# Xtensa Dynamic Library Loader

* [1. Introduction](#XtensaDynamicLibraryLoader-Introduction)
  + [1.1. XRP](#XtensaDynamicLibraryLoader-XRP)
  + [1.2. Multi-section Library Loader](#XtensaDynamicLibraryLoader-Multi-sectio)
  + [1.3. DMA management](#XtensaDynamicLibraryLoader-DMAmanagemen)
  + [1.4. Local Memory Manager](#XtensaDynamicLibraryLoader-LocalMemoryM)
* [2. Dynamically loading libraries](#XtensaDynamicLibraryLoader-Dynamicallyl)
  + [2.1. Types defined by the loader library](#XtensaDynamicLibraryLoader-Typesdefined)
    - [2.1.1. xdyn\_lib\_status\_t](#XtensaDynamicLibraryLoader-xdyn_lib_sta)
    - [2.1.2. xdyn\_lib\_funcs\_t](#XtensaDynamicLibraryLoader-xdyn_lib_fun)
  + [2.2. DSP](#XtensaDynamicLibraryLoader-DSP)
  + [2.3. Structure of a Dynamically Loadable Library](#XtensaDynamicLibraryLoader-Structureofa)
  + [2.4. Host](#XtensaDynamicLibraryLoader-Host)
  + [2.5. Dynamic loading Internals](#XtensaDynamicLibraryLoader-Dynamicloadi)
* [3. Building Dynamic Library Loader](#XtensaDynamicLibraryLoader-BuildingDyna)
* [4. Usage](#XtensaDynamicLibraryLoader-Usage)
* [5. Open Questions](#XtensaDynamicLibraryLoader-OpenQuestion)
* [6. References](#XtensaDynamicLibraryLoader-References)

# 1. Introduction

Xtensa DSPs are often used as off-load engines controlled by a host processor. The main driver application is run on the host and any computationally intensive parts are off-loaded to the DSP. DSPs are commonly programmed either using the built-in intrinsics or using high level frameworks like OpenCL, OpenCV, Halide, Android Neural Network (ANN), or the Xtensa Neural Network Compiler (XNNC). In order to offload these applications from the host, the host application needs to

* load the DSP with the target library that implements the kernels to be offloaded
* issue commands with relevant parameters to invoke the kernels on the DSP

The DSP is usually shared across multiple applications running on the host. To support this, the host needs the ability to load and unload DSP libraries on demand. This allows building modular libraries each targeted for different sets of applications rather than a monolithic binary that implements all possible functionalities.

The DSP kernels commonly use shared resources such as tightly-coupled local memories and the iDMA engine. Software frameworks that manage these shared resources has to be shared across all such libraries so that they can be reused without having to link them separately with the target library.

Xtensa software libraries provide solutions to address most of the above issues. Below briefly describes some of these.

## 1.1. XRP

XRP[1] provides a low-level transport framework to execute kernels remotely on the DSP and manage shared memory between the host and one or more DSPS. The target DSP binary has to be pre-loaded before any commands are enqueued. If another application wants to use the DSP, the DSP has to be reloaded with the target specific binary associated with that application. XRP provides namespaces that allows each application to execute commands targeted to the specific namespace.

## 1.2. Multi-section Library Loader

Xtensa provides a mechanism to build the DSP application as a position-independent library that can be loaded and unloaded on demand by the DSP[2]. The library is required to have a single entry point named \_start. The library is expected to be self-contained, i.e. not have any references to routines external to the library. To call functions defined in any external libraries, references to such functions have to be explicitly passed to the library. Similarly, if the library needs to export any functions callable from the outside, references to such functions have to be explicitly returned to the caller. The library can have dedicated sections that allow any library specific data/code sections to be placed in the DSP's local memories, heap, or the L2-RAM. If the library references functions that are not provided externally, they would need to be explicitly linked into the library (eg: libc, libm).

## 1.3. DMA management

The iDMA library[3] provides functions to manage the iDMA engine including scheduling descriptors, checking status of the queued descriptors, etc. Since iDMA is a shared resource, the iDMA library needs to be shared across all target libraries.

## 1.4. Local Memory Manager

Local memory is a shared resource that needs to be shared across multiple target libraries. For multiple target applications to co-exist, the local memory has to be managed by a common manager that is responsible for allocating and freeing the memory. Each application may have critical data that needs to be allocated in the local memory at load time. Similarly, when executing a kernel, the application may need to dynamically allocate data from local memory until it either terminates or explicitly yields execution.

# 2. Dynamically loading libraries

To manage dynamically loading and unloading libraries, helper loader libraries are provided for the host (libxdyn-lib-host) and the DSP (libxdyn-lib-dev). More details are provided in the following sections.

## 2.1. Types defined by the loader library

The loader library defines the below enum types:

### 2.1.1. xdyn\_lib\_status\_t

This enum is used to define the return values of the libxdyn-lib-\* library APIs. The list of all possible values for this enum are defined below:

| **Enum** | **Value** |
| --- | --- |
| XDYN\_LIB\_LOAD\_FAILED | Failed to load library |
| XDYN\_LIB\_UNLOAD\_FAILED | Failed to unload library |
| XDYN\_LIB\_VERSION\_MISMATCH | Loader version mismatch |
| XDYN\_LIB\_UNKNOWN\_CMD | Unrecognized command |
| XDYN\_LIB\_INIT\_FAILED | Library initialization failed |
| XDYN\_LIB\_ERR\_INTERNAL | Internal error |
| XDYN\_LIB\_OK | No error |

### 2.1.2. xdyn\_lib\_funcs\_t

This enum is used to define the functions required to be defined by the dynamically loaded library. Handles to these functions are required to be returned by the dynamic library to the library loader. The list of all possible values for this enum are defined below:

| **Enum** | **Value** |
| --- | --- |
| XDYN\_LIB\_INIT | Return the initialization function of the dynamically loaded library |
| XDYN\_LIB\_FINI | Return the finalization function of the dynamically loaded library |

## 2.2. DSP

The target library to be loaded to the device is expected to be built as a position independent packaged library (see Chapter 5, Multiple Section Library Loader, in the Xtensa System Software Reference Manual[2] for more details). The DSP boots into a base firmware that includes the main driver. The main driver is linked against shared frameworks such as XRP, multi-section library loader, iDMA library, local memory management library, etc. that are shared across all dynamically loaded libraries.

The DSP side loader library, libxdyn-lib-dev.a, automatically manages loading, unloading, and registering of the dynamic libraries. Initialization of the loader is performed from the main driver via the following API:

**xdyn\_lib\_status\_t t xdyn\_lib\_init(struct xrp\_device \*device, void \*context)**

| **Parameters** | **Description** |
| --- | --- |
| device | XRP device structure corresponding to the DSP |
| context | Generic context that is passed to the initialization function of the dynamically loaded library |
| Returns | XDYN\_LIB\_OK if successful, else returns XDYN\_LIB\_INIT\_FAILED |

The xdyn\_lib\_init function registers a system command handler using a pre-defined XRP namespace. This namespace is used by the host to send special LOAD/UNLOAD commands for loading and unloading the libraries.

## 2.3. Structure of a Dynamically Loadable Library

A dynamically loadable library has four important functions that an application programmer needs to provide

* void \*\_start(xdyn\_lib\_funcs\_t)This is the main entry point to the library. Based on the input type, the \_start function returns either a pointer to an initialization or finalization function.
* void \*init(void \*context)  
  The initialization function is called at library load time and is responsible for any initialization that the library needs to perform prior to execution. The initialization function takes as parameter the generic context that is passed to the xdyn\_lib\_init function and returns a pointer to the XRP namespace handler that is responsible for handling any XRP commands associated with this library.
* void fini(void)  
  The finalization function is called at library unload time and is responsible for any finalization that the library needs to perform prior to unloading.
* XRP namespace handler  
  This function is responsible for handling all XRP commands targeting this library.

## 2.4. Host

The host side loader library, libxdyn-lib-host.a, automatically manages loading and unloading of the libraries from the host to the target DSP. Loading the dynamic library is done via the below API:

***xdyn\_lib\_status\_t xdyn\_lib\_load(struct xrp\_device \*dev, const char \*lib, size\_t lib\_size, const unsigned char \*lib\_uuid)***

| **Parameters** | **Description** |
| --- | --- |
| device | XRP device structure corresponding to the target DSP |
| lib | Pointer to library to be loaded |
| lib\_size | Size of the library in bytes |
| lib\_uuid | UUID (16-bytes) of the library |
| Returns | XDYN\_LIB\_OK if successful, else returns one of XDYN\_LIB\_LOAD\_FAILED, XDYN\_LIB\_VERSION\_MISMATCH, XDYN\_LIB\_UNKNOWN\_CMD, or XDYN\_LIB\_ERR\_INTERNAL |

Unloading the library is done via the below API:

***xdyn\_lib\_status\_t xdyn\_lib\_unload(struct xrp\_device \*dev, const unsigned char \*lib\_uuid)***

| **Parameters** | **Description** |
| --- | --- |
| device | XRP device structure corresponding to the target DSP |
| lib\_uuid | UUID (16-bytes) of the library |
| Returns | XDYN\_LIB\_OK if successful, else returns one of XDYN\_LIB\_UNLOAD\_FAILED, XDYN\_LIB\_VERSION\_MISMATCH, XDYN\_LIB\_UNKNOWN\_CMD, or XDYN\_LIB\_ERR\_INTERNAL |

## 2.5. Dynamic loading Internals

The libxdyn-lib-dev.a loader library is responsible for loading and unloading the dynamic library on the DSP.  This library implements a special system command handler using a pre-defined dynamic loader XRP namespace. The system command handler processes two kinds of commands - LOAD and UNLOAD. The LOAD command is responsible for loading the dynamic library placed in shared memory by the host. Each dynamically loadable library is identified using a unique UUID (universally unique identifier). The library is loaded via the multi-section library loader (libmloader). Code and data sections associated with the dynamic library are first allocated. Local memory space is reserved using the local memory manager for any local data sections in the library. The \_start function is then invoked to get a reference to the dynamic library's initialization and finalization function. After initialization, the XRP namespace handler returned by the initialization function is registered to the library's unique UUID. Following which, the dynamic library is ready to handle any XRP commands targeting the library.

For unloading the library, the host sends the UNLOAD command. The system command handler calls the dynamic library's finalization function and unloads the library. Code and data sections allocated at load time are released. If the dynamic library is to be shared by multiple host applications, they all need to use the same UUID. The system command handler ensures that the library is loaded just once and unloaded only when all references to the library are done.

The libxdyn-lib-host.a loader library is responsible for loading and unloading the dynamic library from the host. The host sets up an XRP system command queue using the pre-defined dynamic loader XRP namespace. It allocates a buffer in shared memory and copies the packaged position independent library.  For each library a unique UUID is generated that identifies the library. A LOAD XRP command is issued for each core with the unique UUID and the buffer in shared memory as parameters. Once the packaged library is successfully loaded, the host can enqueue commands to the library using the UUID namespace. At the end of execution, the host issues the UNLOAD command.

# 3. Building Dynamic Library Loader

The dynamic library is provided as source under <xtensa\_tools\_root>/xtensa-elf/src/libxdynlib. From a command line shell, type ‘xt-make install’ using the Makefile provided in the directory to build the host and DSP side loader libraries. By default, this process creates a bin directory in the current directory where the libraries are built. The libraries and header files are installed in the install directory.

The BUILD\_DIR and INSTALL\_DIR variables can be used to provide alternate build and install locations. The DEBUG variable can be set to build the debug version of the library that generates detailed trace on the load/unloading process. The default builds the optimized version.

The header files containing all the libxdynlib primitive types and API declarations are defined under xdyn\_lib.h in the install directory. The following libraries are available in the install directory:

* libxdyn-lib-dev.a: DSP side dynamic library loader to be used with XTOS
* libxdyn-lib-dev-xos.a: DSP side dynamic library loader to be used with XOS
* libxdyn-lib-host.a: Host side dynamic library loader. The host side is built assuming a X86/Linux based host system. The Makefiles have to be modified appropriately if targeting a different host.

# 4. Usage

See example folder for a complete working example on how to use the xdyn\_lib API. A Makefile is provided to build the example dynamically loadable library, the host side application, and the DSP side driver code. Instructions are provided in the Makefile on how to build and run. The build process will also create a XTSC model and LSPs of a 2-core DSP subsystem with a X86/Linux host. The example provides the subsystem specification files for a 2-core reference Vision-Q7 system (XRC\_Vision\_Q7\_AO). Below briefly sketches the important parts of the example:

**Dynamically loadable library**

#include "xdyn\_lib.h"  
  
// The dynamically loadable library has to be built as a position independent packaged library  
  
// XRP handler for this library  
enum xrp\_status run\_command(void \*context, const void \*in\_data, size\_t in\_data\_size,void \*out\_data, size\_t out\_data\_size, struct xrp\_buffer\_group \*buffer\_group)   
{  
 ...  
 // Implement the actual functionality provided by this library  
 ...  
 return XRP\_STATUS\_SUCCESS;  
}  
  
// Library initialization function. The context received is same as what was passed  
// to the xdyn\_lib\_init function  
void \*lib\_init(void \*context)  
{  
 ...  
 // Returns pointer to the XRP handler for this library  
 return run\_command;  
}  
  
// Library finalization function  
void lib\_fini()   
{  
 ...  
}  
  
// Library main entry point  
void \*\_start(xdyn\_lib\_funcs\_t func)   
{  
 if (func == XDYN\_LIB\_INIT)  
 return lib\_init;  
 else if (func == XDYN\_LIB\_FINI)  
 return lib\_fini;  
 return 0;  
}

**DSP main driver**

int main(void)  
{  
 struct xrp\_device \*device;  
 enum xrp\_status status;  
 xdyn\_lib\_status\_t xdyn\_lib\_status;  
 xmem\_bank\_status\_t xb\_status;  
   
 xrp\_hw\_init();  
  
 // Initialize local memory manager. Reserve 16Kb for stack  
 xb\_status = xmem\_init\_local\_mem(XMEM\_BANK\_HEAP\_ALLOC, 16\*1024);  
  
 // Init xrp device  
 device = xrp\_open\_device(0, &status);  
  
 // Initialize dynamic library manager  
 xdyn\_lib\_status = xdyn\_lib\_init(device, &xdyn\_lib\_cb\_if);  
  
 // XRP Command processing loop  
 for (;;) {  
 status = xrp\_device\_dispatch(device);  
 if (status == XRP\_STATUS\_PENDING)  
 xrp\_hw\_wait\_device\_irq();  
 else if (status != XRP\_STATUS\_SUCCESS)  
 return status;  
 }  
  
 return 0;  
}

**Host application**

#include <uuid/uuid.h>  
#include "xrp\_api.h"  
#include "xdyn\_lib.h"

int main()  
{  
 enum xrp\_status status;  
 xdyn\_lib\_status\_t xdlib\_status;  
 struct xrp\_queue \*queue = NULL;  
 unsigned char lib\_uuid[16];  
 char \*lib = NULL;  
 size\_t lib\_size = 0;  
  
 struct xrp\_device \*device = xrp\_open\_device(0, &status);  
  
 // Generate unique UUID for the library  
 uuid\_generate(lib\_uuid);  
  
 // Load the packaged position independent library  
 int r = load\_file("device.lib.pkg", &lib, &lib\_size);  
  
 // Load library  
 xdlib\_status = xdyn\_lib\_load(device, lib, lib\_size, lib\_uuid);  
  
 // Create queue to push commands to the library  
 queue = xrp\_create\_ns\_queue(device, lib\_uuid, &status);  
  
 // Enqueue a dummy command to the dynamic library  
 int cmd = 0xdeadbeef;  
 int result;  
 xrp\_run\_command\_sync(queue, &cmd, sizeof(cmd), &result, sizeof(result), NULL, &status);  
  
 xrp\_release\_queue(queue);  
  
 // Unload library  
 xdlib\_status = xdyn\_lib\_unload(device, lib\_uuid);  
  
 xrp\_release\_device(device);  
  
 xrp\_exit();  
  
 return 0;  
}

# 5. Open Questions

1. On a multi-core system, when commands are enqueued to each of the cores, XRP does not guarantee that all commands are enqueued to all the cores atomically. It is possible that an application might enqueue a command to say the first core, possibly context switch, and another application enqueues a command to the first and second cores (assuming a two core system), before the first application has had a chance to load the second core. If the command enqueued across each core by an application are expected to work in sync, the lack of atomicity can cause a potential deadlock, where the command on the first core is waiting for its counterpart on the second core, but the second core is executing a different command.
2. The current model assumes that any user process on the host can load a library on to the DSP.  Is this ok to have? Or do we enforce some security in here. Do we allow only privileged process to do the load. If so we will have to put the library in a secure location that can only be read by a privileged user. Or should the security be at the DSP side - that is, it is ok to load any random library as long as it runs in user mode on the DSP and cannot access say another thread's data or possibly crash the DSP. Or alternately, we could 'sign' the binary that can be checked at the DSP. In any case, since we are allowing possibly arbitrary user code to be loaded, we do need some form of 'system' MMU to protect the host from a possible rogue DSP.

# 6. References

1. Xtensa Remote Processing (XRP) Library Reference Manual
2. Multiple Section Library Loader (Chapter 5) in the Xtensa System Software Reference Manual
3. The Integrated DMA Library API (Chapter 7) in the Xtensa System Software Reference Manual