Software User Manual

STB Reference Software

SetTop API Design

STB_RefSw-SWUM300-R

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

09/26/08

REVISION HISTORY

Revision	*Date*	*Change Description*
STB_RefSw-SWUM-300-R	09/26/08	Inital Release

Broadcom Corporation
5300 California Avenue
Irvine, CA 92617

Table of Contents

LIST OF FIGURES

LIST OF TABLES

Section 1: Introduction

INTRODUCTION

This document discusses the design goals and principles used in the development of the Settop API.

The intended audience includes:

- Users of the Settop API who want to know why it is the way it is.
- Users of the Settop API who want to make customizations.
- Developers of the Settop API who need to know how to extend it.

This does not document internal APIs or internal implementation, unless it directly affects the public API.

Where possible, actual Settop API code is used to illustrate the principles. Be aware that the Settop API may change slightly over time; all changes are reported with every release.

Section 2: Requirements

High-Level Requirements

The public API should:

- Be extremely simple to understand and use.
  - The normal case should not be complicated in order to support the exception.
  - Extensive documentation will not improve a complicated API.
- Support all active Broadcom Reference Set-Top platforms.
  - This currently includes satellite, cable, terrestrial, and IP settop.
  - Chip revisions will be supported as needed.
  - On-going development and support for older platforms and chip revisions will be dropped, but code will remain available.
- Support a wide range of operating systems including:
  - Linux
  - VxWorks
  - Micro-C OS
  - No OS mode
- Be written to an application level.
  - This means that it provides the highest level of functionality possible to facilitate the widest range of foreseen applications.
  - This is different from the lower-level Porting Interface, which must expose all hardware functionality.
- Be complete enough to expose all functionality specified by the system architecture specification for a given platform.
- Be flexible enough to expose all usage modes specified by the system architecture specification for a given platform.

Low-Level and Implementation Requirements

In addition to the high-level requirements. the public API:

- Should be binary-compatible across all platforms for a given release.
  - General-purpose utilities don't have to be recompiled for every platform.
  - Functions that have no meaning for a given platform (e.g. btuner_tune_qam on a satellite box) are still defined and always return an error.
  - Note that the API is not guaranteed to be binary compatible between releases. For a given release, when compiled for different platforms, the Settop API will be binary compatible across those platforms.
- Should maintain a clear distinction between public API's and internal and private API's.
  - Only the public API's will be documented and supported.

- Should try to be source-compatible between releases.
  - Should be set up so that most of the time, a user can update the Settop API and just recompile.

    If this cannot be done, the API design should force a compiler error so that the problem can be seen.

    The worst scenario is that a change in the API causes no compiler error but generates a run-time error. This needs to be avoided via good API design.
- Should be implemented in the C programming language for maximum portability.
  - Must compile with no warnings using a C 89 compiler.
  - May not compile at all using a C++ compilier.
- Should not require system-level code to be written in order to exercise functionality.
  - This means that your application does not need to manage interrupts, threads, file I/O, or other systems tasks in order to handle complex operations such as PVR, dynamic picture change, aspect-ratio configurations, double-buffered graphics, etc.
- May use platform-specific threading libraries. (such as pthreads) as required.
- Must allow for no-threads build if not needed (such as no PVR configuration).
- May use global variables.

Section 3: Public API Design Principles

Return Types

There are three possible return types. This makes for very straightforward error handling and no special cases.

- void–no error possible, ever
- bresult–0 is success, else error
- Any handle (for example, `btuner_t`)–not NULL is success, NULL is error

If you are running in debug mode (with `BDBG_DEBUG_BUILD`=1 defined), you should always get an error message printed to the console for errors.

In some cases, error return codes are “normal” and printing an error message would result in a very full log when everything is working well. These cases are documented. It is possible that system errors could occur in these functions, and so the user can decide to test for these non-fatal error codes and distinguish them from fatal error codes.

Setting Structures with Init Functions

When creating a function, using a single settings structure with a well-defined initialization method allows the API to be extended without breaking old code. Consider the following code:

Example 1:

```
bresult bfoo_start(bfoo_t handle, int size, bool enabled);
```

Example 2:

```
struct {
       int size;
       bool enabled;
} bfoo_settings;
bfoo_settings_init(bfoo_settings *settings);
bresult bfoo_start(bfoo_t handle, const bfoo_settings *settings);
```

Now add a new parameter to `bfoo_start` called `int weight`. In Example 1, existing applications will break and will need to add the new parameter. This change may cause a ripple effect in the application as it now has to figure out what that new data should be.

In Example 2, the `init` function could simply provide an appropriate default value, and the existing application simply needs to be recompiled. At first, there’s more overhead with a structure and `init` function, but in most cases in the Settop API, it has allowed the code to extend functionality easily.

SETTINGS VERSUS STATUS

Settings are values selected by the user and given to the Settop API. They reflect what the user wants to do, not what is actually happening. Status values are returned to the user by the Settop API and are read-only. They reflect what is actually happening.

Function return codes are a form of status, because they inform the user whether a change has been accepted or not.

It is very important that the Settop API not blend these two together; otherwise you create a “get/set” bug. If a parameter is used as a setting, it cannot also be used to return status.

A good example of this is `bdisplay_content_mode`. The actual aspect ratio depends on three things:

- The aspect ratio of the display,
- The aspect ratio of the source, and
- The user setting.

If you set `display_settings.content_mode = bdisplay_content_mode_box`, the Settop API will letter box or window box the content, only if it makes sense. Table 1shows some scenarios:

Table 1: Display and Source Aspect Ratios and Content Mode Results

Display A/R	*Source A/R*	*content_mode*	*Result*
4:3	4:3	box	Full-screen
4:3	16:9	box	Letter box
16:9	4:3	box	Window box
16:9	16:9	box	Full-screen
4:3	4:3	zoom	Full-screen
4:3	16:9	zoom	Cut off sides
16:9	4:3	zoom	Cut off top/bottom
16:9	16:9	zoom	Full-screen
4:3	4:3	full	Full-screen
4:3	16:9	full	Full-screen
16:9	4:3	full	Full-screen
16:9	16:9	full	Full-screen

It's important to note that `bdisplay_get` will not return the current aspect ratio being used; otherwise we get the following scenario:

```
bdisplay_get(display, &display_settings);
display_settings.content_mode = bdisplay_content_mode_box;
display_settings.format = bvideo_format_1080i;
bdisplay_set(display, &display_settings);

// start 16:9 source decode, which will be full screen

// user wants to change display format
bdisplay_get(display, &display_settings);
display_settings.format = bvideo_format_ntsc;
bdisplay_set(display, &display_settings);

// start 4:3 source decode. what is the aspect ratio now?
```

If `bdisplay_get` returned actual status, the user will have unknowingly set the aspect ratio to full screen because they were actually full screen at the time of `bdisplay_get`. However, they expect to keep it box, because they haven't changed the setting.

We avoid this error by requiring that settings members be settings only, not status.

Immediate Parameter Validation

Functions parameters will always be immediately validated against themselves or other settings for that handle. They will not be immediately validated against settings from other related handles.

The following scenario will fail:

```
bdisplay_get(display, &display_settings);
display_settings.format = bvideo_format_1080i;
display_settings.svideo = boutput_svideo_open(B_ID(0));
bdisplay_set(display, &display_settings);
```

We can fail it immediately because the parameters passed in are not consistent with each other. You can't drive 1080i output on an S-video connector.

The following scenario will not fail:

```
// ...skipping display config code...

boutput_spdif_get(display_settings.spdif, &spdif_settings);
spdif_settings.pcm = true;
boutput_spdif_set(display_settings.spdif, &spdif_settings);

// ...skipping tune code...

bstream_mpeg mpeg;
mpeg.audio[0].pid = 0x20;
mpeg.audio[0].format = baudio_format_aac;

stream = bstream_open(band, &mpeg);
bdecode_start(decode, stream, window);
```

If the chip does not have an AAC audio decoder, it cannot output PCM on the spdif output. It can only pass-through the AAC (and on some chips, that isn't even possible). Instead of having the Settop API fail the `bdecode_start`, the Settop API makes a decision and outputs what it can. Using this an example, here are some S/PDIF scenarios:

Table 2: S/PDIF Scenarios

SPDIF PCM setting	*Audio stream*	*Decode?*	*Audio Output*
true	AAC	yes	PCM
false	AAC	yes	AAC
true	AAC	no	AAC
false	AAC	no	AAC

This is an example of an "advisory setting," which is discussed in the next subsection.

ADVISORY SETTINGS

An advisory setting is one that may or may not be actually used. It will always depend on some thing else in the system. Here are some examples of advisory settings and their dependencies:

Table 3: Advisory Settings

Structure	*Member*	*Dependencies*
`boutput_spdif_settings`	`pcm`	Audio decoder capabilities Audio format
`bdecode_window_settings`	`deinterlacer`	Chip capabilities Source format is interlaced Output format is not interlaced If decode window is full screen If using decode window 0
`btuner_analog_params`	`btsc_mode`	If source format is analog RF If btsc encoder is present

Every advisory parameter should be documented in the header file with its behavior and dependencies. The only way to know the actual status is if it is provided in a separate status structure. Status for every advisory parameter is not provided. If you feel one is needed, you can request that it be added. You can also `printf` the code during development to assure yourself that's it's being properly handled.

OPAQUE HANDLES AND PRIVATE FUNCTIONS

All public handles are opaque, which means you cannot de-reference them. They all have a size of `sizeof(void *)` and can be NULL.

All settings are set or retrieved using other structures. Every member of these structures is public. Some may be read-only if documented as such. You will never see code similar to the following:

```
typedef struct {
       /* These are public */
       int size;
       int length;

       /* These are private */
       int internal_coeff[4];
       int dont_modify_this_please_critical_setting;
} bfoo_settings;
```

Section 4: API Particulars

BSTREAM_T

The `bstream_t` type wraps all information needed to decode, record, or play a stream. It supports both analog and digital. It is a central element of making the Settop API flexible and easy to understand.

See the Section 7: “Sequence Diagrams” on page 14 to see how `bstream_t` connects the front end and back end of the Settop.

BOBJECT_T AND B_ID()

`bobject_t` provides a way of selecting a particular resource.

If you want to get an indexed resource (such as the 1st decoder, the 2nd display), use the `B_ID` macro and specify the number.

If you want to get a particular resource, use the `bobject_t` handle. The only way to currently select a `bobject_t` handle is from the `bconfig` interface, and this in turn is based on an index. The following two examples do the same thing:

```
brecord_open(bconfig->sources.record.objects[0]);

brecord_open(B_ID(0));
```

The reason for having a non-index-based `bobject_t` system is that we may extend the Settop API to support query-based `bobject_t` retrieval. The following example does not now exist, but might someday:

```
config_query_settings.record.high_through_put = true;
config_query_settings.special_encryption_mode_required = false;

bobject_t robj = bconfig_query_record(&config_query_settings);
brecord_open(robj);
```

CLONED WINDOWS

A “window” is defined to be the presentation of video on a display. A “display” is defined to be one compositor/VEC pair. All BVN-based chips (the BCM7038, BCM7401, etc.) allow for a single video decode to be present on more than one display. Each display requires a separate window to specify the location, size and z-order of that video on the display.

The Settop API accomplishes this through cloning. A cloned window is a additional window which has the same content as the master window from which is was cloned. Only the master window is used for decode. Both the master and the cloned window are available for specifying the location, size, and z-order of the window on that display.

The `B_ID`’s for master and cloned windows is an absolute number which specifies a particular window on a particular display. It is relative to the capabilities of the system. There are two general categories:

For dual-decode systems (like the BCM7038 and BCM7400):

Table 4: B_ID for Dual-Decode Systems

Window	*B_ID*
Main on display()	0
PIP on display0 or Main on display 1	1
Cloned main on display 1	2
Cloned PIP on display 1	3

For single decode systems (like the BCM7401):

Table 5: B_ID for Single-Decode Systems

Window	*B_ID*
Main on display0	0
Cloned main on display1	1

Read the comments for `bdecode_window_clone` (in BSEAV/api/include/bsettop_display.h) for details. Also read the example code in BSEAV/api/examples/clone.c.

Brutus uses cloning to implement the HDSD_SINGLE and HDSD_SIMUL modes. See the Brutus Usage Guide (document number STB_Brutus-SWUM20x) for more information.

BDECODE_WINDOW_SETTINGS.CLIPRECT

The coordinates of `bdecode_window_settings.cliprect` are not in source- or display-relative coordinates; they are window-relative. At first this seems counter intuitive, but there's a simple reason: source size changes dynamically.

Assume you set `cliprect` = {0,0,960,540} and you want the upper left corner of a 1080i source, but then the source dynamically changes to 480i. Now what do you get? The whole thing, or the upper right? If `cliprect` is in source-relative coordinates, you get the entire thing. So cliprect was made in window-relative coordinates to address this.

Please see the `bdecode_window_settings` header documentation. Also see the BSEAV/api/examples/window_clipping.c.

In addition, Figure 1 on page 11 is an illustration. Notice that the size of the `cliprect` is not equal to the size of the portion of the source that is clipped.

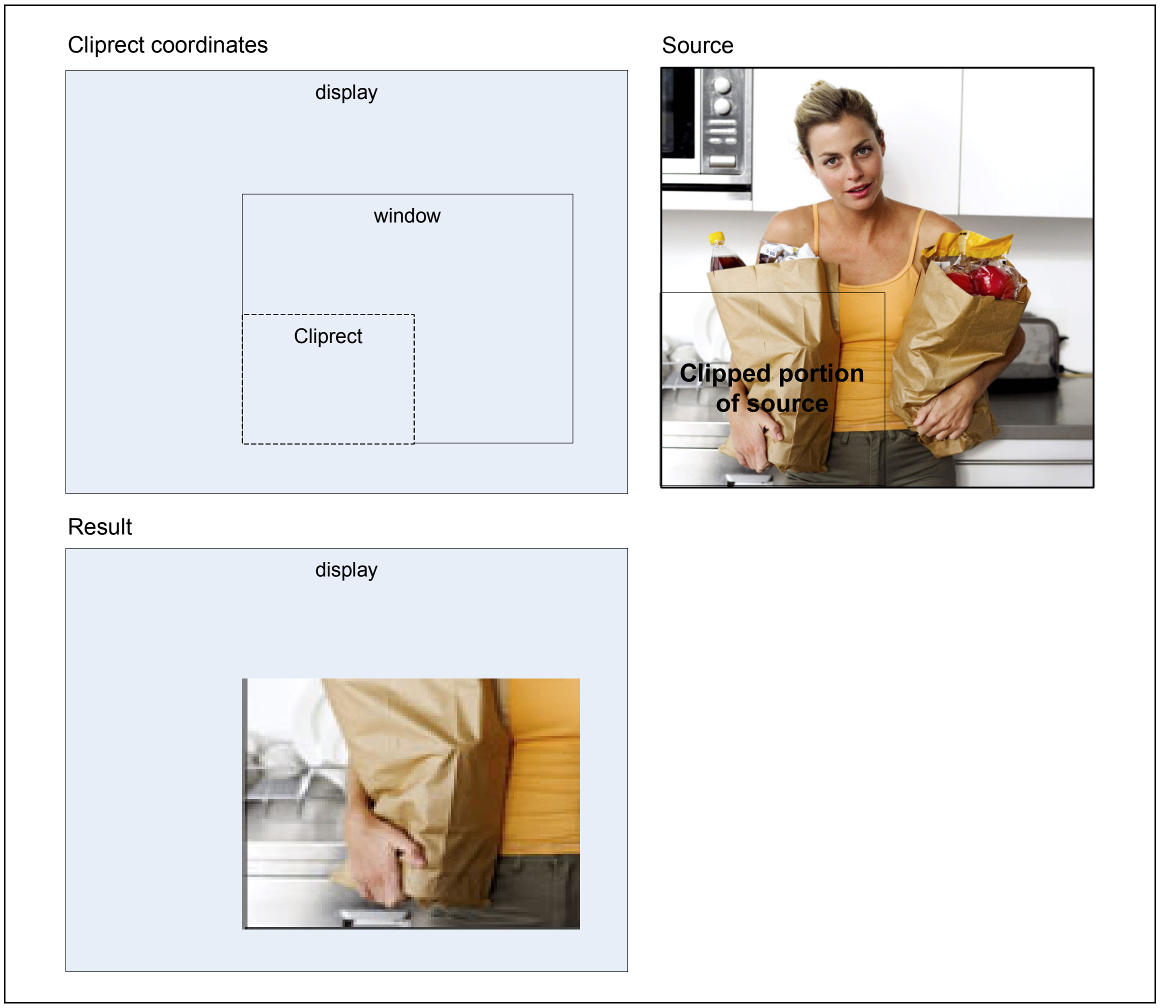

Figure 1: Illustration of Cliprec Function

Section 5: Internals

AVOIDING VDC "GET" FUNCTIONS

In the Settop API, we never use VDC "Get" functions. VDC has two software state machines. One reflects the current user state and the other reflects the actual hardware state. When the user calls a VDC Set function, it changes the current user state. `BVDC_ApplyChanges` applies the current user state to hardware and updates the actual hardware state machine. This happens at interrupt time (using either source or display interrupts).

The following sequence will assert:

```
data = 1;
BVDC_SetSomething(vdc, data);
BVDC_GetSomething(vdc, &data);
BDBG_ASSERT(data == 1);
```

This sequence may work:

```
data = 1;
BVDC_SetSomething(vdc, data);
BVDC_ApplyChanges(vdc);
BVDC_GetSomething(vdc, &data);
BDBG_ASSERT(data == 1);
```

It is possible it may not because there may have been a modification to the state in the process of applying. This means it would vary.

The design issue for Settop API is that we cannot use VDC to provide our state storage in order to have consistent get/set logic in the code. One function may set something, and then call another function, and that function may need to test what was just set.

CALLBACK CONTEXT

All callbacks from the Settop API should be in task context and without `b_lock` held. They should not be in `_isr` context. They should release `b_lock` before firing the callback.

This allows some amount of work to be done in callbacks, including limited `get_status` calls back into the Settop API.

Section 6: Data Flow

INTRODUCTION

The purpose of this section is to describe how data flow works in some portions of the Settop API. This includes the expected calling sequence and the synchronization required.

This applies to the following sections:

- bplayback
- brecord
- bplaypump
- brecpump
- bpcm_play
- buser_input

BPLAYBACK AND BRECORD ADVANTAGES

The bplayback and brecord APIs are high level PVR APIs. They handle file I/O and threading internally. Their advantages include:

- Use of Direct I/O if the Linux kernel supports it (using `O_DIRECT`); this allows the kernel to read from the disk directly into the playback buffer, avoiding any unnecessary CPU-based memory copies in the kernel's page cache. The requirement for this is that all disk reads must be page-aligned on disk and in memory.
- A two-thread file I/O mechanism; this guarantees that when one file I/O request is completed by the kernel, the next I/O request will already be pending. Otherwise there is unneeded application delay.
- A priority scheme to coordinate record and playback; in general, record always has higher priority than playback. More full record buffers are higher priority than less full buffers. More empty playback buffers are higher priority than less empty buffers.
- No extra threads are created for managing the lower level bplaypump and brecpump interfaces; all work is done at interrupt time or using callbacks from the file I/O threads.
- Built in time-shifting support; in time-shifting mode, a bplayback resource is linked to a brecord resource. If data at the end-of-file cannot be read, the bplayback resource will wait on the brecord resource to notify it that data is available. This allows for playback to be very close to the currently recorded data without any loss of efficiency.
- Simple rate-based trick modes that are offered (+1, +2, +3, etc.), along with specific trick-mode types.

Section 7: Sequence Diagrams

BPLAYBACK AND BRECORD DATAFLOW

All threading and synchronization is handled internally. This means you should call bplayback and brecord from the main thread of your application, synchronously with all other Settop API functions.

The `bplayback_params` and `brecord_params` structures do have several callbacks. However, these callbacks are for notification only. They will be called from an internal context, which might be a different thread, and no callback into the Settop API should be attempted from these callbacks.

BPLAYBACK CALLING SEQUENCE

Once you call bplayback_start, an internal state machine starts processing data through a series of callbacks from bplaypump and bsettop_fileio. The process can be interrupted at any time with calls to bplayback_trickmode, bplayback_pause, bplayback_stop and other calls.

Figure 2 shows bplayback's interaction with the lower-level bplaypump API and the file I/O module. Note that the file I/O module is shared with brecord in order to prioritize all reads and writes to and from the disk.

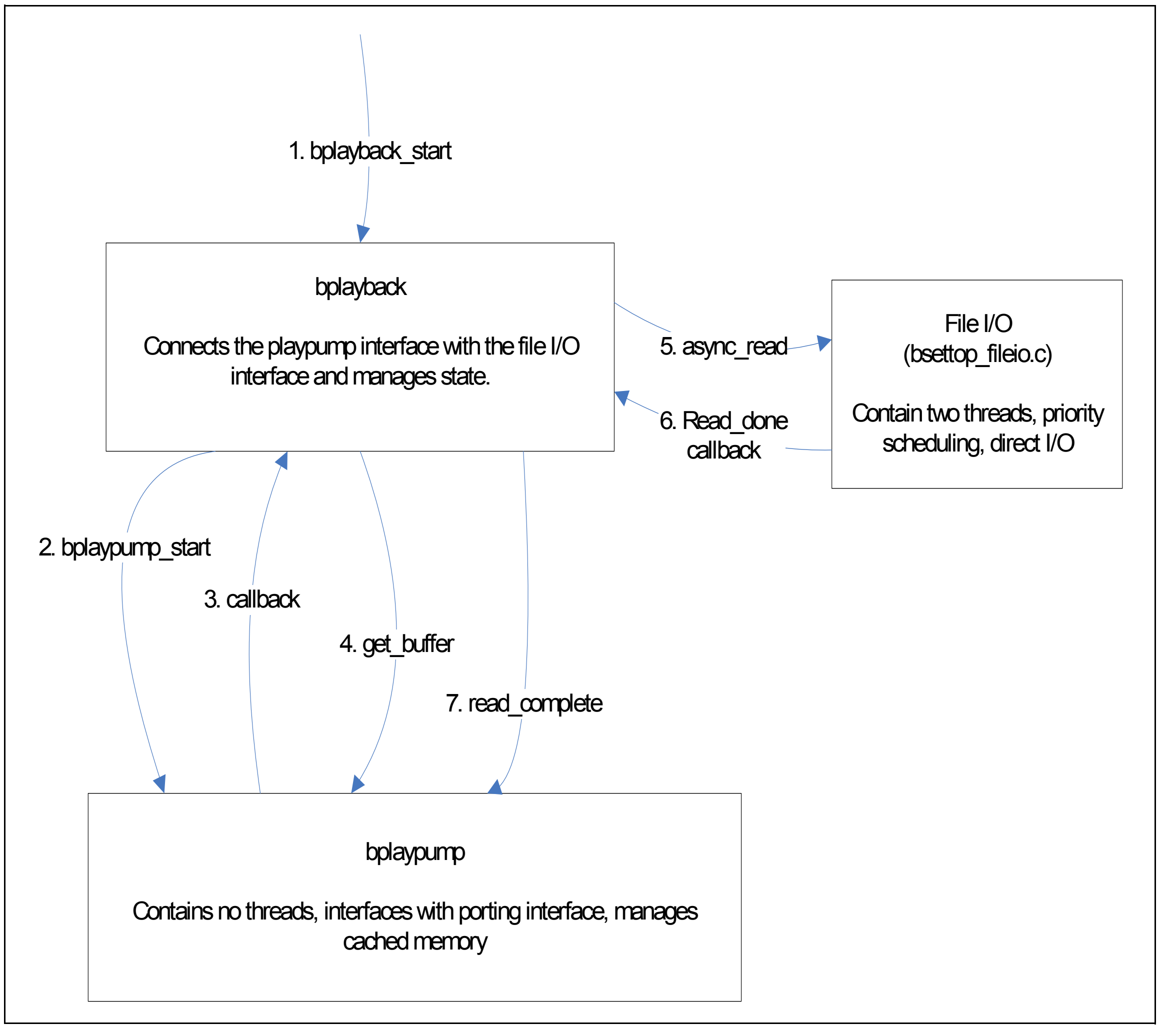
1. bplayback_start
bplayback
Connects the playpump interface with the file I/O interface and manages state.
5. async_read
6. Read_done callback
File I/O (bsettop_fileio.c)
Contain two threads, priority scheduling, direct I/O
2. bplaypump_start
3. callback
4. get_buffer
7. read_complete
bplaypump
Contains no threads, interfaces with porting interface, manages cached memory

Figure 2: bplayback Calling Sequence

BRECORD CALLING SEQUENCE

The brecord state machine handles the interface with brecpump and bsettop_fileio. After starting, the only way to interrupt the flow of brecord is to call brecord_stop.

Figure 3 shows brecord's interaction with the lower-level brecpump API and the file I/O module. The file I/O module is shared with bplayback in order to prioritize all reads and writes to and from the disk.

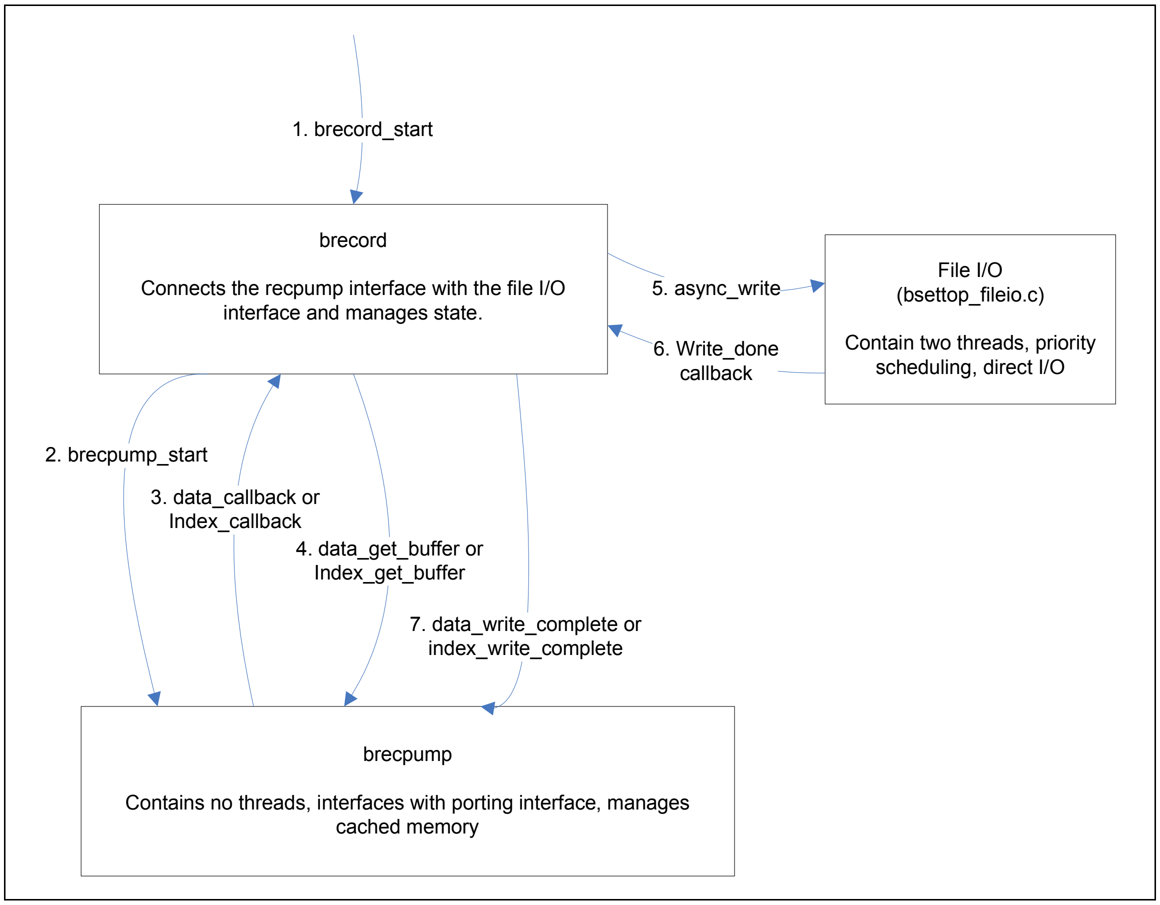
1. brecord_start
brecord
Connects the recpump interface with the file I/O interface and manages state.
5. async_write
6. Write_done callback
File I/O (bsettop_fileio.c)
Contain two threads, priority scheduling, direct I/O
2. brecpump_start
3. data_callback or Index_callback
4. data_get_buffer or Index_get_buffer
7. data_write_complete or index_write_complete
brecpump
Contains no threads, interfaces with porting interface, manages cached memory

Figure 3: brecord Calling Sequence

BPLAYPUMP STATE MACHINE

The playpump module interfaces with the lower-level transport (XPT) porting interface by requesting data and waiting for interrupts. It interacts with the higher level application (for instance, bplayback) by sending callbacks and waiting for read_complete notifications, as shown in Figure 4.

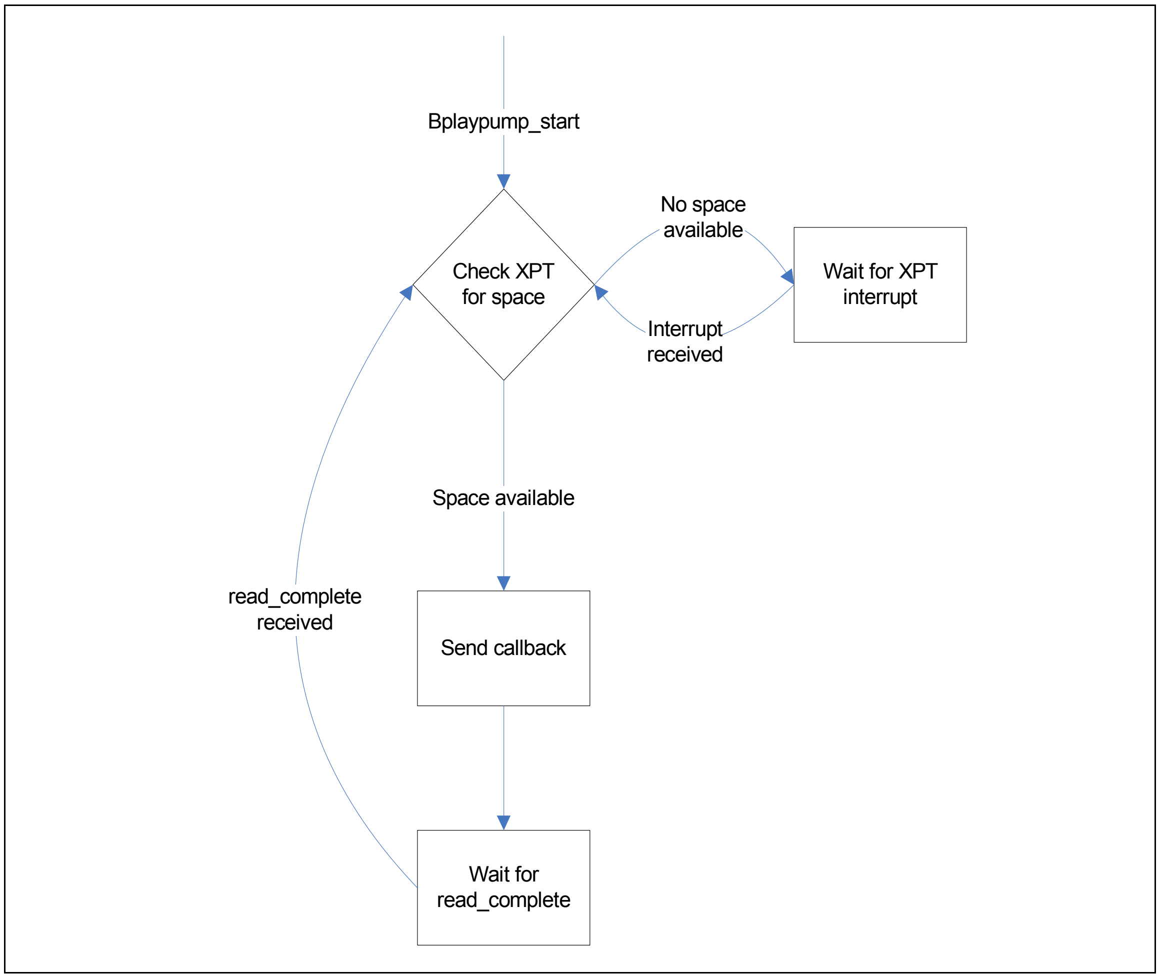

Figure 4: bplaypump State Machine

BRECPUMP STATE MACHINE

The recpump module interfaces with the lower-level transport (XPT) porting interface by requesting space and waiting for interrupts. It interacts with the higher level application (for instance, brecord) by sending callbacks and waiting for write_complete notifications, as shown in Figure 5.

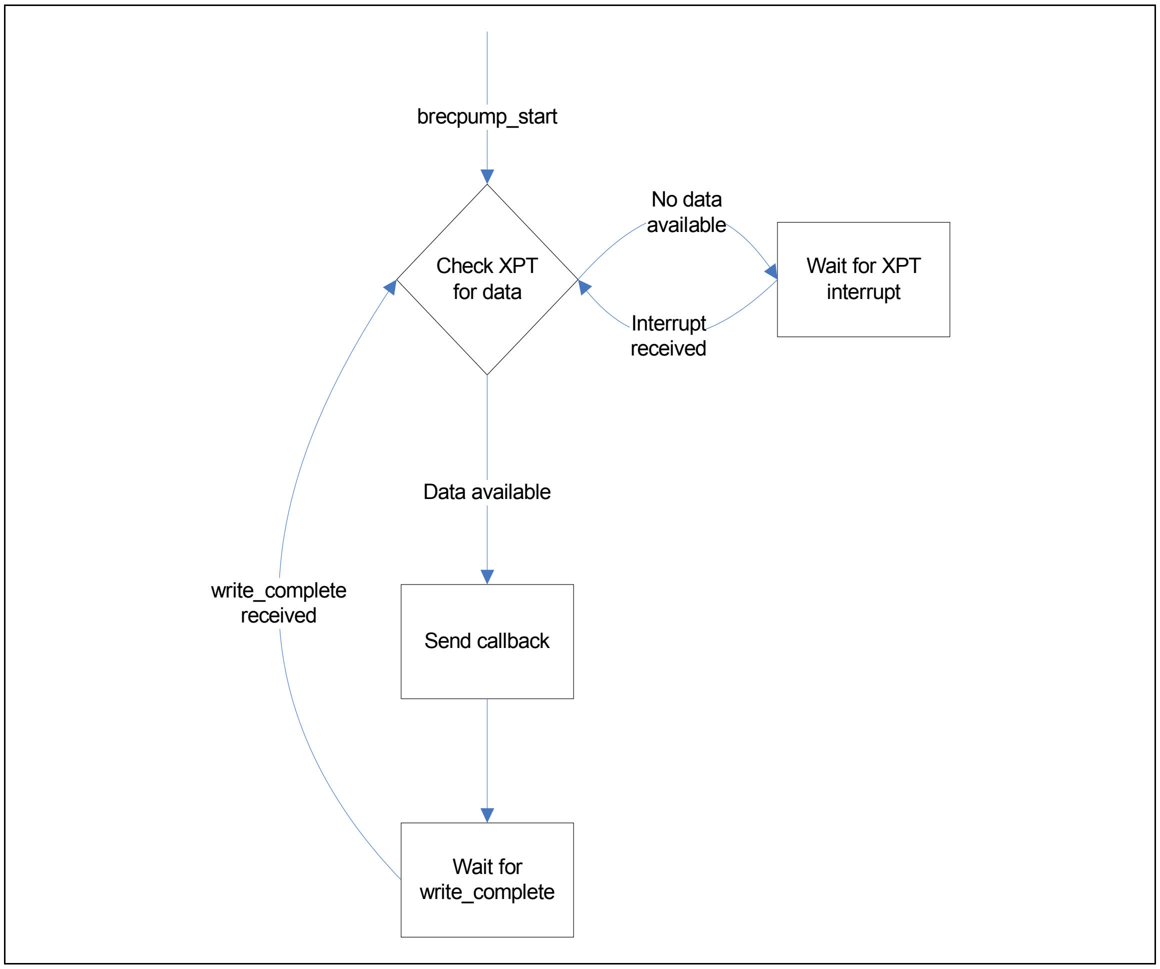

Figure 5: brecpump[State Machine

BPCM_PLAY STATE MACHINE

The bpcm_play module functions exactly like the bplaypump module. The only major difference is that there are no PCM playback trick modes to take into consideration, so the implementation is simpler. This is shown in Figure 6.

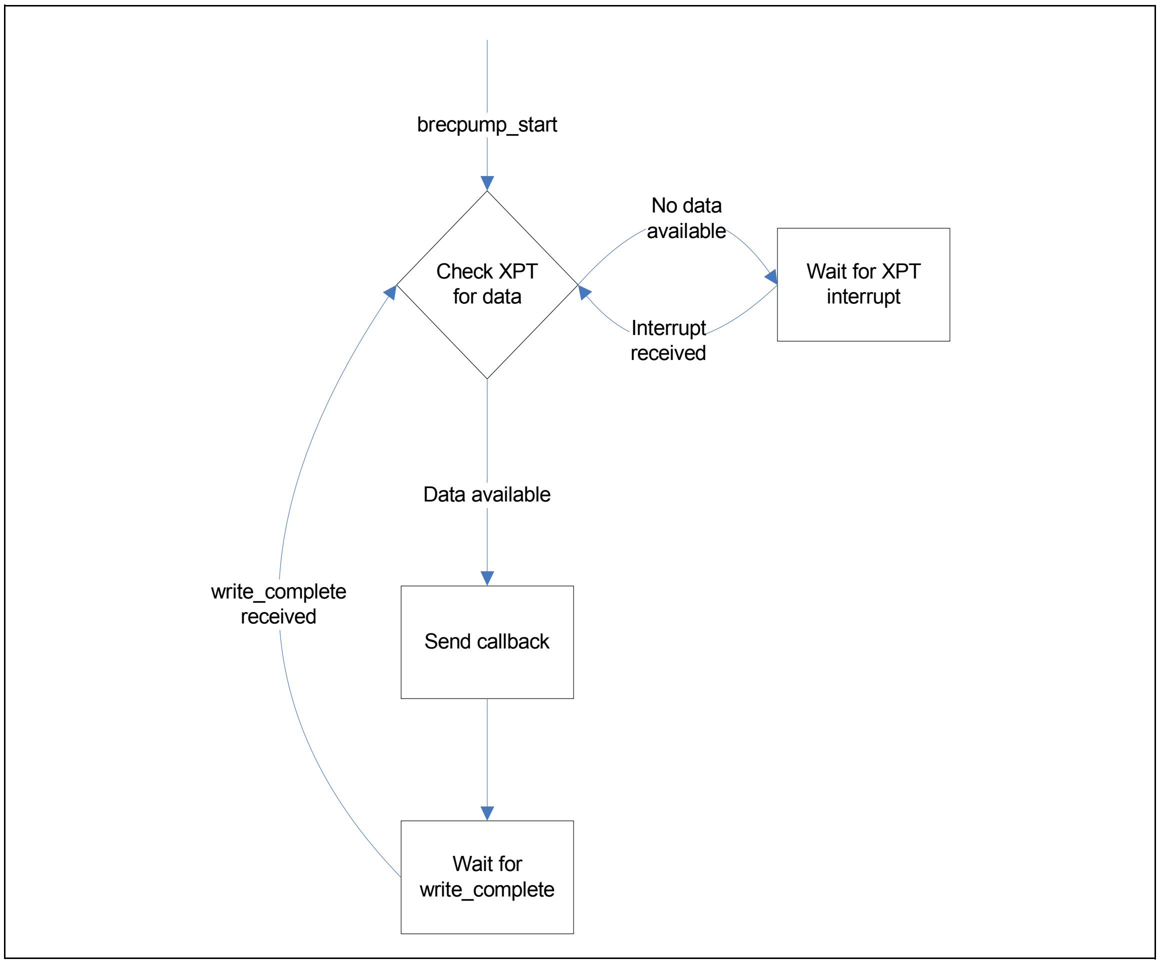

Figure 6: bpcm_play State Machine

BUSER_INPUT CALLING SEQUENCE

The user_input module uses interrupts from lower-level software and calls back to higher level software, as shown in Figure 7. The buser_input_get_event function can be called at any time, but it is usually called in response to a data_ready_callback.

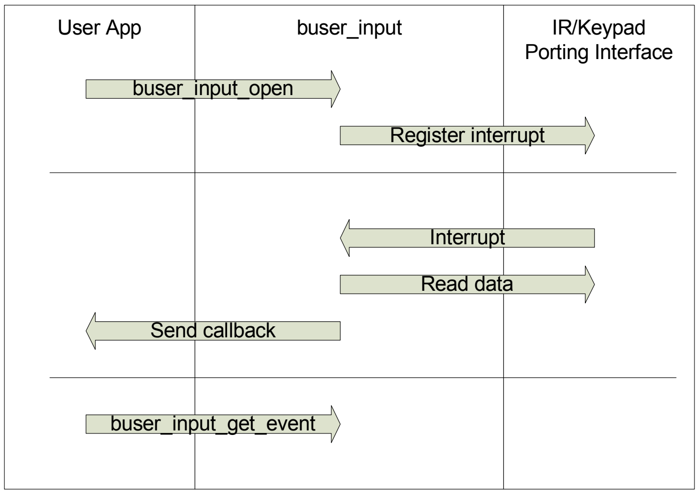

Figure 7: buser_Input Calling Sequence

Broadcom Corporation

5300 California Avenue
Irvine, CA 92617
Phone: 949-926-5000
Fax: 949-926-5203