Writing an InputSprocket Driver

In order to write a driver for InputSprocket, you should already be familiar with the Code Fragment Manager and the InputSprocket application interfaces. If you are writing a driver for an ADB device you should be familiar with the ADB Manager.

An InputSprocket driver is a CFM fragment with specific file type and creator (`'shlb' and 'insp'`). All drivers must be located in the same folder as InputSprocketLib, which must be the Extensions folder. Drivers should link with InputSprocketLib or InputSprocketStubLib (they are identical, previously there was a stub library named InputSprocketDriverStubLib, which contained the entry points for Drivers, you may now use InputSprocketLib). You will need InputSprocket.h and InputSprocketDriver.h. It is recommended that drivers require InputSprocket 1.2 or later, avoiding compatibility issues with certain functions not being available. Drivers which use the Apple Desktop Bus (ADB) will probably want to include InputSprocketDefer.h and link with InputSprocketDeferLib.

A driver must handle six major tasks: CFM initialization, CFM termination, pushing data, 'high level' Init, 'high level' Stop, and the configure dialog user interface. There are two ways an application can use InputSprocket: 'high' or 'low' level. The 'low level' interface corresponds to a representation of the device based on what axis, buttons, etc. it has. The 'high level' interface corresponds to the 'needs' of the application. A driver's low level interface is based on the devices and elements it creates with `ISpDevice_New` and `ISpElement_New`. The high level interface is only valid between Init and Stop calls, and is based on the needs list provided by the application. Only the driver knows exactly how it has been configured, and when it is active, it is responsible for pushing data to those virtual elements for each need to which it is configured, in addition to the low level elements.

CFM Initialization

During CFM initialization, an ISp driver should determine what devices are connected and call `ISpDevice_New` for each detected device. The ISp driver should then call `ISpElement_New` for each element of the device (i.e., button, axis, dpad, delta). The data type that is closest to the actual physical action on the device should be chosen. A device should be initially active, and immediately start pushing data.

One parameter to `ISpDevice_New` is the 'metahandler' function pointer. Within the `ISpDevice_New` call, the device's metahandler function is called once for each driver selector corresponding to an entry point into the driver (e.g. `kISpSelector_Init`). The driver should return `NULL` for the function pointer if the entrypoint is not implemented, otherwise return a pointer to it's corresponding function .

A driver should check using `ISpGetVersion` to verify that InputSprocket at least some minimum version. Drivers should only return an error if CFM does, specifically, they should return `noErr` if they find no devices.

InputSprocket does not currently support 'hot-plugging' when the application is running, but a future version may do so. If this occurs, most likely a driver will have to be marked as 'hot-plug aware' and it will behave differently at CFM Initialization time.

CFM Termination

During CFM termination, for each device created, an ISp driver should deactivate itself if necessary, then call `ISpElement_Dispose` for each element created and then call `ISpDevice_Dispose`.

Pushing Data

A driver is responsible for pushing data to InputSprocket whenever the state of a particular element has changed. If driver Init has been called and has not been followed by a corresponding driver Stop, a driver is also responsible for pushing data to the virtual elements corresponding to the needs to which it has been configured.

When the state of an element changes, a driver should call ISpUptime to determine the time value that the change has occurred and then call either ISpElement_PushSimpleData or ISpElement_PushComplexData to push the data. If the high level is valid, and that element has been configured to a need, the driver should make the same call pushing data to the corresponding virtual element. All three of these API calls are interrupt safe.

The driver calls Tickle and InterruptTickle are provided as a convenience to get regular execution time. They are available with InputSprocket version 1.03 and later. If a driver implements InterruptTickle, if will be called approximately 90 times a second. Tickle will be called once for each time the application calls `ISpTickle`. If a driver relies on Tickle being called, it should set its device class to `kISpDeviceClass_Tickle` when it is created. Some applications will not call `ISpTickle` and so will not work with those drivers. In the currently implementation, InputSprocket deactivates `kISpDeviceClass_Tickle` devices and it is the responsibility of the application to reactivate them if it calls `ISpTickle`. InterruptTickle is only called at 'Virtual Memory (VM) safe' times.

However, you must be aware of any callbacks you install yourself, and whether they can occur during 'VM unsafe' times. If so, your driver must hold itself as resident. Drivers which use ADB can include InputSprocketDefer.h which provides services so that ADB handlers are deferred until paging safe time. This way, the driver does not have to be held (although it will be resident under most conditions if it is actually being called many times a second). Drivers which use USB through USBHIDUniversalModule will automatically be deferred until 'VM safe' times, so do not have to worry about this problem.

When pushing data to axis or delta types, care should be taken to make sure the ISp coordinate system is being followed correctly. This is important so that the (high level) auto-configuration in driver Init works correctly, and so that applications using the low-level interface get a correct 'feel' for what the device really does. Notes on the coordinate system appear later in this document.

SetActive will be called to activate and deactivate a device. Devices should only push data when they are active.

High Level Init

InputSprocket calls each device's Init function after the application calls `ISpInit`. The driver should make a local copy of the needs array and the corresponding virtual elements array.. It should allocate (in the system heap) any memory used only in the high level. At this time, the driver should autoconfigure itself to the needs, taking advantage of the need labels and other provided information. Drivers are responsible for doing the data conversion between the 'native' type of an element and the data type for a need when pushing data to a virtual element.

There are several guidelines which should be followed during autoconfiguration. If the value of the `used` array corresponding to a need is `true`, it means some previous device fulfilled the need. If the value is `true` and the `kISpNeedFlag_NoMultiConfig` bit is set in the need structure for that need, the device should not attempt to autoconfigure to the need. The driver should set `used` to `true` for each need it fulfills on autoconfiguration. The need flags such as `kISpNeedFlag_Button_AlreadyAxis` combined with the need `group` should be used to exclude autoconfiguring multiple elements from the same device to the same need. Needs with the same `group` should be autoconfigured together when possible. If any needs with a non-zero `playerNum` are autoconfigured, then no needs with a different non-zero `playerNum` should be autoconfigured (for the same device).

After autoconfiguring, the device should immediately push initial values to the corresponding virtual elements and from that point push data to those virtual elements whenever data is pushed to the elements to which they are configured.

Until the high level is invalidated with the Stop call, InputSprocket may make the GetState, and SetState calls. The driver should package it's configuration state into a provided buffer when the GetState call is made. If the buffer size is too small, it should just change it to the needed size and return an error. Otherwise, it should change the size to the amount of the buffer used, and place the data into the buffer. This data will later be passed to the driver in a SetState call when the driver should change it's configuration to a previous setting.

High Level Stop

InputSprocket calls each device's Stop function after the application calls `ISpStop`. The driver should suspend pushing data to all virtual elements and deallocate memory allocated in Init.

Configure Dialog User Interface

The majority of driver function calls are related to the Configure dialog. These function calls will only happen while the high level is valid (i.e. `ISpInit` has been called without a subsequent `ISpStop`). When the application calls ISpConfigure, a dialog is presented with a scrolling list of devices with one device selected. The selected device is responsible for the primary pane of the dialog, handling all events and drawing related to that area.

The GetSize function is called to determine the preferred and minimum sizes of the pane area used by the device. If the pane area will not fit on any available display device, then that device is removed from the high level list. The BeginConfiguration function is then called for each device. The configure dialog is resized and shown. The GetIcon

function is called for each device to determine the icon to be shown in the scrolling list. Then the Show function is called for the selected device. This is a good time to call `AppendDITL`.

All events returned in the dialog filter are passed to the HandleEvent function to give the device a chance to handle them. If update events are not handled (the recommended option to avoid extra flicker), InputSprocket will make the device Draw function call from within a `BeginUpdate`/`EndUpdate` pair. Unhandled mouse clicks will be passed to the Click function. If the device called `AppendDITL` to add items to the dialog and those items are returned by InputSprocket's `ModalDialog` call, then it will call the DialogItemHit function.

The device should maintain a 'dirty' variable which is set whenever any configuration information is changed, and returned and cleared when the Dirty call is made. When a different device is selected in the list, the old device will receive a Hide function call and the new device a Show function call. When the device is closed, every device will receive a EndConfiguration call.

ADB Specific Notes

Drivers should identify their devices by handler ID and original ADB address. InputSprocket does not currently support a method for arbitrating between different InputSprocket drivers for the same ADB device other than using InputSprocketDeferLib to install your service routine. If the device can change handler IDs (if it starts out looking like a mouse for example), in the CFM initialization, the driver should change it's handler ID, so later ISp drivers (like the mouse driver) do not try to read the same device. When ADBReInit is called, all devices will get two ReInit function calls, one before and one after the bus reset happens. Just before the bus reset, the device should remove any patches it installed. After the reset, the driver should search the bus to see if the device is still present. Since one driver may be responsible for multiple devices, and ReInit will be called for each device, care must be taken to make sure the remapping happens correctly.

Driver Function Reference

`kISpSelector_Init`	InputSprocket calls each device's (high level) Init function after the application calls `ISpInit`. This signifies that the high level interface is valid.
`kISpSelector_Stop`	InputSprocket calls each device's (high level) Init function after the application calls `ISpStop`. This signifies that the high level interface now invalid.
`kISpSelector_GetSize`	InputSprocket calls each device's GetSize function prior to bringing up the configure dialog to determine the visual area the device needs in the configure dialog.
`kISpSelector_HandleEvent`	While the configure dialog is up, this function is passed events from InputSprocket's dialog filter and should return true if they are handled. In many cases, a driver can always return false.
`kISpSelector_Show`	When a device is selected in the configure dialog, this function is called. The device should present it's user interface in the specified area, possibly calling `AppendDITL`.
`kISpSelector_Hide`	When a different device is selected in the configure dialog, this function will be called, notifying the device to remove it's user interface, and if necessary call `ShortenDITL`.
`kISpSelector_BeginConfiguration`	This function is called once for each device, just prior to bringing up the configure dialog.
`kISpSelector_EndConfiguration`	This function is called once for each device, just prior to closing the configure dialog.
`kISpSelector_GetIcon`	This function should return the icon family ID for the icon to be used to represent the device in the configure dialog.
`kISpSelector_GetState`	This function should return the current device specific configuration in a buffer.

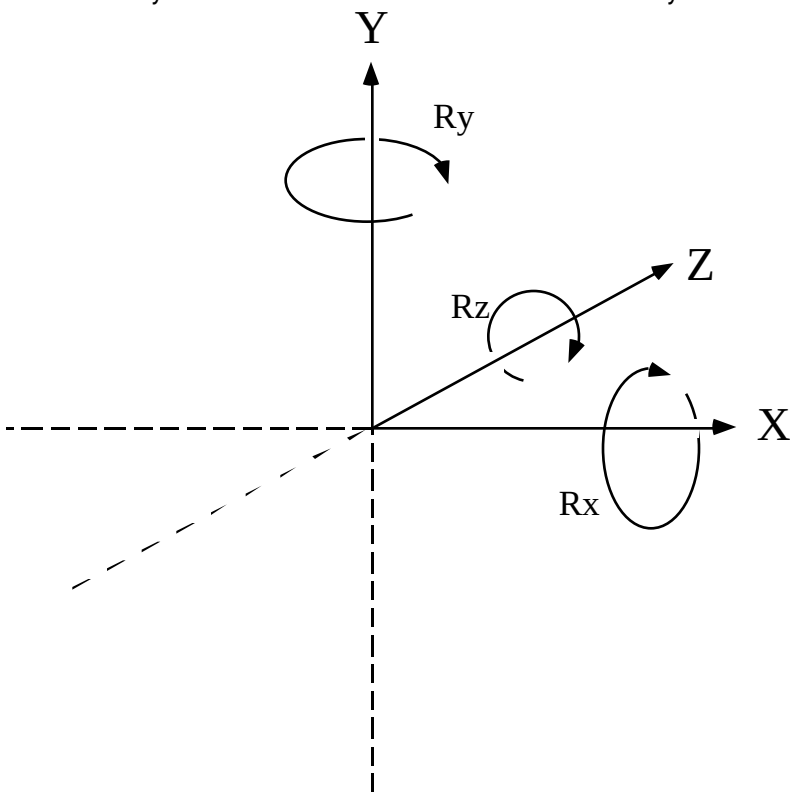
`kISpSelector_SetState`	The device should set it's configuration to that specified by the buffer.
`kISpSelector_Dirty`	The device should return if the configuration has changed and clear it's dirty variable.
`kISpSelector_SetActive`	The device should start/stop pushing data and install/remove any applicable patches to activate/deactivate.
`kISpSelector_DialogItemHit`	A dialog item (from an appended DITL) has been clicked in the configure dialog.
`kISpSelector_Tickle`	The device receives a (wait next event time) tickle every time the application calls `ISpTickle`. Available in InputSprocket 1.03 or later.
`kISpSelector_InterruptTickle`	The device receives a tickle at interrupt time ninety times a second. Available in InputSprocket 1.03 or later.
`kISpSelector_Draw`	The device should draw in the specified area of the configure dialog. Available in InputSprocket 1.03 or later.
`kISpSelector_Click`	A mouse click was detected in the configure dialog. Available in InputSprocket 1.03 or later.
`kISpSelector_ADBReInit`	The ADB bus is being reset, the device receive a call before to remove itself and after to search the bus to see if the hardware is still connected. Available in InputSprocket 1.2 or later.

Driver API Reference

`ISpDevice_New`	Creates a new InputSprocket device. Should only be called from CFM initialization of the ISp driver.
`ISpDevice_Dispose`	Disposes an InputSprocket device. Should only be called from CFM termination of the ISp driver.
`ISpElement_New`	Adds the specified element to the ISp device. Should only be called from CFM initialization of the ISp driver.
`ISpElement_Dispose`	Removes the specified element to the ISp device. Should only be called from CFM termination of the ISp driver.
`ISpElement_PushSimpleData`	Pushes UInt32 size data to the specified element (real or virtual). Can be (and usually is) called at interrupt time.
`ISpElement_PushComplexData`	Pushes arbitrary size data to the specified element (real or virtual). Can be (and usually is) called at interrupt time..
`ISpUptime`	Returns a 64 bit number for the current machine time which is machine specific. This can be converted to microseconds with `ISpTimeToMicroseconds`.

InputSprocket Coordinate System

Below is a picture of the coordinate system which is much easier to understand than any textual description.



The origin is in the bottom left corner, the x-axis is the horizontal aspect of the screen, and the y-axis is the vertical aspect of the screen. The z-axis is straight into the screen with increasing values. Rotation about the x-axis (in the plane of yz) is in the direction of the shortest arc from y to x. Rotation about the y-axis (in the plane of xz) is in the direction of the shortest arc from z to x. Rotation about the z-axis (in the plane of xy) is in the direction of the shortest arc from y to x. The purpose of these choices for direction of rotation was to keep them consistent with the 'left' and 'right' orientation of the x-axis.

Therefore, roll is the same as Rz, with left roll being at value `kISpAxisMinimum`, and right roll being at `kISpAxisMaximum` (and zero roll at `kISpAxisMiddle`); pitch is the same as Rx with pitch up being at value `kISpAxisMinimum`, zero pitch at `kISpAxisMiddle` and pitch down being at kISpAxisMaximum; and yaw is the same as Ry with left yaw being at value `kISpAxisMinimum`, zero yaw at `kISpAxisMiddle`, and right yaw being at `kISpAxisMaximum`. New label constants placed in the header file for convenience and clarity: ISpElementLabel_Axis_Pitch, ISpElementLabel_Axis_Roll and ISpElementLabel_Axis_Yaw, which are equal to the ISpElementLabel_Axis_Rz, ISpElementLabel_Axis_Rx, and ISpElementLabel_Axis_Ry constants.

In the low-level interface, Joysticks should now report to have Rz (roll) and Rx (pitch) axis instead of x and y axis. Mice

should report x and y axis, but trackballs should Rz and Rx.

In auto-configure (high-level interface), when the exact type is not one of the needs of the game, drivers like mice should map their x axis to Rz (roll) and their y axis to Rx (pitch). If the game asks for x and y axis drivers like joysticks should substitute their Rz and Rx. Rudders and yaw should be treated as equivalent. Throttle and trim axis should be mapped (in both directions) to z axis and Rx (pitch) [thus 'positive' z-axis movement or Rx rotation on a device, i.e. away from the user, translates to positive throttle]. Developers using the low level interface should be making similar mappings.

In Apple's drivers, in the configure dialog (high level) unless the option key is down when the user clicks on a popup, the driver will not show axis with labels which are 'inappropriate'. The following table describes which axis assignments are appropriate (this is **only** for axis types, delta types are similar):

Element label	'Appropriate' axis need labels
x axis	x axis, Rz (roll), Ry (yaw), rudder, *
y axis	y axis, z axis, Rx (pitch), *
z axis	z axis, y axis, Rx (pitch), *
Rx (pitch)	Rx (pitch), z axis, y axis, *
Ry (yaw)	Ry (yaw), rudder, x axis, Rz (roll), *
Rz (roll)	Rz (roll), x axis, Ry (yaw), rudder, *
rudder	rudder, Ry (yaw), x axis, Rz (roll), *
all others	any label

*gas, brake, clutch, throttle, trim, and none labels will be allowed to be assigned to any axis