



DirectFB 1.4.17

Software Users Guide v1.3

Broadcom Corporation Proprietary and Confidential

Broadcom Corporation
5300 California Avenue
Irvine, California, USA 92677
Phone: 949-926-5000
Fax: 949-926-5203

Web: www.broadcom.com

Revision History

Revision	Date	Change Description
1.0	May 14, 2012	Updated from 1.4.15 document
1.1	May 23, 2012	1.4.17 changes
1.2	June 22, 2012	Updated template and minor fixes
1.3	October 18, 2012	Updated multi app & XS guidance

Table of Contents

References	3
Introduction	4
Overview	4
Audience	4
Prerequisites	4
Main Changes From DirectFB-1.4.15 Phase 2.1.....	5
Deliverables.....	7
Installation	8
Introduction	8
Building	9
Step 1: Host machine tools check.....	9
Step 2: Environment variables	9
Step 3: Driver build check	10
Step 4A: Building DirectFB in single-application mode	11
Step 4B: Building DirectFB in multi-application mode	12
Step 4C: Building DirectFB-XS	13
Building DirectFB tests	14
Building DirectFB examples	14
Building ++DFB	14
Building Insignia test harness	14
Building Tacho test harness	15
Building external applications.....	15
Additional make targets.....	16
Additional make flags.....	18
Running DirectFB on the target platform	24
Standard DirectFB single application mode.....	24
DirectFB multi-application mode.....	25
DirectFB-XS (Nexus Surface Compositor)	26
Running texture mapped graphics applications	27
Running OpenGL ES 1.0 graphics applications.....	27
Running OpenGL ES 2.0 graphics applications.....	28
Running SaWMan (multi-application mode)	29
Running with kernel-space (proxy mode) drivers.....	29
Running with user-mode drivers	29
Running DirectFB examples	30
Running ++DFB.....	30
Running audio/video tests	30
playback_dfb	30
decode_server_dfb	31
decode_client_dfb	31
Run-time environment variables	32
Additional information.....	33
Build system information.....	33
Multi-application support with DirectFB	34
Running non-DirectFB and DirectFB applications.....	35

Multi-application support with DirectFB-XS	39
DirectFB memory management.....	41
Changes to DirectFB-1.4.17	42
Platform library usage for standard DirectFB	43
Platform library usage for DirectFB-XS	43
Graphics driver	44
IR and front panel drivers	44
DirectFB Nexus input router	46
System driver	47
ImageProvider driver	48
Core changes.....	49
New pixel formats.....	49
3DTV stereoscopic support.....	49
Colour-space support.....	49
Screen changes	49
DirectFB unit tests.....	49
Graphics changes	51
Image provide changes	51
Font changes	51
Build system	51
Input devices	52
Other	52
Public API changes to DirectFB-1.4.17	53
Testing DirectFB	56
Testing the IR input.....	56
Testing the front panel input	56
Testing different blitting and drawing modes	57
Performance tests.....	58
Supported platforms.....	59
Frequently asked questions (FAQ).....	60
How do I enable debugging on a per-module basis in DirectFB?.....	60
How do I enable back-tracing in DirectFB?.....	60
How can I disable hardware acceleration and use the generic DirectFB software graphics functions instead?.....	61
How can I tell what size surfaces are being created?	61
Why can't I see memory for my surface being allocated on creation?	61
Blending multiple windows together doesn't look right - why?	61
How do I change the cursor in DirectFB?	61

List of Tables

Table 1: Software deliverables.....	7
Table 2: Documentation deliverables.....	7
Table 3: Make targets	16
Table 4: Make flags	18
Table 5: Run-time environment variables	32
Table 6: Public Function API changes	53
Table 7: Public definition API changes.....	53
Table 8: Supported platforms	59

References

Reference	Description	Version/ Date
1	DirectFB-1.4.17_v1.5_Feature_List.pdf	A16
2	BroadcomReferencePlatformSetup.pdf	STB_Platform_SWUM101-R
3	BrutusIntallationGuide.pdf	STB_Brutus_SWUM202-R
4	Nexus Usage Guide	STB_Nexus-SWUM204-R
5	Nexus Architecture Guide	STB_Nexus-SWUM104-R
6	Nexus Development Guide	STB_Nexus-SWUM302-R
7	Nexus_MultiProcess.pdf	
8	http://www.directfb.org	N / A

Introduction

Overview

DirectFB stands for Direct Frame Buffer. "DirectFB is a thin library that provides hardware graphics acceleration, input device handling and abstraction, integrated windowing system with support for translucent windows and multiple display layers, on top of not only the Linux Frame buffer Device.

It is a complete hardware abstraction layer with software fallbacks for every graphics operation that is not supported by the underlying hardware. DirectFB adds graphical power to embedded systems and sets a new standard for graphics under Linux." (See www.directfb.org for more details).

This document describes how to build, install and run DirectFB 1.4.17 on a Broadcom set-top box reference platform.

Audience

This document is aimed for individuals who have an engineering background and already know how to build the standard Broadcom reference software for a set-top platform. This document assumes the user is familiar with a standard Unix environment and build tools such as "make" and "gcc".

Prerequisites

You must have the following before building and running DirectFB on a reference platform:

- A host PC or build server upon which to install the DirectFB source code and build it. It must have the Broadcom MIPS cross-compilation tool chain installed.
- A DHCP server running on the same network as the reference platform.
- Knowledge of the "vi" UNIX editor to be able to edit text on the reference platform.
- Bash shell to execute the installation and build steps on the host / build server.
- It may be desirable to have a USB mouse and keyboard to use with the reference platform.

Main Changes From DirectFB-1.4.15 Phase 2.1

This release officially only supports a subset of Broadcom STB chipsets, namely the 7231, 7241, 7346, 7358, 7409, 7420, 7425, 7429, 7435 and 7552. Other chipsets or platforms may work without problems, but these will not have been rigorously tested, a warning message will be issued informing the user that the platform may not be useable.

DirectFB-1.4.17 supports compositing to external frame buffers with Nexus surface compositor (NSC). This is also known as “DirectFB-XS”, a terminology used by Broadcom to refer to “eXternal Surface” composition. DirectFB and DirectFB-XS can be built to run with the Nexus drivers either in the kernel or in user-space.

Internal to the Broadcom DirectFB release there has been a radical change in the layout of the code. In previous releases the Broadcom specific code was overlaid on top of the vanilla DirectFB release and the hybrid tree was then compiled and installed. With the new system all of the Broadcom drivers are built in a separate source tree. So now when compiling the source code you will see the generic DirectFB code compiled first, then the DirectFB-Broadcom package and finally any extra packages such as the example applications or SaWMan. The aim of this new system is to eventually completely decouple the version of DirectFB from the Broadcom drivers, allowing customers to select their own version of DirectFB. However this will take some time as the current generation of public DirectFB releases do not contain all of the APIs required by Broadcom customers, so we will continue to patch the DirectFB releases until all of our changes have been rolled back into the open source project.

In order to help separate the Broadcom drivers from the vanilla DirectFB code, two new features have been implemented internally. It is now possible to register a default version of an interface which is tried first when a request for that type of interface is made. This allows us to register the Nexus Still Image Decoder image provider implementation as the default interface for decoding images by the usual range of option passing methods such as the `directfbrc` file. A further change is to allow external libraries such as the Broadcom system driver to parse command line, `directfbrc` and `DFBARGS` options passed into the application, so we can keep Broadcom specific options e.g. IR remote control protocol inside the Broadcom driver.

Refactoring the build system has given us a chance to improve the robustness of the system and improve the stability when switching between build types. There has also been an effort to try and produce more useful error messages when invalid configurations are detected.

The DirectFB signal handling code has been significantly reworked to help with the shutdown of applications via signals or CTRL-C. This should mean that applications forced to quit under unexpected circumstances will close down in a more orderly fashion and not hold onto resources allowing other applications to launch successfully.

1.4.17 is the first version to support capturing input events from the Nexus Input Router (NIR) module. The Nexus Input Router is designed to work alongside the Nexus Surface Compositor and provide a mechanism to pass input events such as keyboard, IR, keypad and mouse events to a series of registered clients. The server API of NIR allows you to filter events and control which client receives what type of event.

This release supports the “secure-fusion” multi-application architecture. As with the last release, this release only supports running applications with “secure-fusion” enabled. This is to provide maximum security for multi-application environments. Bug-fixes to secure-fusion have also been incorporated into this release. For example, it is now possible to set the shape of the cursor “SetCursorShape()” for a window using the create palletised surfaces API in the context of a DirectFB slave application without receiving a segmentation fault. It is also possible to use pre-allocated system memory to create a DirectFB surface and update the contents of it during the execution of a DFB slave application.

This release contains fixes for all of the image providers including the Nexus Still Image Decoder implementation to attempt to detect broken images and stop decoding as soon as an error is detected, preventing a range of undetected decode errors and segmentation faults.

Improvements have been made to the DirectFB Screen APIs available for the secondary display. It is now possible to use the Encoder API to control the secondary screen in the same manner as the primary screen. It is also possible to disconnect display outputs from one screen and connect to the other. This may be useful to customers who might be interested in using the component output on platforms to provide an RGB SCART signal on the secondary display. See the new test tool `df_screen_encoder` for example usage.

Internal testing has shown issues with small sized pixel formats such as LUT4 under a range of circumstances inside the generic software graphics library. These have now been fixed and a number of improvements have been pushed into the mainline DirectFB code.

The ancillary libraries like zlib, jpeg, png, ffmpeg and freetype have been upgraded to newer more secure versions. In addition, utilities and tests making use of the PNG library have been refactored to ensure that they correctly use the public API and not the deprecated private accessor functions.

Many of the unit test applications have been updated to work in an automated build environment. New tests have also been added to exercise different areas of the core DFB code (e.g. pre-allocation of system and video memory).

Deliverables

This section describes what is contained in the release including software and documents.

Table 1: Software deliverables

Description	Version/file	Licensing
DirectFB 1.4.17 v 1.5 reference software	v 1.5 / 20121018	Broadcom SLA

Table 2: Documentation deliverables

Description	Version/date
DirectFB 1.4.17 Software Users Guide (this document)	1.3
DirectFB-1.4.17_v1.5_Feature_List.pdf	A16

Installation

Introduction

The DirectFB-1.4.17 release contains both the open source software and Broadcom SLA specific driver code.

All of the code inside the DirectFB-Broadcom directory, typically the DirectFB graphics, system, input and image provider drivers come under the Broadcom SLA and are built as shared libraries. Refer to section 3.2 for information on installing from a full release.

The standard reference software (Nexus/Magnum) source code should be available (untared) prior to installation of the DirectFB-1.4.17 Version 1.5 reference software. If you are unsure of how to do this, refer to the “Brutus Installation Guide” that comes as part of the reference software release.

On the host (build) machine, navigate to the root of the reference software source tree and type:

```
tar xzvf DirectFB-1.4.17_v1.5_20121018.tgz
```

This will overwrite any existing “AppLibs/opensource/directfb” dir (and subdirs).

You will now be ready to build the DirectFB source code from the “AppLibs/opensource/directfb/build” directory.

Building

This section explains how to build standard DirectFB-1.4.17 in single, multi-application and DirectFB-XS modes.

Step 1: Host machine tools check

Before commencing the build, ensure that the version of GNU make is 3.80 or higher on your host build machine. You can test what version of make you are using by issuing the following command:

```
make -version
```

An earlier version of this will result in DirectFB not being built and you will need to upgrade your make package on the host build machine.

Step 2: Environment variables

Make sure you have the reference software environment variables setup correctly. The important ones are the following:

NEXUS_PLATFORM, BCHP_VER, LINUX

Example:

```
export NEXUS_PLATFORM=97425
export BCHP_VER=B2
export LINUX=/opt/brcm/linux-2.6.37-2.8
```

By default the Nexus and magnum drivers will be built in user-space, however you can override this behaviour by setting the “NEXUS_MODE” envvar to “proxy” and set “KERNELMODE=y”. This will ensure the drivers are built in kernel-space with a “proxy” shim layer to translate the API calls to Linux syscalls (ioctl’s) and back again.

Example:

```
export NEXUS_MODE=proxy
export KERNELMODE=y
```



You can speed up the build process on multi-processor machines by ensuring that either MULTI_BUILD=y or MAKE_OPTIONS=-j? is set where ? specifies how many make jobs can be run in parallel (e.g. make MAKE_OPTIONS=-j4).

By default, DirectFB and the drivers will be built in DEBUG mode. This can have performance penalties and it is strongly recommended that the user switch to a non-DEBUG mode after monitoring the drivers and DirectFB for any warnings or errors. To switch to a non-DEBUG mode (a.k.a. RELEASE mode); ensure that the environment variable “B_REFSW_DEBUG” is set to “n”.

Example:

```
export B_REFSW_DEBUG=n
```

The default is to build DirectFB and the drivers in little-endian mode. If the user wishes to run the platform and DirectFB/drivers in BIG endian mode, then the “B_REFSW_ARCH” environment variable needs to be specified as “mips-linux”. Little endian mode is the preferred option.

Example:

```
export B_REFSW_ARCH=mips-linux
```

Finally, make sure your PATH environment variable is setup correctly to point to your MIPS cross-compilation tool chain.

Example:

```
export PATH=/opt/toolchains/stbgcc-4.5.3-1.3/bin/:$PATH
```

Step 3: Driver build check

The DirectFB build system will build the Nexus/Magnum drivers for you automatically honouring the environment variable settings setup in Step 1 above. For special cases you can manually do it this way:

```
make -C nexus/build
```




You can speed up the build process on multi-processor machines by specifying the “-j” option to make (e.g. make -j4)

Step 4A: Building DirectFB in single-application mode

Standard DirectFB can be built in different modes of operation known as “single-application” and “multi-application”. Single-application mode is typically used in situations where there is only a single application accessing the DirectFB API’s. A single application is typically a single process with or without multiple threads. If more than one application or process is required to access DirectFB concurrently, then DirectFB will need to be built in multi-application mode, skip to step 4B.

To build DirectFB in single application mode you need to follow these steps:

```
cd AppLibs/opensource/directfb/build
make default tarball
```

 This will build DirectFB along with the zlib, libpng, libjpeg and freetype libraries from the “AppLibs/opensource” directory.

The build process will first check to see whether your host build tools are at least at the correct minimum version before proceeding. It will then build and install the Nexus/Magnum drivers with the correct configuration based on the environment variables set in step 2.

The build process will then check to see whether the DirectFB source tree already exists in “AppLibs/opensource/directfb/src/DirectFB-1.4.17”. If not, then the generic DirectFB-1.4.17.tar.gz tarball from the “AppLibs/opensource/directfb/src/directfb_tarballs” directory will be untared to create the “AppLibs/opensource/directfb/src/DirectFB-1.4.17” directory. The next step will then be to copy the contents of the “AppLibs/opensource/directfb/src/broadcom_files/public/DirectFB/1.4.17” directory on top of the newly created DirectFB-1.4.17 source tree. This step is necessary, as the standard DirectFB tarball doesn’t contain any of the Broadcom specific changes. This public directory contains only open-source components, which have been released to DirectFB.org or are back ports from newer versions.

Before the DirectFB source code can be built, the freetype, jpeg, zlib and png libraries need to be built. The DirectFB source code will then be configured to auto-generate the Makefiles and finally the DirectFB source code will be built and installed.

After the vanilla DirectFB code is built, the Broadcom specific DirectFB drivers are configured automatically for the platform chosen based on build time options and parsing of the Magnum/Nexus driver code. The Broadcom specific DirectFB drivers, for things that use the Broadcom Nexus drivers such as the Still Image decoder (SID), M2MC, and platform layer are now built from within the folder “AppLibs/opensource/directfb/src/DirectFB-Broadcom”.

The final build stage will produce a tarball that can then be copied to the target platform for extracting and running. This tarball will be called something similar to the example below:

e.g. DirectFB-1.4.17_debug_build.97425B2.tgz

 In non-DEBUG (RELEASE) mode, the word “debug” will be replaced with “release”.

Step 4B: Building DirectFB in multi-application mode


Multi-application mode will allow multiple applications/processes to use the DirectFB API concurrently. The build process is the same as for Step 4A above, except that the additional make build option called “DIRECTFB_MULTI=y” needs to be set:

Example:

```
cd AppLibs/opensource/directfb/build
make DIRECTFB_MULTI=y default tarball
```

Multi-application mode is supported for both kernel-space (proxy mode) and user-space Nexus drivers.

The steps in the build process are slightly different, in that the fusion IPC kernel module (linux-fusion) will be built prior to freetype (and the other libraries) being built. The last stage of the build process is also different in that the SaWMan window manager will be built. SaWMan allows finer control over the placement and lifecycle of multiple applications’ windows on the screen. It replaces the default window manager that comes with DirectFB.

 By default when DirectFB is built for multi-application mode, the SaWMan window manager will also be built and will override the “default” window manager. If the user would rather use the “default” window manager instead of “SaWMan” for multi-application mode, then the following make options can be specified:

```
DIRECTFB_MULTI=y BUILD_SAWMAN=n
```

The resulting tarball will be named slightly differently to the one produced in step 4A above. The word “multi” will appear immediately after the “DirectFB-1.4.17”

e.g. DirectFB-1.4.17_multi_debug_build.97425B2.tgz

This tarball can be copied to the target platform in the same way as for step 4A above.

Step 4C: Building DirectFB-XS

DirectFB-XS allows non-DirectFB Nexus applications to be composited together with DirectFB applications using the Nexus Surface Compositor (NSC).

In this mode, DirectFB will be built in single-application mode and multiple instances of DirectFB can run concurrently with other non-DirectFB applications. The Broadcom “Trellis” framework architecture uses this build mode of DirectFB to allow a mix of DFB and non-DFB applications to execute simultaneously.

To distinguish between DirectFB and DirectFB-XS builds, a new environment variable has been introduced named “DIRECTFB_NSC_SUPPORT”. When this is set to “y”, DFB will be built in DirectFB-XS mode and all DirectFB applications will be run as Nexus clients.

Example:

```
cd AppLibs/opensource/directfb/build
make DIRECTFB_NSC_SUPPORT=y default tarball
```

To be able to run a DirectFB application, a Nexus server application must be built and launched first. DirectFB includes a sample Nexus server called “nxsmaster”, which will be built when DIRECTFB_NSC_SUPPORT=y and DIRECTFB_MASTER_LIB=y are set.

N.B. For Nexus kernel mode builds DIRECTFB_MASTER_LIB is automatically set. For user space Nexus builds you will manually need to set this variable as it requires a second complete build of Nexus, so to reduce build time and target file system size this option isn’t enabled by default.

Once successfully built, the resulting tarball will be named in the same way to the one produced in step 4A above.

e.g. DirectFB -1.4.17_debug_build.97425B2.tgz

This tarball can be copied to the target platform in the same way as for step 4A above.

Building DirectFB tests

DirectFB test applications that reside in the DirectFB-1.4.17/tests directory are not built by default. To enable these unit tests ensure that the make build option “BUILD_TESTS” is set to “y”.

e.g. `make BUILD_TESTS=y`

Building DirectFB examples

DirectFB examples are separate test/demo applications that are built by default and can be run on the Broadcom reference platforms. To disable these additional test/demo applications from being built, ensure that the make build option “BUILD_EXAMPLES” is set to “n”.

e.g. `make BUILD_EXAMPLES=n`

Building ++DFB

++DFB (a.k.a. ppDFB), is a library that C++ application can call into to make DirectFB API calls (effectively a set of C++ bindings). ++DFB is a more advanced version of DFB++, and is incompatible in the way applications can call methods/functions.

To build ++DFB-1.4.2, ensure that the make build option “BUILD_PPDFB” is set to “y”.


e.g. `make BUILD_PPDFB=y`

 ++DFB requires the standard C++ libraries be present on the target platform.

Building Insignia test harness

The Insignia test harness is only available to certain customers who have signed an SLA with YouView. This test harness will check that the Broadcom DirectFB graphics driver matches the software fall back implementation for over 600 test cases. If the Insignia tarball is present, then this package can be built by setting the make build option “BUILD_INSIGNIA” to “y”.

e.g. `make BUILD_INSIGNIA=y`

 It is advisable to also set `DIRECTFB_HW_DITHERING=n` as running without this option will cause failures in certain 16bit pixel format blits due to added rounding differences.

Building Tacho test harness

The Tacho test harness is only available to certain customers who have signed an SLA with YouView. This test harness will check the graphics performance of the Broadcom DirectFB graphics driver. If the Tacho tarball is present, then this package can be built by setting the make build option “BUILD_TACHO” to “y”.

e.g. `make BUILD_TACHO=y`

Building external applications

Third party applications and/or external applications/tests can be built with the correct DirectFB compiler flags by using the “directfb-config” utility. The example below shows how a test application called “my_test.c” can be built.

```
mipsel-linux-gcc `./DirectFB-1.4.17/directfb-config --cflags --libs` my_test.c -o my_test
```



If the application you want to compile includes any Nexus headers and makes any Nexus calls, then the following additional flags can be passed to “directfb-config” in order for the application to be built more easily:

```
--extra-cflags --extra-libs
```

Additional make targets

The DirectFB build system does allow for partial steps or targets to be chosen. These build targets are listed below along with a description:

Table 3: Make targets

Make Target	Description
help	List the DirectFB make targets that can be called along with options.
default	This is the default make target and will attempt to install all DirectFB software modules, if they haven't already been installed. If a module hasn't been compiled, then it will be compiled first.
release	This will create a full release of the DirectFB software including both open source and Broadcom SLA specific code. The result is a dated tarball.
all	This will force every DirectFB software module to be reconfigured, rebuilt and installed.
tarball	This will create a tarball of the target output directory that can then be copied over to the target platform for unpacking and running.
install	This option is the same as the default target option and will only install software modules that need installing.
compile	This will attempt to compile all DirectFB software modules that need compiling. If a module hasn't already been configured, then it will be configured first.
config	This will attempt to configure all DirectFB software modules that need configuring. If a module hasn't already been unpacked, then it will be unpacked first.
uninstall	This will cause all installed intermediate files to be uninstalled.
uninstall-target	This will remove all target output directory installed files.
clean	Remove all generated object files, dependencies, binaries and temporary object directories.
distclean	Remove everything including the generated source code. The user will be prompted first to remove any source code. This target should always be called prior to installing a new release of DirectFB.
mrproper	Remove everything including generated source code. No user prompts will be displayed. This target should always be called prior to installing a new release of DirectFB.
check-tools	Do a quick check to make sure you have up-to-date tools to build DirectFB.
check-autogen-tools	Do a quick check to make sure you have up-to-date tools to auto generate the autoconf *.in files needed to build DirectFB.

Make Target	Description
directfb-defines	This will update the graphics and system defines files in the DirectFB-1.4.17 source tree.
xxx-all	Where xxx can be "", "directfb", "directfb-brcm", "directfb-examples", "sawman", "fusion", "ppdfb", "divine", "insignia", "tacho", "ffmpeg", "freetype", "jpeg", "png" and "zlib". This will re-build the chosen target software module including re-configuration and re-compilation if necessary.
xxx-source	Like "xxx-all" above, but the source code for the chosen module "xxx" will be created if not already present.
xxx-config	Like "xxx-source" above, but the configuration step will be called.
xxx-compile	Like "xxx-source" above, but the compilation step will be called.
xxx-install	Like "xxx-source" above, but the installation step will be called.
xxx-uninstall	Like "xxx-source" above, but the uninstallation step will be called.
xxx-clean	Only clean the required software module specified by "xxx".
xxx-distclean	Perform a complete clean of the chosen software module specified by "xxx" with user prompt.
xxx-mrproper	Perform a complete clean of the chosen software module specified by "xxx".
yyy-autogen	Where "yyy" can be "directfb", "directfb-examples", "sawman", "FFmpeg" and "ppdfb". This will regenerate the "Makefile.in" and "configure" scripts from the *.am files. This option is only useful for developers who want to change the way the chosen software module is built (e.g. build new test application).

Additional make flags

The following table lists the complete set of flags that can be passed to the DirectFB build system to modify the default build behaviour. The flags are normally passed on the “make” command line, but can also be set as environment variables.

e.g. `make DIRECTFB_MULTI=y`
e.g. `export DIRECTFB_MULTI=y`

Table 4: Make flags

Make Target	Description
DIRECTFB_VERSION	The version of DirectFB to be compiled can be overridden (e.g. <code>DIRECTFB_VERSION=1.4.17</code>)
DIRECTFB_MULTI	The default is to build DirectFB in single application mode. Setting this flag to “y” will build DirectFB in multi-application mode.
DIRECTFB_NSC_SUPPORT	The default is to build DirectFB in “non-XS” mode. Setting this flag to “y” will build in DirectFB-XS mode. .
DIRECTFB_MASTER_LIB	The default is to build DirectFB with master Nexus libraries and to build DirectFB-XS with only client Nexus libraries. Setting this flag to “y” when “ <code>DIRECTFB_NSC_SUPPORT=y</code> ” is also set, will ensure that the DFB Nexus master server application, <code>nxsmaster</code> , is also built.
DIRECTFB_CLIENT_LIB	The default is to build single-application DirectFB with only master Nexus libraries and DirectFB-XS with client only Nexus libraries. This flag is not normally used by the end user.
DIRECTFB_SHARED	The default is to build the zlib, freetype, png and jpeg utility libraries as shared libraries that can then be dynamically linked with DirectFB at run time. Setting this flag to “n” will instead build these libraries as static (.a) and DirectFB will statically link with them at compile time. Setting this flag to “n” generally increases code size, but can be useful if applications use different versions of these utility libraries.
DIRECTFB_PREFIX	The default target prefix is “ <code>/usr/local</code> ” but this can be overridden using this flag (e.g. <code>DIRECTFB_PREFIX=/usr</code>). This specifies the path to the DirectFB installation on the target platform.
DIRECTFB_IR_PROTOCOL	The default IR protocol can be overridden using this flag. The IR protocol should be the name of the NEXUS

Make Target	Description
	IR protocol (e.g. "Generic", "RemoteA", and "CirNec").
DIRECTFB_IR_INPUT	The default is to enable the DirectFB IR input if the platform supports IR input. However, the user can specify "n" to disable it.
DIRECTFB_KEY_INPUT	The default is to disable the DirectFB front-panel keypad input. However, the user can specify "y" to enable it.
DIRECTFB_SID	The default is to build the DirectFB still image decoder (SID) image provider module/lib, if the platform supports a SID hardware decoder. However, the user can specify "n" to prevent this module from ever being built.
DIRECTFB_SW_DITHERING	Enable software dithering (currently only supported with RGB16 and ARGB4444 formats). When set to "y" advanced software dithering for RGB16 and ARGB4444 formats will be enabled. The downside is that this will increase the size of the data section by at least 64KB. Default is "n".
DIRECTFB_HW_DITHERING	Enable hardware dithering support for RGB16 and ARGB4444 pixel formats. The default is "y". It is best to set this to "n" when running the Insignia test suite as some of the tests will fail due to the dithering.
DIRECTFB_SW_SMOOTH_SCALING	Enable software smooth scaling. When set to "y" software smooth scaling will be enabled and the size of the text section will increase by at least 100K bytes.
DIRECTFB_GFX_PACKET_BUFFER	The default is to enable the packet buffer interface in the Broadcom DirectFB graphics driver. Setting this to "n" will cause the legacy graphics driver to be used that will have lower graphics performance.
DIRECTFB_GFX_TRAPEZOID_SUPPORT	The default is to enable drawing of trapezoids in the graphics driver if the packet buffer interface is enabled too. Setting this option to "n" will result in trapezoids being drawn using only software.
DIRECTFB_GFX_SOFT_MATRIX_SUPPORT	The default is to enable support for the SetMatrix() function in our graphics driver using the PX3D hardware to perform rotation and shearing. If the PX3D hardware is not present or this option is set to "n", then a much limited feature set for SetMatrix() will be available.

GL_SUPPORT	This flag controls whether the nexus/magnum drivers and DirectFB are built with support for the PX3D 3D graphics core. By default the drivers and DirectFB are not built with support for 3D graphics. To enable support for “DrawLine()”, “TextureTriangles()”, “FillTriangle()” and “FillTriangles()” set this flag to “y” when building nexus and also when building DirectFB. This flag does not have any effect for chips that don’t have the PX3D core.
DIRECTFB_GLES_SUPPORT	The default is not to build DirectFB with OpenGL ES 1.0 and EGL support. However, setting both this flag and the “GL_SUPPORT” flag to “y” will build DirectFB with 3D OpenGL ES 1.0 support. This support is only available on devices that have a PX3D graphics core. The nexus/magnum drivers must have also been built with both these flags set to “y” in order to have OpenGL ES 1.0 support.
BUILD_TESTS	The default is not to build the DirectFB unit test applications that are located in the “tests” directory. Setting this flag to “y” will instead build and install the additional DirectFB tests.
BUILD_EXAMPLES	The default is to build the additional DirectFB examples. Setting this flag to “n” will disable the building and installing of the additional DirectFB examples.
BUILD_SAWMAN	The default is to build SaWMan only when “DIRECTFB_MULTI=y”. Setting this flag to “n” will disable building SaWMan and will ensure that the default window manager of DirectFB is used. This option is only meaningful when building in multi-application mode.
BUILD_PPDFB	The default is not to build the ++DFB library. Setting this flag to “y” will build and install the library.
BUILD_FUSION	The default is to build the “linux-fusion” kernel module only when DirectFB is built in multi-application mode. Setting this flag to “n” will prevent this module from being built and installed and if used in conjunction with “DIRECTFB_MULTI=y”, the experimental multi-application mode of DirectFB will be built instead.
BUILD_VOODOO	The default is not to build the voodoo library that is part of DirectFB. Setting this flag to ‘y’ will build and install the library.

BUILD_FFMPEG	The default is “n”, to build the FFmpeg library that is used to decode MPEG-2 and H.264 I/IDR pictures set it to “y”. If you do not need to display MPEG-2/H.264 I/IDR still pictures, then you may leave it unset.
BUILD_INSIGNIA	The default is not to build the Insignia library. Setting this flag to “y” will build and install the library if the source tarball is present.
BUILD_TACHO	The default is not to build the Tacho library. Setting this flag to “y” will build and install the library if the source tarball is present.
NEXUS_MODE	The default is to build NEXUS for user-space. Setting this option to “proxy” will result in the NEXUS drivers being compiled for kernel-space.
B_REFSW_ARCH	The default is to build DirectFB and the associated libraries in little endian mode. Setting this flag to “mips-linux” will result in Nexus, DirectFB and its libraries being built in big endian mode instead. .
B_REFSW_DEBUG	The default is to build DirectFB in debugging mode. However, setting this flag to “n” will build DirectFB in release mode and no debugging information will be available. Setting this option to “n” will also improve graphics performance.
B_REFSW_VERBOSE	The default is to build DirectFB with minimal information. Setting this flag to “y” will increase the amount of information available during the building stages.
TRACE	The default is to build DirectFB without any tracing information. Setting this flag to “y” will allow tracing information to be enabled.
DIRECTFB_EXAMPLES_VERSION	The default is to build the DirectFB examples 1.6.0pre software package. If a different version is available, then seeing this flag will result in that version being built instead (e.g. DIRECTFB_EXAMPLES_VERSION=1.2.1). The alternative software tarball should be placed in the “directfb_tarballs” directory before building the software.
FUSION_VERSION	The default is to build linux-fusion version 8.9.10. If an alternative tarball version is available to be built, it should first be placed in the “directfb_tarballs” directory and this flag should be set accordingly (e.g. FUSION_VERSION=8.1.1).

SAWMAN_VERSION	The default is to build SaWMan version 1.5.4, but if a different version of SaWMan is available, then setting this flag will result in that version being built instead (e.g. SAWMAN_VERSION=1.4.15). The alternative software tarball should be placed in the “directfb_tarballs” directory before building the software.
FFMPEG_VERSION	The default is to build FFmpeg version 0.9.1. Setting this flag will allow an alternative version of FFmpeg to be built (e.g. FFMPEG_VERSION=1.0.0). The alternative software tarball should be placed in the “directfb_tarballs” directory before building the software.
PPDFB_VERSION	The default is to build ++DFB version 1.4.2. However, this behaviour can be overridden by specifying an alternative version (e.g. PPDFB_VERSION=1.4.0). The alternative software tarball should be placed in the “directfb_tarballs” directory before building the software.
INSIGNIA_VERSION	The default is to build Insignia version 0.1.2. However, this behaviour can be overridden by specifying an alternative version (e.g. INSIGNIA_VERSION=0.1.3). The alternative software tarball should be placed in the “directfb_tarballs” directory before building the software.
TACHO_VERSION	The default is to build Insignia version 0.1.2. However, this behaviour can be overridden by specifying an alternative version (e.g. TACHO_VERSION=0.1.3). The alternative software tarball should be placed in the “directfb_tarballs” directory before building the software.
DFB_FREETYPE_VERSION	The default is to build Freetype version 2.4.9 from “AppLibs/opensource/freetype”. This behaviour can be overridden by setting this flag appropriately. For example, to build a different Freetype library version you can set “DFB_FREETYPE_VERSION=2.4.9”.
DFB_JPEG_VERSION	The default is to build JPEG version “8d” from “AppLibs/opensource/jpeg”. This behaviour can be overridden by setting this flag appropriately. For example, to build a different JPEG library you can set “DFB_JPEG_VERSION=8d”.
DFB_PNG_VERSION	The default is to build PNG version 1.5.10 from “AppLibs/opensource/libpng”. This behaviour can be

	overridden by setting this flag appropriately. For example, to build a different PNG library you can set "DFB_PNG_VERSION=1.5.10".
DFB_ZLIB_VERSION	The default is to build zlib version 1.2.6 from "AppLibs/opensource/zlib". This behaviour can be overridden by setting this flag appropriately. For example, to build a different zlib library you can set "DFB_ZLIB_VERSION=1.2.6".
APPLIBS_INSTALL_PREFIX	The default is to install all files relative to "/usr/local" on the target platform. This option can be overridden to place the target files in a different directory structure.
APPLIBS_TARGET_TOP	This specifies the final output directory on the host build machine in which the DirectFB binaries and libraries are to be installed prior to being packed ready for transfer to the target platform. The default is "AppLibs/target", but this can be overridden
APPLIBS_COMMON_INC	This specifies whether the AppLibs or DirectFB build process will be used to compile the zlib, libpng, libjpeg and freetype ancillary libraries. The default is to "n", to use the DirectFB build process.
OPENSOURCE_TOP	This specifies the top-level directory where all open-source software components/libraries reside on the host machine. By default this is "AppLibs/opensource".
NEXUS_TOP	This specifies the top-level directory where the Nexus reference software resides.


Running DirectFB on the target platform

This section explains how to run DirectFB and DirectFB-XS on different target platforms.

Standard DirectFB single application mode

Once you have generated the tarball in Step 4A you need to copy it to the target platform. The most straightforward method is to run an NFS server on your build machine and mount the extracted file system onto the reference board.

```
mkdir -p /export/nfs/97425/  
tar -zxvf DirectFB-1.4.17_debug_build.97425B2.tgz -C /export/nfs/97425/
```

 You will need to make sure that the NFS server is running on your build machine and that the directory you have extracted the root file system into is exported in the `/etc/exports` configuration file.

Once the NFS server is configured you will need to mount the file system on the reference platform and create a link to place the files in the correct location.

```
mount 192.168.0.1:/export/nfs/97425/usr /mnt/nfs;  
ln -s /mnt/nfs /usr
```

With the file system correctly mounted, you are now ready to run the installation script.

```
cd /usr/local/bin/directfb/1.4  
./rundfb.sh install
```

You are now ready to run any DirectFB application from this directory. For example, to run the `df_andi` (penguins) test, enter the following command:

```
./rundfb.sh df_andi
```

You can also specify the output resolution of the connected display (the default is 720p on most chipsets). For example, to run with a 1080i output resolution, you can enter the following command:

```
./rundfb.sh df_andi --dfb:res=1080i
```

DirectFB multi-application mode

With DirectFB running in multi-application mode, more than one application can access the DirectFB API's at the same time from different processes. The same steps can be taken as for single-application mode when it comes to running the first application. However, for any subsequent application, the "rundfb.sh" script needs to be used with the "join" option. An example of running both df_andi and df_window in multi-application mode using different processes is given below:

```
cd /usr/local/bin/directfb/1.4
./rundfb.sh install
./rundfb.sh df_window &
./rundfb.sh join df_andi --dfb:force-windowed,mode=640x480
```

You should now be able to see the DirectFB Penguins application on top of df_window. If you have a USB keyboard and mouse connected to the platform, you should be able to move the windows created by df_window around the screen. Pressing Q or ESC on the USB keyboard will quit the application(s). Pressing EXIT on an infra-red handset will also quit the application(s).

You can also specify the size of the graphics surface/layer independent of the output resolution of the display. For example, if you would like to have a 640x480 graphics layer with a 1280x720p output resolution, you can use the "mode" DirectFB option. An example of this is given below:

```
./rundfb.sh join df_andi --dfb:force-windowed,mode=640x480
```

In some multi-application usage modes, the graphics will be stretched horizontally and vertically to fill the display window. If you would prefer not to have the graphics stretched to fill the display window, then the "scaled" option can be used in conjunction with "force-windowed". An example of this is given below:

```
./rundfb.sh join df_andi --dfb:force-windowed,mode=640x480,scaled=640x480
```



The first DirectFB application that runs is known as the "master" and subsequent DirectFB applications are known as "slaves". The "master" application is normally a module that should not under normal circumstances be terminated. Internally within DirectFB, it will manage the Nexus display settings and will be responsible for creating and destroying all Nexus surfaces and memory. If this application is terminated before any of the client applications are closed, then the system may be left in an unrecoverable state and may require a reboot.

DirectFB-XS (Nexus Surface Compositor)

Before you can run any DirectFB applications in this mode, you need to launch the Nexus server application. If you have built DirectFB with the “DIRECTFB_NSC_SUPPORT=y” option (and DIRECTFB_MASTER_LIB=y for userspace Nexus drivers, then the “nxsmaster” application (like “dfbmaster”) will have also been built.

```
cd /usr/local/bin/directfb/1.4
./rundfb.sh install
./rundfb.sh nxsmaster &
```

You are now ready to run any DirectFB application from this directory as a Nexus slave. You will need to specify the “join” command for all Nexus/DFB slave applications to ensure that the Nexus drivers are not reloaded.

For example, to run the df_andi (Penguins) test application as full screen, enter the following command:

```
./rundfb.sh join df_andi
```

The Nexus server application (“nxsmaster” in this case) is the process that composites the different applications framebuffers together using the Nexus Surface Compositor (NSC). It is responsible for the size and position of the client applications’ framebuffer. There are some presets in the “nxsmaster” application that allow multiple applications to be positioned in different quadrants of the screen. You can specify the client/quadrant by using the “dfb_clientid” envvar before you launch an application. For example, to launch df_andi in the top left-hand corner of the screen, enter the following command:

```
dfb_clientid=1 ./rundfb.sh join df_andi
```

To launch another instance of df_andi in the bottom right-hand corner of the screen, you may enter:

```
export sw_picture_decode=1 (the current Nexus SID driver doesn't support multiple
instances)
dfb_clientid=4 ./rundfb.sh join df_andi
```

Running texture mapped graphics applications

DirectFB can be built with hardware acceleration support for “TextureTriangles()”, “FillTriangle()” and “FillTriangles()” graphics functions. This hardware acceleration support is only available for Broadcom devices that have the PX3D 3D graphics core (e.g. BCM7420). Both Nexus and DirectFB must be built with the “GL_SUPPORT=y” environment flag set.

To test “TextureTriangles()”, the user can run the “df_texture” unit test.

e.g. `./rundfb.sh df_texture`

It is possible to run “df_texture” or perform matrix transformations on primitives and surfaces without 3D capable graphics hardware. This is achieved by using software fall back graphics operations which are enabled by default on non PX3D systems.

Running OpenGL ES 1.0 graphics applications

DirectFB can be built with support for OpenGL ES 1.0 and EGL for chips that have the PX3D 3D graphics core (e.g. BCM7420). Both Nexus and DirectFB must be built with the following two environment flags set:

```
"GL_SUPPORT=y"
"DIRECTFB_GLES_SUPPORT=y"
```

To test the OpenGL ES 1.0 and EGL support within DirectFB, you may run any of the following test applications (build with BUILD_TOOLS=y):

1. `dfbtest_egl_only`
2. `dfbtest_egl_pixmap` (use mouse to XYZ rotation)
3. `dfbtest_gl` (use mouse to change XYZ rotation)


e.g. `./rundfb.sh dfbtest_gl`

Running OpenGL ES 2.0 graphics applications

The DirectFB release does not contain any OpenGL ES 2.0 applications, but instead the “rockford/applications/opengles_v3d/v3d/directfb” directory contains a few example applications. These applications must be built after DirectFB has been built and they will only run on chips that have a VC-4 3D graphics core (e.g. BCM7425) and won’t run on chips that have the PX3D core.

To build any of these OpenGL ES 2.0 applications, simply type “make” in the appropriate application directory (ensuring that “V3D_SUPPORT=y” envvar is set first):

```
e.g. make -C rockford/applications/khronos/v3d/directfb/cube
```

 If you have built DirectFB in multi-application mode, then you need to ensure you pass “DIRECTFB_MULTI=y” on the make command line:

```
make DIRECTFB_MULTI=y -C rockford/applications/khronos/v3d/directfb/cube
```

or

```
make DIRECTFB_MULTI=y -C  
rockford/applications/khronos/v3d/directfb/earth_es2
```

After this step, you can type “make tarball” in the DirectFB build directory to create a tarball that will contain the application(s) and OpenGL driver and platform code.

This release now supports running multiple OpenGL ES 2.0 applications simultaneously. For example, you can run df_window, earth_es2 and cube together when DFB is built in multi-application mode as follows:

Example:

```
./rundfb.sh df_window &  
./rundfb.sh earth_es2 --dfb:force-windowed,mode=640x480 &  
./rundfb.sh cube --dfb:force-windowed,mode=320x240
```

For newer platforms which support Nexus Surface Compositor (NSC) you may want to look at this approach rather than using DirectFB to display OpenGL applications. This approach offers more flexibility and removes any dependency of the OpenGL application on DirectFB, reducing code size and increasing performance. This also means you can use DirectFB-XS which uses DirectFB in single application mode which offers better performance and reduced CPU usage than multi-application mode.

Running SaWMan (multi-application mode)

SaWMan is the **Shared application and Window Manager** that overrides the “default” window manager of DirectFB. It can act as an application lifecycle manager, deciding what application/processes can be spawned or terminated and which application(s) receive input events. Many multi-application environments use SaWMan to help fulfil their requirements for displaying and managing multiple applications simultaneously.

If DirectFB has been built in multi-application mode, then the SaWMan becomes the default window manager. There are two specific applications that can be used to test SaWMan functionality. They are “testman” and “testrun”. “testman” is the main application manager and is used to register what applications can be spawned or terminated. It also has full control over the layout of multiple applications on the display. “testrun” on the other-hand, is used to signal what pre-registered application can be run. “testrun” can be called from different processes multiple times, thus helping to simulate a real-world multi-process / multi-application environment. To test SaWMan follow the steps below:

Running with kernel-space (proxy mode) drivers

```
cd /usr/local/bin/sawman/1.5
./runsaw.sh testman &
./runsaw.sh join testrun Penguins
./runsaw.sh join testrun Penguins2
./runsaw.sh join testrun Penguins3
./runsaw.sh join testrun Penguins4
```

Running with user-mode drivers

```
cd /usr/local/bin/sawman/1.5
./runsaw.sh testman &
cd /usr/local/bin/directfb/1.4
./rundfb.sh join df_andi --dfb:force-windowed &
./rundfb.sh join df_andi --dfb:force-windowed &
./rundfb.sh join df_andi --dfb:force-windowed &
./rundfb.sh join df_andi --dfb:force-windowed &
```

On the screen you should see 4 windows each with their own df_andi (Penguins) moving around. You can move the mouse over any of the windows and press <Q> to quit the application. Each of the applications is running in a separate process.

Running DirectFB examples

DirectFB examples are additional example applications and tests that are built when the “BUILD_EXAMPLES=y” option is set. To test any of these additional example applications, follow the example steps below:

```
cd /usr/local/bin/directfb/1.4
./rundfb.sh df_matrix
```

Running ++DFB

++DFB requires that the C++ standard libraries are installed on the target platform. These libraries are usually located in the “/lib” directory and are called “libstdc++.so

If you have the C++ libraries installed you should be able to run any of the ++DFB test applications. For example, you can run the “dfbshow” test application as follows:

```
cd /usr/local/bin/++dfb/1.4
./runppd.sh dfbshow /usr/local/share/directFB-1.4.17/images/biglogo.png
```

Running audio/video tests

When building for multi-application DirectFB (non-DirectFB-XS), there are three audio/video playback examples tests:

1. playback_dfb
2. decode_server_dfb
3. decode_client_dfb

playback_dfb

This is a DirectFB master and Nexus server application that is built when the Nexus server library (libnexus.so) is present.

This application can read in an MPEG-2 transport stream file and decode a particular packetized elementary stream within it specified by the audio and video PIDs. The audio and video codecs can also be specified on the command line.

It is possible to start this application running and then run any further DirectFB applications as slaves.

Example:

1. Run the playback example as a master DirectFB application (Nexus server):


```
./rundfb.sh playback_dfb --file <path to MPEG-2TS file> --vpid <video PID in decimal> --apid <audio PID in decimal> --vcodec <video codec in decimal> --acodec <audio codec in decimal>.
```

2. Run df_andi as a DirectFB slave application (Nexus client):

```
./rundfb.sh join df_andi -dfb:force-windowed,mode=640x480
```



You can find out what the audio and video codecs arguments are available by using the “--help” argument to “playback_dfb”.

e.g. `./rundfb.sh playback_dfb --help`

decode_server_dfb

This is a Nexus server / DirectFB master application that is only built when the Nexus server library (libnexus.so) is present and DirectFB is compiled in multi-application mode. This application initializes Nexus, opens up server-side Nexus modules/interfaces (e.g. display, simple audio/video decoders) and starts the Nexus server. This application must be the first Nexus/DirectFB application in the system to be executed.

decode_client_dfb

This is a Nexus client application that is used to playback audio/video from an MPEG-2 TS file. It connects to the already running “decode_server_dfb” application on the target platform. It accepts the same command line arguments as for the “playback_dfb” application. The advantage this application has over “playback_dfb” is that it can be run as a Nexus client (slave) application. For example, to be able to playback audio/video in a Nexus client application as well as running a DirectFB slave application (e.g. df_andi), you can follow the steps below:

From the main console:

```
1. ./rundfb.sh decode_server_dfb
```

From a new virtual console (e.g. telnet session):

```
2. ./rundfb.sh join decode_client_dfb --file <path to MPEG-2 TS file> --vpid <video PID in decimal> --apid <audio PID in decimal> --vcodec <video codec in decimal> --acodec <audio codec in decimal>.
```

From a new virtual console (e.g. telnet session):

```
3. ./rundfb.sh join df_andi --dfb:force-windowed,mode=640x480
```



You can find out what the audio and video codecs are available by using the “--help” argument to “decode_client_dfb”.

e.g. `./rundfb.sh join decode_client_dfb --help`

Run-time environment variables

The table below lists the environment variables that affect the run-time behaviour of DirectFB. These environment variables can be set using the “export” command.

Table 5: Run-time environment variables

Runtime Option	Description
dfb_slave	This determines whether the DirectFB application should “join” Nexus (set to “y”) or initialize Nexus (set to “n”). This option can be set to “y” if another non-DirectFB application is the primary application in the system and has already initialized Nexus with “NEXUS_Platform_Init()”. After initializing or joining Nexus, DirectFB can decide whether to open the display, graphics and picture decoder Nexus handles itself or use the handles provided to it in the DFB_Platform_Init() call.
dfb_clientid	Used by client applications when running in DirectFB-XS mode to determine which client is attaching to the server.
sw_picture_decode	This envvar only affects platforms that have a still image decoder (SID). Normally, JPEG, GIF and PNG images are rendered using the SID. However, setting this envvar to any value will result in the software DirectFB picture decoding functions being used instead.
hdsd_mode	Set the HD/SD display mode. If this envvar is set to “0”, then the composite/CVBS output is connected to the primary display 0 output. In this configuration, only SD display output resolutions are supported on both primary and secondary display outputs (e.g. “res=576i”). If this envvar is not set or is a value other than “0”, then the composite/CVBS output is connected to the secondary display 1 output and the primary display output can be configured to be either HD or SD. (Not available in DirectFB-XS)
DFBARGS	This is the standard DirectFB arguments envvar that can be used to specify the DirectFB run-time options (e.g. export DFBARGS=“res=1080i”).

It is worth noting that the build system does not track modified source code files between the stages in green. For example, if the user built and installed DirectFB by typing “make” and then modified a DirectFB-1.4.17 source code file, the build system does NOT know that a source code file was modified if the user were to type “make” again. Instead, the user can type “make directfb-compile” and this will rebuild only the source files needed within DirectFB. The user can then type “make” and the build system is intelligent enough to know that the directfb installation phase needs to be completed next.

This approach saves time during the build process when making lots of source code modifications. If the makefile had to call each software modules “compile” stage, then it would also force an installation which would all consume valuable time. The recommended approach is to make source code modifications, type “make xxx-compile” (where xxx is the software module like “directfb”) and then type “make” for the build system to complete any further necessary steps (e.g. installation). The same can be said if the user wants to reconfigure DirectFB or a software module. The user should type “make xxx-config” first to re-configure the software module, and then type “make”. Usually this will involve the source code being re-compiled and re-installed.

Multi-application support with DirectFB

Standard DirectFB can be built in what is known as “multi-application” mode. This mode allows multiple DirectFB and non-DirectFB applications to run in separate processes simultaneously. DirectFB multi-process support is available when the Nexus drivers are built for “proxy” mode (kernel mode) and when the drivers are built for user-space.

The first DirectFB or non-DirectFB application that is executed on the target system is known as the “master” application and this application always need to be running and must be the last application terminated on the target system. If this first application is a DirectFB application, then it will be responsible for receiving remote procedure calls (RPC) from client/slave DirectFB applications. This “master” DirectFB application is responsible for creating and destroying Nexus surfaces, allocating and freeing Nexus memory, graphics rendering using the underlying graphics hardware and handling Nexus display settings (e.g. setting the frame-buffer). When a DirectFB slave application (Nexus client) tries to create a surface, set the graphics frame-buffer or use the DirectFB graphics operation API’s (e.g. Blit()), the core DirectFB code will use “secure-fusion” to issue a RPC to the “master” DirectFB application to service the request. By default, “secure-fusion” is enabled so many of the DirectFB API’s are actually executed in a dispatch thread in the context of the master DirectFB process.

If the “master” application is terminated either intentionally or unintentionally, then the system will be in an unstable state as client/slave applications won’t be able to have their RPC requests serviced.

Running non-DirectFB and DirectFB applications

There are some situations where the system may have non-DirectFB applications and DirectFB applications running concurrently. In this scenario, the non-DirectFB application may have already initialized Nexus and opened Nexus modules that the DirectFB application(s) rely on (e.g. Nexus display, graphics2d, graphics3d, picture decoder).

(For newer systems you may want to consider using Nexus Surface Compositor to handle this situation).

To allow for this usage scenario, there is a “dfb_platform.h” file that contains a light-weight API that non-DirectFB and DirectFB applications can use.

Running a non-DirectFB master application

There are two ways in which a non-DirectFB “master” application can initialize Nexus:

1. Can continue to use the existing “NEXUS_Platform_GetDefaultSettings()” and “NEXUS_Platform_Init()” API. It will also need to start the Nexus server using the “NEXUS_Platform_GetDefaultStartServerSettings()” and “NEXUS_Platform_StartServer()” APIs (see “decode_server_dfb” example).

It can then inform the Broadcom DirectFB platform layer code that Nexus has already been initialized by making a call to “DFB_Platform_GetDefaultSettings(DFB_PlatformClientType_eMasterNexusInitialized, pSettings)” followed by a call to “DFB_Platform_Init(pSettings)”. The client type in the call to “DFB_Platform_GetDefaultSettings()” specifies that this is a master application and that Nexus has already been initialized.

By default, the “DFB_Platform_Init()” function will automatically attempt to open up the Nexus display, graphics2d, graphics3d and picture decoder modules and will place these module handles in System V shared memory to allow other processes to access them. If the “master” non-DirectFB application has already opened up the Nexus modules, then the Nexus handles can be passed into the “DFB_Platform_Init()” function and the DFB platform code will know not to attempt to open the Nexus modules again.

The “DFB_Platform_Init()” function will also try to automatically connect up certain outputs to the display(s). Normally, the HDMI/component outputs will automatically be connected to the primary HD display and the composite output will be connected to the secondary SD display. If the non-DirectFB “master” application wants to override this default behaviour, then it can indicate what outputs should be connected to what display by setting appropriate flags in the “DFB_PlatformSettings” structure.

2. Can replace the calls to “NEXUS_Platform_GetDefaultSettings()”, “NEXUS_Platform_Init()”, “NEXUS_Platform_GetDefaultStartServerSettings()” and “NEXUS_Platform_StartServer()” with “DFB_Platform_GetDefaultSettings(DFB_PlatformClientType_eMasterNexusUninitialized, pSettings)” and “DFB_Platform_Init(pSettings)”. The application can still pass in certain Nexus platform settings through this API.

By default, the display, graphics2d, graphics3d and picture decoder Nexus modules will automatically be initialized and outputs will be connected to the display (e.g. HDMI, component). This is the simplest approach if the non-DirectFB application does not need to modify Nexus platform settings that are not exposed within the “DFB_PlatformSettings” structure.

Running a DirectFB master application

For a “master” DirectFB application, there are three ways in which Nexus can be initialized:

1. The simplest approach is not to do anything and rely on the internal Broadcom DirectFB system driver to automatically call into the DFB Platform code with default values to initialize Nexus and open up the display, graphics2d, graphics3d and picture decoder Nexus modules. Outputs like HDMI and component will automatically be connected to the primary display. This is the preferred method as it means the DirectFB application does not need to be modified to make any explicit “DFB_Platform_xxx()” calls and does not need to link with the DFB platform shared library (libinit.so).
2. Use the “DFB_Platform_GetDefaultSettings(DFB_PlatformClientType_eMasterNexusUninitialized, pSettings)” and “DFB_Platform_Init(pSettings)” API’s to explicitly initialize Nexus. Again, by default the Nexus display, graphics2d, graphics3d and picture decoder modules will automatically be opened and outputs connected to the display. This option is useful, if the default platform values are not suitable and need modifying.
3. Use the “NEXUS_Platform_GetDefaultSettings()”, “NEXUS_Platform_Init()”, “NEXUS_Platform_GetDefaultStartServerSettings()” and “NEXUS_Platform_StartServer()” API’s to explicitly initialize Nexus with non-default values. Then use “DFB_Platform_GetDefaultSettings(DFB_PlatformClientType_eMasterNexusInitialized, pSettings)” and “DFB_Platform_Init(pSettings)” to inform DirectFB that Nexus as already been initialized and that the Nexus server has been started. This option is useful if non-default Nexus platform settings need to be provided. This case is rarely used for DirectFB “master” applications.

Running a non-DirectFB slave application

A non-DirectFB “slave” application (Nexus client) is one in which another DirectFB or non-DirectFB application is the master and initialized Nexus. A “slave” application cannot initialize Nexus, but can “join” it and can open up additional Nexus modules for its own use. Any Nexus handles that were opened in either the DirectFB master application or another Nexus application cannot be shared and used in future Nexus calls in a different process, if the Nexus client process(es) are run in either “untrusted” or “protected” Nexus security modes. Refer to the Nexus Multi-Application document for more information (NEXUS_MultiProcess.pdf).

There are two ways in which a non-DirectFB “slave” application can be run:

1. Simply call “DFB_Platform_GetDefaultSettings(DFB_PlatformClientType_eSlaveNexusUninitialized, pSettings)” followed by “DFB_Platform_Init(pSettings)”.
2. Make an explicit call to “NEXUS_Platform_Join()” to join Nexus. Then call “DFB_Platform_GetDefaultSettings(DFB_PlatformClientType_eSlaveNexusInitialized, pSettings)” followed by “DFB_Platform_Init(pSettings)”. This case would rarely be used, as it requires an additional call to Nexus.

Running a DirectFB slave application

For “slave” DirectFB applications, Nexus will have already been initialized. However, “slave” DirectFB applications can still have control over whether a particular Nexus module that DirectFB depends on should be opened, as long as only that process consumes the Nexus module (in “untrusted” and “protected” Nexus security modes).

There are two ways in which a DirectFB slave application can be run:

1. If the “slave” DirectFB application(s) doesn’t need to perform any additional platform initialization or configuration, then they can simply ignore the Broadcom DirectFB platform API entirely. Internally, the DirectFB system driver will call into the platform code to “join” Nexus and to obtain already opened Nexus module handles.
2. If the “slave” DirectFB application(s) would like to perform any additional initialization or configuration, then they can do so by using “DFB_Platform_GetDefaultSettings(DFB_PlatformClientType_eSlaveNexusUninitialized, pSettings)” followed by “DFB_Platform_Init(pSettings)” API calls. The application would have to explicitly link against the DirectFB platform library (libinit.so).

Terminating a non-DirectFB application

Non-DirectFB applications that have called “DFB_Platform_Init()” must explicitly call “DFB_Platform_Uninit()” to ensure the system is in a consistent state. This function will release resources that were previously acquired by the “DFB_Platform_Init()” call, such as Nexus modules and outputs. It will only release resources that were opened/created in the call to DFB_Platform_Init() and in the context of that process.

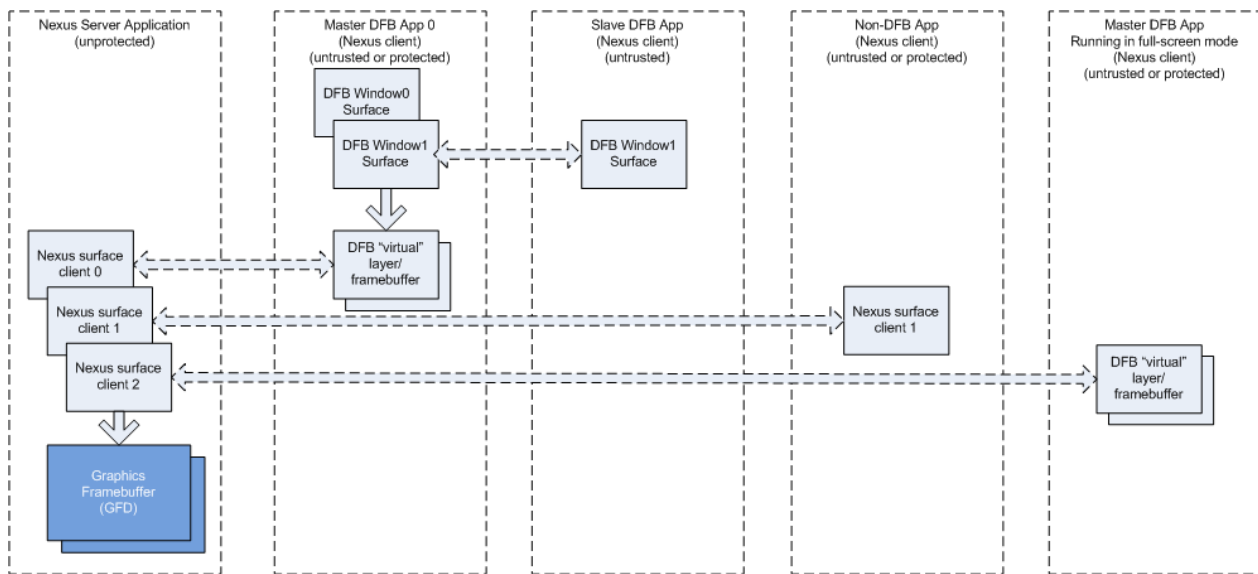
Terminating a DirectFB application

When a DirectFB application is to be terminated, it can either explicitly call “DFB_Platform_Uninit()” to ensure the system is in a consistent state or can let the DirectFB system driver automatically call this function upon shutdown. This function will release resources that were previously acquired by the implicit or explicit call to “DFB_Platform_Init()” call, such as Nexus modules and outputs.

Multi-application support with DirectFB-XS

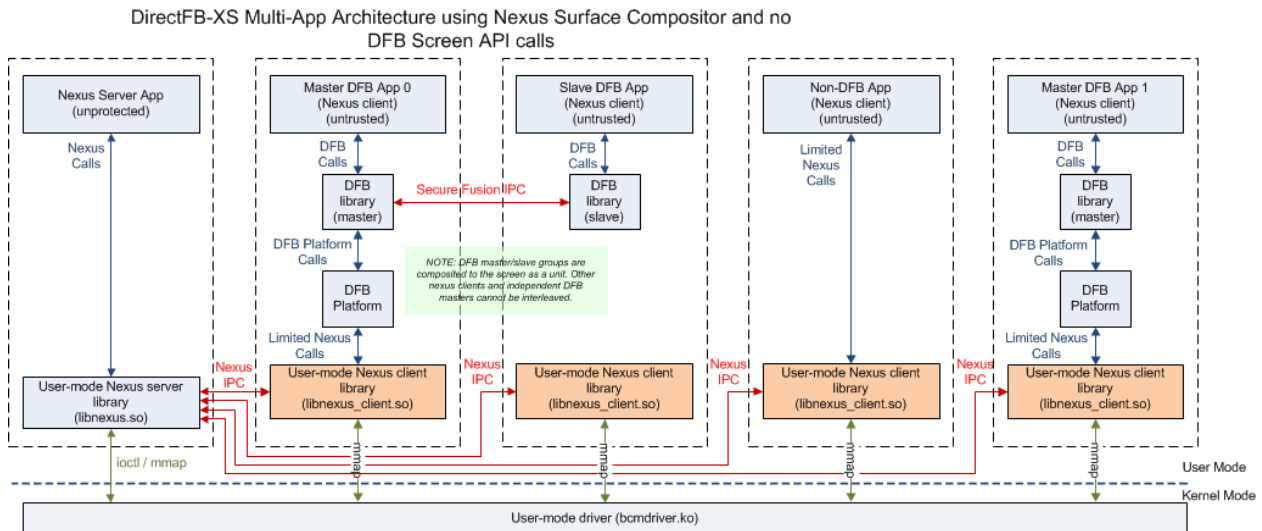
DirectFB-XS is a Broadcom terminology for running DirectFB with an eXternal Surface compositor (Nexus surface compositor). In this usage scenario, multiple DirectFB and non-DirectFB applications can run concurrently and their graphics outputs will be positioned, sized and blended together by the Nexus surface compositor before provided to the graphics feeder for output on the associated display. DirectFB can only be built in single-application mode and multiple instances of single-app DirectFB applications can be run simultaneously with non-DirectFB applications.

Instead of DirectFB accessing a physical framebuffer when it composites graphics on the DirectFB “layer”, a “virtual” layer / framebuffer is provided. This “virtual” framebuffer is in fact just a Nexus surface that is blended with other Nexus surfaces by the Nexus surface compositor. The diagram below helps to illustrate this behaviour.



On the diagram above, a “Nexus server application” contains the Nexus surface compositor. This is used to composite Nexus surfaces from different process application spaces. The “Master DFB App 0” and “Slave DFB App” processes help to illustrate what occurs when DirectFB is built and run in multi-application mode. Here, there is a conventional DFB master application that uses the DirectFB window compositor to composite graphics from its application and a DFB slave application on to a “virtual” frame-buffer. This “virtual” frame-buffer is then “pushed” to the Nexus surface compositor and blended with other “virtual” frame-buffers from “non-DFB App” and other “Master DFB App” applications. The “Master DFB App” application shown on the far right of the diagram can be another instance of DirectFB that has been built in single-application mode.

The diagram below helps to illustrate what software modules each type of application can call.



What can be seen is that neither the DirectFB nor the non-DirectFB applications directly call the Broadcom DFB platform library. Instead, the DFB platform layer is only called internally by the DirectFB application through the core DFB code (during the system initialize/join call). This means that non-DirectFB applications no longer need to call the DFB platform API's in order to inform DFB whether the system has been initialized or not.

Each of the DirectFB and non-DirectFB applications are treated as Nexus clients. They will all join Nexus, rather than initializing Nexus. Only the Nexus server application is used to initialize Nexus and it is typically the application that also opens any audio/video outputs. It is the process that will instantiate the Nexus surface compositor and will typically be the process that instantiates the Nexus simple audio/video decoders (for playing back of audio/video from a Nexus client application(s)).

The diagram also helps to illustrate how Nexus client applications communicate with a Nexus server. In the diagram, Nexus has been built to run in user-space. Each of the DFB and non-DFB applications will inherently link with the client Nexus user-space shared library. Nexus calls that affect the underlying hardware will automatically be "marshalled" across to the "Nexus server" application using the auto-generated thunk (Nexus IPC in the diagram). Here, a "master server" instance of the Nexus library will actually make the call to the underlying hardware.

For more information about the Nexus multi-application architecture, refer to the "Nexus_MultiProcess.pdf" document that is bundled as part of the reference software release.

DirectFB memory management

This is a brief overview of some of the principles of how Broadcom's DirectFB implementation uses memory.

DirectFB uses three sources of memory:

1. Linux memory (ignored for this discussion).
2. Primary on screen display 'framebuffers'.
3. Off-screen graphics memory.

Memory for cases 2 and 3 are available from the Nexus heaps and are requested using the "NEXUS_Platform_GetFramebufferHeap()" function in the Nexus platform code.

For case 2, the call used is "NEXUS_Platform_GetFramebufferHeap(0)" for the main display and this needs to return a handle to a heap which is of type "NEXUS_MemoryType_eFull".

For case 3, the memory is available from the call to "NEXUS_Platform_GetFramebufferHeap(NEXUS_OFFSCREEN_SURFACE)" and this should return a heap handle of type "NEXUS_MemoryType_eApplication". If DirectFB fails to get a valid heap for case 3, the offscreen memory, we can set the offscreen heap to be the same as for case 2, the primary heap.

In a DirectFB multi-application system when we set the client application permissions we only pass across the heap settings for the offscreen heap (case 3). All the allocations the slave app makes should be from this heap. If the heap is full, it returns an out of memory error.

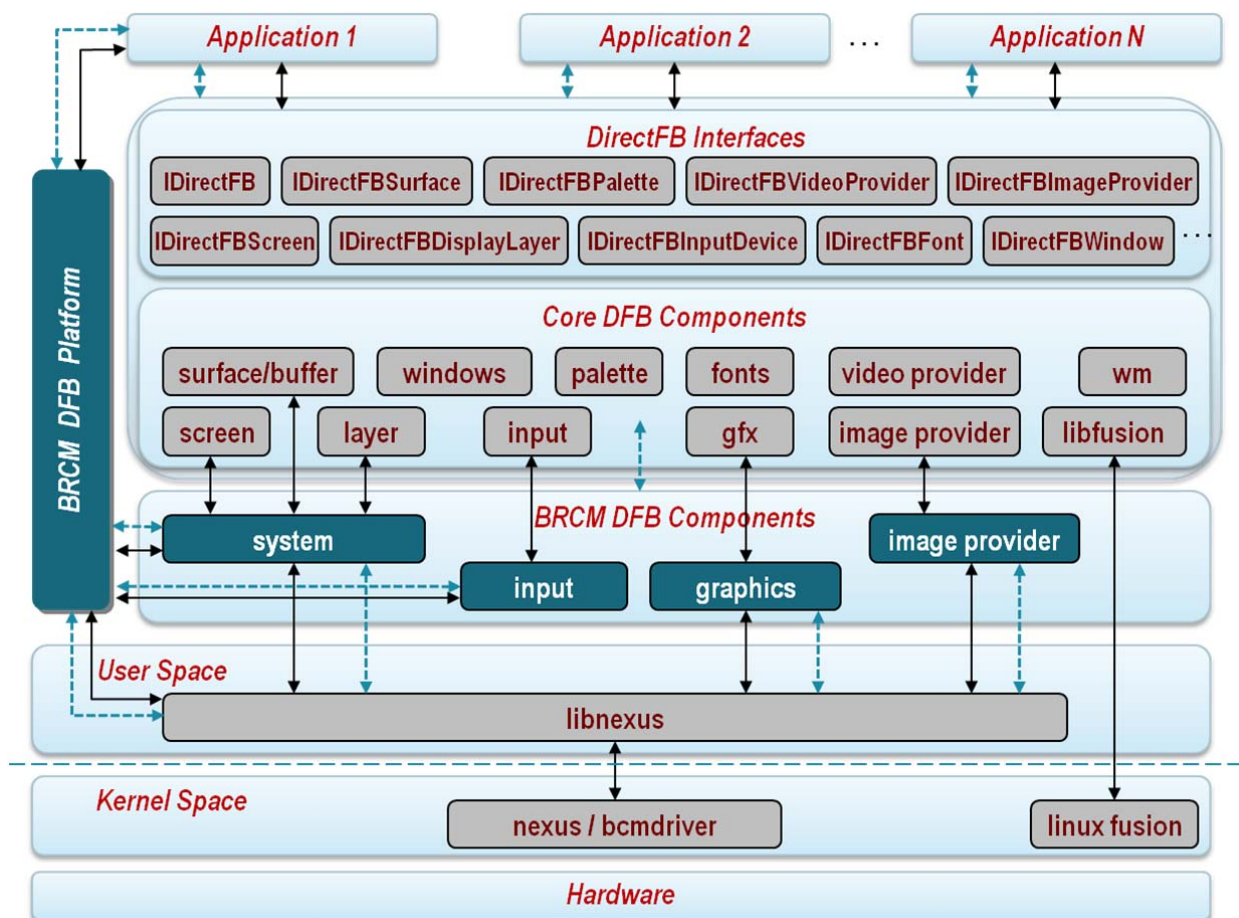
For a DirectFB master application if the offscreen heap is full, we are able attempt to allocate from the primary heap (2), but not for a slave as there is no virtual memory mapping in the slave applications address space. For security reasons we try and make sure that the slave apps have no access to hardware or display buffers to reduce the chance of a malicious slave app causing the system any damage or hijacking the display. If you want to sandbox the slave applications memory usage you could create a separate heap inside Nexus and return this handle via "NEXUS_Platform_GetFramebufferHeap(NEXUS_OFFSCREEN_SURFACE)". This way the slave applications would have the most limited access to any memory from which they could affect other system components.

Changes to DirectFB-1.4.17

This section lists the changes and additions that Broadcom has made to DirectFB-1.4.17 when compared with the stock DirectFB-1.4.17 release (from <http://www.directfb.org>).

The diagram below helps to show the overall DirectFB software architecture, with Broadcom proprietary DFB software components highlighted in dark blue. These components in dark blue are all shared libraries that are dynamically loaded into the system at run-time. They are dynamically linked with the rest of DirectFB to provide additional features and performance improvements. Only the Broadcom DirectFB platform shared library is accessible to applications. All the remaining Broadcom DirectFB components are not directly accessible by applications, but instead are accessible through the DirectFB API (interfaces).

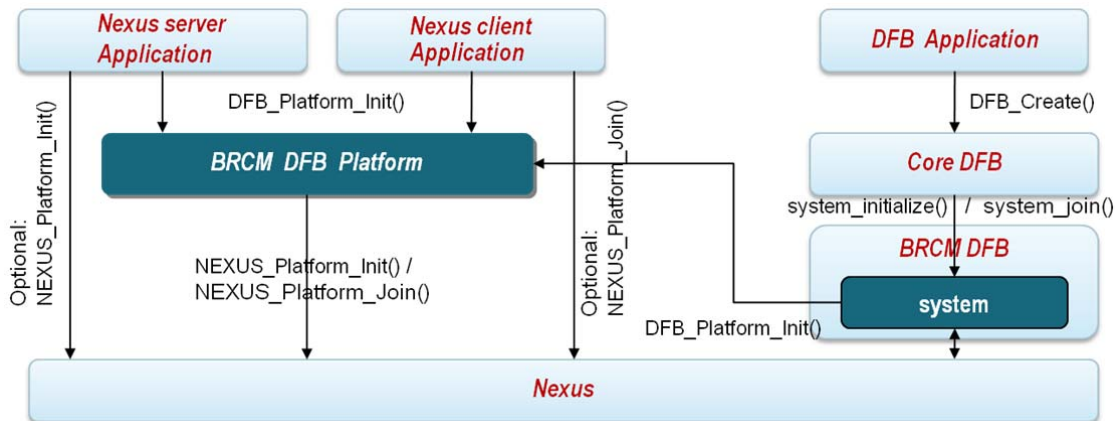
All these Broadcom proprietary software components rely on “Nexus” and is accessible through a shared library (“libnexus”). Nexus provides access to the underlying hardware through a well-defined API.



Platform library usage for standard DirectFB

Broadcom provides a DirectFB platform software library that is used to help initialize/join Nexus (and its required modules) and to support sharing of resources between DirectFB and non-DirectFB applications. This software library has its own light-weight API that is not part of the original DirectFB API. DirectFB applications can run without modification, but non-DirectFB applications (e.g. Nexus applications) must use this API to keep Nexus and DirectFB synchronized with each other.

Non-DirectFB applications must link against this shared library (libinit.so or libinit_client.so) and must call “*DFB_Platform_GetDefaultSettings()*” followed by “*DFB_Platform_Init()*”. DirectFB applications themselves need not be aware of this platform library, as the Broadcom system driver will automatically call these functions during the system initialization stage. The diagram below helps to illustrate these concepts:



Platform library usage for DirectFB-XS

The Broadcom DirectFB platform software library is also used when DirectFB is built to use the Nexus surface compositor. This is also known as “DirectFB-XS” in Broadcom terminology. In this case, the platform library is used to “join” Nexus, as all DirectFB applications must be run as Nexus client applications. The platform library is also used to open certain Nexus modules (e.g. graphics2d, graphics3d, picture decoder), but the handle for each of these modules is not shared between different processes like they are for standard DirectFB. DirectFB-XS also does not open the Nexus display, but instead acquires an instance of a Nexus surface compositor client interface. This interface is used to provide a “virtual framebuffer” for DirectFB-XS to render in to. The platform library also does not open any Nexus display windows or outputs. These interfaces are opened and configured in the Nexus server application.

Standard DirectFB applications can run unmodified when DirectFB-XS is built. These applications will all run as Nexus clients. Non-DirectFB applications no longer need to call the DirectFB platform

layer API, as no sharing of handles or resources is permitted (or enabled). The DirectFB platform layer adds no additional functionality.

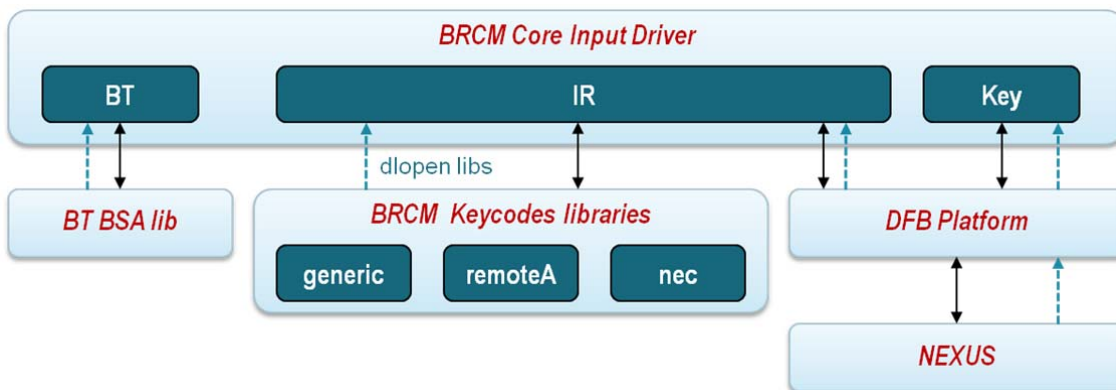
Graphics driver

Broadcom provides a highly optimized 2D and 3D graphics driver that will accelerate most drawing and blitting operations. This graphics driver will accelerate 2D operations when the underlying hardware is using the M2MC core (blitter). When using the M2MC core, the driver defaults to using a packet-buffer based API to gain maximum performance. However, the legacy NEXUS_Graphics2D_xxx() API can still be enabled, but the graphics performance won't be as high as for the packet-buffer implementation.

For chips that have a PX3D 3D graphics core (e.g. BCM7420, BCM7413), 3D transformations like texture mapping, triangle fills, rotation of primitives, etc. are hardware accelerated. For chips that don't have this 3D graphics card, software fall back operations are provided. For chips that have the VC-4 graphics core, customers are encouraged to use the OpenGL ES 2.0 or OpenGL ES 1.1 API's to provide these types of operations.

IR and front panel drivers

Broadcom provides infra-red (IR), and keypad (front-panel) drivers to allow interaction with DirectFB applications. These input device drivers can be optionally compiled in during the build process. The diagram below helps to illustrate the software architecture.



The IR input driver allows run-time selection of the IR protocol and IR keycodes mapping file. The keycodes mapping file is used to convert from native IR command codes to DirectFB input event codes. The default IR protocol and keycodes mapping file are defined in the build system, but the user can override this default at compile-time and/or run-time. This means that it is possible to change the protocol or keycodes mapping file used at run-time without having to recompile DirectFB.

The run-time selection is available with the following DirectFB config options:

```
bcmnexus-ir-protocol
bcmnexus-ir-keycodes
```

By default, the “CirNec” keycodes and “CirNec” IR protocol are used that support the Broadcom silver handset. Previous DirectFB releases supported the “RemoteA” One-For-All handset or black slim handset by default.

If the user would rather use this older remote control handset to control STBs, then the “RemoteA” IR protocol and keycodes file can be specified at run-time as follows:

```
./rundfb.sh df_input --dfb:bcmnexus-ir-protocol=RemoteA,bcmnexus-ir-  
keycodes=RemoteA
```

If DirectFB-XS is being used, then the run time option cannot be used. In this case, keyboard and ir protocols can be specified in the directfb_common.inc file found at ‘opensource/directfb/build’ directory or as a compile time option (see below).

The choice of the IR protocol name can be found in the “DirectFB-Broadcom/inputdrivers/bcmnexus/core/bcmnexus_ir_inputmode.h” header file. This is an auto-generated header file that extracts the name of the input modes from Nexus. The choice of IR keycode mapping file can be found in what keycodes modules are built and present in the “/usr/local/lib/directfb-1.4-17/inputdrivers/bcmnexus” target directory. For example, if “libdirectfb_bcmnexus_ir_keycodes_cirnec.so” is present, then “cirnec” can be specified as the keycodes mapping file at run-time.

If the user wishes to support a different IR protocol or handset, then a new keycodes mapping file will have to be created and added to the DirectFB build system. If the user would like to override the default IR protocol and keycodes file at build time, then the following environment variables can be set to override the defaults:

```
DIRECTFB_IR_PROTOCOL=xxxxxx  
DIRECTFB_IR_KEYCODES=yyyyyy
```

The front panel and IR receiver drivers have been designed to mimic the behaviour of a keyboard by default. This means that if the user presses a key, a single DIET_KEYPRESSED event is generated and when the key is released a single DIET_KEYRELEASED event is generated. If the key is held down for longer than the “skip” count, single DIET_KEYPRESSED events are generated with the DIET_REPEAT flag set.

If the user would like to revert to using the original mechanism whereby both DIET_KEYPRESSED and DIET_KEYRELEASED events are generated together, then the following runtime DirectFB options can be specified in the directfbrc file (or set in the DFBARGS envvar):

```
bcmnexus-ir-timeout=0  
bcmnexus-key-timeout=0
```

The list of all IR and KEYPAD options can be found at run-time by using the help option:

```
e.g. ./rundfb.sh <app> --dfb-help
```

For example, the IR repeat filter time can be specified with the following run-time option:

```
bcmnexus-ir-repeat-time=xxx
```

DirectFB Nexus input router

DirectFB 1.4.17 version 1.1 onwards features Nexus Input Router (NIR) as an additional input driver. NIR allows a server process to capture various input events such as IR, keyboard, and mouse and send them via IPC to client applications. This allows multiple clients to share system inputs without contention over the actual hardware devices.

This feature can be used when DirectFB Nexus Surface Compositor (NSC) support is enabled. To enable DirectFB NIR support, following build flags need to be set.

```
DIRECTFB_NSC_SUPPORT=y  
DIRECTFB_NIR_SUPPORT=y
```

Enabling DirectFB NIR support suppresses run time loading of the linux_input input driver by adding an entry during build in directfbrc file as “disable-module=linux_input”.

DirectFB NIR feature requires Nexus server input router application running in the background. The server NIR application captures various input events like keypad, keyboard and mouse then sends them via IPC to client applications. Client NIR application then registers for desired input events and receives them via IPC using Nexus APIs. DirectFB NIR client side code can be found in platform_nexus_input_router.c in DirectFB Broadcom package in platform directory.

An example NIR server application nxsmaster is provided in DirectFB-Broadcom package. nxsmaster app gets built when DIRECTFB_NSC_SUPPORT is set to “y” during the DirectFB build.

To test DirectFB NIR input driver, DirectFB example application df_input.c can be used. This can be tested by first starting the nxsmaster server app as:

```
./rundfb.sh nxsmaster
```

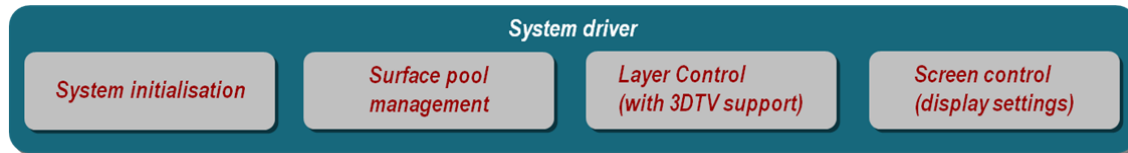
Then launch the dfb_input DirectFB example app from separate window/terminal as:

```
./rundfb.sh join df_input
```

For more details on Nexus Input Router server and client applications, refer to the Nexus apps input_router.c and input_client.c in Nexus/examples/multiprocess. Details on how to build these applications can be found in “Nexus_Usage.pdf” as part of Reference Software Release.

System driver


Broadcom provides what is known as a “system” driver (shared library) to help initialize the system, control the layer(s) (framebuffers), manage memory and surface buffers, setup up the display pipeline (e.g. setting correct video format) and allow different outputs to be added or removed from a given screen/display. The diagram below helps to depict these key responsibilities.



The system driver is multi-process aware and has the responsibility to marshal memory allocation/deallocation requests, surface buffer creation/destruction, setting of the graphics framebuffer(s) and setting of display/screen settings (for standard DirectFB mode) from client/slave DirectFB applications to the master DirectFB application using fusion IPC. The system driver also parses all Broadcom specific application options passed in from the command line, DFBARGS envvar or directfbrc file.

For standard DirectFB (non DirectFB-XS) builds, Broadcom supports the DirectFB Screen Encoder API and Mixer API to allow a master DirectFB application to setup the video output resolution, frequency, scan-mode, background colour, etc. The default start up video output resolution is specified in the “/usr/local/etc/directfbrc” file with the “res=” run-time option. The default is 720p/60Hz, but the user can override this in a number of ways:

1. Modify the directfbrc file.
2. Specify “res=xxx” on the command line (e.g. `./rundfb.sh df_andi --dfb:res=720p50`).
3. Specify the “res=xxx” option in the DFBARGS envvar (e.g. `export DFBARGS="res=720p50"`).

 The complete list of resolutions is available if you type:

```
“./rundfb.sh df_andi -dfb:help”.
```

For standard DirectFB mode, the layer handling code supports both mono and stereo surface layer buffers. The latter is required when 3DTV stereoscopic graphics are enabled. For chipsets that don't have 3DTV capable video processing, the left/right buffers are provided to the display in "top/bottom half" or "left/right half" orientation. For newer STB chipsets that have 3DTV capable video processing, the left/right buffers can be provided to the display hardware in additional formats such as frame-packed. There are new 3DTV DirectFB API's that Broadcom has provided to directfb.org and have been incorporated into the DirectFB-1.5.x series. Broadcom have back-ported these changes to DirectFB-1.4.17. Refer to DirectFB-1.5.x API's on the DirectFB website (http://directfb.org/docs/DirectFB_Reference_1_5/index.html) for more information.

ImageProvider driver

Broadcom provides a hardware accelerated ImageProvider based on the Nexus Picture Decoder. This can decode GIF, JPEG and PNG images using the underlying Still Image Decoder (SID) hardware.

The ImageProvider() driver for Nexus uses the "Picture Decoder" API to decode and render an image into a temporary buffer before being blit (or stretch blit) to the final destination surface. This last blitting operation is done using the internal DirectFB API's and does not directly use the Nexus graphics2d or packet-buffer API directly. This final blit also provides any colour format conversion between the typical YUV output from the hardware and the final destination surface's format.

If the image cannot be decoded and rendered (e.g. SID hardware doesn't support the image), then the software image providers will render the image to the final destination surface.

The ImageProvider driver for Nexus now supports segmented and streaming decoding. This is useful when trying to decode and display large sized images that would otherwise require large amounts of memory.

Core changes

Broadcom has also fixed bugs and made modifications to the core DirectFB codebase. Below is a list of changes that have been made against the stock DirectFB-1.4.17 release tarball.

New pixel formats

1. Support for “DSPF_ABGR” pixel format needed to support chips with the VC-4 3D graphics core.
2. Support for “DSPF_ALUT8” pixel format needed to support the output of the picture decoder for indexed pixel formats.
3. Support for “DSPF_LUT4” needed for some chipsets that have a reduced graphics feeder pixel format capability.

3DTV stereoscopic support

The core DirectFB code that handles layers, windows, regions and surfaces has been updated to understand stereoscopic graphics. New API's have been added to DirectFB to allow the client application to specify options like stereo depth and which “eye” the rendering should be made to. SaWMan-1.5.4 has support for the stereoscopic API's and it the version that should be used in conjunction with this release.

Colour-space support

DirectFB surfaces now have a colour-space associated with them (e.g. RGB or BT.601). It is now possible to convert between different colour-space formats (e.g. BT.601 and BT.709) when blitting between these surfaces. The call to create a surface now accepts a colour-space argument.

Screen changes

1. Ability to specify the picture framing with the screen encoder API (good for 3DTV modes).
2. Ability to specify the aspect ratio for the screen.

DirectFB unit tests

The following additional tests applications have been modified / provided:

1. **df_andi**: with modifications to allow the user to toggle power mode (press “P”) and/or cycle around different video output formats (press “O”).
2. **df_andi3d**: 3DTV stereoscopic graphics tests (“3D penguins”).

3. **df_stereo3d**: enhanced 3DTV stereoscopic graphics test.
4. **dfbtest_stereo**: a 3DTV stereoscopic full-screen test app.
5. **dfbtest_stereo_window**: a 3DTV stereoscopic window test app.
6. **df_texture3d**: 3DTV stereoscopic version of “df_texture”.
7. **df_brcmTest**: graphics conformance test comparing h/w vs. s/w blits and fill operations.
8. **df_dok**: benchmark test that has been modified to support flipping, fill triangles and fill trapezoid test cases.
9. **df_flip**: this test has been created to check whether flipping a surface is VSync locked or not.
10. **df_input**: is used to test the various input devices (e.g. IR or USB keyboard).
11. **df_texture3d**: is a 3DTV stereoscopic version of texture mapping.
12. **dfbtest_layer**: is used to test layer repositioning and opacity change.
13. **dfbtest_gl**: is only available on PX3D capable chips and tests the OpenGL ES 1.0 API.
14. **dfbtest_egl_only**: is only available on PX3D capable chips and test the EGL API.
15. **dfbtest_egl_pixmap**: is only available on PX3D capable chips and tests the EGL API.
16. **dfbtest_alloc**: is used to test memory surface allocations/deallocations.
17. **dfbtest_prealloc**: is used to test pre-allocation of system or video memory.
18. **dfbtest_resize**: is used to test resizing of surfaces.
19. **df_window_prealloc**: used to test system and video memory pre-allocated surfaces.
20. **decode_server_dfb**: this is a Nexus server/DirectFB master based application available for standard DirectFB builds that allows a video/audio playback client to be attached
21. **decode_client_dfb**: this is a Nexus client based application that allows video/audio content to be played back from a transport stream file. It connects to the already running decode_server_dfb application and is only available for standard DirectFB builds.
22. **playback_dfb**: this is a Nexus master/DFB master based video/audio playback application (standalone) that is only available for standard DirectFB builds.
23. **nxsmaster**: this is a Nexus master server application only available for DirectFB-XS master library builds. It is the first application that should be launched before any further DirectFB-XS or non-DirectFB-XS applications are started.

24. **df_screen_encoder**: this example shows how to connect and disconnect various output connectors from the primary and secondary displays.

Graphics changes

1. Support triple-buffered pre-allocated surfaces from video memory.
2. Added “dfb_gfx_clear()” function to clear a surface buffer. This is used to clear a layer surface buffer after it has been allocated.
3. Support for drawing multiple triangles in one operation (“FillTriangles()”).
4. Ability to dump out an image in raw format.
5. Support for setting the texture surface within the IDirectFBGL API.
6. Support for colour-space conversion within the core graphics code.
7. Performance optimizations to the generic software-based graphics driver.
8. Ability to obtain the size and constraints from the graphics card when allocating a buffer.

Image provide changes

1. Changes to software based image providers to use the hardware image provider if available (and enabled) instead.
2. Added support for FFMPEG based image provider (for displaying MPEG-1/2 I-frames and H.264/AVC I/IDR still pictures).

Font changes

1. Support for italics, reverse italics and bold font attributes.
2. Support for ABGR font pixel-format.

Build system

1. Many compiler warnings have been removed from the code.
2. The build system now uses “silent make” and shows a “CC” or “CCLD” when a file is being compiled or linked rather than the verbose compilation information. This makes it much easier to spot errors or warnings.
3. Added the ability to link DirectFB with external shared libraries (e.g. zlib, libpng, libjpeg, freetype) rather than statically compile them in (saves space).

4. Added support for setting the vendor version string as part of the DirectFB version information.

Input devices

1. Input Device fixes to allow the GetLockState() API to work correctly in the context of a DFB slave application.
2. Support hot-plugging of USB keyboard/mice properly (hot-plug thread now exits).
3. Support for additional picture-in-picture (PIP) input key types.

Other

1. Surface hints have been added during the creation of a surface.
2. Surface memory permissions have been added to help improve security.
3. A new “PreAlloc()” surface pool function has been added to allow pre-allocation of video and system memory to work reliably in the context of DFB slave applications.
4. Triple and double buffered window surfaces can now be flipped without artefacts or tearing.
5. Fusion updated from 8.8.0 to 8.10.4.
6. Fusion main pool area increased to 16MB to allow more DFB surfaces to be created without running out of memory in the main pool.

Public API changes to DirectFB-1.4.17

Table 6: Public Function API changes

Function	Change
GetStereoDepth()	Used on the display layer or window to retrieve the current stereo depth between the left/right eyes.
SetStereoDepth()	Used to shift the left and right eyes in opposite directions horizontally to provide a 3DTV stereoscopic effect on the layer or window.
GetStereoEye()	Only applicable to layer/window surfaces that have a stereo pair on underlying buffers. This API will retrieve which set of buffers (left or right) is active for graphics operations on this surface.
SetStereoEye()	Only applicable to layer/window surfaces that have a stereo pair on underlying buffers. This API will specify which set of buffers (left or right) is used for future graphics operations on this surface.
FlipStereo()	Flip/update stereo surface buffers simultaneously to ensure synchronisation between the two left/right buffers. This API is only applicable to surfaces that have a stereo pair of underlying buffers.
DumpRaw()	The API will dump the raw contents of the specified surface to a file.

Table 7: Public definition API changes

Definition	Change
DFB_NOALLOCATION	Error to signify that a buffer allocation could not succeed.
DSPF_ABGR	This new pixel format is necessary to support VC-4's texture pixel output format.
DSPF_LUT4	Provides 16 entry palette pixel format.
DSPF_ALUT8	This new format is necessary to support the still image decoder's output.
DLCAPS_LR_MONO	Specifies that the layer can support a L/R mono stereoscopic display.
DLCAPS_STEREO	Specifies that the layer can support an independent L/R stereoscopic display.
DLCAPS_FOLLOW_VIDEO	Used to signal that video metadata can be used to specify the z-depth on layers with DLOP_LR_MONO or DLOP_STEREO.
DLOP_LR_MONO	Specifies that a layer has a single set of surface buffers and a stereo depth.

Definition	Change
DLOP_STEREO	Specifies that a layer has two independent surface buffers (left/right) with unique content.
DSDDESC_COLORSPACE	Specifies that the colour space field for the surface description is valid.
DSCAPS_GL	Specifies that the surface data is stored in memory that is accessible by a GL (e.g. OpenGL / OpenVG) accelerator.
DSCAPS_STEREO	Specifies that the surface contains both left/right buffers.
DFXL_FILLTRIANGLES	Signals that the “ <i>FillTriangles()</i> ” API is accelerated.
DWDESC_COLORSPACE	Specifies that the window description’s colour space field is valid.
DWCAPS_LR_MONO	Specifies that a window has a single set of surface buffers and a stereo depth.
DWCAPS_STEREO	Specifies that a window has two independent left/right buffers each with unique content.
DFFA_ITALIC	Specifies that the Font should be displayed with italics.
DFFA_REVERSE_ITALIC	Specifies that the Font should be displayed with reverse italics.
DFFA_BOLD	Specifies that the Font should be displayed in Bold.
DFDESC_RESOURCE_ID	Provides a unique resource ID for the font.
DLCONF_COLORSPACE	Specifies that the layer colour-space can be set.
DLSO_FIXED_LIMIT	Specifies the absolute maximum positive or negative value used for setting the stereo depth on a layer
DSECAPS_FRAMING	Allows the HDMI picture framing to be specified in the encoder config.
DSECAPS_ASPECT_RATIO	Allows the display aspect ratio to be specified.
DFBScreenEncoderPictureFraming	New structure that specifies the picture framing that can be sent to the screen encoder.
DSEPF_MONO	Screen encoder picture framing is non-stereoscopic.
DSEPF_STEREO_SIDE_BY_SIDE_HALF	Also known as “L/R” frame packed mode. The left and right stereo pair images are decimated by 50% and placed side-by-side in an existing frame and output every VSync period.
DSEPF_STEREO_TOP_AND_BOTTOM	Also known as “Over-Under” frame packed mode. The left and right stereo pair images are decimated vertically

Definition	Change
	by 50% and placed in an existing frame that is output every VSync.
DSEPF_STEREO_FRAME_PACKING	The left/right images are output sequentially at full-resolution This stereoscopic mode is only available for HDMI 1.4a capable STB platforms and TVs.
DSEPF_STEREO_SIDE_BY_SIDE_FULL	The left/right images are output at full resolution, but are packed in a frame that is twice the width of a single frame. This stereoscopic mode is only available for HDMI 1.4a capable STB platforms and TVs.
DFBDisplayAspectRatio	New structure that specified the aspect ratio of the screen.
DSECONF_ASPECT_RATIO	Specifies that the aspect ratio should be sent for the screen set encode configuration.
DSECONF_FRAMING	Specifies that the picture framing should be sent for the screen set encoder configuration.
DFBSurfaceStereoEye	Enumeration for left/right stereo eye buffer.
DSSE_LEFT	Specifies that the left eye buffer should be used for all subsequent graphics operations on this surface.
DSSE_RIGHT	Specifies that the right eye buffer should be used for all subsequent graphics operations on this surface.
DWSO_FIXED_LIMIT	Specifies the absolute maximum positive or negative value used for setting the stereo depth on a window.

Testing DirectFB

Testing the IR input

The current DirectFB IR driver supports the Broadcom silver remote control by default (NEC protocol).

The best application to test the IR remote control is “df_input”. You can run this test application by entering in the following command:

```
cd /usr/local/bin/directfb/1.4
./rundfb.sh df_input
```

Then you can press any key on the handset and you should see the name of the key and key code displayed on the display. If the key is held down on the remote control, the repeat event should be set.

Testing the front panel input

Before the front panel input can be tested, DirectFB needs to be built with the keypad driver enabled. This can be achieved by ensuring that “DIRECTFB_KEY_INPUT” environment variable is set.

```
e.g. export DIRECTFB_KEY_INPUT=y
```

Secondly, there is a known issue with the Nexus/magnum drivers that requires the LED controller to be initialized first. Instead of placing the burden on DirectFB to initialize the LED controller (which could interfere with an application that already opens the LED controller), it was decided instead to wait for the drivers to rectify this initialization problem. As a result, it is necessary to run a test application first to initialize the front panel LED controller prior to running any DirectFB test application that requires front panel input control. The nexus “frontpanel” example application is recommended to be run first prior to running the require DirectFB application.

Example:

```
./nexus frontpanel
./rundfb.sh df_input
```



This assumes you have already compiled the Nexus example applications and have copied the executables and “nexus” script to “/usr/local/bin/directfb/1.4”. If you are unsure of how to compile the Nexus example applications, then refer to the document “Nexus_Usage.pdf” that should be part of the reference software release.

Testing different blitting and drawing modes


There is a specific test called “df_brcmTest” that runs through many different blitting/blending and drawing operations with the hardware acceleration output displayed in a window on the left-hand side the screen and the software fall back mechanism displayed in its own window on the right-hand side the screen (side-by-side for comparison). The user simply needs to press the <OK> or <SELECT> key on the IR handset to progress through the different tests.

The test also accepts setting the “blittingflags” and “drawingflags” environment variables to test additional blitting and drawing scenarios (such as Destination colour keying). For example, to test source colour keying on all blitting/blending test cases, you need to set the “blittingflags” environment variable as follows (prior to running the test).

```
export blittingflags=0x08
```

To test destination colour keying for all blitting test cases, you need to set the environment variable as follows:

```
export blittingflags=0x10
```

 These values are determined by looking at the “DirectFB-1.4.17/include/directfb.h” header file and reviewing the “DFBSurfaceBlittingFlags” typedef.

The test defaults to an ARGB pixel format for the graphics layer/plane, but any valid pixel format can be set by modifying the df_brcmTest.c file in the “DirectFB-1.4.17/tools” directory and setting the following define to the required format:

```
#define PRIMARY_PIXELFORMAT    DSPF_XXXX
```



Where: XXXX is one of the pixel formats as defined in “DirectFB-1.4.17/include/directfb.h”.

Performance tests

Both “df_andi” (Penguins) and “df_dok” are good benchmarking tests to check the overall graphics performance of the target platform.

“df_andi” measures real-world blitting performance by seeing how many penguin blits can be sustained at a given number of frames per second. The graphics performance is proportional to the number of penguins and fps (frames per second) displayed at the top left-hand corner of the screen. The number of penguins can be increased by pressing the **<S>** key on the target platform’s USB keyboard. Also, the number of penguins can be decreased by pressing the **<D>** key on the keyboard. Pressing the **<SPACE>** bar will cause the penguins to form a logo and pressing **<R>** will cause them to revert to moving around the screen. Pressing **<P>** will power up/down the screen. Pressing **<O>** will cause the output resolution to change. Pressing **<M>** will toggle mirroring of the primary graphics frame-buffer to the secondary display. Pressing **<Q>** will quit the application.

“df_dok” is a true benchmarking test that measures CPU load and graphics blitting / drawing performance. It shows the CPU load in square brackets along with a print out of the number of graphics operations per second of each test (e.g. Mpixels/s, Kchars/s).

-  There will be a significant difference in CPU load when running DirectFB in release mode vs. debug mode. To obtain the best results, always build DirectFB with “B_REFSW_DEBUG=n” (release mode).
-  Building the Nexus and magnum drivers in release mode (B_REFSW_DEBUG=n) will also provide a significant increase in some hardware accelerated operations.

Supported platforms

The table below lists the Set-Top platforms that this release of DirectFB supports.

Table 8: Supported platforms

Platform	Comments
BCM97231	OpenGL ES2.0 support (VC4) / Stereoscopic hardware support
BCM97241	OpenGL ES2.0 support (VC4) / Stereoscopic hardware support
BCM97346	OpenGL ES2.0 support (VC4) / Stereoscopic hardware support
BCM97358	No OpenGL 3D support / Stereoscopic hardware support
BCM97409	OpenGL ES1.0 support (PX3D)
BCM97420	OpenGL ES1.0 support (PX3D)
BCM97425	OpenGL ES2.0 support (VC4) / Stereoscopic hardware support
BCM97429	OpenGL ES2.0 support (VC4) / Stereoscopic hardware support
BCM97435	OpenGL ES2.0 support (VC4) / Stereoscopic hardware support
BCM97552	No OpenGL 3D support / Stereoscopic hardware support

Frequently asked questions (FAQ)

How do I enable debugging on a per-module basis in DirectFB?

To enable debugging for “module_identifier”, you can do it 3 different ways:

1. On the target platform export DFBARGS like this:

```
export DFBARGS="debug=module_identifier,debug=module2_identifier"
```

2. Edit “/usr/local/etc/directfbrc” and add the lines:

```
debug=module_identifier debug=module2_identifier ...
```

3. Pass the debug options on the command line:

```
./rundfb.sh df_dok --dfb:debug=module_identifier,debug=module2_identifier
```

For the DirectFB platform layer the Nexus / Magnum debug system is used. The exact syntax used varies depending on whether you are using kernel or user space Nexus drivers.

Here is an example of how to set some variables in Linux user mode:

```
export msg_modules=platform_nexus_init
```

In the kernel mode configuration, variables can be passed in using the configuration environment variable, using the following syntax:

```
export config="msg_modules=platform_nexus_init"
```

The config variable should then be passed to the nexus module when it is being inserted.

```
e.g. insmod nexus.ko config=$config
```

How do I enable back-tracing in DirectFB?

One other useful debug tool is to enable a back-trace whenever DirectFB receives a signal. Add the option "**TRACE=y**" to you make command. This is also the option that should be set when using gdb to help debug DirectFB issues.

How can I disable hardware acceleration and use the generic DirectFB software graphics functions instead?

You can pass the "no-hardware" option to DirectFB like this for example:

```
e.g. export DFBARGS="no-hardware"
```

This is useful if you are unsure whether the hardware accelerated version is doing the correct graphics operation, or you want to measure the performance with and without hardware acceleration.

How can I tell what size surfaces are being created? Why can't I see memory for my surface being allocated on creation?

DirectFB implements a lazy surface allocation scheme. The Broadcom DirectFB system driver uses NEXUS_Surface_Create() to create a buffer that is associated with each DirectFB surface. This buffer is normally only allocated at the very first Lock() call on the surface. This Lock() call can be made either by the CPU explicitly or internally by the GPU/blitter. When a surface is first locked, a buffer will be allocated for it and the code to do this is in bcmnexus_pool.c. To see the size of the buffer (Nexus surface), you can enable debugging for the bcmNexus/Pool module identifier.

```
e.g. export DFBARGS="debug=bcmNexus/Pool"
```

Blending multiple windows together doesn't look right - why?

Have a look at the following twiki page to better understand how your application should use Porter-Duff and Blitting flag calls to blend multiple surfaces together:

http://directfb.org/wiki/index.php/Blending_HOWTO

How do I change the cursor in DirectFB?

The best method is to load in an image and then set the cursor using the layer or window "SetCursorShape()" function.

The sample code below presumes you want to use a PNG file called cursor.png from "/usr/local/share/directfb-1.4.17/images" which is 32x32 pixels.

```
IDirectFBDisplayLayer *layer;
IDirectFBSurface      *cursurface;
IDirectFBImageProvider *provider;
DFBSurfaceDescription desc;

DFB_CHECK(dfb->CreateImageProvider(dfb,
                                   DATADIR"/cursor.png",
                                   &provider));
```

```
desc.flags = DSDESC_WIDTH | DSDESC_HEIGHT | DSDESC_CAPS;
desc.width = 32;
desc.height = 32;
desc.caps = DSCAPS_NONE;

DFB_CHECK(dfb->CreateSurface( dfb, &desc, &cursurface ) );

DFB_CHECK(provider->RenderTo( provider, cursurface, NULL ) );
provider->Release( provider );

DFB_CHECK(layer->SetCursorShape( layer, cursurface, 0, 0 ) );
```

When using the SaWMan window manager, you can have different cursors for different windows, see `df_window.c` for an example.