Button Configuration

Version 1.1

5300 California Avenue • Irvine, CA 92617-7013 USA • Phone: (949) 926-5000 • Fax: (949) 926-5203

Table of Contents

Broadcom Button Configuration

REVISION HISTORY

Revision Number	*Date*	*Change Description*
V1.0	02/07/2014	Initial Release.
V1.1	15/09/2016	Fixed header, and clarified wording for release

1.0 INTRODUCTION

Recently, new support for buttons was added to the code. This support allows a programmer to associate multiple actions to the same button, based on how long the button was pressed etc. The new method allows a user to associate button presses both via the boardparms, or dynamically, during run time.

2.0 THE ORIGINAL STYLE

In the past, buttons were configurable via boardparms. A typical configuration would look like this:

```
{bp_usExtIntrResetToDefault, .u.us = BP_EXT_INTR_0 |
BP_EXT_INTR_TYPE_IRQ_LOW_LEVEL },
{bp_usGpio_Intr,             .u.us = BP_GPIO_0_AL},
{bp_usExtIntrSesBtnWireless, .u.us = BP_EXT_INTR_1},
{bp_usGpio_Intr,             .u.us = BP_GPIO_1_AL},
```


With this configuration, when the reset button was pushed, the signal from GPIO 1 would fall to low, and external interrupt 0 would get triggered. This in turn would cause the associated reset interrupt handler to get triggered, which would then schedule a bottom half, and cause the board to reset it's configuration to default, and reset. The restore to default action would occur as soon as the button was pressed down.

The second part to this would tie the SES Button to External Interrupt 1, and GPIO 1. Notice that in the past, the SES button was also used for initiating PLC key exchange, and 1905 configuration as well. Again, the action would occur as soon as the button was pressed.

While this configuration is convenient, it was limited. Having the restore-to-default occur as soon as the button is pressed could lead to some accidental restores. It may be preferable to require a user to push and hold the button for five seconds instead. This would help prevent some accidental resets.

One might also want to randomize the security keys if the security button was held down for more than 10 seconds. Finally, in the past, there was no way to add vendor-specific functionality to the buttons.

3.0 NEW BUTTON SUPPORT

The new button support separates the button functionality from the buttons themselves. It allows a user to register several actions against various button based on various triggers (button is held for 10 seconds, button is released before 5 seconds, etc).

3.1 Button Actions

Button actions are callbacks which are associated against a button trigger. All button actions will be invoked from a real-time kernel thread, with no locks held. If there are multiple actions registered against a single trigger, the actions will get called sequentially (they do not run concurrently). Blocking within a button action may delay other button actions from running.

3.2 Button Triggers

There will be three types of triggers:

Button Press: Button press actions will get trigger immediately when the button is first pressed. You can register multiple actions against a button press event, and each of them will trigger when the button is first pressed.

Button Hold: Button hold actions will get triggered when a button has been held down for a period of time. If you register 1s, 3s, and 5s hold actions, and then press the button for four seconds, then the 1s hold action will trigger 1s after you press the button, the 3s will trigger 3s after you press the button, and the 5s event will not trigger (as the button is not held down for 5s).

Button Release: Button release actions will get triggered when the button is released after a certain period of time. On a button release, only the actions(s) with the longest associated timeouts will be invoked (if there are multiple hooks with the same timeout then all hooks with the same timeout will be invoked). If you register 0s, 3s, 3s, 5s, then both 3s events will get triggered if you release the button after 4s), however, neither the 0s nor the 5s events will be triggered.

3.3 Registering Button Actions through BoardParms

There are a common set of actions which will commonly be registered against the button events. To make it simpler to manage the buttons, these can be registered through boardparms. Notice that registering actions in boardparms does not prevent you from dynamically registering/deregistering other actions against the same buttons going forward. Also, you must first declare your buttons in boardparms before being able to registering other buttons.

Registering buttons in boardparms requires multiple contiguous parameters. You must first declare the button, and then declare any button actions directly beneath the definition.

3.3.1 Declaring the button

You must first declare the button. The button requires an id, a gpio and an external interrupt number as so:

```
{bp_usButtonIdx,              .u.us = 0},
{  bp_usButtonExtIntr,        .u.us = BP_EXT_INTR_1 |
BP_EXT_INTR_TYPE_IRQ_LOW_LEVEL},
{  bp_usGpio_Intr,            .u.us = BP_GPIO_1_AL },
```

The first line represents the button index. This must be between 0 and PB_BUTTON_MAX (defined in pushbutton.h, Currently defined as 3). The second line represents the external interrupt number, or'd with the HIGH/LOW detection level. The last line represents the GPIO line number to which the button is attached. Again, this requires you to specify whether the button is active when high or low.

3.3.2 Declaring the actions

Once defined, you may register some actions against the button. The actions must follow immediately after the button definition, and will apply to that button.

To register an action, you must set bp_usButtonAction (which is an unsigned short). This should be set to an action enumeration, ored with a trigger type enumeration, and potentially ored with a trigger time. So, for example you could have:

```
{    bp_usButtonAction,         .u.us = BP_BTN_ACTION_PLC_UKE |
BP_BTN_TRIG_RELEASE | BP_BTN_TRIG_0S  },
```

This tells boardparms to register the button action BP_BTN_ACTION_PLC_UKE (which initiates a PLC key exchange), if the button is released after 0 seconds.

It is possible to pass parameters to actions as well (though only one action currently supports this). To do this, you can specify a bp_ulButtonActionParm immediately after you specify the action. For example:

```
{    bp_usButtonAction,        .u.us = BP_BTN_ACTION_PRINT |
BP_BTN_TRIG_PRESS },
{      bp_ulButtonActionParm,    .u.ul = (unsigned long)"Button 1
Press -- Hold for 5s to do restore to default" },
```

This will cause the PRINT action to be triggered directly when the button is pressed. The print action takes a string parameter (cast to a long), and will print that string to the screen.

Currently, the supported functions are as follow:

BP_BTN_ACTION_NONE	Does nothing. This is useful if you want to invalidate a previous release action after a period of time. So, if you registered a release action, with time 0, and then registered a NONE action with time 3, then the original release action would only get invoked if the button was held for less than three seconds.
BP_BTN_ACTION_SES	Initiates a wireless SES key exchange. If 1905 is compiled in, this will initiate key exchanges on all 1905 interfaces instead.
BP_BTN_ACTION_PLC_UKE	Initiates a PLC key exchange.
BP_BTN_ACTION_RANDOMIZE_PLC	Causes the PLC to select a new random key
BP_BTN_ACTION_RESTORE_DEFAULTS	Restores the device to factory default settings, and resets the board
BP_BTN_ACTION_RESET	Causes the board to reset
BP_BTN_ACTION_PRINT	Causes a message to be printed to the CLI. Takes a parameter, which will be a pointer to a string (cast to unsigned long)

You may also or in a time, using the BP_BTN_TRIG_1S set of macros. This should be ored in with the action and trigger type.

3.4 Registering Button Actions Dynamically

It is also possible to register push button actions which are not in the list above, and to specify them dynamically during runtime. This can be done through the interfaces defined in pushbutton.h (registerPushButtonPressNotifyHook, registerPushButtonHoldNotifyHook, registerPushButtonReleaseNotifyHook). For example, you could register a hook foo, using the following call:

```
registerPushButtonHoldNotifyHook(PB_BUTTON_0, foo, 3000, NULL);
```

This would cause function foo to get called if button 0 was held down for 3000ms. foo would get passed NULL as it's parameter. This must be called from kernel space, but otherwise, may be called from any module. The function can also be deregistered in the same fashion. The function should be of type pushButtonNotifyHook_t:

```
typedef void (* pushButtonNotifyHook_t)(unsigned long timeInMs,
unsigned long param);
```