Gestures Library Reference

QNX Software Systems Limited
1001 Farrar Road
Ottawa, Ontario
K2K 0B3
Canada

Voice: +1 613 591-0931
Fax: +1 613 591-3579
Email: info@qnx.com
Web: http://www.qnx.com/

Electronic edition published: Tuesday, October 7, 2014

Table of Contents

About Gestures

The Gestures library provides gesture recognizers to detect gestures through touch events that occur when you place one or more fingers on a touch screen.

The *Gestures Library Reference* is intended for application developers. This table may help you find what you need in this guide:

To find out about:	See:
Gestures Overview	*Gestures Library Overview* (p. 11)
Gesture Recognition	*Gesture Recognition* (p. 17)
Creating custom gestures	*Custom Gestures* (p. 25)
Tutorials	*Gesture Tutorials* (p. 31)
Gestures API	*Gestures Library Reference* (p. 47)

Typographical conventions

Throughout this manual, we use certain typographical conventions to distinguish technical terms. In general, the conventions we use conform to those found in IEEE POSIX publications.

The following table summarizes our conventions:

Reference	Example
Code examples	`if( stream == NULL)`
Command options	`-lR`
Commands	`make`
Constants	`NULL`
Data types	`unsigned short`
Environment variables	***PATH***
File and pathnames	`/dev/null`
Function names	*exit()*
Keyboard chords	**Ctrl–Alt–Delete**
Keyboard input	`Username`
Keyboard keys	**Enter**
Program output	`login:`
Variable names	*stdin*
Parameters	*parm1*
User-interface components	**Navigator**
Window title	**Options**

We use an arrow in directions for accessing menu items, like this:

You'll find the Other... menu item under **Perspective**   **Show View**.

We use notes, cautions, and warnings to highlight important messages:

Notes point out something important or useful.

Cautions tell you about commands or procedures that may have unwanted or undesirable side effects.

Warnings tell you about commands or procedures that could be dangerous to your files, your hardware, or even yourself.

Note to Windows users

In our documentation, we typically use a forward slash (/) as a delimiter in pathnames, including those pointing to Windows files. We also generally follow POSIX/UNIX filesystem conventions.

Technical support

Technical assistance is available for all supported products.

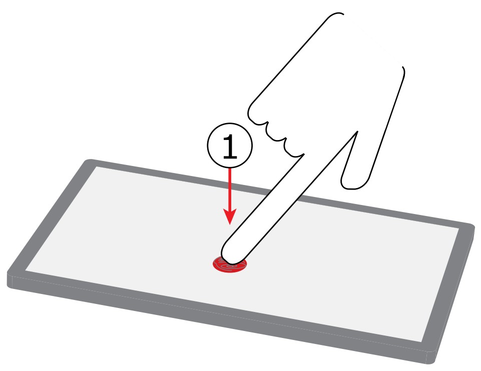
To obtain technical support for any QNX product, visit the Support area on our website (*www.qnx.com*). You'll find a wide range of support options, including community forums.

Chapter 1
Gestures Library Overview

The Gestures library provides gesture recognizers to detect gestures through touch events that occur when you place one or more fingers on a touch screen.

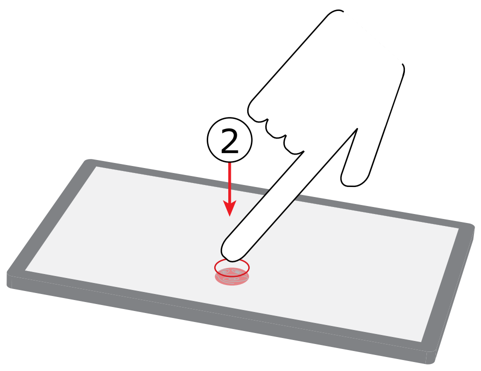
What are gestures?

A gesture is an interaction between users and your application through a touch screen.



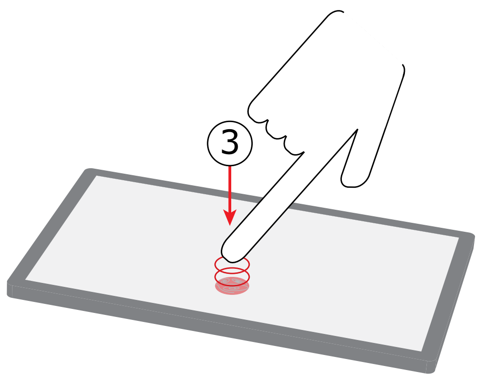
Tap

One touch with one finger.



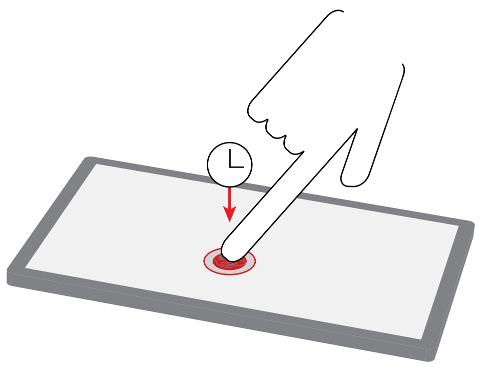
Double tap

Two touches in quick succession with one finger.



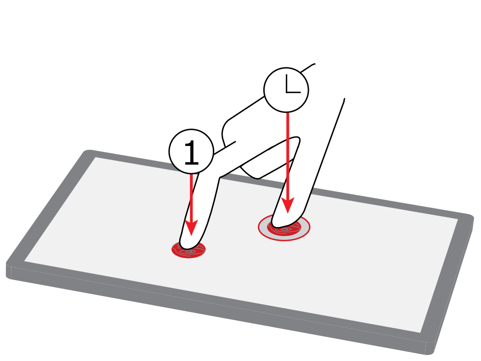
Triple tap

Three touches in quick succession with one finger.



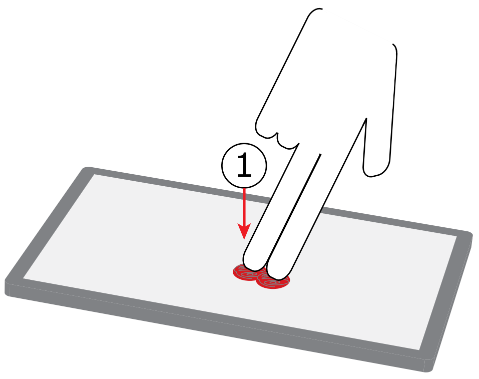
Long press

One touch with a pause before releasing.



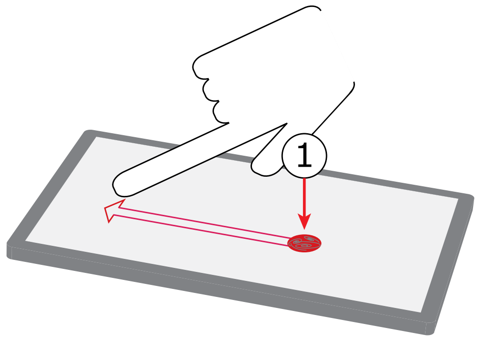
Press and tap

One long press with one finger, followed by a touch with a second finger.



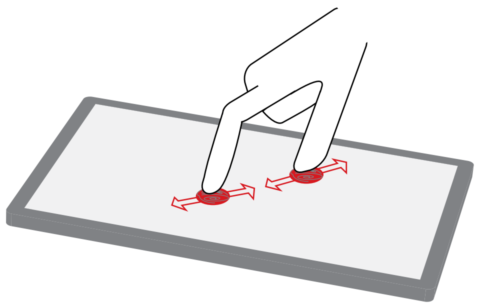
Two-finger tap

One touch with two fingers.



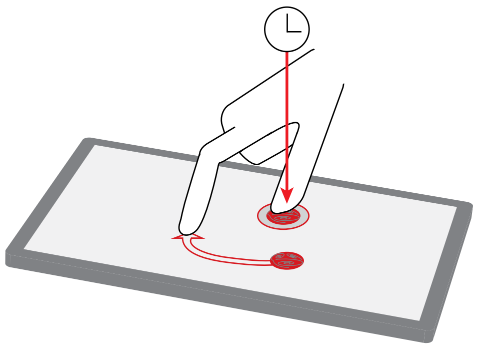
Swipe

One continuous horizontal or vertical motion consisting of a tap, a movement, and a release with one finger.



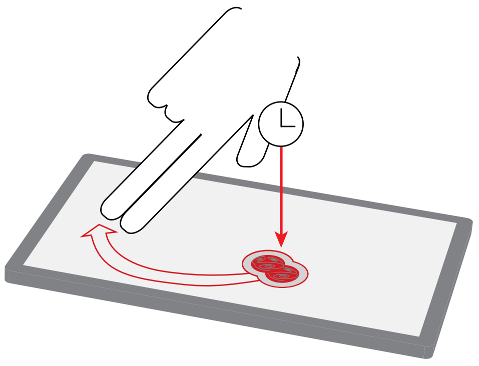
Pinch

One continuous motion consisting of a two-finger tap, a movement inward or outward, and a release.



Rotate

One long press with one finger, followed shortly by a swipe in an arc by your second finger



Two-finger pan

One continuous motion consisting of a long press, a movement, and a release with two fingers

There are two classes of gestures:

Composite

A composite (also called transform or continuous) gesture is one that may send multiple notifications to the application as you continue to touch or move your fingers on the screen. The composite gestures are:

- Swipe
- Pinch
- Rotate
- Two-finger pan

Discrete

Discrete gestures send only a single notification to the application. The discrete gestures are:

- Tap
- Double tap
- Triple tap
- Long press
- Press and tap
- Two-finger tap

What are gesture recognizers?

A gesture recognizer is a self-contained state machine. It progresses through its various states in reaction to the touch events that it receives. Depending on the gesture, a gesture recognizer may need to interpret single- or multiple-touch events in order to detect a single gesture. That is, a gesture recognizer could transition through multiple states before being able to determine the gesture that the user intended. Once the gesture recognizer detects a gesture, its gesture callback is invoked. It is the responsibility of your application to handle the detected gesture in your gesture callback function. In the context of the Gestures library, you will see that the gesture recognizers are simply referred to as gestures.

Gesture recognizers for some of the widely used gestures are already provided for you by the Gestures library:

Gesture Recognizer	Gesture class	Uses timer(s)
Tap	Discrete	
Double tap	Discrete	

Gesture Recognizer	Gesture class	Uses timer(s)
Triple tap	Discrete	
Long press	Discrete	
Press and tap	Discrete	
Two-finger tap	Discrete	
Swipe	Composite	
Pinch	Composite	
Rotate	Composite	
Two-finger pan	Composite	

Some gesture recognizers use timers as part of their detection of gestures. If you need to detect a gesture that isn't supported by the Gestures library, you can define your own gesture recognizer.

What are touch events?

Touch events are events generated by the touch controller. In the Gestures library, touch events are represented by the data structure `mtouch_event_t`. `mtouch_event_t` contains various types of information about the touch event, such as the type of touch event (touch, move, or release), the timestamp of the event, the coordinates of the touch, and so on. Touch events that are represented by this data structure are referred to as mtouch events. See the file input/event_types.h for more information.

What are gesture sets?

A gesture set is a set of gesture recognizers; it can detect multiple types of gestures. Your application defines a gesture set by allocating, to the set, the gesture recognizers that detect the gestures that are of interest to you.

You can think of a gesture set as the interface between gesture recognizers and the application. The application sends mtouch events to a gesture set, not to individual gesture recognizers. Individual gesture recognizers must belong to a gesture set to be able to receive mtouch events and invoke callback functions when the gesture recognizer state transitions happen.

Inter-gesture recognizer relationships and dependencies are to be managed completely by the gesture set. Therefore, individual gesture recognizers are simple to implement and allow the application to customize the desired gesture-recognizer relationships for its own needs. Also, applications that are interested only in a small subset of gestures can choose its gesture recognizers of interest.

A gesture set handles all that is required of the state transition. Examples of what the gesture set handles are:

- invoking the gesture recognizer callback function when valid
- resetting each gesture recognizer when all those in the gesture set have transitioned to either `GESTURE_STATE_COMPLETE` or `GESTURE_STATE_FAILED`
- managing failure dependencies between gesture recognizers
- timer handling for gesture recognizers that need timer events (as opposed to mtouch events)
- failure notification when all gestures recognizers in a set have transitioned to `GESTURE_STATE_FAILED`

For information about:	See:
Recognizing gestures	*Gesture recognition* (p. 17) *Simultaneous recognition* (p. 19)
State transitions of gesture recognizers	*State transitions* (p. 17)
The gesture callback	*Callback invocation* (p. 19)
Timers	*Timer support* (p. 19) *Timer callback* (p. 27)
Failures	*Failure dependencies* (p. 21) *Failure notifications* (p. 22)
Resets	*Reset* (p. 22)
Defining your own gesture recognizer	*Custom gestures* (p. 25)
Tutorials	*Gestures Tutorials* (p. 31)
What's in the Gestures library?	*Gestures Library Reference* (p. 47)

Chapter 2
Gesture Recognition

Gesture recognition is performed by gesture recognizers that detect gestures through touch events.

Applications are responsible for selecting the gestures they're interested in and adding the corresponding gesture recognizers to one or more gesture sets to achieve the desired recognition behavior. When a gesture-recognizer state transition occurs, the gesture set invokes the application callback function for that gesture. Applications can also add their own gesture recognizers if they need specialized or custom gesture recognition.

State transitions

It's important to understand the states and valid state transitions of the gesture recognizers. Understanding is especially important when you're defining your own custom gesture recognizer because you need to provide the function to handle these state transitions:

```
gesture_state_e (*process_event)(struct contact_id_map* map,
                                 struct gesture_base* gesture,
                                 mtouch_event_t* event,
                                 int* consumed)
```

Note that composite gestures and discrete gestures go through different state transitions.

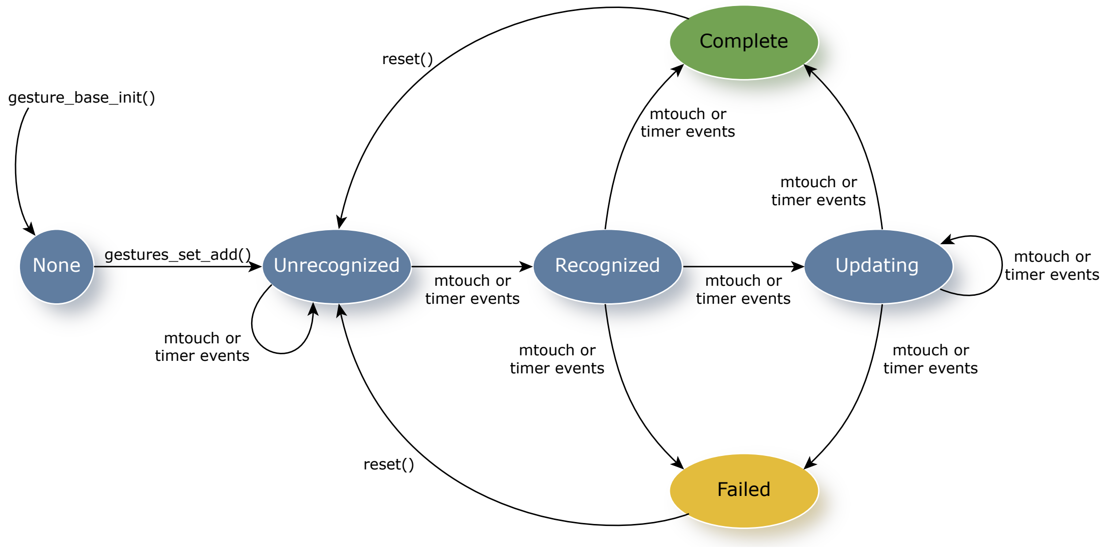


Figure 1: Gesture-recognizer state transitions for composite gestures

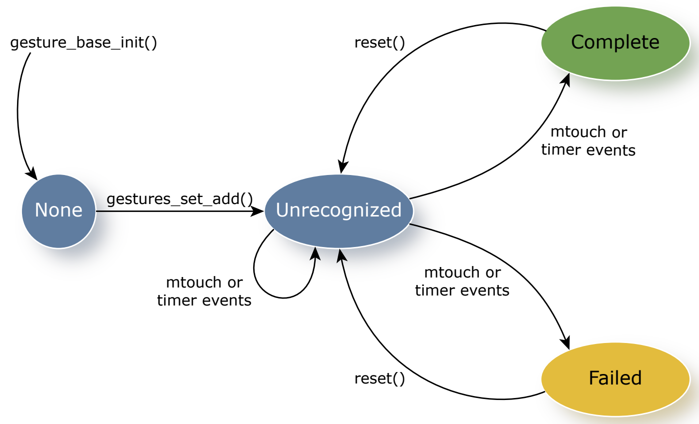


Figure 2: Gesture-recognizer state transitions for discrete gestures

---

`GESTURE_STATE_RECOGNIZED` and `GESTURE_STATE_UPDATING` aren't valid states for recognizers of discrete gestures; these gesture recognizers transition directly from `GESTURE_STATE_UNRECOGNIZED` to either `GESTURE_STATE_FAILED` or `GESTURE_STATE_COMPLETE`.

State transitions are specific to the gesture that the recognizer is detecting. Most state transitions are a result of mtouch or timer events that are of interest to the gesture recognizer.

None (`GESTURE_STATE_NONE`)

The initial state of a gesture recognizer; you can intialize the state to `GESTURE_STATE_NONE` by calling *gesture_base_init()*.

Unrecognized (`GESTURE_STATE_UNRECOGNIZED`)

The state of a gesture recognizer after it has been added to a gesture set; it is now ready to receive mtouch and timer events. The gesture recognizer returns to this state after *reset()* is called by the gesture set.

Recognized (`GESTURE_STATE_RECOGNIZED`)

The state of a gesture recognizer after it has received one mtouch or timer event that moves the gesture recognizer to `GESTURE_STATE_COMPLETE`.

Updating (`GESTURE_STATE_UPDATING`)

The state of a gesture recognizer while it's receiving mtouch or timer events that move the gesture recognizer to `GESTURE_STATE_COMPLETE`.

Complete (`GESTURE_STATE_COMPLETE`)

The state of a gesture recognizer when it has received all mtouch or timer events that fulfill the requirements of detecting its gesture.

Failed (`GESTURE_STATE_FAILED`)

The state of a gesture recognizer when requirements of detecting its gesture aren't fulfilled.

Simultaneous recognition

Unless there is a failure dependency indicated, each gesture recognizer in a gesture set receives all mtouch events sent to the gestures set; this means your application can simultaneously recognize several gestures.

A single mtouch event can result in the invocation of multiple gesture-recognizer callback functions because several gestures can be recognized simultaneously.

Callback invocation

The gesture-recognizer callback function is provided by your application. It defines what the application does when a gesture is recognized or updated:

```
void(*gesture_callback_f)(struct gesture_base* gesture,
                          mtouch_event_t* event,
                          void* param, int async);
```

The parameter *gesture* contains information about the gesture, *event* contains information about the mtouch event that caused the gesture, and *async* identifies whether this callback was invoked from an mtouch event (*async* = 0) or from a timer event (*async* = 1).

This callback function is invoked as a result of either an mtouch or a timer event. Although your gesture callback will be invoked on other state transitions, your application is usually interested only when the gesture has been recognized or not. That is, you would usually implement application behavior based on your gesture recognizer in the `GESTURE_STATE_COMPLETE` or the `GESTURE_STATE_FAILED` state.

The goal of this callback function is to identify the gesture that is recognized based on mtouch events that are received and to define your application's actions based on the gesture that is recognized. Typically, your application copies information from the incoming mtouch event to a local structure and uses that information accordingly.

Timer support

Events

A gesture recognizer can change states on timer events. Gesture callback functions are invoked after the invocation of the timer callback from the context of the timer thread. The timer thread doesn't detect mtouch events.

Gesture callbacks of gesture recognizers that have timers (e.g., double tap, triple tap, or long press), can be invoked from both the application thread or the timer thread.

The processing performed while in your gesture callback will block either mtouch or timer events, depending on which thread invoked the callback.

The following diagrams show that the gesture set is shared between two separate threads. Each of these threads (application and timer) can be blocked by the other thread when accessing the shared data.

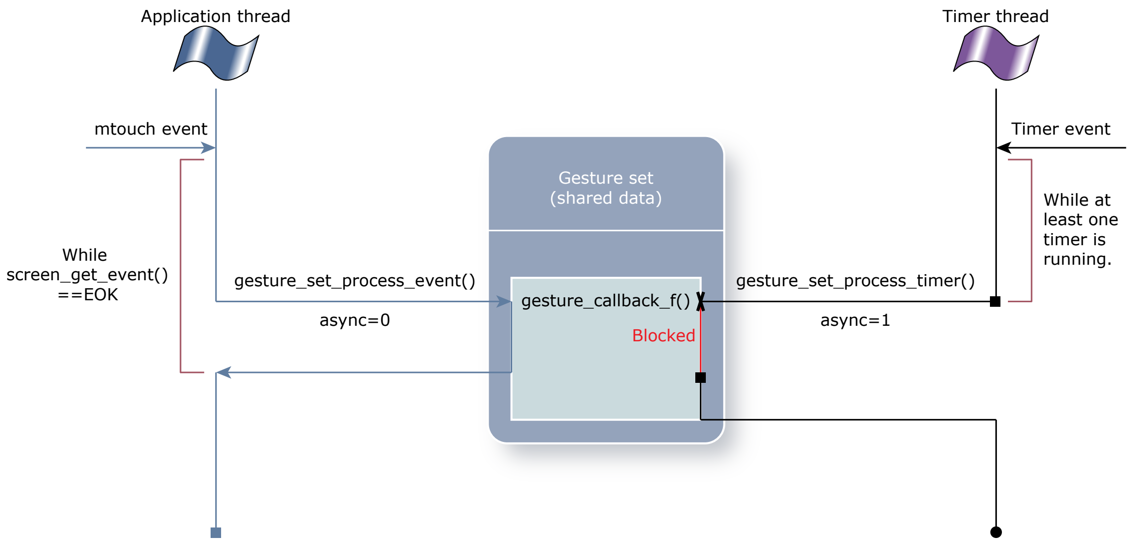

Application thread
Timer thread
mtouch event
Timer event
Gesture set
(shared data)
While at least one timer is running.
While screen_get_event() ==EOK
gesture_set_process_event()
gesture_callback_f()
gesture_set_process_timer()
async=0
Blocked
async=1

Figure 3: Example of mtouch event-invoked *gesture_callback_f()* blocking a timer event

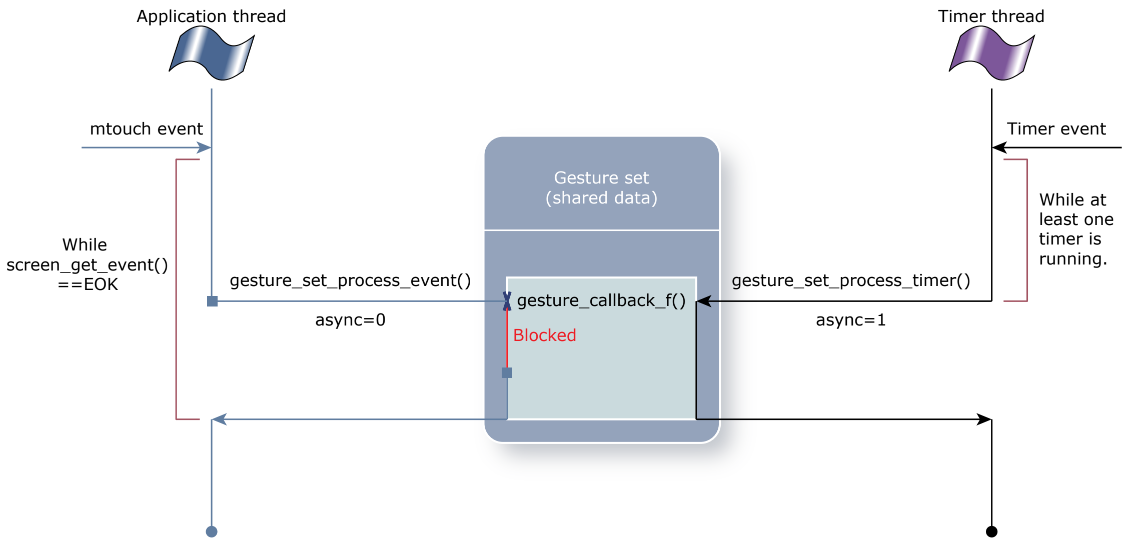

Application thread
Timer thread
mtouch event
Timer event
Gesture set
(shared data)
While at least one timer is running.
While screen_get_event() ==EOK
gesture_set_process_event()
gesture_callback_f()
gesture_set_process_timer()
async=0
Blocked
async=1

Figure 4: Example of timer event-invoked *gesture_callback_f()* blocking an mtouch event

The Gestures library is thread-safe and assures that any data shared between multiple threads will not be accessed simultaneously. However, you need to consider the possibility of threads blocking, or being blocked. For example, if your gesture callback function performs rendering operations, you are likely blocking important mtouch or timer events from being processed. As a result, the behavior of your application may become unpredictable.

To help with the synchronization between mtouch-invoked and timer-invoked gesture callback functions, gesture recognizers and gesture sets provide a parameter, *async*, in the *gesture_callback_f()* and *gestures_set_fail_f()* functions. This *async* parameter is set to `1` when the gesture callback is called from the timer thread.

Gesture recognizers that don't use timers aren't guaranteed to never have their gesture callback function invoked asynchronously. Your callback functions are synchronous to the thread where *gestures_set_process_event()* is called only when none of the gestures in your gesture set use timers. If you are using gesture recognizers that are provided by the Gestures Library, note that the double tap, triple tap and long press gesture recognizers use timers.

Lists

Timers behave differently, depending on what the application passes:

- event list (e.g., *gestures_set_process_event_list()*) or,
- single event (e.g., *gestures_set_process_event()*)

It doesn't make sense to use the wall clock for timer events when a gesture recognizer set is passed a list of events that occurred in the past. Gesture recognizers will receive events at a much higher rate than when the events come in at real time. For this reason, when processing event lists, the gesture set will cause timers to expire and invoke their callback functions using the event timestamps as the timebase.

If unexpired timers are left after the event-list processing finishes, they will be converted to realtime timers based on the differences between the current wall clock time and the timestamp of the last event in the list.

Timer handling for gestures recognizers with failure dependencies behave in the same way.

Failure dependencies

You can define a gesture recognizer to have failure dependencies on other gestures.

A failure dependency is when the detection of one gesture recognizer is dependent on the failure of another. That is, if one gesture recognizer moves to `GESTURE_STATE_FAILED` and it has failure dependents, then the gesture recognizers that are dependent on this failed gesture recognizer will be processed.

For example, you have an application that has one gesture set. This gesture set includes both tap and double-tap gesture recognizers. Your application need only one of the tap or double-tap gesture to be recognized at a time. Your application sets a failure dependency to indicate that in this gesture set, the tap gesture recognition depends on the failure of the double-tap gesture recognition. To set this failure dependency in your application, you must use the *gesture_add_mustfail()* function. In this particular example, you include the following code in the initialization of your gesture recognizers:

```
...
gesture_tap_t* tap;
gesture_double_tap_t double_tap;
...
gesture_add_mustfail(&tap->base, &double_tap->base);
```

The above code snippet indicates that when a double-tap gesture fails, the gesture recognizer will try to recognize the mtouch event as a tap gesture.

Failure notification and event lists

Applications can register gesture sets for failure notifications. These notifications are delivered by the gesture set by a failure callback function that is separate from the gesture recognizer callback function:

```
void(*gestures_set_fail_f)(struct gestures_set* set, struct event_list* list, int async);
```

This failure callback function is invoked only if all gesture recognizers in the gesture set have transitioned to `GESTURE_STATE_FAILED`. If at least one gesture recognizer is in the `GESTURE_STATE_COMPLETE` state, the failure notification callback function isn't invoked.

The *event_list* parameter contains the list of events that were delivered to the gesture set that subsequently caused all gestures to fail. This list of events is passed to the failure callback function of the gesture set. These events can either be processed individually or delivered to another gesture set for further processing.

Event lists are used to keep copies of events should failure dependencies need to be fulfilled or failure notifications need to be delivered. Event lists can contain up to 1024 events. If more events come in after the list is full, the oldest non-key (mtouch or release) events are dropped from the list.

Reset

Once a gesture recognizer in a gesture set transitions its state to either `GESTURE_STATE_COMPLETE` or `GESTURE_STATE_FAILED`, it will be reset only when all other gesture recognizers in the same gesture set have also transitioned to either of these states.

For example, if an application defines a gesture set with tap and double-tap gestures, the application would receive two callbacks:

- single tap after first release
- double tap after second release

Assuming that no failure dependencies have been configured, the application would not receive two callbacks for two single taps because the completed single tap will already be in the state, GESTURE_STATE_COMPLETE, after the first tap. When the double tap either completes or fails, the gesture set calls

```
void (*reset)(struct gesture_base* gesture);
```

on each of its gesture recognizers.

Chapter 3 Custom Gestures

You can define gesture recognizers to detect gestures that are not already supported by the Gestures library (custom gestures). Gesture recognizers that are used to detect custom gestures can be compiled with the application code and added to a gesture set in the same way that system gesture recognizers are added to a gesture set.

Custom gesture recognizer data types and functions

If you're defining your own custom gesture recognizers, you will need to provide the following as part of the definition and implementation of your own gesture recognizer:

Definition of your custom gesture

```
typedef struct {
   gesture_base_t base; /* The gesture base data structure. */
   ...                  /* The information specific to your custom gesture */
   int timer_id; /* The ID of the timer for this custom gesture (if needed) */
} gesture_custom_t;
```

This structure represents information for your custom gesture recognizer; this structure must include `gesture_base_t` followed by additional members to capture your specific information. For example, if your custom gesture recognizer uses a timer, you need to include the ID of the timer as a specific parameter.

Definition and implementation of the *alloc()* function

```
gesture_custom_t* custom_gesture_alloc(gesture_custom_params_t* params,
                                       gesture_callback_f callback,
                                       struct gestures_set* set);
```

where types and parameters are as follows:

gesture_custom_t

The structure of your custom gesture recognizer.

custom_gesture_alloc

The name of your gesture's *alloc()* function.

gesture_custom_params_t

A structure that represents the parameters of your custom gesture recognizer.

callback

The application gesture callback.

set

A gesture set that your custom gesture recognizer is to be added to.

Your *alloc()* function must:

1. Allocate the memory necessary for your custom gesture recognizer.
2. Invoke *gesture_base_init()* to initialize the gesture base data structure.
3. Invoke *gesture_set_add()* to add your custom gesture to the gesture set.
4. Set the gesture recognizer type as GESTURE_USER.
5. Set the *process_event()*, the *reset()*, and the *free()* functions.
6. Set the gesture callback: *callback*.
7. Perform any custom gesture-specific initialization that isn't part of the reset.
8. Store the custom gesture-specific parameters with the gesture recognizer, and set default parameters. If your custom gesture recognizer uses a timer, use *gesture_timer_create()* to obtain the ID for your timer.

Definition and implementation of the *process_event()* function

```
gesture_state_e (*process_event)(struct contact_id_map* map,
                                 struct gesture_base* gesture,
                                 mtouch_event_t* event, int* consumed)
```

Your *process_event()* function must handle state transitions and return the new, or unchanged, gesture recognizer state. If your custom gesture is time-based, you will need to adjust the timers accordingly. The Gestures library provides API functions for you to set and reset your timers.

Definition and implementation of the *free()* function

```
void (*free)(struct gesture_base* gesture)
```

Your *free()* function must release all the memory that's associated with your gesture recognizer that was allocated by your *alloc()* function.

Definition and implementation of the *reset()* function

```
void (*reset)(struct gesture_base* gesture);
```

Your *reset()* function must reset the gesture-specific data structures to their initial states.

Definition of parameters specific to your custom gesture (optional)

This is a structure that represents the parameters specific to your custom gesture recognizer.

Definition of states specific to your custom gesture (optional)

These are constants that represent the states specific to your custom gesture recognizer; these states are **in addition** to the set of states defined in `gesture_state_e`.

Definition and implementation of the *gesture_timer_callback()* function (optional)

```
gesture_state_e(*gesture_timer_callback_t)(struct gesture_base* gesture, void* param);
```

Gesture sets provide time-based notifications to gesture recognizers that use timers. A notification is implemented as a callback function to the gesture recognizer. If your custom gesture recognizer is timer-based, you need to implement this timer callback function.

A gesture recognizer can transition states on timer events. Similar to the *process_event()* function for mtouch events, the gesture recognizer's time-based state transitions are accomplished by its timer callback function. This function returns the new, or unchanged, state based on the timer event received.

Contact ID map

A contact ID is an identifier that is used to identify mtouch events. The mtouch event data structure contains the *contact_id* element that is assigned a value from a zero-based index and corresponds to the individual fingers that touch the screen. The contact ID doesn't change until that finger is released.

User-defined gestures typically need to associate a specific mtouch event with a contact ID for the purpose of associating the streams of events with the finger that caused them. The contact ID from the mtouch event can't be used directly by the custom gesture. Instead, user-defined gestures need to invoke *map_contact_id()* to obtain a zero-based contact ID that is remapped from the gesture set's perspective.

The remapping is necessary because a contact ID of 1 for an mtouch event could actually correspond to a gesture set's contact ID 0. This mapping could be the case if there are multiple gesture sets in play, or if the user's finger is resting on the touch-sensitive bevel.

Helper functions

Helper functions are available if you are defining your own gestures. These functions are:

void save_coords(mtouch_event_t *event,gesture_coords_t *coords)

This function saves the coordinates of a mtouch event in the specified `gesture_coords_t` data structure. This is useful if your gesture is sensitive to the placement of the touch event. For example, in a double-tap gesture, the coordinates of the first tap are saved and compared to the coordinates of the second tap. If these coordinates are within an acceptable range, the gesture recognizer can consider the gesture to be a double tap.

int32_t diff_time_ms(gesture_coords_t *coords1, gesture_coords_t *coords2)

This function returns the elapsed time between the two specified gesture events. This is useful if your gesture is dependent on the receipt of multiple mtouch events. For example, in a double-tap gesture, the time elapsed between the first and second tap cannot exceed an acceptable time. If too much time has elapsed between the two taps, the double-tap gesture is considered to have failed.

uint32_t max_displacement_abs(gesture_coords_t *coords1, gesture_coords_t *coords2)

This function returns the maximum displacement, in pixels, between two gesture events. For example, if the absolute value of the difference between the x coordinates of the two gestures is greater than the absolute value of the difference between the y coordinates, the former is returned by the function. For example, in a double-tap gesture, this function can be used to help determine whether or not the two taps received are close enough together on the screen to be considered a double-tap gesture.

int map_contact_id(struct contact_id_map *map, unsigned contact_id)

This function remaps contact identifiers from mtouch events to contact identifiers to be used by gestures. This function is typically one of the first calls in your custom gesture recognizer's *process_event()* function. You need to first map the *contact_id* from the mtouch event received to a *contact_id* that can be used by your custom gesture recognizer.

int gesture_timer_create(struct gesture_base* gesture, gesture_timer_callback_t callback, void* param)

This function creates a new timer that invokes your gesture recognizer's timer callback function when expired. You need to use this function if your gesture recognizer is time-based.

int gesture_timer_set_now(struct gesture_base* gesture, int timer_id, unsigned ms)

This function sets a timer using the current time as the reference time. You can use this if your gesture recognizer is time-based.

int gesture_timer_set_ms(struct gesture_base* gesture, int timer_id, unsigned ms, _Uint64t base_nsec)

This function sets a timer using a specified timestamp as the reference time. You can use this if your gesture recognizer is time-based.

int gesture_timer_set_event(struct gesture_base* gesture, int timer_id, unsigned ms, struct mtouch_event* base_event)

This function sets a timer using an mtouch event timestamp as the reference time. You can use this if your gesture recognizer is time-based.

void gesture_timer_destroy(struct gesture_base* gesture, int timer_id)

This function resets the specified timer. You can use this if your gesture recognizer is time-based.

int gesture_timer_query(struct gesture_base* gesture, int timer_id, int* pending, _Uint64t* expiry)

This function queries the information for the specified timer. You can use this if your gesture recognizer is time-based.

Chapter 4
Gesture Tutorials

Gestures tutorials aim to help you understand how to use the Gestures library in your own applications by providing step-by-step guides.

Tutorial: Create a gesture-handling application

This tutorial shows you the basics for creating a gesture application using system-supported gesture recognizers.

You will learn to:

- create a gesture callback function
- create a gesture set failure callback function
- initialize gesture sets
- detect gestures
- clean up gestures

Create your gesture callback function

The gesture callback function defines what the application does when a gesture is recognized or updated:

```
void(*gesture_callback_f)(struct gesture_base* gesture,
                          mtouch_event_t* event,
                          void* param, int async);
```

The argument *gesture* contains information about the gesture and the parameter *event* contains information about the mtouch event that caused the gesture. The parameter *async* identifies whether this callback was invoked from an event (*async* = 0) or from a timer (*async* = 1).

If you have gestures that are transitioning based on timer events, this callback function could be invoked as a result of either a timer event (from the context of the timer thread) or an mtouch event. Your application code needs to implement the synchronization mechanism between mtouch callback functions and timer-event callback functions. Your application needs to check the *async* parameter and implement synchronization accordingly.

This function's main component is a `switch` statement that defines the application's actions based on the gesture received. Typically, your gesture application copies information from the incoming gesture to a local structure and uses that information accordingly. The type of local structure depends on the incoming gesture. Usually, you're interested only if a certain gesture has been detected (i.e., the state of the gesture recognizer is `GESTURE_STATE_COMPLETE`). However, your callback function may look for other states and behave accordingly for your application. For an example of such `switch` statement, see the following code:

```
switch (gesture->type) {
    case GESTURE_TWO_FINGER_PAN: {
        gesture_tfpan_t* tfpan = (gesture_tfpan_t*)gesture;
        if (tfpan->base.state == GESTURE_STATE_COMPLETE)
        {
           printf("Two-finger pan gesture detected: %d, %d",
                                                tfpan->centroid.x, tfpan->centroid.y);
        }
        break;
    }
    case GESTURE_ROTATE: {
        gesture_rotate_t* rotate = (gesture_rotate_t*)gesture;
        if (rotate->base.state == GESTURE_STATE_COMPLETE)  {
           if (rotate->angle != rotate->last_angle) {
               printf("Rotate: %d degs", rotate->angle - rotate->last_angle);
           }
        }
        break;
    }
    case GESTURE_SWIPE: {
        gesture_swipe_t* swipe = (gesture_swipe_t*)gesture;
        if (swipe->base.state == GESTURE_STATE_COMPLETE)  {
           if (swipe->direction & GESTURE_DIRECTION_UP) {
               printf("up %d", swipe->last_coords.y - swipe->coords.y);
           } else if (swipe->direction & GESTURE_DIRECTION_DOWN) {
              printf("down %d", swipe->coords.y - swipe->last_coords.y);
           } else if (swipe->direction & GESTURE_DIRECTION_LEFT) {
               printf("left %d", swipe->last_coords.x - swipe->coords.x);
           } else if (swipe->direction & GESTURE_DIRECTION_RIGHT) {
               printf("right %d", swipe->coords.x - swipe->last_coords.x);
           }
        }
        break;
    }
    case GESTURE_PINCH: {
        gesture_pinch_t* pinch = (gesture_pinch_t*)gesture;
        if (pinch->base.state == GESTURE_STATE_COMPLETE)  {
           printf("Pinch %d, %d", (pinch->last_distance.x - pinch->distance.x),
                                  (pinch->last_distance.y - pinch->distance.y));
        }
        break;
    }
    case GESTURE_TAP: {
        gesture_tap_t* tap = (gesture_tap_t*)gesture;
        if (tap->base.state == GESTURE_STATE_COMPLETE)  {
           printf("Tap x:%d y:%d",tap->touch_coords.x, tap->touch_coords.y);
        }
        break;
    }
    case GESTURE_DOUBLE_TAP: {
        gesture_double_tap_t* d_tap = (gesture_double_tap_t*)gesture;
        if (d_tap->base.state == GESTURE_STATE_COMPLETE)  {
           printf("Double tap first_touch x:%d y:%d", d_tap->first_touch.x,
                                                      d_tap->first_touch.y);
           printf("Double tap first_release x:%d y:%d", d_tap->first_release.x,
                                                        d_tap->first_release.y);
           printf("Double tap second_touch x:%d y:%d", d_tap->second_touch.x,
                                                       d_tap->second_touch.y);
           printf("Double tap second_release x:%d y:%d", d_tap->second_touch.x,
                                                         d_tap->second_release.y);
        }
        break;
    }
    case GESTURE_TRIPLE_TAP: {
        gesture_triple_tap_t* t_tap = (gesture_triple_tap_t*)gesture;
        if (t_tap->base.state == GESTURE_STATE_COMPLETE)  {
           printf("Triple tap first_touch x:%d y:%d", t_tap->first_touch.x,
                                                      t_tap->first_touch.y);
           printf("Triple tap first_release x:%d y:%d", t_tap->first_release.x,
                                                        t_tap->first_release.y);
           printf("Triple tap second_touch x:%d y:%d", t_tap->second_touch.x,
                                                       t_tap->second_touch.y);
           printf("Triple tap second_release x:%d y:%d", t_tap->second_touch.x,
                                                         t_tap->second_release.y);
           printf("Triple tap third_touch x:%d y:%d", t_tap->third_touch.x,
                                                      t_tap->second_touch.y);
           printf("Triple tap third_release x:%d y:%d", t_tap->third_touch.x,
                                                        t_tap->second_release.y);
        }
        break;
    }
    case GESTURE_PRESS_AND_TAP: {
        gesture_pt_t* pt_t = (gesture_pt_t*)gesture;
        if (pt_t->base.state == GESTURE_STATE_COMPLETE)  {
           printf("Initial press x:%d y:%d", pt_t->initial_coords[0].x,
                                             pt_t->initial_coords[0].y);
           printf("Initial tap x:%d y:%d", pt_t->initial_coords[1].x,
                                           pt_t->initial_coords[1].y);
           printf("Press x:%d y:%d", pt_t->coords[0].x, pt_t->coords[0].y);
           printf("Tap x:%d y:%d", pt_t->coords[1].x, pt_t->coords[1].y);
        }
        break;
    }
    case GESTURE_TWO_FINGER_TAP: {
        gesture_tft_t* tft_t = (gesture_tft_t*)gesture;
        if (tft_t->base.state == GESTURE_STATE_COMPLETE)  {
           printf("Coordinates of touch event (finger 1) x:%d y:%d",
                                                         tft_t->touch_coords[0].x,
                                                         tft_t->touch_coords[0].y);
```

```
            printf("Coordinates of touch event (finger 2) x:%d y:%d",
                                                  tft_t->touch_coords[1].x,
                                                  tft_t->touch_coords[1].y);
            printf("Coordinates of release event (finger 1) x:%d y:%d",
                                                  tft_t->release_coords[0].x,
                                                  tft_t->release_coords[0].y);
            printf("Coordinates of release event (finger 2) x:%d y:%d",
                                                  tft_t->release_coords[1].x,
                                                  tft_t->release_coords[1].y);
            printf("Midpoint between two touches x:%d y:%d", tft_t->centroid.x,
                                                     tft_t->centroid.y);
        }
        break;
    }
    case GESTURE_LONG_PRESS: {
        gesture_long_press_t* lp_t = (gesture_long_press_t*)gesture;
        if (lp_t->base.state == GESTURE_STATE_COMPLETE)  {
            printf("Long press x:%d y:%d",lp_t->coords.x, lp_t->coords.y);
            printf("Timer ID:%d",lp_t->success_timer);
        }
        break;
    }
    case GESTURE_USER: {
        printf("User-defined gesture detected.");
        break;
    }
    default:
        printf("Unknown");
        break;
    }
```

Create your failure callback function (optional)

Sometimes, you may want your application to create a failure callback function to be invoked when all gestures in a gesture set have transitioned to the GESTURE_STATE_FAILED state:

```
void(*gestures_set_fail_f)(struct gestures_set* set, struct event_list* list, int async);
```

This sample failure callback function shows how to copy an event list that's received as part of a gesture set failure callback.

```
void fail_callback(struct gestures_set* set, struct event_list* list, int async)
{
    /* first_set should be defined in your application as
     * struct gestures_set* first_set;
     * and then be allocated and initialized in your application.
     */
    if (set == first_set) {
        /* An example of list copy.
         * This isn't necessary if events don't need to be kept following the
         * call to gestures_set_process_event_list()
         */
        struct event_list* new_list = event_list_alloc(0, 0, 0, 1);
        if (new_list) {
            event_list_copy(list, new_list);
            gestures_set_process_event_list(second_set, new_list, NULL);
            event_list_free(new_list);
        }
    }
}
```

Initialize your gesture sets

Your application needs to register the callback function with the Gestures library so that it can be invoked when a gesture occurs.

To register the gesture callback function, your application needs to first allocate the gesture set. If you have defined a failure callback for your gesture set, then you must call:

```
gestures_set_register_fail_cb(struct gestures_set* set, gestures_set_fail_f callback);
```

to register your failure callback function with your gesture set.

In this example, two gesture sets are initialized and a failure callback is registered with the first gesture set:

```
struct gestures_set* first_set;
struct gestures_set* second_set;

static void init_gestures()
{
    gesture_tap_t* tap;
    gesture_double_tap_t* double_tap;
    gesture_triple_tap_t* triple_tap;
    gesture_tft_t* tft;

    first_set = gestures_set_alloc();
        long_press_gesture_alloc(NULL, gesture_callback, first_set);
        tap = tap_gesture_alloc(NULL, gesture_callback, first_set);
        double_tap = double_tap_gesture_alloc(NULL, gesture_callback, first_set);
        triple_tap = triple_tap_gesture_alloc(NULL, gesture_callback, first_set);
        tft = tft_gesture_alloc(NULL, gesture_callback, first_set);
        gesture_add_mustfail(&tap->base, &double_tap->base);
        gesture_add_mustfail(&double_tap->base, &triple_tap->base);
        gestures_set_register_fail_cb(first_set, fail_callback);

    second_set = gestures_set_alloc();
        swipe_gesture_alloc(NULL, gesture_callback, second_set);
        pinch_gesture_alloc(NULL, gesture_callback, second_set);
        rotate_gesture_alloc(NULL, gesture_callback, second_set);
        pt_gesture_alloc(NULL, gesture_callback, second_set);
        tfpan_gesture_alloc(NULL, gesture_callback, second_set);
        tft_gesture_alloc(NULL, gesture_callback, second_set);
}
```

If you're using custom gestures that you have defined yourself, then you need to allocate your custom gesture recognizer and add it to the gesture set as part of the gesture-set initialization using the *alloc()* function you've defined. For example, in the code snippet shown, you can add your custom gesture recognizer to your second gesture set by calling your *custom_gesture_alloc()* function in *init_gestures()*:

```
static void init_gestures()
{
   ...
   second_set = gestures_set_alloc();
   ...
      gesture_custom_t* user_gesture = custom_gesture_alloc(custom_params,
                                                            gesture_callback,
                                                            set);
}
```

Detect the gestures

Now, your application needs a way of triggering the gesture callback function when a touch event occurs. When such a touch event is detected, your application calls *gestures_set_process_event()*.

Touch events can be detected through Screen events in a main application loop. If the incoming event is a touch, move, or release event, you need to populate an mtouch event with data from the Screen event. The helper function, *screen_get_mtouch_event()*

can do this for you; it's part of the Input Events library. See file input/screen_helpers.h for more information. Then, your application calls *gestures_set_process_event()*.

```
while (1) {
    while (screen_get_event(screen_ctx, screen_ev, ~0L) == EOK) {
        rc = screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_TYPE, &screen_val);
        if (rc || screen_val == SCREEN_EVENT_NONE) {
            break;
        }
        switch (screen_val) {
            case SCREEN_EVENT_MTOUCH_TOUCH:
            case SCREEN_EVENT_MTOUCH_MOVE:
            case SCREEN_EVENT_MTOUCH_RELEASE:
                rc = screen_get_mtouch_event(screen_ev, &mtouch_event, 0);
                if (rc) {
                    fprintf(stderr, "Error: failed to get mtouch event\n");
                    continue;
                }
                gestures_set_process_event(first_set, &mtouch_event, NULL);
                break;
        }
    }
}
```

Clean up gestures

Before exiting your application, you need to free the memory associated with your gesture sets:

```
static void gestures_cleanup()
{
    if (NULL != first_set) {
        gestures_set_free(first_set);
        first_set = NULL;
    }
    if (NULL != second_set) {
        gestures_set_free(second_set);
        second_set = NULL;
    }
}
```

Example: Code snippets of a gesture-handling application

This example contains most of the code snippets that illustrate how to create a gesture application and cascade multiple gesture sets using failure callbacks.

```
#include <stdio.h>
#include <screen/screen.h>
#include "input/screen_helpers.h"
#include "gestures/types.h"
#include "gestures/set.h"
#include "gestures/event_list.h"
#include "gestures/swipe.h"
#include "gestures/pinch.h"
#include "gestures/press_and_tap.h"
#include "gestures/rotate.h"
#include "gestures/two_finger_pan.h"
#include "gestures/tap.h"
#include "gestures/double_tap.h"
#include "gestures/triple_tap.h"
#include "gestures/long_press.h"
#include "gestures/two_finger_tap.h"

/* The callback invoked when a gesture is recognized or updated. */
void gesture_callback(gesture_base_t* gesture, mtouch_event_t* event, void* param, int async)
{
    if (async) {
        printf("[async] ");
    }
    switch (gesture->type) {
        case GESTURE_TWO_FINGER_PAN: {
            gesture_tfpan_t* tfpan = (gesture_tfpan_t*)gesture;
            printf("Two-finger pan: %d, %d", tfpan->centroid.x, tfpan->centroid.y);
            break;
        }
        case GESTURE_ROTATE: {
            gesture_rotate_t* rotate = (gesture_rotate_t*)gesture;
            if (rotate->angle != rotate->last_angle) {
                printf("Rotate: %d degs", rotate->angle - rotate->last_angle);
            } else {
                return;
            }
            break;
        }
        case GESTURE_SWIPE: {
            gesture_swipe_t* swipe = (gesture_swipe_t*)gesture;
            printf("Swipe ");
            if (swipe->direction & GESTURE_DIRECTION_UP) {
                printf("up %d", swipe->last_coords.y - swipe->coords.y);
            } else if (swipe->direction & GESTURE_DIRECTION_DOWN) {
                printf("down %d", swipe->coords.y - swipe->last_coords.y);
            } else if (swipe->direction & GESTURE_DIRECTION_LEFT) {
                printf("left %d", swipe->last_coords.x - swipe->coords.x);
            } else if (swipe->direction & GESTURE_DIRECTION_RIGHT) {
                printf("right %d", swipe->coords.x - swipe->last_coords.x);
            }
            break;
        }
        case GESTURE_PINCH: {
            gesture_pinch_t* pinch = (gesture_pinch_t*)gesture;
            printf("Pinch %d, %d", (pinch->last_distance.x - pinch->distance.x),
                                   (pinch->last_distance.y - pinch->distance.y));
            break;
        }
        case GESTURE_TAP:
            printf("Tap");
            break;
        case GESTURE_DOUBLE_TAP:
            printf("Double tap");
            break;
        case GESTURE_TRIPLE_TAP:
            printf("Triple tap");
            break;
        case GESTURE_PRESS_AND_TAP:
            printf("Press and tap");
            break;
        case GESTURE_TWO_FINGER_TAP:
            printf("Two-finger tap");
            break;
        case GESTURE_LONG_PRESS:
            printf("Long press");
            break;
        case GESTURE_USER:
            printf("User");
            break;
        default:
            printf("Unknown");
            break;
    }
    printf("\n");
}

/**
 * The set failure callback that's invoked when all gestures in the first set have
 * transitioned to the failed state.
 */

void fail_callback(struct gestures_set* set, struct event_list* list, int async)
{
    if (set == first_set) {
        /* Sample list copy - not necessary if events don't need to be kept
           following the call to gestures_set_process_event_list() */
        struct event_list* new_list = event_list_alloc(0, 0, 0, 1);
        if (new_list) {
            event_list_copy(list, new_list);
            gestures_set_process_event_list(second_set, new_list, NULL);
            event_list_free(new_list);
        }
    }
}

/** Initialize the gesture sets */
static void init_gestures()
{
    gesture_tap_t* tap;
    gesture_double_tap_t* double_tap;
    gesture_triple_tap_t* triple_tap;
    gesture_tft_t* tft;

    first_set = gestures_set_alloc();
    if (NULL != first_set) {
        long_press_gesture_alloc(NULL, gesture_callback, first_set);
        tap = tap_gesture_alloc(NULL, gesture_callback, first_set);
        double_tap = double_tap_gesture_alloc(NULL, gesture_callback, first_set);
        triple_tap = triple_tap_gesture_alloc(NULL, gesture_callback, first_set);
        tft = tft_gesture_alloc(NULL, gesture_callback, first_set);
        gesture_add_mustfail(&tap->base, &double_tap->base);
        gesture_add_mustfail(&double_tap->base, &triple_tap->base);
        gestures_set_register_fail_cb(first_set, fail_callback);

        second_set = gestures_set_alloc();

        if (NULL != second_set) {
            swipe_gesture_alloc(NULL, gesture_callback, second_set);
            pinch_gesture_alloc(NULL, gesture_callback, second_set);
            rotate_gesture_alloc(NULL, gesture_callback, second_set);
            pt_gesture_alloc(NULL, gesture_callback, second_set);
            tfpan_gesture_alloc(NULL, gesture_callback, second_set);
            tft_gesture_alloc(NULL, gesture_callback, second_set);
        } else {
            gestures_set_free(first_set);
            first_set = NULL;
        }
    } else {
        fprintf(stderr, "Failed to allocate gesture sets\n");
    }
}

static void gestures_cleanup()
{
    if (NULL != first_set) {
        gestures_set_free(first_set);
        first_set = NULL;
    }
    if (NULL != second_set) {
        gestures_set_free(second_set);
        second_set = NULL;
    }
}

int main(int argc, const char* argv[])
{
    int screen_val;
    screen_event_t screen_ev;
    int rc;
    ...
    mtouch_event_t mtouch_event;
    init_gestures();
    while (1) {
        while (screen_get_event(screen_ctx, screen_ev, ~0L) == EOK) {
            rc = screen_get_event_property_iv(screen_ev, SCREEN_PROPERTY_TYPE, &screen_val);
            if (rc || screen_val == SCREEN_EVENT_NONE) {
                break;
            }
            switch (screen_val) {
                case SCREEN_EVENT_MTOUCH_TOUCH:
                case SCREEN_EVENT_MTOUCH_MOVE:
                case SCREEN_EVENT_MTOUCH_RELEASE:
                    rc = screen_get_mtouch_event(screen_ev, &mtouch_event, 0);
                    if (rc) {
                        fprintf(stderr, "Error: failed to get mtouch event\n");
                        continue;
                    }
                    gestures_set_process_event(first_set, &mtouch_event, NULL);
                    break;
            }
        }
    }
    gestures_cleanup();
    return 0;
}
```

Tutorial: Create a custom gesture

This tutorial shows you the basics for defining a custom gesture recognizer.

You will learn to:

- define a custom gesture recognizer
- create a timer callback function
- create an *alloc()* function
- create a *process_event()* function
- create a *free()* function
- create a *reset()* function

Define your custom gesture recognizer

The definition of your custom gesture recognizer includes the following:

Custom gesture recognizer parameters (optional)

A structure that represents the parameters specific to your custom gesture recognizer. This structure is optional. If your custom gesture recognizer has no specific parameters, then you don't need to define this structure. Here's an example of what parameters might be in this custom gesture-recognizer structure:

```
typedef struct {
 unsigned max_displacement; /** The maximum distance your finger can move before
                                 this custom gesture fails. */
 unsigned max_hold_ms;      /** The maximum time your finger can remain touching
                                 the screen before this custom gesture fails. */
 unsigned max_delay_ms;     /** The time between the first release and the second
                                touch. */

} custom_gesture_params_t;
```

Custom gesture-recognizer states (optional)

Constants that represent the states specific to your custom gesture recognizer; these states are **in addition** to the set of states defined in `gesture_state_e`. Additional states for your gesture recognizer are optional. If your gesture recognizer doesn't need additional states, these constants aren't necessary. Here's an example of what the definition of additional states might look like:

```
typedef enum {
 CUSTOM_STATE_INIT = 0,
 CUSTOM_STATE_FIRST_TOUCH,
 CUSTOM_STATE_FIRST_RELEASE,
 CUSTOM_STATE_SECOND_TOUCH,
 CUSTOM_STATE_SECOND_RELEASE
} custom_gesture_state_e;
```

Custom gesture-recognizer information

A structure that contains information about your custom gesture recognizer. This structure must list *gesture_base_t* as its first data member. Here's an example of what information might be included in your gesture recognizer:

```
typedef struct {
 gesture_base_t base;              /* The gesture base data structure. */
 custom_gesture_params_t params;  /** Your custom gesture recognizer parameters. */
 gesture_coords_t first_touch;    /** The coordinates of the first touch. */
 gesture_coords_t first_release;  /** The coordinates of the first release. */
 gesture_coords_t second_touch;   /** The coordinates of the second touch. */
 gesture_coords_t second_release; /** The coordinates of the second release. */
 custom_gesture_state_e ct_state; /** The intermediate state of your recognizer. */
 int timer;                        /** The ID of the timer for this gesture. */
} gesture_custom_t;
```

Create the timer callback function (optional)

If your custom gesture recognizer is timer-based, you need to implement a callback function that will be called upon receipt of a timer event. Your *gesture_timer_callback_t* function returns the new, or unchanged, state based on the timer event received and might look like this:

```
static gesture_state_e custom_gesture_timer_callback(gesture_base_t* base, void* param)
{
 return GESTURE_STATE_FAILED;
}
```

Create the *alloc()* function

Your *alloc()* function must:

1. Allocate the memory necessary for your custom gesture recognizer.
2. Invoke *gesture_base_init()* to initialize the gesture base data structure.
3. Invoke *gesture_set_add()* to add your custom gesture to the gesture set.
4. Set the gesture recognizer type as GESTURE_USER.
5. Set the *process_event()*, the *reset()*, and the *free()* functions.
6. Set the gesture callback: *callback*.
7. Perform any custom gesture-specific initialization that isn't part of the reset.
8. Store the custom gesture-specific parameters with the gesture recognizer, and set default parameters. If your custom gesture recognizer uses a timer, use *gesture_timer_create()* to obtain the ID for your timer.

If your custom gesture recognizer uses timers, then use *gesture_timer_create()* to create the timer and register your timer callback with your gesture recognizer.

Here's an example of what your custom gesture recognizer *alloc()* might look like:

```
gesture_custom_t*
custom_gesture_alloc(custom_gesture_params_t* params,
                     gesture_callback_f callback,
                     struct gestures_set* set)
{
 gesture_custom_t* user_gesture = calloc(1, sizeof(*user_gesture));
 if (NULL == user_gesture) {
  return NULL;
 }

 gesture_base_init(&user_gesture->base);
 gestures_set_add(set, &user_gesture->base);
 user_gesture->base.type = GESTURE_USER;
 user_gesture->base.funcs.free = user_gesture_gesture_free;
 user_gesture->base.funcs.process_event = user_gesture_gesture_process_event;
 user_gesture->base.funcs.reset = user_gesture_gesture_reset;
 user_gesture->base.callback = callback;
 user_gesture->timer = gesture_timer_create(&user_gesture->base,
                                            custom_gesture_timer_callback, NULL);
 user_gesture_gesture_reset(&user_gesture->base);

 if (NULL != params) {
  user_gesture->params = *params;
 } else {
  user_gesture_gesture_default_params(&user_gesture->params);
 }

 return user_gesture;
}
```

Create the *process_event()* function

You need to implement a *process_event()* function that's responsible for state-handling and returning the new (or unchanged) gesture set:

```
gesture_state_e (*process_event)(struct contact_id_map* map,
                                 struct gesture_base* gesture,
                                 mtouch_event_t* event,
                                 int* consumed);
```

Ensure that your state transitions are valid according to the Gestures library. Refer to State transitions.

A *process_event()* function for a custom gesture may look like this:

```
gesture_state_e
custom_gesture_gesture_process_event(struct contact_id_map* map,
                                     gesture_base_t* gesture,
                                     mtouch_event_t* event,
                                     int* consumed)
{
 gesture_custom_t* custom_gesture = (gesture_custom_gesture_t*)gesture;
 gesture_coords_t coords;
 gesture_coords_t* compare_coords;
 int set_id = map_contact_id(map, event->contact_id);

 if (set_id < 0) {
  error("process_event() called with event with invalid contact_id");
  goto failed;
 }

 switch (user_gesture->base.state) {
  case GESTURE_STATE_UNRECOGNIZED:
   switch (event->event_type) {
    case INPUT_EVENT_MTOUCH_TOUCH:
     if (set_id > 0) {
      goto failed;
     } else if (user_gesture->ct_state > CUSTOM_STATE_FIRST_TOUCH) {
      gesture_timer_clear(gesture, user_gesture->fail_timer);
      save_coords(event, &user_gesture->second_touch);

      if (!((max_displacement_abs(&user_gesture->first_release,
                                  &user_gesture->second_touch)
              <= user_gesture->params.max_displacement) &&
                               (diff_time_ms(&user_gesture->first_release,
                           &user_gesture->second_touch)
                               <= user_gesture->params.max_delay_ms))) {
       goto failed;
      }
      user_gesture->ct_state = CUSTOM_STATE_SECOND_TOUCH;
     } else {
      save_coords(event, &user_gesture->first_touch);
      user_gesture->ct_state = CUSTOM_STATE_FIRST_TOUCH;
     }

     goto nochange;
    case INPUT_EVENT_MTOUCH_MOVE:
     save_coords(event, &coords);
     if (user_gesture->ct_state >= CUSTOM_STATE_SECOND_TOUCH) {
      compare_coords = &user_gesture->second_touch;
     } else {
      compare_coords = &user_gesture->first_touch;
     }

        if (!((max_displacement_abs(compare_coords,
                                 &coords)
             <= user_gesture->params.max_displacement) &&
                            (diff_time_ms(compare_coords,
                               &coords)
                            <= user_gesture->params.max_hold_ms)))
                    {
      goto failed;
     }

     goto nochange;
    case INPUT_EVENT_MTOUCH_RELEASE:
     if (user_gesture->ct_state >= CUSTOM_STATE_SECOND_TOUCH) {
      save_coords(event, &user_gesture->second_release);

      if (!((max_displacement_abs(&user_gesture->second_touch,
                                     &user_gesture->second_release)
             <= user_gesture->params.max_displacement) &&
                            (diff_time_ms(&user_gesture->second_touch,
                               &user_gesture->second_release)
                            <= user_gesture->params.max_hold_ms)))
                        {
        goto failed;
         }

      user_gesture->ct_state = CUSTOM_STATE_SECOND_RELEASE;
      goto complete;
     } else {
      save_coords(event, &user_gesture->first_release);

      if (!((max_displacement_abs(&user_gesture->first_touch,
                                     &user_gesture->first_release)
             <= user_gesture->params.max_displacement) &&
                            (diff_time_ms(&user_gesture->first_touch,
                               &user_gesture->first_release)
                            <= user_gesture->params.max_hold_ms)))
                        {
       goto failed;
      }
      user_gesture->ct_state = CUSTOM_STATE_FIRST_RELEASE;

      /* Set a timer in case an event doesn't come in */
      gesture_timer_set_event(gesture, user_gesture->timer,
                              user_gesture->params.max_delay_ms, event);
     }

     goto nochange;
    default:
     warn("Unhandled switch/case: %d", event->event_type);
     goto failed;
   }
  case GESTURE_STATE_RECOGNIZED:
   error("GESTURE_STATE_RECOGNIZED is an invalid state for double tap");
   break;
  case GESTURE_STATE_UPDATING:
   error("GESTURE_STATE_UPDATING is an invalid state for double tap");
   break;
  case GESTURE_STATE_COMPLETE:
   error("process_event() called on complete gesture");
   break;
  case GESTURE_STATE_FAILED:
   error("process_event() called on failed gesture");
   break;
  case GESTURE_STATE_NONE:
   error("process_event() called on uninitialized gesture");
   break;
 }

nochange:
 *consumed = 0;
 return user_gesture->base.state;

failed:
 *consumed = 0;
 return GESTURE_STATE_FAILED;

complete:
 *consumed = 1;
 return GESTURE_STATE_COMPLETE;
}
```

Create the *free()* function

Release the memory that's associated with the custom gesture:

```
void custom_gesture_free(struct gesture_base* gesture);
```

A *free()* function for a custom gesture may simply look like this:

```
void custom_gesture_free(gesture_base_t* gesture)
{
    free(gesture);
}
```

Create the *reset()* function

Reset any specific data structures that are associated with the custom gesture:

```
void custom_gesture_reset(struct gesture_base* gesture);
```

A *reset()* function for a custom gesture may be empty if there are no specific data structures associated with your custom gesture.

```
void
custom_gesture_reset(gesture_base_t* gesture)
{
 gesture_custom_t* user_gesture = (gesture_custom_t*)gesture;
 user_gesture->ct_state = CUSTOM_STATE_INIT;
}
```

Example: Code snippets of a defining a custom gesture

This example contains most of the code snippets that illustrate what you need in order to define a custom gesture.

custom_gesture.h

```
#include "gestures/types.h"

/* The stucture custom_gesture_params_t represents the parameters for the custom gesture. */
typedef struct {
   unsigned max_displacement; /* The maximum distance the finger can move before
                                 the custom gesture fails. */
   unsigned max_hold_ms;      /* The maximum time the finger can remain touching
                                 the screen before the custom gesture fails. */
   unsigned max_delay_ms;     /* The time between the first release and the second touch. */
} custom_gesture_params_t;

/* The enumeration custom_gesture_state_e defines additional states the custom
 * gesture can transition between. */
typedef enum {
   CT_STATE_INIT = 0,
   CT_STATE_FIRST_TOUCH,
   CT_STATE_FIRST_RELEASE,
   CT_STATE_SECOND_TOUCH,
   CT_STATE_SECOND_RELEASE
} custom_gesture_state_e;

/* The structure gesture_custom_gesture_t carries data about the custom gesture. */
typedef struct {
   gesture_base_t base;             /* The gesture base data structure. */
   custom_gesture_params_t params;  /* The custom gesture parameters. */
   gesture_coords_t first_touch;    /* The coordinates of the first touch. */
   gesture_coords_t first_release;  /* The coordinates of the first release. */
   gesture_coords_t second_touch;   /* The coordinates of the second touch. */
   gesture_coords_t second_release; /* The coordinates of the second release. */
   custom_gesture_state_e dt_state; /* The intermediate state of the custom gesture. */
   int fail_timer;                  /* The ID of the timer for this gesture. */
} gesture_custom_gesture_t;

/* Allocate and initialize the custom gesture structure */
gesture_custom_gesture_t* custom_gesture_gesture_alloc(custom_gesture_params_t* params,
                                                       gesture_callback_f callback,
                                                       struct gestures_set* set);

/* Initialize the custom parameters */
void custom_gesture_gesture_default_params(custom_gesture_params_t* params);
```

custom_gesture.c

```
/*The example below shows the implementation of a custom user-defined gesture. */

#include <sys/types.h>
#include <stdlib.h>
#include <malloc.h>
#include <string.h>
#include <stdio.h>
#include "gestures/set.h"
#include "gestures/timer.h"
#include "custom_gesture.h"
#include "input/event_types.h"
#include "gestures/defaults.h"

/* Custom free() function */
void custom_gesture_gesture_free(gesture_base_t* gesture)
{
   free(gesture);
}

/* Custom helper function */
static int _check_valid(gesture_coords_t* coords1, gesture_coords_t* coords2,
                        unsigned max_displacement, unsigned max_ms)
{
   return ((max_displacement_abs(coords1, coords2) <= max_displacement) &&
           (diff_time_ms(coords1, coords2) <= max_ms));
}

/* Custom process_event() function */
gesture_state_e custom_gesture_gesture_process_event(struct contact_id_map* map,
                                                     gesture_base_t* gesture,
                                                     mtouch_event_t* event,
                                                     int* consumed)
{
   gesture_custom_gesture_t* custom_gesture = (gesture_custom_gesture_t*)gesture;
   gesture_coords_t coords;
   gesture_coords_t* compare_coords;
   int set_id = map_contact_id(map, event->contact_id);

   if (set_id < 0) {
      error("process_event() called with event with invalid contact_id");
      goto failed;
   }

   switch (custom_gesture->base.state) {
      case GESTURE_STATE_UNRECOGNIZED:
         switch (event->event_type) {
            case INPUT_EVENT_MTOUCH_TOUCH:
               if (set_id > 0) {
                  goto failed;
               } else if (custom_gesture->dt_state > CT_STATE_FIRST_TOUCH) {
                  gesture_timer_clear(gesture, custom_gesture->fail_timer);
                  save_coords(event, &custom_gesture->second_touch);
                  if (!_check_valid(&custom_gesture->first_release,
                                    &custom_gesture->second_touch,
                                    custom_gesture->params.max_displacement,
                                    custom_gesture->params.max_delay_ms)) {
                     goto failed;
                  }
                  custom_gesture->dt_state = CT_STATE_SECOND_TOUCH;
               } else {
                  save_coords(event, &custom_gesture->first_touch);
                  custom_gesture->dt_state = CT_STATE_FIRST_TOUCH;
               }

               goto nochange;
            case INPUT_EVENT_MTOUCH_MOVE:
               save_coords(event, &coords);
               if (custom_gesture->dt_state >= CT_STATE_SECOND_TOUCH) {
                  compare_coords = &custom_gesture->second_touch;
               } else {
                  compare_coords = &custom_gesture->first_touch;
               }

               if (!_check_valid(compare_coords, &coords,
                                 custom_gesture->params.max_displacement,
                                 custom_gesture->params.max_hold_ms)) {
                  goto failed;
               }

               goto nochange;
            case INPUT_EVENT_MTOUCH_RELEASE:
               if (custom_gesture->dt_state >= CT_STATE_SECOND_TOUCH) {
                  save_coords(event, &custom_gesture->second_release);
                  if (!_check_valid(&custom_gesture->second_touch,
                                    &custom_gesture->second_release,
                                    custom_gesture->params.max_displacement,
                                    custom_gesture->params.max_hold_ms)) {
                     goto failed;
                  }
                  custom_gesture->dt_state = CT_STATE_SECOND_RELEASE;
                  goto complete;
               } else {
                  save_coords(event, &custom_gesture->first_release);
                  if (!_check_valid(&custom_gesture->first_touch,
                                    &custom_gesture->first_release,
                                    custom_gesture->params.max_displacement,
                                    custom_gesture->params.max_hold_ms)) {
                     goto failed;
                  }
                  custom_gesture->dt_state = CT_STATE_FIRST_RELEASE;

                  /* Set a timer in case an event doesn't come in */
                  gesture_timer_set_event(gesture,
                                          custom_gesture->fail_timer,
                                          custom_gesture->params.max_delay_ms,
                                          event);
               }

               goto nochange;
            default:
               warn("Unhandled switch/case: %d", event->event_type);
               goto failed;
         }
      case GESTURE_STATE_RECOGNIZED:
         error("GESTURE_STATE_RECOGNIZED is an invalid state for double tap");
         break;
      case GESTURE_STATE_UPDATING:
         error("GESTURE_STATE_UPDATING is an invalid state for double tap");
         break;
      case GESTURE_STATE_COMPLETE:
         error("process_event() called on complete gesture");
         break;
      case GESTURE_STATE_FAILED:
         error("process_event() called on failed gesture");
         break;
      case GESTURE_STATE_NONE:
         error("process_event() called on uninitialized gesture");
         break;
   }

nochange:
   *consumed = 0;
   return custom_gesture->base.state;

failed:
   *consumed = 0;
   return GESTURE_STATE_FAILED;

complete:
   *consumed = 1;
   return GESTURE_STATE_COMPLETE;
}

/* Custom timer callback */
static gesture_state_e
timeout(gesture_base_t* base, void* param)
{
   return GESTURE_STATE_FAILED;
}

/* Custom reset() function */
void custom_gesture_gesture_reset(gesture_base_t* gesture)
{
   gesture_custom_gesture_t* custom_gesture = (gesture_custom_gesture_t*)gesture;
   custom_gesture->dt_state = CT_STATE_INIT;
}

/* Initialize custom gesture parameters with default values. */
void custom_gesture_gesture_default_params(custom_gesture_params_t* params)
{
   params->max_displacement = GESTURE_MAX_MOVE_TOLERANCE_PIX;
   params->max_hold_ms = GESTURE_MAX_TAP_DELAY_MS;
   params->max_delay_ms = GESTURE_MIN_HOLD_DELAY_MS;
}

/* Custom alloc() function */
gesture_custom_gesture_t* custom_gesture_gesture_alloc(custom_gesture_params_t* params,
                                                       gesture_callback_f callback,
                                                       struct gestures_set* set)
{
 gesture_custom_gesture_t* custom_gesture = calloc(1, sizeof(*custom_gesture));
```

```
    if (NULL == custom_gesture) {
       return NULL;
    }

    gesture_base_init(&custom_gesture->base);
    gestures_set_add(set, &custom_gesture->base);
    custom_gesture->base.type = GESTURE_USER;
    custom_gesture->base.funcs.free = custom_gesture_gesture_free;
    custom_gesture->base.funcs.process_event = custom_gesture_gesture_process_event;
    custom_gesture->base.funcs.reset = custom_gesture_gesture_reset;
    custom_gesture->base.callback = callback;
    custom_gesture->fail_timer = gesture_timer_create(&custom_gesture->base, timeout, NULL);
    custom_gesture_gesture_reset(&custom_gesture->base);

    if (NULL != params) {
       custom_gesture->params = *params;
    } else {
       custom_gesture_gesture_default_params(&custom_gesture->params);
    }

    return custom_gesture;
}
```

Chapter 6
Gestures Library Reference

The Gestures library primarily provides gesture recognizers to detect gestures through mtouch events that occur when you place one or more fingers on a touch screen.

Here is a summary of what the Gestures library provides:

Gesture Recognizers

self-contained state machines that detect gestures through mtouch events

Gesture Sets

collections of gesture recognizers that interface between the gesture recognizers and the application

Gesture buckets

lists of gesture recognizers that have not yet been processed

Gesture timers

data type definitions and functions for manipulating timers used for determining the time elapsed between touches or the length of a touch

Event lists

data type definitions and functions for lists of mtouch events to be processed by the gesture sets

Data types and Helper functions

data type definitions and helper functions for recognizing gestures from the touch screen

bucket.h

Data types and functions for gesture buckets.

The `bucket.h` header file provides data type definitions and functions for the gesture bucket, that is, the set of gestures that have not yet been processed.

Definitions in *bucket.h*

Preprocessor macro definitions for the bucket.h header file in the library.

Definitions:

```
#define GESTURES_BUCKET_GROW_INCREMENT 4
```

The number of gestures by which the gestures bucket grows each time it reaches its `size`.

Library:

`libgestures`

gestures_bucket_add()

Add a gesture to a gesture bucket.

Synopsis:

```
#include <gestures/bucket.h>

int gestures_bucket_add(gestures_bucket_t *bucket,
                        struct gesture_base *gesture)
```

Arguments:

bucket

A pointer to the gesture bucket.

gesture

A pointer to the gesture to add.

Library:

`libgestures`

Description:

This function adds a gesture to a gestures bucket.

Returns:

0 on success, or -1 on failure.

gestures_bucket_clear()

Clear a gesture bucket.

Synopsis:

```
#include <gestures/bucket.h>

void gestures_bucket_clear(gestures_bucket_t *bucket)
```

Arguments:

bucket

A pointer to the gesture bucket to clear.

Library:

`libgestures`

Description:

This function removes all entries from the gestures array and sets the gesture count and bucket size to zero.

Returns:

Nothing.

gestures_bucket_count()

Return the number of gestures in a gesture bucket.

Synopsis:

```
#include <gestures/bucket.h>

int gestures_bucket_count(gestures_bucket_t *bucket)
```

Arguments:

bucket

A pointer to the gesture bucket.

Library:

`libgestures`

Description:

This function returns the number of gestures in the specified gesture bucket.

Returns:

The number of gestures in the bucket.

gestures_bucket_del()

Remove a gesture from a gesture bucket.

Synopsis:

```
#include <gestures/bucket.h>

int gestures_bucket_del(gestures_bucket_t *bucket,
                        uint_t idx)
```

Arguments:

bucket

A pointer to the gesture bucket.

idx

The index of the gesture to delete.

Library:

`libgestures`

Description:

This function deletes the gesture at the specified index from the gesture bucket.

Returns:

0 on success, or -1 on failure.

gestures_bucket_get()

Return the gesture at the specified index.

Synopsis:

```
#include <gestures/bucket.h>

struct gesture_base* gestures_bucket_get(gestures_bucket_t
*bucket,
                                         uint_t idx)
```

Arguments:

bucket

A pointer to the gesture bucket.

idx

The index of the gesture to return.

Library:

```
libgestures
```

Description:

The function gestures_bucket_get() returns a pointer to the gesture at the specified index of the gesture bucket.

Returns:

A pointer to the gesture at the specified index.

gestures_bucket_init()

Initialize a gesture bucket.

Synopsis:

```
#include <gestures/bucket.h>

void gestures_bucket_init(gestures_bucket_t *bucket)
```

Arguments:

bucket

A pointer to the gesture bucket structure to initialize.

Library:

```
libgestures
```

Description:

This function initializes the gesture bucket structure.

Returns:

Nothing.

gestures_bucket_t

Structure representing the gesture bucket.

Synopsis:

```
typedef struct  {
    struct gesture_base ** gestures ;
    _Uint32t gestures_count ;
    _Uint32t size ;
}gestures_bucket_t;
```

Data:

struct gesture_base ** gestures

The array of gestures.

_Uint32t gestures_count

The number of gestures currently in the bucket.

_Uint32t size

The total allocated size of the bucket (measured in number of increments of gesture_base_t size)

Library:

```
libgestures
```

Description:

This structure represents the set of gestures awaiting processing. The gesture bucket grows indefinitely by `GESTURES_BUCKET_GROW_INCREMENT` every time it reaches `size`.

defaults.h

Global settings for gesture timings.

The `defaults.h` header file provides global settings for hold delay, move tolerance, touch interval, and tap delay.

Definitions in *defaults.h*

Preprocessor macro definitions for the defaults.h header file in the library.

Definitions:

`#define GESTURE_MIN_HOLD_DELAY_MS 400`

The amount of time the finger must remain touching the screen to qualify as a hold.

`#define GESTURE_MAX_MOVE_TOLERANCE_PIX 16`

The maximum number of pixels in any direction a touch gesture can move before failing.

`#define GESTURE_TWO_FINGER_INTERVAL_MS 100`

The maximum time interval between two fingers touching the screen for the gesture to be considered a two-finger touch, as opposed to two single-finger touches.

`#define GESTURE_MAX_TAP_DELAY_MS 300`

The maximum delay between taps for a double-tap or triple-tap gesture.

A longer delay results in multiple single taps.

Library:

`libgestures`

double_tap.h

Definition of the double tap gesture.

The `double_tap.h` header file provides data type definitions and functions for the double tap gesture. Your application must provide the callback function to handle changes in gesture state.

double_tap_gesture_alloc()

Allocate and initialize the double tap gesture structure.

Synopsis:

```
#include "gestures/double_tap.h"


gesture_double_tap_t*
double_tap_gesture_alloc(double_tap_params_t *params,

gesture_callback_f callback,
                                        struct
gestures_set *set)
```

Arguments:

params

The double tap gesture parameters.

callback

The function to invoke when the double tap gesture changes state.

set

The gesture set to add this double tap gesture to.

Library:

`libgestures`

Description:

This function allocates a new double tap gesture data structure, initializes it with the specified parameters and callback function, and adds it to the specified gesture set.

Returns:

An initialized double tap gesture.

double_tap_gesture_default_params()

Initialize the double tap parameters.

Synopsis:

```
#include "gestures/double_tap.h"


void double_tap_gesture_default_params(double_tap_params_t
*params)
```

Arguments:

params

The double tap gesture parameter structure to initialize.

Library:

`libgestures`

Description:

This function initializes the double tap parameters to default values.

Returns:

Nothing.

double_tap_params_t

Double tap gesture parameters.

Synopsis:

```
typedef struct  {
    unsigned max_displacement ;
    unsigned max_hold_ms ;
    unsigned max_delay_ms ;
}double_tap_params_t;
```

Data:

unsigned max_displacement

The maximum distance the finger can move before the double tap gesture fails.

unsigned max_hold_ms

The maximum time the finger can remain touching the screen before the double tap gesture fails.

unsigned max_delay_ms

The time between the first release and the second touch.

Library:

`libgestures`

Description:

This stucture represents the parameters for the double tap gesture.

double_tap_state_e

States for the double tap gesture.

Synopsis:

```
#include "gestures/double_tap.h"


typedef enum {
      DT_STATE_INIT = 0
      DT_STATE_FIRST_TOUCH
      DT_STATE_FIRST_RELEASE
      DT_STATE_SECOND_TOUCH
      DT_STATE_SECOND_RELEASE
} double_tap_state_e;
```

Data:

DT_STATE_INIT

The initial state of the double tap gesture.

DT_STATE_FIRST_TOUCH

The state of the double tap gesture after the first touch was detected.

DT_STATE_FIRST_RELEASE

The state of the double tap gesture after the first release was detected.

DT_STATE_SECOND_TOUCH

The state of the double tap gesture after the second touch was detected.

DT_STATE_SECOND_RELEASE

The state of the double tap gesture after the second release was detected.

Library:

`libgestures`

Description:

This enumeration defines additional states the double tap gesture can transition between.

gesture_double_tap_t

The double tap gesture data type.

Synopsis:

```
typedef struct  {
    gesture_base_t base ;
    double_tap_params_t params ;
    gesture_coords_t first_touch ;
    gesture_coords_t first_release ;
    gesture_coords_t second_touch ;
    gesture_coords_t second_release ;
    double_tap_state_e dt_state ;
    int fail_timer ;
}gesture_double_tap_t;
```

Data:

gesture_base_t base

The gesture base data structure.

double_tap_params_t params

The double tap parameters.

gesture_coords_t first_touch

The coordinates of the first touch.

gesture_coords_t first_release

The coordinates of the first release.

gesture_coords_t second_touch

The coordinates of the second touch.

gesture_coords_t second_release

The coordinates of the second release.

double_tap_state_e dt_state

The intermediate state of the double tap.

int fail_timer

The ID of the timer for this gesture.

Library:

`libgestures`

Description:

This structure carries data about the double tap gesture.

event_list.h

Data types and functions for event lists.

The `event_list.h` header file provides data type definitions and functions for lists of touch events to be processed by the gesture sets.

This file makes use of list and queue macros defined in the header file sys/queue.h. For more information about these macros, see the documentation in the sys/queue.h file.

event_list_add()

Add an event to an event list.

Synopsis:

```
#include <gestures/event_list.h>

int event_list_add(struct event_list *list,
                   mtouch_event_t *event)
```

Arguments:

list

A pointer to the event list.

event

A pointer to the event to add.

Library:

`libgestures`

Description:

This function adds an event to an event list.

Returns:

0 on success, -1 on failure

event_list_alloc()

Allocate and initialize an event list.

Synopsis:

```
#include <gestures/event_list.h>

struct event_list* event_list_alloc(unsigned init_size,
                                    unsigned grow_size,
                                    unsigned max_size,
                                    int allow_compress)
```

Arguments:

init_size

The initial size of the list, in number of events.

grow_size

The size by which a list is to be grown when full, in number of events.

max_size

The maximum size a list will be allowed to grow to, in number of events.

allow_compress

Allow move events to be dropped to make room in the list.

Library:

`libgestures`

Description:

This function allocates and initializes a new event list. If `init_size`, `grow_size` and `max_size` are all zero, the defaults values of 256, 128 and 1024, respectively, are used. If the `allow_compress` parameter is non-zero, touch events of type `INPUT_EVENT_MTOUCH_MOVE` will be deleted from the list when it is full to make room for more items.

Returns:

The newly allocated event list on success, NULL on failure

event_list_alloc_copy()

Allocate and initialize an event list from an existing list.

Synopsis:

```
#include <gestures/event_list.h>

struct event_list* event_list_alloc_copy(struct event_list
*list)
```

Arguments:

list

A pointer to the event list to copy

Library:

`libgestures`

Description:

This function allocates and initializes a new event list by copying the entries from an existing event list.

Returns:

The newly allocated event list on success, NULL on failure

event_list_clear()

Clear an event list.

Synopsis:

```
#include <gestures/event_list.h>

void event_list_clear(struct event_list *list)
```

Arguments:

list

A pointer to the even list to clear.

Library:

`libgestures`

Description:

This function returns an event list to the empty state. It does not free any of the associated memory.

Returns:

Nothing.

event_list_copy()

Copy an event list.

Synopsis:

```
#include <gestures/event_list.h>

int event_list_copy(struct event_list *from_list,
                    struct event_list *to_list)
```

Arguments:

from_list

A pointer to the event list to copy from.

to_list

A pointer to the event list to copy to.

Library:

`libgestures`

Description:

This function copies the events from one event list to another.

Returns:

0 on success, -1 if to_list is too small to hold all events.

event_list_free()

Free an event list.

Synopsis:

```
#include <gestures/event_list.h>

void event_list_free(struct event_list *list)
```

Arguments:

list

A pointer to the event list.

Library:

`libgestures`

Description:

This function resets the members of the specified list and frees the associated memory.

Returns:

Nothing.

event_list_get_first()

Return the first item in an event list.

Synopsis:

```
#include <gestures/event_list.h>

event_list_item_t* event_list_get_first(struct event_list *list)
```

Arguments:

list

A pointer to an event list.

Library:

`libgestures`

Description:

This function gets the first element in an event list. Use the macro `STAILQ_NEXT(element, field)` to walk the list, where `element` is a pointer to an event list item, and `field` is the `link` member of the event list item.

Returns:

A pointer to the first item in the event list.

event_list_item

Event list item.

Synopsis:

```
typedef struct event_list_item {
    mtouch_event_t event ;
}event_list_item_t;
```

Data:

mtouch_event_t event

A touch event.

STAILQ_ENTRY(event_list_item) link

A macro that resolves to a an event_list_item pointer.

Use the `event_list_*`() functions and the `STAILQ` macros to manipulate the event list, rather than manipulating it directly.

Library:

`libgestures`

Description:

This structure represents an item in the list of touch events that need to be handled. Use the `event_list_*`() functions to manipulate the event list.

event_list_item_t

Event list item.

Synopsis:

```
#include <gestures/event_list.h>
```

```
typedef struct event_list_item  event_list_item_t;
```

Library:

`libgestures`

Description:

This structure represents an item in the list of touch events that need to be handled. Use the `event_list_*`() functions to manipulate the event list.

long_press.h

Definition of the long press gesture.

The `long_press.h` header file provides data type definitions and functions for the long press gesture.

gesture_long_press_t

The long press data structure.

Synopsis:

```
typedef struct  {
    gesture_base_t base ;
    long_press_params_t params ;
    gesture_coords_t coords ;
    int success_timer ;
}gesture_long_press_t;
```

Data:

gesture_base_t base

The gesture base data structure.

long_press_params_t params

The long press parameters.

gesture_coords_t coords

The coordinates of the touch.

int success_timer

The ID of the timer for this gesture.

Library:

`libgestures`

Description:

This structure carries data about the long press gesture.

long_press_gesture_alloc()

Allocate a new long press gesture.

Synopsis:

```
gesture_long_press_t*
long_press_gesture_alloc(long_press_params_t *params,

gesture_callback_f callback,
                                                struct
gestures_set *set)
```

Arguments:

params

A pointer to the long press gesture parameters.

callback

The function to invoke when the long press gesture changes state.

set

A pointer to the gesture set to add this long press gesture to.

Library:

`libgestures`

Description:

This function allocates a new long press gesture data structure and initializes it with the specified parameters and callback function, and adds it to the specified gesture set.

Returns:

A pointer to an initialized pinch gesture.

long_press_gesture_default_params()

Initialize the long press parameters.

Synopsis:

```
void long_press_gesture_default_params(long_press_params_t
*params)
```

Arguments:

params

The long press gesture parameters.

Library:

`libgestures`

Description:

This function initializes the gestures parameters to default values.

Returns:

Nothing.

long_press_params_t

Parameters for the long press gesture.

Synopsis:

```
typedef struct  {
    unsigned max_displacement ;
    unsigned min_press_time_ms ;
}long_press_params_t;
```

Data:

unsigned max_displacement

The maximum number of pixels from the initial coordinates of the touch before the press fails.

unsigned min_press_time_ms

The minumum time in millisecond for the gesture to be considered a long press and not a tap.

Library:

`libgestures`

Description:

This structure represents the parameters for the long press gesture.

pinch.h

Definition of the pinch gesture.

The `pinch.h` header file provides data type definitions and functions for the pinch gesture. Your application must provide the callback function to handle changes in gesture state.

gesture_pinch_t

The pinch gesture data type.

Synopsis:

```
typedef struct  {
    gesture_base_t base ;
    pinch_params_t params ;
    gesture_coords_t coords [2];
    gesture_coords_t centroid ;
    gesture_coords_t last_centroid ;
    gesture_coords_t distance ;
    gesture_coords_t last_distance ;
}gesture_pinch_t;
```

Data:

gesture_base_t base

The gesture base data structure.

pinch_params_t params

The swipe parameters.

gesture_coords_t coords[2]

The coordinates of the touch events for the two fingers.

gesture_coords_t centroid

The coordintes of the midpoint between the two touches.

gesture_coords_t last_centroid

The coordintes of the midpoint between the previous two touches.

gesture_coords_t distance

The distance between the current touches.

gesture_coords_t last_distance

The distance between the previous touches.

Library:

`libgestures`

Description:

This structure carries data about the pinch gesture.

pinch_gesture_alloc()

Allocate and initialize the pinch gesture structure.

Synopsis:

```
#include "gestures/pinch.h"

gesture_pinch_t* pinch_gesture_alloc(pinch_params_t *params,
                                     gesture_callback_f
callback,
                                    struct gestures_set *set)
```

Arguments:

params

A pointer to the pinch gesture parameters.

callback

The function to invoke when the pinch gesture changes state.

set

A pointer to the gesture set to add this pinch gesture to.

Library:

`libgestures`

Description:

This function allocates a new pinch gesture data structure, initializes it with the specified parameters and callback function, and adds it to the specified gesture set.

Returns:

A pointer to an initialized pinch gesture.

pinch_gesture_default_params()

Initialize the pinch parameters.

Synopsis:

```
#include "gestures/pinch.h"

void pinch_gesture_default_params(pinch_params_t *params)
```

Arguments:

params

A pointer to the pinch gesture parameter structure to initialize.

Library:

`libgestures`

Description:

This function initializes the pinch parameters to default values.

Returns:

Nothing.

pinch_params_t

Pinch gesture parameters.

Synopsis:

```
struct  {
    int none ;
};
```

Data:

int none

Not used.

Library:

`libgestures`

Description:

This structure is provided for consistency with other gesture implementations. Although it carries no information, it cannot be empty because common functions rely on its existence.

press_and_tap.h

Definition of the press and tap gesture.

The `press_and_tap.h` header file provides data type definitions and functions for the press and tap gesture. Your application must provide the callback function to handle changes in gesture state.

gesture_pt_t

The press and tap gesture data type.

Synopsis:

```
typedef struct  {
    gesture_base_t base ;
    pt_params_t params ;
    gesture_coords_t initial_coords [2];
    gesture_coords_t coords [2];
}gesture_pt_t;
```

Data:

gesture_base_t base

The gesture base data structure.

pt_params_t params

The swipe parameters.

gesture_coords_t initial_coords[2]

Index of the two sets of coordinates; 0 = press; 1 = tap.

gesture_coords_t coords[2]

The coordinates of the press and the tap.

Library:

`libgestures`

Description:

This structure carries data about the press and tap gesture.

pt_gesture_alloc()

Allocate and initialize the press and tap gesture structure.

Synopsis:

```
gesture_pt_t* pt_gesture_alloc(pt_params_t *params,
                               gesture_callback_f callback,
                               struct gestures_set *set)
```

Arguments:

params

The press and tap gesture parameters.

callback

The function to invoke when the press and gesture changes state.

set

The gesture set to add this press and tap gesture to.

Library:

`libgestures`

Description:

This function allocates a new press and tap gesture data structure, initializes it with the specified parameters and callback function, and adds it to the specified gesture set.

Returns:

An initialized press and tap gesture.

pt_gesture_default_params()

Initialize the press and tap parameters.

Synopsis:

```
void pt_gesture_default_params(pt_params_t *params)
```

Arguments:

params

A pointer to the press and tap gesture parameter structure to initialize.

Library:

`libgestures`

Description:

This function initializes the press and tap parameters to default values.

Returns:

Nothing.

pt_params_t

Press and touch gesture parameters.

Synopsis:

```
typedef struct  {
    unsigned max_tap_time ;
    unsigned min_press_tap_interval ;
    unsigned max_press_tap_interval ;
    unsigned max_displacement ;
}pt_params_t;
```

Data:

unsigned max_tap_time

The maximum time the second finger can be held down.

unsigned min_press_tap_interval

The minimum time between the pressing finger touching down and the tapping finger touching down.

unsigned max_press_tap_interval

The maximum time between the pressing finger touching down and the tapping finger touching down.

unsigned max_displacement

Maximum distance either finger can move before the gesture fails.

Library:

`libgestures`

Description:

This structure represents the parameters for the press and tap gesture.

rotate.h

Definition of the rotate gesture.

The `rotate.h` header file provides data type definitions and functions for the rotate gesture. Your application must provide the callback function to handle changes in gesture state.

gesture_rotate_t

The rotate gesture data type.

Synopsis:

```
typedef struct  {
    gesture_base_t base ;
    rotate_params_t params ;
    gesture_coords_t coords [2];
    gesture_coords_t centroid ;
    gesture_coords_t last_centroid ;
    int angle ;
    int last_angle ;
}gesture_rotate_t;
```

Data:

gesture_base_t base

The gesture base data structure.

rotate_params_t params

The swipe parameters.

gesture_coords_t coords[2]

The coordinates of the touch events for the two fingers.

gesture_coords_t centroid

The coordintes of the midpoint between the two touches.

gesture_coords_t last_centroid

The coordintes of the midpoint between the previous two touches.

int angle

The angle of the current two touches relative to the x axis.

int last_angle

The previous angle.

Library:

`libgestures`

Description:

This structure carries data about the rotate gesture.

rotate_gesture_alloc()

Allocate and initialize the rotate gesture structure.

Synopsis:

```
#include "gestures/rotate.h"

gesture_rotate_t* rotate_gesture_alloc(rotate_params_t *params,
                                       gesture_callback_f
callback,
                                       struct gestures_set
*set)
```

Arguments:

params

A pointer to the rotate gesture parameters.

callback

The function to invoke when the rotate gesture changes state.

set

A pointer to the gesture set to add this rotate gesture to.

Library:

`libgestures`

Description:

This function allocates a new rotate gesture data structure and initializes it with the specified parameters and callback function, and adds it to the specified gesture set.

Returns:

A pointer to an initialized rotate gesture.

rotate_gesture_default_params()

Initialize the rotate parameters.

Synopsis:

```
#include "gestures/rotate.h"

void rotate_gesture_default_params(rotate_params_t *params)
```

Arguments:

params

A pointer to the rotate gesture parameter structure to initialize.

Library:

```
libgestures
```

Description:

This function initializes the rotate parameters to default values.

Returns:

Nothing.

rotate_params_t

Rotate gesture parameters.

Synopsis:

```
struct  {
    int none ;
};
```

Data:

int none

Not used.

Library:

`libgestures`

Description:

This structure is provided for consistency with other gesture implementations. Although carries no information, it cannot be empty because common functions rely on its existence.

set.h

Data types and functions for gesture sets.

The `set.h` header file provides data type definitions and functions for the gesture set.

gestures_set_add()

Add a gesture to a gesture set.

Synopsis:

```
#include "gestures/set.h"

void gestures_set_add(struct gestures_set *set,
                      gesture_base_t *gesture)
```

Arguments:

set

A pointer to the gesture set.

gesture

A pointer to the gesture to add.

Library:

`libgestures`

Description:

This function adds the specified gesture to the specified gesture set.

Returns:

Nothing.

gestures_set_alloc()

Allocate and initialize a new gesture set.

Synopsis:

```
#include "gestures/set.h"

struct gestures_set* gestures_set_alloc()
```

Arguments:

Library:

`libgestures`

Description:

This function creates a new gesture set and initializes.

Returns:

Nothing.

gestures_set_fail_f

Callback function for gesture set failure.

Synopsis:

```
#include "gestures/set.h"

typedef void(* gestures_set_fail_f)(struct gestures_set *set,
             struct event_list *list,
             int async);
```

Library:

`libgestures`

Description:

This callback function is invoked when the gesture set fails. The gesture set fails when all gestures in the set transition to the FAILED state. The callback is passed the list of events leading up to the failure, so that those events can be passed to another gesture set, if necessary.

The `async` parameter indicates which thread invoked the callback:

- the thread that called gestures_set_process_event() (`async` == 0)

- the timer thread (`async` == 1)

Returns:

Nothing.

gestures_set_free()

Free a gesture set.

Synopsis:

```
#include "gestures/set.h"

void gestures_set_free(struct gestures_set *set)
```

Arguments:

set

A pointer to the gesture set to free.

Library:

`libgestures`

Description:

This function frees the memory associated with a gesture set.

Returns:

Nothing.

gestures_set_process_event()

Process a touch event at the gesture set level.

Synopsis:

```
#include "gestures/set.h"

int gestures_set_process_event(struct gestures_set *set,
                               struct mtouch_event *event,
                               void *param)
```

Arguments:

set

A pointer to the gesture set.

event

The touch event to process.

param

A pointer to the parameter list for the callback functions.

Library:

`libgestures`

Description:

This function processes incoming touch events by adding them to the event list and passing them to the individual gestures so that their processing callback functions can be invoked as appropriate.

Returns:

The number of callback functions invoked.

gestures_set_process_event_list()

Process the event list.

Synopsis:

```
#include "gestures/set.h"

int gestures_set_process_event_list(struct gestures_set *set,
                                    struct event_list *list,
                                    void *param)
```

Arguments:

set

A pointer to the gesture set.

list

The event list to process.

param

A pointer to the parameter list for the callback functions.

Library:

`libgestures`

Description:

This function processes the event list by adding the events to the gesture set, updating various properties associated with the events, evaluating timers, and passing each event to the individual gestures for processing.

Returns:

The number of callback functions invoked.

gestures_set_register_fail_cb()

Add a failure callback to a gesture set.

Synopsis:

```
#include "gestures/set.h"

void gestures_set_register_fail_cb(struct gestures_set *set,
                                   gestures_set_fail_f callback)
```

Arguments:

set

A pointer to the gesture set.

callback

The callback function to add.

Library:

`libgestures`

Description:

This function adds the specified callback function to the specified gesture set.

Returns:

Nothing.

swipe.h

Definition of the swipe gesture.

The `swipe.h` header file provides data type definitions and functions for the swipe gesture. Your application must provide the callback function to handle changes in gesture state.

gesture_swipe_t

The swipe gesture data type.

Synopsis:

```
typedef struct  {
    gesture_base_t base ;
    swipe_params_t params ;
    gesture_coords_t initial_coords ;
    gesture_coords_t coords ;
    gesture_coords_t last_coords ;
    int moving ;
    int direction ;
}gesture_swipe_t;
```

Data:

gesture_base_t base

The gesture base data structure.

swipe_params_t params

The swipe parameters.

gesture_coords_t initial_coords

The coordinates of the first touch.

gesture_coords_t coords

The coordinates of an intermediate point in the swipe.

gesture_coords_t last_coords

The coordinates of the point where the finger was lifted from the screen.

int moving

Indicates whether the last event was a move.

int direction

The direction of the swipe.

Library:

`libgestures`

Description:

This structure carries data about the swipe gesture.

swipe_gesture_default_params()

Initialize the swipe parameters.

Synopsis:

```
#include "gestures/swipe.h"

void swipe_gesture_default_params(swipe_params_t *params)
```

Arguments:

params

The swipe gesture parameter structure to initialize.

Library:

`libgestures`

Description:

This function initializes the swipe parameters to default values.

Returns:

Nothing.

swipe_params_t

Swipe gesture parameters.

Synopsis:

```
typedef struct  {
    unsigned directions ;
    unsigned off_axis_tolerance ;
    unsigned min_distance ;
    unsigned min_velocity ;
}swipe_params_t;
```

Data:

unsigned directions

Bitmask of the directions in which the swipe can occur.

unsigned off_axis_tolerance

The number of touch units the swipe can occur off axis with the direction.

unsigned min_distance

The minimum distance traveled in the touch direction for the gesture to be considered a swipe.

unsigned min_velocity

The minimum velocity between any two points in the swipe.

Library:

`libgestures`

Description:

This structure represents the parameters for the swipe gesture.

tap.h

Definition of the tap gesture.

The `tap.h` header file provides data type definitions and functions for the tap gesture. Your application must provide the callback function to handle changes in gesture state.

gesture_tap_t

The tap gesture data type.

Synopsis:

```
typedef struct  {
    gesture_base_t base ;
    tap_params_t params ;
    gesture_coords_t touch_coords ;
}gesture_tap_t;
```

Data:

gesture_base_t base

The gesture base data structure.

tap_params_t params

The swipe parameters.

gesture_coords_t touch_coords

The coordinates of the touch event.

Library:

`libgestures`

Description:

This structure carries data about the tap gesture.

tap_gesture_alloc()

Allocate and initialize the tap gesture structure.

Synopsis:

```
#include "gestures/tap.h"

gesture_tap_t* tap_gesture_alloc(tap_params_t *params,
                                 gesture_callback_f callback,
                                 struct gestures_set *set)
```

Arguments:

params

The tap gesture parameters.

callback

The function to invoke when the tap gesture changes state.

set

The gesture set to add this tap gesture to.

Library:

`libgestures`

Description:

This function allocates a new tap gesture data structure and initializes it with the specified parameters and callback function, and adds it to the specified gesture set.

Returns:

An initialized tap gesture.

tap_gesture_default_params()

Initialize the tap parameters.

Synopsis:

```
#include "gestures/tap.h"
```

```
void tap_gesture_default_params(tap_params_t *params)
```

Arguments:

params

The tap gesture parameter structure to initialize.

Library:

`libgestures`

Description:

This function initializes the tap parameters to default values.

Returns:

Nothing.

tap_params_t

Tap gesture parameters.

Synopsis:

```
typedef struct  {
    unsigned max_displacement ;
    unsigned max_hold_ms ;
}tap_params_t;
```

Data:

unsigned max_displacement

Maximum distance the finger can move before the tap gesture fails.

unsigned max_hold_ms

Maximum time the finger can remain touching the screen before the tap gesture fails.

Library:

`libgestures`

Description:

This structure represents the parameters for the tap gesture.

timer.h

Functions and datatypes for gestures timers.

The `timer.h` header file provides data type definitions and functions for manipulating timers used for determining the time elapsed between touches or the length of a touch.

gesture_timer_callback_t

Gesture timer callback function type.

Synopsis:

```
#include "gestures/timer.h"


typedef gesture_state_e(* gesture_timer_callback_t)(struct
gesture_base *gesture,
                        void *param);
```

Arguments:

gesture

A pointer to the gesture.

param

A pointer to the parameter list.

Library:

`libgestures`

Description:

This callback function is invoked when a timer expires.

Returns:

The current gesture state.

gesture_timer_clear()

Clear an armed timer.

Synopsis:

```
#include "gestures/timer.h"

int gesture_timer_clear(struct gesture_base *gesture,
                        int timer_id)
```

Arguments:

gesture

A pointer to the gesture.

timer_id

The timer to clear.

Library:

`libgestures`

Description:

This function clears an armed timer (a timer previously set by one of the `gesture_timer_set_*`() functions). If *the timer is unarmed, this function does nothing and returns 0.

Returns:

0 on success, or -1 on failure.

gesture_timer_create()

Create a gesture timer.

Synopsis:

```
#include "gestures/timer.h"

int gesture_timer_create(struct gesture_base *gesture,
                         gesture_timer_callback_t callback,
                         void *param)
```

Arguments:

gesture

A pointer to the gesture.

callback

The gesture timer callback function.

param

A pointer to the parameter list.

Library:

`libgestures`

Description:

This function creates a new gesture timer that invokes the callback function when it expires.

Returns:

The timer ID, or -1 on failure.

gesture_timer_destroy()

Destroy a timer.

Synopsis:

```
#include "gestures/timer.h"

void gesture_timer_destroy(struct gesture_base *gesture,
                           int timer_id)
```

Arguments:

gesture

A pointer to the gesture.

timer_id

The timer to clear.

Library:

`libgestures`

Description:

This function destroys the specified timer by resetting the timer data structure. It does not free any memory associated with the timer.

Returns:

0 on success, or -1 on failure.

gesture_timer_query()

Query a timer.

Synopsis:

```
#include "gestures/timer.h"

int gesture_timer_query(struct gesture_base *gesture,
                        int timer_id,
                        int *pending,
                        _Uint64t *expiry)
```

Arguments:

gesture

A pointer to the gesture.

timer_id

The id of the timer to query.

pending

Returns the current state of the timer.

expiry

Returns the expiry time of the timer.

Library:

`libgestures`

Description:

This function returns information about the timer. If the timer is valid, the return value is set to 0, the parameter `pending` is set to the timer's current pending state (destroyed timer will result in a -1 return value), and `expiry` is set to the timer's expiry time.

Returns:

0 if the timer is valid, -1 otherwise.

gesture_timer_set_event()

Set a gesture timer from a touch event timestamp.

Synopsis:

```
#include "gestures/timer.h"


int gesture_timer_set_event(struct gesture_base *gesture,
                            int timer_id,
                            unsigned ms,
                            struct mtouch_event *base_event)
```

Arguments:

gesture

A pointer to the gesture.

timer_id

The timer to set.

ms

The expiry time in milliseconds.

base_event

A pointer to the touch event whose timestamp is to be used as the base time.

Library:

`libgestures`

Description:

This function sets a timer using a touch event timestamp as the reference time.

Returns:

0 on success, or -1 on failure.

gesture_timer_set_ms()

Set a gesture timer from a timestamp.

Synopsis:

```
#include "gestures/timer.h"

int gesture_timer_set_ms(struct gesture_base *gesture,
                         int timer_id,
                         unsigned ms,
                         _Uint64t base_nsec)
```

Arguments:

gesture

A pointer to the gesture.

timer_id

The timer to set. 8

ms

The expiry time in milliseconds.

base_nsec

The base time in nanoseconds. The `ms` parameter is compared to this time to determine whether the timer has expired.

Library:

`libgestures`

Description:

This function sets a timer using a timestamp as the reference time.

Returns:

0 on success, or -1 on failure.

gesture_timer_set_now()

Set a gesture timer from now.

Synopsis:

```
#include "gestures/timer.h"

int gesture_timer_set_now(struct gesture_base *gesture,
                          int timer_id,
                          unsigned ms)
```

Arguments:

gesture

A pointer to the gesture.

timer_id

The timer to set.

ms

The expiry time in milliseconds (from now).

Library:

`libgestures`

Description:

This function sets a timer using the current time as the reference time.

Returns:

0 on success, or -1 on failure.

triple_tap.h

Definition of the triple tap gesture.

The `triple_tap.h` header file provides data type definitions and functions for the triple tap gesture. Your application must provide the callback function to handle changes in gesture state.

gesture_triple_tap_t

The triple tap gesture data type.

Synopsis:

```
typedef struct  {
    gesture_base_t base ;
    triple_tap_params_t params ;
    gesture_coords_t first_touch ;
    gesture_coords_t first_release ;
    gesture_coords_t second_touch ;
    gesture_coords_t second_release ;
    gesture_coords_t third_touch ;
    gesture_coords_t third_release ;
    triple_tap_state_e tt_state ;
    int fail_timer ;
}gesture_triple_tap_t;
```

Data:

gesture_base_t base

The gesture base data structure.

triple_tap_params_t params

The triple tap parameters.

gesture_coords_t first_touch

The coordinates of the first touch event.

gesture_coords_t first_release

The coordinates of the first release event.

gesture_coords_t second_touch

The coordinates of the second touch event.

gesture_coords_t second_release

The coordinates of the second release event.

gesture_coords_t third_touch

The coordinates of the third touch event.

gesture_coords_t third_release

The coordinates of the third release event.

triple_tap_state_e tt_state

The intermediate state of the triple tap.

int fail_timer

The ID of the timer for this gesture.

Library:

```
libgestures
```

Description:

This structure carries data about the triple tap gesture.

triple_tap_gesture_alloc()

Allocate and initialize the triple tap gesture structure.

Synopsis:

```
#include "gestures/triple_tap.h"


gesture_triple_tap_t*
triple_tap_gesture_alloc(triple_tap_params_t *params,

gesture_callback_f callback,
                                              struct
gestures_set *set)
```

Arguments:

params

A ponter to the triple tap gesture parameters.

callback

The function to invoke when the triple tap gesture changes state.

set

A pointer to the gesture set to add this tap gesture to.

Library:

`libgestures`

Description:

This function allocates a new triple tap gesture data structure, initializes it with the specified parameters and callback function, and adds it to the specified gesture set.

Returns:

A pointer to an initialized triple tap gesture.

triple_tap_gesture_default_params()

Initialize the triple tap parameters.

Synopsis:

```
#include "gestures/triple_tap.h"

void triple_tap_gesture_default_params(triple_tap_params_t
*params)
```

Arguments:

params

A pointer to the triple tap gesture parameter structure to initialize.

Library:

`libgestures`

Description:

This function initializes the triple tap parameters to default values.

Returns:

Nothing.

triple_tap_params_t

Triple tap gesture parameters.

Synopsis:

```
typedef struct  {
    unsigned max_displacement ;
    unsigned max_hold_ms ;
    unsigned max_delay_ms ;
}triple_tap_params_t;
```

Data:

unsigned max_displacement

Maximum distance the finger can move before the triple tap gesture fails.

unsigned max_hold_ms

Maximum time the finger can remain touching the screen before the tap gesture fails.

unsigned max_delay_ms

The maximum time between release and touch.

Library:

```
libgestures
```

Description:

This structure represents the parameters for the triple tap gesture.

triple_tap_state_e

States for the triple tap gesture.

Synopsis:

```
#include "gestures/triple_tap.h"


typedef enum {
      TT_STATE_INIT = 0
      TT_STATE_FIRST_TOUCH
      TT_STATE_FIRST_RELEASE
      TT_STATE_SECOND_TOUCH
      TT_STATE_SECOND_RELEASE
      TT_STATE_THIRD_TOUCH
      TT_STATE_THIRD_RELEASE
} triple_tap_state_e;
```

Data:

TT_STATE_INIT

TT_STATE_FIRST_TOUCH

TT_STATE_FIRST_RELEASE

TT_STATE_SECOND_TOUCH

TT_STATE_SECOND_RELEASE

TT_STATE_THIRD_TOUCH

TT_STATE_THIRD_RELEASE

Library:

`libgestures`

Description:

This enumeration defines additional states the triple tap gesture can transition between.

two_finger_pan.h

Definition of the two finger pan gesture.

The `two_finger_pan.h` header file provides data type definitions and functions for the two-finger pan gesture. Your application must provide the callback function to handle changes in gesture state.

gesture_tfpan_t

The two-finger pan gesture data type.

Synopsis:

```
typedef struct  {
    gesture_base_t base ;
    tfpan_params_t params ;
    gesture_coords_t coords [2];
    gesture_coords_t centroid ;
    gesture_coords_t last_centroid ;
}gesture_tfpan_t;
```

Data:

gesture_base_t base

The gesture base data structure.

tfpan_params_t params

The two-finger pan parameters.

gesture_coords_t coords[2]

The coordinates of the touch events for the two fingers.

gesture_coords_t centroid

The coordinates of the midpoint between the two touches.

gesture_coords_t last_centroid

The coordintes of the midpoint between the previous two touches.

Library:

`libgestures`

Description:

This structure carries data about the pinch gesture.

tfpan_gesture_alloc()

Allocate and initialize the two-finger pan gesture structure.

Synopsis:

```
gesture_tfpan_t* tfpan_gesture_alloc(tfpan_params_t *params,
                                     gesture_callback_f
callback,
                                    struct gestures_set *set)
```

Arguments:

params

A pointer to the two-finger pan gesture parameters.

callback

The function to invoke when the two-finger pan gesture changes state.

set

A pointer to the gesture set to add this two-finger pan gesture to.

Library:

`libgestures`

Description:

This function allocates a new two-finger pan gesture data structure, initializes it with the specified parameters and callback function, and adds it to the specified gesture set.

Returns:

A pointer to an initialized two-finger pan gesture.

tfpan_gesture_default_params()

Initialize the two-finger pan parameters.

Synopsis:

```
void tfpan_gesture_default_params(tfpan_params_t *params)
```

Arguments:

params

A pointer to the two-finger pan gesture parameter structure to initialize.

Library:

`libgestures`

Description:

This function initializes the two-finger pan parameters to default values.

Returns:

Nothing.

tfpan_params_t

Two-finger pan gesture parameters.

Synopsis:

```
struct  {
    int none ;
};
```

Data:

int none

Not used.

Library:

`libgestures`

Description:

This structure is provided for consistency with other gesture implementations. Although carries no information, it cannot be empty because common functions rely on its existence.

two_finger_tap.h

Definition of the two finger tap gesture.

The `two_finger_tap.h` header file provides data type definitions and functions for the two finger tap gesture. Your application must provide the callback function to handle changes in gesture state.

gesture_tft_t

The two-finger tap gesture data type.

Synopsis:

```
typedef struct  {
    gesture_base_t base ;
    tft_params_t params ;
    gesture_coords_t touch_coords [2];
    gesture_coords_t release_coords [2];
    gesture_coords_t centroid ;
    unsigned down_count ;
}gesture_tft_t;
```

Data:

gesture_base_t base

The gesture base data structure.

tft_params_t params

The two-finger tap parameters.

gesture_coords_t touch_coords[2]

The coordinates of the first touch event.

gesture_coords_t release_coords[2]

The coordinates of the second release event.

gesture_coords_t centroid

The midpoint between the two touches.

unsigned down_count

The time in milliseconds that the fingers remained touching.

Library:

`libgestures`

Description:

This structure carries data about the two-finger tap gesture.

tft_gesture_alloc()

Allocate and initialize the two-finger tap gesture structure.

Synopsis:

```
gesture_tft_t* tft_gesture_alloc(tft_params_t *params,
                                 gesture_callback_f callback,
                                 struct gestures_set *set)
```

Arguments:

params

The two-finger tap gesture parameters.

callback

The function to invoke when the two-finger tap gesture changes state.

set

The gesture set to add this two-finger tap gesture to.

Library:

`libgestures`

Description:

This function allocates a new two-finger tap gesture data structure, initializes it with the specified parameters and callback function, and adds it to the specified gesture set.

Returns:

An initialized two-finger tap gesture.

tft_gesture_default_params()

Initialize the two-finger tap parameters.

Synopsis:

```
void tft_gesture_default_params(tft_params_t *params)
```

Arguments:

params

The two-finger tap gesture parameter structure to initialize.

Library:

```
libgestures
```

Description:

This function initializes the two-finger tap parameters to default values.

Returns:

Nothing.

tft_params_t

Two-finger tap gesture parameters.

Synopsis:

```
typedef struct  {
    unsigned max_touch_interval ;
    unsigned max_release_interval ;
    unsigned max_tap_time ;
    unsigned max_displacement ;
}tft_params_t;
```

Data:

unsigned max_touch_interval

The maximum time that can elapse between when the first and second fingers touch the screen.

unsigned max_release_interval

The maximum time that can elapse between when the first and second fingers are released.

unsigned max_tap_time

The maximum time both fingers can stay down.

unsigned max_displacement

The maximum distance either finger can move before the two-finger tap gesture fails.

Library:

`libgestures`

Description:

This structure represents the parameters for the two-finger tap gesture.

types.h

Common data types and helper functions.

The `types.h` header file provides data type definitions and helper functions for recognizing gestures from the touch screen.

Definitions in *types.h*

Preprocessor macro definitions for the types.h header file in the library.

Definitions:

`#define GESTURE_DIRECTION_UP (1 << 0)`

Detection of touch coordinates in the up direction.

`#define GESTURE_DIRECTION_DOWN (1 << 1)`

Detection of touch coordinates in the down direction.

`#define GESTURE_DIRECTION_LEFT (1 << 2)`

Detection of touch coordinates in the left direction.

`#define GESTURE_DIRECTION_RIGHT (1 << 3)`

Detection of touch coordinates in the right direction.

Library:

`libgestures`

diff_time_ms()

Return the elapsed time between two events.

Synopsis:

```
#include <gestures/types.h>


int32_t diff_time_ms(gesture_coords_t *coords1,
                     gesture_coords_t *coords2)
```

Arguments:

coords1

A pointer to the first gesture event.

coords2

A pointer to the second gesture event.

Library:

`libgestures`

Description:

This function returns the elapsed time between the two specified gesture events. You will likely need this function only if you are defining your own gestures.

Returns:

The elapsed time in milliseconds.

gesture_add_mustfail()

Add a gesture to the 'must fail' list.

Synopsis:

```
#include <gestures/types.h>

int gesture_add_mustfail(gesture_base_t *target,
                         gesture_base_t *mustfail)
```

Arguments:

target

The gesture dependent on the failure of the second gesture.

mustfail

The gesture that must fail. This gesture is added to the 'must fail' list of the target gesture.

Library:

`libgestures`

Description:

This function adds a gesture to the 'must fail' list of another gesture. That is, the gesture added to the list must fail in order for the gesture that owns the list to complete.

Returns:

0 on sucess; -1 on error.

gesture_base

Common data structure for all gestures.

Synopsis:

```
typedef struct gesture_base {
    struct gestures_set * set ;
    gesture_e type ;
    gesture_state_e state ;
    gesture_funcs_t funcs ;
    gesture_callback_f callback ;
    gestures_bucket_t mustallfail ;
    gestures_bucket_t faildependents ;
}gesture_base_t;
```

Data:

struct gestures_set * set

A pointer to the gesture set.

gesture_e type

The gesture type.

gesture_state_e state

The current state of the gesture.

gesture_funcs_t funcs

The state and memory handling functions.

gesture_callback_f callback

The gesture handling function, triggered when a gesture changes state.

gestures_bucket_t mustallfail

List of gestures that must fail for this gesture to complete.

gestures_bucket_t faildependents

List of gestures that can only complete after this gesture fails.

TAILQ_ENTRY(gesture_base) glink

A macro that resolves to pointers into the gestures bucket.

Use the `gestures_bucket_*`() functions to manipulate the gestures bucket, rather than manipulating it directly.

Library:

`libgestures`

Description:

This structure represents information that is common to all gestures. Specific gestures include the gesture base in their representation, and also include additional members to capture gesture-specific information.

It is up to the application to define the failure dependencies between gestures and to add gestures to a gesture set.

gesture_base_init()

Initialize the gesture base data structure.

Synopsis:

```
#include <gestures/types.h>

void gesture_base_init(gesture_base_t *gesture)
```

Arguments:

gesture

The gesture to initialize.

Library:

`libgestures`

Description:

This function initializes the gesture base data structure, `gesture_base_t`.

Returns:

None.

gesture_base_t

Common data structure for all gestures.

Synopsis:

```
#include <gestures/types.h>


typedef struct gesture_base  gesture_base_t;
```

Library:

```
libgestures
```

Description:

This structure represents information that is common to all gestures. Specific gestures include the gesture base in their representation, and also include additional members to capture gesture-specific information.

It is up to the application to define the failure dependencies between gestures and to add gestures to a gesture set.

gesture_callback_f

Gesture callback function prototype.

Synopsis:

```
#include <gestures/types.h>


typedef void(* gesture_callback_f)(struct gesture_base *gesture,

             mtouch_event_t *event,
             void *param,
             int async);
```

Arguments:

gesture

A pointer to the gesture.

event

A pointer to the last touch event.

param

A pointer to the parameter list.

async

Indicator which thread invoked the callback:

- 0: the thread that called gestures_set_process_event()
- 1: the timer thread

Library:

```
libgestures
```

Description:

This callback function is invoked every time a gesture changes state, with the exception of the transition from UNRECOGNIZED to FAILED.

Note that if the event passed to the callback is NULL, it means the callback was invoked following a timer callback (as opposed to an event coming in).

gesture_coords_t

Gesture coordinates.

Synopsis:

```
typedef struct  {
    _Int32t x ;
    _Int32t y ;
    _Uint64t timestamp ;
}gesture_coords_t;
```

Data:

_Int32t x

The x coordinate of the touch.

_Int32t y

The y coordinate of the touch.

_Uint64t timestamp

The time when the touch occurred.

Library:

```
libgestures
```

Description:

This structure carries the x and y coordinates of a touch gesture, as well as the time that it occured.

gesture_e

Gesture type enumeration.

Synopsis:

```
#include <gestures/types.h>


typedef enum {
      GESTURE_NONE = 0
      GESTURE_TWO_FINGER_PAN
      GESTURE_ROTATE
      GESTURE_SWIPE
      GESTURE_PINCH
      GESTURE_TAP
      GESTURE_DOUBLE_TAP
      GESTURE_TRIPLE_TAP
      GESTURE_PRESS_AND_TAP
      GESTURE_TWO_FINGER_TAP
      GESTURE_LONG_PRESS
      GESTURE_USER
} gesture_e;
```

Data:

GESTURE_NONE

No gesture.

GESTURE_TWO_FINGER_PAN

The two finger pan gesture.

GESTURE_ROTATE

The rotate gesture.

GESTURE_SWIPE

The swipe gesture.

GESTURE_PINCH

The pinch gesture.

GESTURE_TAP

The tap gesture.

GESTURE_DOUBLE_TAP

The double tap gesture.

GESTURE_TRIPLE_TAP

The triple tap gesture.

GESTURE_PRESS_AND_TAP

The press and tap gesture.

GESTURE_TWO_FINGER_TAP

The two finger gesture.

GESTURE_LONG_PRESS

The long press gesture.

GESTURE_USER

The custom gesture.

Library:

`libgestures`

Description:

This enumeration the set of possible gestures.

gesture_funcs_t

Touch event handling functions.

Synopsis:

```
typedef struct  {
    void(* free )(struct gesture_base *gesture);
    gesture_state_e(* process_event )(struct contact_id_map
*map,
                           struct gesture_base *gesture,
                           mtouch_event_t *event,
                           int *consumed);
    void(* reset )(struct gesture_base *gesture);
}gesture_funcs_t;
```

Data:

void(* free)(struct gesture_base *gesture)

Free all the memory associated with the gesture that was allocated by the gesture's alloc() function.

Arguments

- `gesture` The gesture whose memory is to be freed.

gesture_state_e(* process_event)(struct contact_id_map *map, struct gesture_base *gesture, mtouch_event_t *event, int *consumed)

Manage the state transitions and return the new, or unchanged, gesture state.

Arguments

- `map` Identifier used to identify the mtouch event.
- `gesture` Gesture recognizer whose state is being managed.
- `event` Mtouch event that has been received.
- `consumed` Indicator that gesture recognizer is finished recoginzing a gesture based on mtouch events received.

Returns

- The new state of the gesture recognizer.

void(* reset)(struct gesture_base *gesture)

Reset the gesture-specific data structures to their initial state.

Arguments

- `gesture` Gesture recognizer whose specific data structures are to be returned to initial state.

Library:

`libgestures`

Description:

This data type consists of pointers to functions that handle mtouch events by updating the gesture state, resetting the gesture data structures, or freeing memory associated with the gesture. The following functions must be defined for each of your custom gesture recognizer:

- (*process_event)()
- (*reset)()
- (*free)()

gesture_state_e

Gesture state enumeration.

Synopsis:

```
#include <gestures/types.h>


typedef enum {
      GESTURE_STATE_NONE = 0
      GESTURE_STATE_UNRECOGNIZED
      GESTURE_STATE_RECOGNIZED
      GESTURE_STATE_UPDATING
      GESTURE_STATE_COMPLETE
      GESTURE_STATE_FAILED
} gesture_state_e;
```

Data:

GESTURE_STATE_NONE

The initial state at which a gesture recognizer starts.

GESTURE_STATE_UNRECOGNIZED

The state of a gesture recognizer after it has been added to a gesture set; it is now ready to receive mtouch and timer events.

GESTURE_STATE_RECOGNIZED

The state of a gesture recognizer after it has received one mtouch or timer event.

This state is valid only for composite gestures.

GESTURE_STATE_UPDATING

The state of a gesture recognizer while it is receiving mtouch or timer events.

This state is valid only for composite gestures.

GESTURE_STATE_COMPLETE

The state of a gesture recognizer when it has received all mtouch or timer events that fulfill the requirements of detecting its gesture.

GESTURE_STATE_FAILED

The state of a gesture recognizer when requirements of detecting its gesture is not fulfilled.

Library:

`libgestures`

Description:

This enumeration represents state of a gesture as it is being processed. The possible next state depends on the type of gesture.

map_contact_id()

Remap the contact id from the touch evemt.

Synopsis:

```
#include <gestures/types.h>

int map_contact_id(struct contact_id_map *map,
                   unsigned contact_id)
```

Arguments:

map

A pointer to the gesture map. The gesture passes the map to the process_event() callback function.

contact_id

A touch event contact id.

Library:

`libgestures`

Description:

This function remaps contact identifiers from touch events to contact identifiers to be used by gestures. The touch event data structure contains a contact_id element that is assigned a value from a 0-based index that corresponds to the individual fingers that touch the screen. The ID assigned to a finger will not change until that finger is released. The contact ID from the touch event should not be used directly by the gesture recognizer. Instead, user gestures should call map_contact_id() to get a 0-indexed contact ID remapped from the gesture set's perspective. This is necessary because, for example, the event contact ID 1 could actually correspond to the gesture set's contact ID 0 if there are multiple gesture sets in play, or if the user's finger is resting on the touch-sensitive bezel. You will likely need this function only if you are defining your own gestures.

Returns:

The remapped contact ID.

max_displacement_abs()

Calculate the maximum displacement between two gesture events.

Synopsis:

```
#include <gestures/types.h>

uint32_t max_displacement_abs(gesture_coords_t *coords1,
                              gesture_coords_t *coords2)
```

Arguments:

coords1

A pointer to the first gesture event.

coords2

A pointer to the second gesture event.

Library:

```
libgestures
```

Description:

This function returns the the maximum displacement, in pixels, between two gesture events. For example, if the absolute value of the difference between the x coordinates of the two gestures is greater than the absolute value of the difference between the y coordinates, the former is returned by the function. You will likely need this function only if you are defining your own gestures.

Returns:

The maximum displacement, in pixels.

save_coords()

Save the touch event coordinates.

Synopsis:

```
#include <gestures/types.h>

void save_coords(mtouch_event_t *event,
                 gesture_coords_t *coords)
```

Arguments:

event

A pointer to the event for which to save the coordinates.

coords

A pointer to the gesture coordinates structure to use for saving.

Library:

`libgestures`

Description:

This function saves the coordinates of a touch event in the specified gesture_coords_t data structure. You will likely need this function only if you are defining your own gestures.

Returns:

Nothing.