```
float libct_cal_curve_t::val3
```

Value 3.

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

- caretaker.h

8.10 libct_cal_t Struct Reference

Structure used to pass calibration data to [libct_start_measuring\(\)](#).

Data Fields

- int [type](#)
Calibration type.
- union {
 - struct {
 - short [posture](#)
Patient posture.
 - } [auto_cal](#)
Calibration configuration when [type](#) is LIBCT_AUTO_CAL.
 - struct {
 - [libct_bp_settings_t](#) **settings**
 - } [manual_cal](#)
Calibration configuration when [type](#) is LIBCT_MANUAL_CAL.
- } [config](#)

Calibration data.

8.10.1 Detailed Description

Structure used to pass calibration data to [libct_start_measuring\(\)](#).

8.10.2 Field Documentation

8.10.2.1 type

```
int libct_cal_t::type
```

Calibration type.

Set to one of the [calibration types](#).

8.10.2.2 posture

```
short libct_cal_t::posture
```

Patient posture.

Set to one of the [patient postures](#).

8.10.2.3 config

```
union { ... } libct_cal_t::config
```

Calibration data.

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

- caretaker.h

8.11 libct_cal_type_t Class Reference

The Caretaker calibration types.

8.11.1 Detailed Description

The Caretaker calibration types.

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

- caretaker.h

8.12 libct_context_t Class Reference

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

8.12.1 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

8.13 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.

8.13.1 Detailed Description

Cuff pressure data point within the [libct_stream_data_t](#) packet.

8.13.2 Field Documentation

8.13.2.1 valid

```
bool libct_cuff_pressure_t::valid
```

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

8.13.2.2 value

```
int libct_cuff_pressure_t::value
```

cuff pressure actual value.

8.13.2.3 target

```
int libct_cuff_pressure_t::target
```

cuff pressure target value.

8.13.2.4 snr

```
int libct_cuff_pressure_t::snr
```

signal to noise ratio.

8.13.2.5 timestamp

```
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

8.14 libct_device_class_t Class Reference

Classes of devices that can be monitored by this library.

8.14.1 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

8.15 libct_device_state_t Class Reference

The Caretaker device states.

8.15.1 Detailed Description

The Caretaker device states.

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

- caretaker.h

8.16 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](#)
The 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.

8.16.1 Detailed Description

Device status data point within the `libct_stream_data_t` packet.

8.16.2 Field Documentation

8.16.2.1 `simulation_enabled`

```
bool libct_device_status_t::simulation_enabled
```

An indicator of whether the system is in simulation mode.

8.16.2.2 `inflated_indicator`

```
bool libct_device_status_t::inflated_indicator
```

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

8.16.2.3 `clock_wrap_around`

```
bool libct_device_status_t::clock_wrap_around
```

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

8.16.2.4 battery_voltage_low

```
bool libct_device_status_t::battery_voltage_low
```

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

8.16.2.5 pump_overrun

```
bool libct_device_status_t::pump_overrun
```

The pump has violated an overrun condition.

8.16.2.6 ble_temperature_sensor_paired

```
bool libct_device_status_t::ble_temperature_sensor_paired
```

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

8.16.2.7 ble_stream_control

```
bool libct_device_status_t::ble_stream_control
```

The current stream control status bit of the BLE stream.

8.16.2.8 manual_cal_mode

```
bool libct_device_status_t::manual_cal_mode
```

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

8.16.2.9 motion_event

```
bool libct_device_status_t::motion_event
```

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

8.16.2.10 poor_signal

```
bool libct_device_status_t::poor_signal
```

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

8.16.2.11 data_valid

```
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.

8.16.2.12 calibration_offset_failed

```
bool libct_device_status_t::calibration_offset_failed
```

Too much movement.

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

8.16.2.13 weak_signal

```
bool libct_device_status_t::weak_signal
```

Calibration oscillometric curve amplitude is too weak to verify reading.

8.16.2.14 bad_cuff

```
bool libct_device_status_t::bad_cuff
```

The cuff is not holding pressure as expected.

8.16.2.15 ble_adv

```
bool libct_device_status_t::ble_adv
```

The bluetooth module is advertising.

8.16.2.16 recal_soon

```
bool libct_device_status_t::recal_soon
```

An automatic recalibration will be occurring shortly.

8.16.2.17 too_many_fails

```
bool libct_device_status_t::too_many_fails
```

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

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

- caretaker.h

8.17 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.

8.17.1 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.

8.17.2 Field Documentation

8.17.2.1 get_state

```
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

<i>thiz</i>	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\(\)](#)

8.17.2.2 get_class

```
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

<i>thiz</i>	The device instance.
-------------	----------------------

See also

[libct_device_get_class\(\)](#)

8.17.2.3 get_name

```
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

<i>thiz</i>	The device instance.
-------------	----------------------

See also

[libct_device_get_name\(\)](#)

8.17.2.4 get_address

```
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

<i>thiz</i>	The device instance.
-------------	----------------------

See also

[libct_device_get_address\(\)](#)

8.17.2.5 get_serial_number

```
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

<i>thiz</i>	The device instance.
-------------	----------------------

See also

[libct_device_get_serial_number\(\)](#)

8.17.2.6 get_hw_version

```
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

<i>thiz</i>	The device instance.
-------------	----------------------

See also

[libct_device_get_hw_version\(\)](#)

8.17.2.7 get_fw_version

```
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

<i>thiz</i>	The device instance.
-------------	----------------------

See also

[libct_device_get_fw_version\(\)](#)

8.17.2.8 get_context

```
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

<i>this</i>	The device instance.
-------------	----------------------

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

- caretaker.h

8.18 libct_init_data_t Struct Reference

Structure defining initialization data passed to [libct_init\(\)](#).

Data Fields

- int [device_class](#)
The [device class](#).

8.18.1 Detailed Description

Structure defining initialization data passed to [libct_init\(\)](#).

8.18.2 Field Documentation

8.18.2.1 device_class

```
int libct_init_data_t::device_class
```

The [device class](#).

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

- caretaker.h

8.19 libct_monitor_flags_t Class Reference

Data monitor flags passed to [libct_start_monitoring\(\)](#)

8.19.1 Detailed Description

Data monitor flags passed to [libct_start_monitoring\(\)](#)

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

- caretaker.h

8.20 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.

8.20.1 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.

8.20.2 Field Documentation

8.20.2.1 valid

```
bool libct_param_pulse_t::valid
```

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

8.20.2.2 protocol_header

```
short libct_param_pulse_t::protocol_header
```

Date transfer control byte used internally to assemble the data.

8.20.2.3 t0

```
short libct_param_pulse_t::t0
```

Pulse onset time (index).

8.20.2.4 t1

```
short libct_param_pulse_t::t1
```

First pulse peak time (index).

8.20.2.5 t2

```
short libct_param_pulse_t::t2
```

Second pulse peak time (index).

8.20.2.6 t3

```
short libct_param_pulse_t::t3
```

Third pulse peak time (index).

8.20.2.7 p0

```
int libct_param_pulse_t::p0
```

Integrated pulse onset value.

8.20.2.8 p1

```
int libct_param_pulse_t::p1
```

First integrated pulse peak value.

8.20.2.9 p2

```
int libct_param_pulse_t::p2
```

Second integrated pulse peak value.

8.20.2.10 p3

```
int libct_param_pulse_t::p3
```

Third integrated pulse peak value.

8.20.2.11 ibi

```
short libct_param_pulse_t::ibi
```

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

8.20.2.12 as

```
short libct_param_pulse_t::as
```

Arterial stiffness.

8.20.2.13 sqe

```
short libct_param_pulse_t::sqe
```

Signal quality estimate.

8.20.2.14 pressure

```
short libct_param_pulse_t::pressure
```

The most recent raw ADC cuff pressure.

8.20.2.15 time

```
unsigned int libct_param_pulse_t::time
```

Relative system time of occurrence.

8.20.2.16 waveform_len

```
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

8.21 libct_posture_t Class Reference

Patient postures.

8.21.1 Detailed Description

Patient postures.

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

- caretaker.h

8.22 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.

8.22.1 Detailed Description

Pulse oximetry data point within the [libct_stream_data_t](#) packet.

Note

Reserved for future use.

8.22.2 Field Documentation

8.22.2.1 sao2

```
int libct_pulse_ox_t::sao2
```

Blood oxygen level (percentage).

8.22.2.2 pulse_rate

```
int libct_pulse_ox_t::pulse_rate
```

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

8.22.2.3 timestamp

```
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

8.23 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.

8.23.1 Detailed Description

Raw or integrated pulse data point within the [libct_stream_data_t](#) packet.

8.23.2 Field Documentation

8.23.2.1 timestamp

```
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

8.24 libct_pulse_waveform_t Struct Reference

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

Data Fields

- struct {
 libct_pulse_t * datapoints
 unsigned int count
} int_pulse

 Array of integrated waveform pulse data points.
- struct {
 libct_param_pulse_t * datapoints
 unsigned int count
} param_pulse

 Array of parameterize pulse waveform data points.
- long receive_time
 Value of local clock (in milliseconds) when this stream packet was received.

8.24.1 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.

8.24.2 Field Documentation

8.24.2.1 receive_time

```
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

8.25 libct_status_t Class Reference

Function return status codes.

8.25.1 Detailed Description

Function return status codes.

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

- caretaker.h

8.26 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.

8.26.1 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()`

8.26.2 Field Documentation

8.26.2.1 device

```
libct_device_t* libct_stream_data_t::device
```

Reference to the device that generated this data.

8.26.2.2 device_status

```
libct_device_status_t libct_stream_data_t::device_status
```

Device status information.

8.26.2.3 battery_info

```
libct_battery_info_t libct_stream_data_t::battery_info
```

Battery information.

8.26.2.4 datapoints [1/8]

```
libct_vitals_t* libct_stream_data_t::datapoints
```

Array of vital sign datapoints.

8.26.2.5 count

```
unsigned int libct_stream_data_t::count
```

The count of datapoints.

The count of data points.

8.26.2.6 vitals

```
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
    }
}
```

8.26.2.7 datapoints [2/8]

```
libct_cuff_pressure_t* libct_stream_data_t::datapoints
```

Array of cuff pressure data points.

8.26.2.8 cuff_pressure

```
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
    }
}
```

8.26.2.9 datapoints [3/8]

```
libct_temperature_t* libct_stream_data_t::datapoints
```

Array of temperature data points.

8.26.2.10 temperature

```
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
    }
}
```

8.26.2.11 datapoints [4/8]

```
libct_pulse_ox_t* libct_stream_data_t::datapoints
```

Array of spo2 data points.

8.26.2.12 pulse_ox

```
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
    }
}
```

8.26.2.13 datapoints [5/8]

```
libct_vitals2_t* libct_stream_data_t::datapoints
```

Array of secondary vital sign data points.

8.26.2.14 vitals2

```
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
    }
}
```

8.26.2.15 datapoints [6/8]

```
libct_pulse_t* libct_stream_data_t::datapoints
```

Array of raw pulse (pulse rate) data points.

Array of integrated pulse (pulse pressure) data points.

8.26.2.16 raw_pulse

```
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
    }
}
```

8.26.2.17 int_pulse

```
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
    }
}
```

8.26.2.18 datapoints [7/8]

```
libct_param_pulse_t* libct_stream_data_t::datapoints
```

Array of parameterize pulse (pulse snapshot) data points.

8.26.2.19 param_pulse

```
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
    }
}
```


8.26.2.20 datapoints [8/8]

```
libct_cal_curve_t* libct_stream_data_t::datapoints
```

Array of calibration curve data points.

8.26.2.21 cal_curve

```
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
    }
}
```

8.26.2.22 receive_time

```
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

8.27 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.

8.27.1 Detailed Description

Temperature data point within the [libct_stream_data_t](#) packet.

Note

Reserved for future use.

8.27.2 Field Documentation

8.27.2.1 value

```
int libct_temperature_t::value
```

The temperature value.

8.27.2.2 timestamp

```
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

8.28 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

8.28.1 Detailed Description

CareTaker version information.

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

- caretaker.h

8.29 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.

8.29.1 Detailed Description

Secondary Vitals data point within the [libct_stream_data_t](#) packet.

Note

The secondary vitals are for internal use or research only.

8.29.2 Field Documentation

8.29.2.1 blood_volume

```
unsigned short libct_vitals2_t::blood_volume
```

Blood volume in mS.

8.29.2.2 timestamp

```
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

8.30 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.

8.30.1 Detailed Description

Vitals data point within the [libct_stream_data_t](#) packet.

8.30.2 Field Documentation

8.30.2.1 valid

```
bool libct_vitals_t::valid
```

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

8.30.2.2 bp_status

```
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.

8.30.2.3 systolic

```
short libct_vitals_t::systolic
```

Systolic measurement.

8.30.2.4 diastolic

```
short libct_vitals_t::diastolic
```

Diastolic measurement.

8.30.2.5 map

```
short libct_vitals_t::map
```

Mean arterial pressure value.

8.30.2.6 heart_rate

```
short libct_vitals_t::heart_rate
```

Heart rate measurement.

8.30.2.7 respiration

```
short libct_vitals_t::respiration
```

Respiration measurement.

8.30.2.8 as

```
short libct_vitals_t::as
```

AS factor.

8.30.2.9 sqe

```
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 %.

8.30.2.10 timestamp

```
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

8.31 Caretaker::Device::LibraryCallback Class Reference

This is an internal class representing a callback into the unmanaged library code.

8.31.1 Detailed Description

This is an internal class representing a callback into the unmanaged library code.

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

- CaretakerDevice.h

8.32 Caretaker::PrimaryVitals Class Reference

Managed class defining the primary vitals measured by the Caretaker.

Data Fields

- Int16 [systolic](#)
Systolic measurement in mmHg.
- Int16 [diastolic](#)
Diastolic measurement in mmHg.
- Int16 [map](#)
Mean arterial pressure in mmHg.
- Int16 [heartRate](#)
Heart rate measurement in beats per minute (bpm).
- Int16 [respiration](#)
Respiration measurement in breaths per minute (BPM).
- Int16 [asFactor](#)
AS factor.
- Int16 [signalQualityEstimate](#)
Signal quality estimate expressed in percentage.
- Int32 [timestamp](#)
Milliseconds time-stamp from the device associated with the data.

8.32.1 Detailed Description

Managed class defining the primary vitals measured by the Caretaker.

8.32.2 Field Documentation

8.32.2.1 [systolic](#)

```
Int16 Caretaker::PrimaryVitals::systolic
```

Systolic measurement in mmHg.

8.32.2.2 [diastolic](#)

```
Int16 Caretaker::PrimaryVitals::diastolic
```

Diastolic measurement in mmHg.

8.32.2.3 map

```
Int16 Caretaker::PrimaryVitals::map
```

Mean arterial pressure in mmHg.

8.32.2.4 heartRate

```
Int16 Caretaker::PrimaryVitals::heartRate
```

Heart rate measurement in beats per minute (bpm).

8.32.2.5 respiration

```
Int16 Caretaker::PrimaryVitals::respiration
```

Respiration measurement in breaths per minute (BPM).

8.32.2.6 asFactor

```
Int16 Caretaker::PrimaryVitals::asFactor
```

AS factor.

8.32.2.7 signalQualityEstimate

```
Int16 Caretaker::PrimaryVitals::signalQualityEstimate
```

Signal quality estimate expressed in percentage.

Divide by 100 to convert to percentage.

```
displaySQE = MIN(100, vitals.signalQualityEstimate / 100);  
displaySQE = MAX(0, displaySQE);
```

8.32.2.8 timestamp

Int32 Caretaker::PrimaryVitals::timestamp

Milliseconds time-stamp from the device associated with the data.

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

- CaretakerDevice.h

8.33 Caretaker::SecondaryVitals Class Reference

Managed class defining the secondary vitals measured by the Caretaker.

Data Fields

- UInt16 [bloodVolume](#)
Blood volume in mS.
- Byte [cardiacOutput](#)
Cardiac output in L/min.
- UInt16 [interbeatInterval](#)
Inter-beat Interval in mS.
- UInt16 [lvet](#)
Left ventricular ejection time.
- Single [p2p1Ratio](#)
P ratio.
- Single [hrComp](#)
hrComp
- Single [pr](#)
pr
- Int32 [timestamp](#)
Milliseconds time-stamp from the device associated with the data.

8.33.1 Detailed Description

Managed class defining the secondary vitals measured by the Caretaker.

Only for internal or research use.

8.33.2 Field Documentation

8.33.2.1 bloodVolume

```
UInt16 Caretaker::SecondaryVitals::bloodVolume
```

Blood volume in mS.

8.33.2.2 timestamp

```
Int32 Caretaker::SecondaryVitals::timestamp
```

Milliseconds time-stamp from the device associated with the data.

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

- CaretakerDevice.h

8.34 Caretaker::WaveformDataPoints Class Reference

Pulse waveform data points Each data point is of format (t,v), where t is a monotonically incrementing counter representing time-stamp, and v is the pulse sample at time t.

Public Member Functions

- **WaveformDataPoints** (array< Int32 > ^ [values](#), array< Int32 > ^ [timestamps](#))

Data Fields

- array< Int32 > [values](#)
The waveform v values.
- array< Int32 > [timestamps](#)
The waveform t values.

8.34.1 Detailed Description

Pulse waveform data points Each data point is of format (t,v), where t is a monotonically incrementing counter representing time-stamp, and v is the pulse sample at time t.

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

- CaretakerDevice.h