X2E C++ API Guide

Common Conventions

- `id` is an unsigned 32-bit integer (-1 is invalid) representing the marker's ID
- `time` is an unsigned 64-bit integer (-1 is invalid) representing the time when data was acquired in frames (for libowlsock, time is different for other libowl/owlserver implementations)
- `pose` is the position `[x, y, z]` and quaternion rotation `[s, x, y, z]`
- `options`
  - Options are represented by key-value pairs: `key=value`
  - Multiple pairs are separated by spaces: `key1=value1 key2=value2 ...`
  - Multiple values belonging to one key are separated by commas: `key=value1,value2,value3,...`

Data Structures

Tracker Data Structures

Camera	`uint32_t id;`
	`uint32_t flags;`
	`float pose[7];`
	`float cond;`

Peak	`uint32_t id;`
	`uint32_t flags;`
	• `0x0001` = LED too high/low
	• `0x0010` = Predicted
	• `0x0100` = Amplitude too low
	• `0x0200` = Amplitude too high
	• `0x0400` = 3D distance rejected
	• `0x0800` = Unidentified pattern
	• `0x1000` = Internal
	`int64_t time;`
	`uint16_t camera;`
	`uint16_t detector;`
	`uint32_t width;`
	`float pos;`
	`float amp;`

Plane	`uint32_t id;`
	`uint32_t flags;`
	• `0x0001` = LED too high/low
	• `0x0010` = Predicted
	• `0x0100` = Amplitude too low

		• 0x0200 = Amplitude too high
		• 0x0400 = 3D distance rejected
		• 0x0800 = Unidentified pattern
		• 0x1000 = Internal
		`int64_t time;`
		`uint16_t camera;`
		`uint16_t detector;`
		`float plane[4];`
		`float cond;`

Marker		`uint32_t id;`
		`uint32_t flags;`
		• 0x000F = Slot number
		• 0x0010 = Predicted
		• 0x0100 = 3D rejected
		`int64_t time;`
		`float x, y, z;`
		`float cond;`
		• Condition number of plane intersection matrix
		• Low positive numbers are good condition values, negative numbers are not

Rigid		`uint32_t id;`
		`uint32_t flags;`
		• Mask 0x4 = Kalman estimate
		`int64_t time;`
		`float pose[7];`
		`float cond;`
		• Number of constraint planes
		• Higher positive numbers are good condition values, negative numbers are not

Input		`uint64_t hw_id;`
		`uint64_t flags;`
		`int64_t time;`
		`std::vector<uint8_t> data;`
		• Arbitrary data transport
		• Associated with DeviceInfo by hw_id

Typedef Declarations for Tracker Structures

```
std::vector<Camera> Cameras;
std::vector<Peak>   Peaks;
std::vector<Plane>  Planes;
std::vector<Marker> Markers;
std::vector<Rigid>  Rigids;
std::vector<Input>  Inputs;
```

Information Data Structures

MarkerInfo		`uint32_t id;`
		`uint32_t tracker_id;`
		`std::string name;`
		`std::string options;`

TrackerInfo		`uint32_t id;`
		`std::string type;`
		Type can be `point` or `rigid`
		`std::string name;`
		`std::string options;`
		`std::vector<uint32_t> marker_ids;`

FilterInfo		`uint32_t period;`
		`std::string name;`
		`std::string options;`
		• Options key-value pairs: `Type=value`
		• Multiple filter types supported
		• Filter Types:
		◦ `noop` - No operation
		◦ `linear` - Linear interpolation
		◦ `spline` - Cubic hermite spline interpolation
		◦ `average` - Averaging interpolation

DeviceInfo		`uint32_t hw_id;`
		• Unique hardware identifier
		`uint32_t id;`
		• Unique identifier within same type
		`std::string name;`
		`std::string options;`
		`std::string status;`
		`int64_t time;`
		`std::string type;`
		• Device type: none, hub, camera, stylus, puck, controller

Polymorphic Data Structures

Type		• Polymorphic data structure accessor • Converts data and type identifier to type (const T*) or (T) • Type operators: ◦ `template <typename T> operator const T*() const;` ▪ Return value: • Pointer to value converted to `T` on success • 0 on error ◦ `template <typename T> operator T() const;` ▪ Return value: • Value converted to `T` on success

		• Value of `T()` on error
Variant		• Stores one or more values of a single type • Data accessors: ◦ `uint16_t type_id() const;` ◦ `uint32_t flags() const;` ◦ `const char* type_name() const;` ◦ `bool valid() const;` ◦ `bool empty() const;` ◦ `const Type begin() const;` ◦ `const Type end() const;` ◦ `template <typename T> size_t get(T &v) const;` ▪ `T` is STL compatible container ▪ Fill container `v` with values ▪ Return value: • Number of elements in `v` on success • 0 on error ◦ `std::string str() const;` ▪ Recomended for string types ▪ Return value: • value converted to string on success • empty string on error • Operators (can also be used to access data): ◦ `Variant& operator=(const Variant &v);` ◦ `template <typename T> operator T() const;` ▪ Return value: • Value converted to `T` on success • Value of `T()` on error ◦ `template <typename T> operator std::vector<T>() const;` ▪ Return value: • `std::vector<T>` containing value(s) on success • Empty `std::vector<T>` on error
Event		• Named polymorphic data structure • Primary data transport from OWL server to OWL client • Can contain child Events • Data accessors: ◦ `begin()` and `end()` iterators ◦ `uint16_t type_id() const;` ◦ `uint16_t id() const;` ◦ `uint32_t flags() const;` ◦ `int64_t time() const;` ◦ `const char* type_name() const;`

		◦ `const char* name() const;` ◦ `template <typename T> size_t size() const;` ▪ Return value: • Number of elements of type `T` on success • 0 on error ◦ `template <typename T> size_t get(T &v) const;` ▪ `T` is STL compatible container ▪ Fill `v` with values ▪ Return value: • Number of elements in `v` on success • 0 on error ◦ `std::string str() const;` ▪ Return value: • Value converted to string on success • Empty string on error ◦ `const Event* find(uint16_t type_id, const std::string &name) const;` ▪ Find self or child event of either `type_id` or `name` ▪ Return value: • `const Event` pointer to self or child `Event` on success • 0 on error

Context Class

Functions & Accessors

`int open(`
`  const std::string &name,`
`  const std::string &open_options=std::string()`
`)`	

• Opens connection to name • Blocks until success/failure or timeout expired • Clears properties • Sets properties: `name`, `opened` • `open_options`: ◦ `timeout` (in μsec, default 5s)	• Return value: ◦ > 0 on success ◦ 0 on timeout ◦ < 0 on error (`lastError` is set)
`bool close()`	
• Calls `done()` and closes connection	• Return value: success or failure
`bool isOpen() const`	
	• Return value: connection open or closed
`int initialize(`	

```
    const std::string &init_options
)
```

- Initializes opened connection
- Blocks until success/failure or timeout expired
- Clears properties
- Sets properties: `initializing`, `initialized`
- Multiple connections are handled master/slave(s)
  - Single master connection is allowed
  - First connection defaults to master (`slave=0`)
  - Subsequent connectsion are slave (`slave=1`)
  - First connection can be forced to be slave with `slave=1`, will wait for master connection until timeout
- `init_options`:
  - `timeout` (in μsec, default 5s)
  - `slave`
  - `profile`
  - `event`
  - `streaming`
  - `frequency`
  - `timebase` (numerator, denominator)
  - `scale`
  - `pose` (7 values)

- Return value:
  - > 0 on success (Creates Event BYTE `initialized`)
  - 0 on timeout
  - < 0 on error (Sets `lastError`)

```
int done(
    const std::string &done_options=std::string()
)
```

- Un-initializes initialized connection
- Blocks until all remaining events are received or timeout expires
- Sets properties: `flushing`, `initialized`, `streaming`, `frequency`
- done_options:
  - timeout (in μsec, default 1s)

- Return value:
  - > 0 on success (Creates Event BYTE `done`)
  - 0 on timeout = 0
  - < 0 on failure (Sets `lastError`)

```
int streaming() const
```

	• Return value: stream enabled/disabled

`bool streaming(`
`  int enable`
`)`	

• Enables/disables streaming
• Valid parameters:
  ◦ 1: Streaming over TCP
  ◦ 2: Streaming over UDP
  ◦ 3: Streaming over UDP broadcast
	• Return value: success/failure (Upon success, creates Event INT `streaming`)

`float frequency() const`	
	• Return value: stream frequency

`bool frequency(`
`  float freq`
`)`	

• Sets stream frequency	• Return value: success/failure (Upon success, creates Event FLOAT `frequency`)
`const int* timeBase() const`	

	• Return value:
	  ◦ Pointer to timebase pair on success
	  ◦ 0 on error

`bool timeBase(`
`  int num,`
`  int den`
`)`	

• Set timebase numerator and denominator	• Return value: success/failure (Upon success, creates Event INT `timebase`)
`float scale() const`	
	• Return value: scale factor

`bool scale(`
`  float scale`
`)`	

• Set scale factor	• Return value: success/failure (Upon success, creates Event FLOAT `scale`)
`const float* pose() const`	

	• Return value:
	  ◦ Pointer to pose (7 values)
	  ◦ 0 on error

```
bool pose(
     const float *pose
)
```

- Set pose (7 values)

- Return value: success/failure (Upon success, creates Event FLOAT `scale`)

```
std::string option(
     const std::string &option
) const
```

- Return value:
  - Option's value on success
  - Empty string on error

```
std::string options() const
```

- Return value:
  - All options, separated by spaces
  - Empty string on error

```
bool option(
     const std::string &option,
     const std::string &value
)
```

- Sets option to value

- Return value: success/failure

```
bool options(
     const std::string &options
)
```

- Sets options separated by space

- Return value: success/failure

```
std::string lastError() const
```

- Return value: last error, or empty string

```
bool markerName(
     uint32_t marker_id,
     const std::string marker_name
)
```

- Set name of marker

- Return value: success/failure

```
bool markerOptions(
     uint32_t marker_id,
     const std::string &marker_options
)
```

- For markers in rigid body tracker:
  - `pos=vector (x, y, z)` rigid

- Return value: success/failure

body coordinates	

`const MarkerInfo markerInfo(`
`  uint32_t marker_id`
`) const`	

	• Return value: MarkerInfo of marker with given ID

`bool createTracker(`
`  uint32_t tracker_id,`
`  const std::string &tracker_type,`
`  const std::string &tracker_name = std::string(),`
`  const std::string &tracker_options = std::string()`
`)`	

• Create a tracker	• Return value: success/failure

`bool createTrackers(`
`  const TrackerInfo *first,`
`  const TrackerInfo *last`
`)`	

• Create multiple trackers	• Return value: success/failure

`bool destroyTracker(`
`  uint32_t tracker_id`
`)`	

• Remove a tracker	• Return value: success/failure

`bool destroyTrackers(`
`  const uint32_t *first,`
`  const uint32_t *last`
`)`	

• Remove multiple trackers	• Return value: success/failure

`bool assignMarker(`
`  uint32_t tracker_id,`
`  uint32_t marker_id,`
`  const std::string &marker_name = std::string(),`
`  const std::string &marker_options = std::string()`
`)`	

• Assign marker to tracker • Removes marker from previous tracker	• Return value: success/failure

`bool assignMarkers(`
`  const MarkerInfo *first,`
`  const MarkerInfo *last`
`)`	

• Assign markers to tracker	• Return value: success/failure

```
bool trackerName(
     uint32_t tracker_id,
     const std::string &tracker_name
)
```

• Set name of tracker	• Return value: success/failure

```
bool trackerOptions(
     uint32_t tracker_id,
     const std::string &tracker_options
)
```

• Set tracker options
• For rigid body tracker:
  ◦ `init=value` (4 numbers)
  ◦ default `init=1e-3, 1, 0.5, 100`
	• Return value: success/failure

```
const TrackerInfo trackerInfo(
     uint32_t tracker_id
) const
```

	• Return value: TrackerInfo of tracker with given ID

```
bool filter(
     uint32_t period,
     const std::string &name,
     const std::string &filter_options
)
```

• Create filter	• Return value: success/failure

```
bool filters(
     const FilterInfo *first,
     const FilterInfo *last
)
```

• Create multiple filters	• Return value: success/failure

```
const FilterInfo filterInfo(
     const std::string &name
) const
```

	• Return value: FilterInfo of filter with given name

```
const DeviceInfo deviceInfo(
     uint64_t hw_id
) const
```

	• Return value: DeviceInfo of device with given hardware ID

```
const Event* peekEvent(
```

```
    long timeout = 0
);
```

• Look at next event without removing it from the event queue	• Return value: Pointer to next Event

```
const Event* nextEvent(
    long timeout = 0
);
```

• Get next event, removes it from the event queue	• Return value: Pointer to next Event

```
const Variant property(
    const std::string &name
) const
```

• Property types: ◦ Integers ▪ opened ▪ initializing ▪ initialized ▪ slave ▪ streaming ▪ systemtimebase ▪ timebase ▪ markers ▪ trackers ◦ Floats ▪ maxfrequency ▪ systemfrequency ▪ frequency ▪ scale ▪ systempose ▪ pose ◦ String ▪ name ▪ profile ▪ filters ▪ profiles ▪ defaultprofile ▪ profiles.json ◦ Camera	• Return value: ◦ Read-only property value on success ◦ Empty/invalid Variant on error

▪ `systemcameras` ▪ `cameras` ◦ MarkerInfo `markerInfo` ◦ TrackerInfo `trackerInfo` ◦ FilterInfo `filterInfo` ◦ DeviceInfo `deviceInfo`	

`template <typename T> T property(`
`  const std::string &name`
`) const`	

	• Return value: ◦ Property value converted to `T` on success ◦ Value of `T()` on error

Scan Class

Scan for OWL servers (owld)

`bool send(`
`  const std::string &message`
`);`	

• Send message to all OWL servers on local network	• Return value: success/failure

`std::vector<std::string> listen(`
`  long timeout = 0`
`);`	

• Receive messages from one or more OWL servers after `timeout` • Value: `ip=address` (message from server)	• Return value: ◦ `std::vector<std::string>` containing OWL server messages on success ◦ Empty `std::vector<std::string>` on error or timeout