AOCL Cryptography

Software Design Document

Contents

List of Figures

Part I

Introduction

Chapter 1

Preface

AOCL Cryptography library described in this document is a part of AOCL Library that provides a portable interface to cryptographic operations. The interface is designed to be user friendly, while still providing low-level primitives. This document includes: - System Architecture - High-level functional overview - Design considerations - Detailed definition of API's provided by the library

AOCL Cryptography library here in after referred to as 'the library' or crypto or cryptolib for short.

This document provides details of APIs in the following categories.

- Key Management
- Digests (One-way Hash Functions)
- Symmetric Ciphers
- Public Key Algorithms
- Message Authentication Codes (MAC)
- Key Derivation Functions (KDF)
- Random Number Generator (RNG)
- Digest Signing and Verification
- Padding

AOCL Cryptography also provides compatibility layer which translates libcrypto and IPP-CP APIs to its own.

Part II

System Overview

AOCL Cryptography is designed to be future compatible and extendable. AOCL-Cryptography library has following components.

1. Algorithm - Describes any algorithm that deals with one cryptographic function. For example, AES CBC Encode is an algorithm, supporting one mode.
2. Module - Module is a collection of algorithms grouped together (logically). For example, AES and DES will be under module "Symmetric Key"
3. Plugin - Is a loadable module, contains one or more algorithms, which registers itself with the library to extend its functionality.
4. Compatibility layer - Compatibility layer allows any application compiled and linked against other libraries to work with AOCL Cryptography without modifications. AOCL Cryptography provides compatibility layer for IPP-CP and libcrypto(from OpenSSL).

Dynamic dispatcher in the library optimally dispatches to the best possible function for a given architecture, this decision is made once in the lifetime of the process and would not add any overhead due to decision making process. Each algorithm would exist in at least 2 forms

1. A Reference Implementation
2. An Optimized Implementation.
    - AVX SIMD
    - AVX2 SIMD
    - AVX512 SIMD
    - AESNI accelerated instructions
    - Hardware off-load processing

Any x86_64 machine that doesn't support AVX, the reference implementation (very very slow) will be available, but we dont commit to support any machine before full implementation of AVX.

Each of them are dynamically dispatched to at runtime based on CPUID features.

Chapter 2

Design Consideration

AOCL Cryptography is expected to cater new as well as existing customers. Current customers may already be using other solutions like IPP-CP, OpenSSL-crypto, BoringSSL-crypto, MbedTLS etc, and may not want to recompile their entire software stack with AOCL Cryptography. A solution must be provided to experiment with AOCL Cryptography and to enable existing software stacks to easily migrate.

All are version checked, and time to time libraries are updated and upgraded so that all versions need not be maintained.

Chapter 3

Assumptions and Dependencies (TODO: TBD)

AOCL Cryptography assumes following libraries/tools available on system where it is built or running.

- Required Dependancies
  - CMake (3.22 or later)
  - GCC (11.0 or later)
  - Git (2.30.2 or later)
  - OpenSSL (3.0.7 or later)
  - LSB Release
  - Make (4.0 or later)
  - 7zip (15.0 or later)
- Optional Dependancies
  - Pandoc (+ LaTeX for generating pdf docs)
  - Doxygen
  - Sphinx

Chapter 4

General Constraints (TODO: TBD)

The library will contain all the listed algorithms eventually.

OpenSSL compatibility library needs to be co-developed along with AOCL Cryptography, as the requirement for drop-in replacement is crucial for AOCL Cryptography to succeed.

Part III

Architectural Strategies (TODO: TBD)

Chapter 5

Programming Details

The AOCL Cryptography library provides C99 like API described in detail in API. Though the internal structures are implemented using C++ with no advanced features. This decision is taken to avoid writing primitive library functions like stacks/queues or heaps to [<0;199;17M]manage the module/algorithms. Also the C++ STL provides enough gears to achieve the needed functionality with least efforts.

AOCL Cryptography makes use of AMD's CPUID identification library and RNG (random number generator) library to provide additional functionality like dynamic dispatcher. The RNG library also provides needed seeds for the algorithms in need.

Chapter 6

Apps for testing

- Nginx(pronounced Engine-X)
- gRPC
- QATv2

Chapter 7

System Architecture

To simplify the object access types, we introduce following notion

1. Types - Each category (module) will have many types of schemes, this needs to be highlighted using one of the `type` mechanisms.
2. Attributes - All the above mentioned components have attributes, an attribute defines properties for a given object or a context, may it be an algorithm or a module.
3. Operations - The operations that can be performed using that object or on that object. For example an cipher algorithm provides encrypt()/decrypt() kind of operations, where as an hash algorithm provides hash() or digest() kind of operation. Modules provides load()/unload()/search()/init() and other operations and so on.
4. Parameters - Parameters are passed to Operations to perform the same operation slightly differently. Some cases the distinction between attributes and parameters vanishes, as the attribute itself defines the parameter. However it is maintained throughout to provide uniform interface.

7.1 Plugins

The future of cryptography cannot be easily foreseen. New types of communication/certificate mechanisms may emerge, new types of messages may be introduced. Plugins are provide flexible way to integrate both while experimenting and deploying. Design of the plugins and its interfaces are discussed in detail in later sections of this document.

Chapter 8

Policies and Tactics

For this library, GCC is the choice of compiler with LLVM/Clang also in support, Designers and developers are made sure that no compiler-specific features are used, as it looses big on portability. On Windows VC compiler (latest version as of writing VS2019) is used.

Code will honor multiple operating systems, including Linux and Windows to start with.

Library will be provided as a static archive (libalcrypto.a on Linux and alcrypto.lib on Windows) as well as a dynamic version (libalcrypto.so on Linux and alcrypto.dll on Windows)

For build system we have opted for industry standard CMake (version >=3.18.4), and for testing 'Gtest' (Google Test) framework is used.

This library depends on libaoclutils (A CPU Identification Library), version >= 1.0 used by the dynamic dispatcher to select appropriate function.

Documentation is maintained in 'markdown' format, 'pandoc' (version >= 2.9.2.1) command is used to generate pdfs.

Chapter 9

Library Conventions

AOCL Cryptography is designed to be compliant with C99 API, hence uses all standard datatypes like `uint8_t` , `uint16_t`, however we avoid using `size_t` kind of datatypes as there is no clear mention of its size w.r.t ILP64 and LP64.

Library Defines following types - User Data types - Operation types - Attribute types

All types have prefix of `alc_` followed by type/module and end with `_t` , for example - Error type : `alc_error_t` and `alc_key_t` `alc_algorithm_t` - Operation type: `alc_cipher_ops_t` and `alc_hashing_ops_t` - Attributes: `alc_key_info_t` `alc_module_info_t`

9.1 Directory Structure

This section details the very initial directory structure layout, though heavily subjected to change, overall structure would be comparable to following

- *docs/* : Contains various documentation both for application developers and library developers.
  - *docs/internal* : AMD's internal documentation such as design / architecture etc.
- *examples/* : sub-divided into its own directories to contain examples pertaining to a logical group of algorithms
  - *examples/symmetric/* : symmetric key algorithm examples
  - *examples/digest/* : One way hash function examples
  - etc...
- *include/* : Contains all the headers
  - *include/external* : API header, C99 based
  - *include/alcp* : Internal headers for library
- *lib/* : The library itself
  - *lib/compat* : Compatibility layers
    - *lib/compat/openssl* : OpenSSL Compatibility layer
    - *lib/compat/ippcp* : Intel IPP CP compatibility layer

Part IV

Detailed System Design

Chapter 10

Error Reporting

10.1 Design

10.2 API

Error in AOCL Cryptography library is handled using an `uint64_t` value. It has few possible values which is defined in `alcp/error.h`. Errors are defined in an enum `alc_error_generic_t`.

```
typedef uint64_t alc_error_t;
```

```
typedef enum _alc_error_generic
{
    /*
     * All is well
     */
    ALC_ERROR_NONE = 0UL,

    /*
     * An Error,
     *    but cant be categorized correctly
     */
    ALC_ERROR_GENERIC,

    /*
     * Not Supported,
     *  Any of Feature, configuration,  Algorithm or  Keysize not supported
     */
    ALC_ERROR_NOT_SUPPORTED,

    /*
     * Not Permitted,
```

```
     *  Operation supported but not permitted by this module/user etc.
     *  Kind of permission Denied situation, could be from the OS
     */
    ALC_ERROR_NOT_PERMITTED,

    /*
     * Exists,
     *  Something that is already exists is requested to register or replace
     */
    ALC_ERROR_EXISTS,

    /*
     * Does not Exist,
     *   Requested configuration/algorithm/module/feature  does not exists
     */
    ALC_ERROR_NOT_EXISTS,

    /*
     * Invalid argument
     */
    ALC_ERROR_INVALID_ARG,

    /*
     * Bad Internal State,
     *   Algorithm/context is in bad state due to internal Error
     */
    ALC_ERROR_BAD_STATE,

    /*
     * No Memory,
     *  Not enough free space available, Unable to allocate memory
     */
    ALC_ERROR_NO_MEMORY,

    /*
     * Data validation failure,
     *   Invalid pointer / Sent data is invalid
     */
    ALC_ERROR_INVALID_DATA,

    /*
     * Size Error,
     *   Data/Key size is invalid
     */
```

```
    ALC_ERROR_INVALID_SIZE,

    /*
     * Hardware Error,
     *   not in sane state, or failed during operation
     */
    ALC_ERROR_HARDWARE_FAILURE,

    /* There is not enough entropy for RNG
        retry needed with more entropy */
    ALC_ERROR_NO_ENTROPY,

    /*
     *The Tweak key and Encryption is same
     *for AES-XTS mode
     */
    ALC_ERROR_DUPLICATE_KEY,

    /*
     * Mismatch is tag observed in Decrypt
     */
    ALC_ERROR_TAG_MISMATCH,

} alc_error_generic_t;
```

Chapter 11

Dispatcher

The dynamic dispatcher will populate each kind of algorithm with best suitable implementation for the architecture(on which it is currently running). During the initialization phase of the library, it scans through available implementation and selects the best possible option.

Once the best algorithm is selected, its initialization is called, which then registers itself with the module manager. Once the registration is done, any request for a given algorithm will be returned with the already selected algorithm.

The dynamic dispatcher will allow debug mode to override the selection of the function.

If a plugin is loaded, its implementation will overwrite all the algorithms that are currently selected by the dynamic dispatcher. Hence plugins to be loaded with caution.

Since plugins are dynamic, there is no way to know/distinguish loaded plugin with existing algorithm. Also it will become difficult if plugins are distinguishable by the Application developer.

In cases when the plugin registers an algorithm that is not currently part of the library, it will be treated as an extension and applications can request for the algorithms supported by the newly loaded plugin.

Part V

Detailed Subsystem Design

Chapter 12

Digests

12.1 Design

The datatype `alc_digest_mode_t` describes the digest that is being requested or operated on.

```
typedef enum _alc_digest_mode
{
    ALC_MD5,
    ALC_SHA1,
    ALC_MD5_SHA1,
    ALC_SHA2_224,
    ALC_SHA2_256,
    ALC_SHA2_384,
    ALC_SHA2_512,
    ALC_SHA2_512_224,
    ALC_SHA2_512_256,
    ALC_SHA3_224,
    ALC_SHA3_256,
    ALC_SHA3_384,
    ALC_SHA3_512,
    ALC_SHAKE_128,
    ALC_SHAKE_256,
} alc_digest_mode_t,
```

12.2 The *Digest* class

This is the C++ interface to the Digests. All digest algorithms will inherit this class.

```
class IDigest
{
  public:
```

```
    IDigest() = default;

  public:
    virtual void        init(void)                              = 0;
    virtual alc_error_t update(const Uint8* pBuf, Uint64 size) = 0;
    virtual alc_error_t finalize(Uint8* pBuf, Uint64 size)     = 0;
    /**
     * @return The input block size to the hash function in bytes
     */
    Uint64 getInputBlockSize() { return m_block_len; }

    /**
     * @return The digest size in bytes
     */
    Uint64 getHashSize() { return m_digest_len; }

    virtual ~IDigest() {}

  protected:
    Uint64 m_digest_len = 0; /* digest len in bytes */
    Uint64 m_block_len  = 0;
    bool   m_finished   = false;
    Uint64 m_msg_len    = 0;
    /* index to m_buffer of previously unprocessed bytes */
    Uint32            m_idx = 0;
    alc_digest_mode_t m_mode;
};
```

All algorithms are expected to implement the `IDigest` abstract base class.

```
template<alc_digest_len_t digest_len>
class Sha2 final : public IDigest
{
    static_assert(ALC_DIGEST_LEN_224 == digest_len
                  || ALC_DIGEST_LEN_256 == digest_len);

  public:
    static constexpr Uint64 /* define word size */
        cWordSizeBits   = 32,
        cNumRounds      = 64,                /* num rounds in sha256 */
        cChunkSizeBits  = 512,               /* chunk size in bits */
        cChunkSize      = cChunkSizeBits / 8, /* chunks to process */
        cChunkSizeMask  = cChunkSize - 1,
        cChunkSizeWords = cChunkSizeBits / cWordSizeBits, /* same in words */
        cHashSizeBits   = ALC_DIGEST_LEN_256,            /* same in bits */
```

```
        cHashSize       = cHashSizeBits / 8, /* Hash size in bytes */
        cHashSizeWords  = cHashSizeBits / cWordSizeBits;

  public:
    ALCP_API_EXPORT Sha2();
    ALCP_API_EXPORT Sha2(const Sha2& src);
    virtual ALCP_API_EXPORT ~Sha2() = default;

  public:
    ALCP_API_EXPORT void init(void) override;

    ALCP_API_EXPORT alc_error_t update(const Uint8* pMsgBuf,
                                       Uint64       size) override;

    ALCP_API_EXPORT alc_error_t finalize(Uint8* pBuf, Uint64 size) override;

  private:
    alc_error_t processChunk(const Uint8* pSrc, Uint64 len);
    /* Any unprocessed bytes from last call to update() */
    alignas(64) Uint8 m_buffer[2 * cChunkSize]{};
    alignas(64) Uint32 m_hash[cHashSizeWords]{};
};
```

12.3 API

Digests are computed like the other subsystem, All APIs are prefixed with `alcp_digest`, `alcp` being project prefix and `digest` being subsystem prefix. Following are the C99 APIs.

1. Context size determination: The context size needs to be queried from the library, this helps the Application designer to allocate and free the memory needed for the 'Handle'. `c Uint64 alcp_digest_context_size();`
2. Request : The application needs to request the 'Handle', and supposed to send required configuration via the input of type `alc_digest_mode_t`. `c alc_error_t alcp_digest_request(alc_digest_mode_t mode, alc_digest_hand p_digest_handle);`
3. Init : Initializes the digest object. This is called to start a new sequence of digest creation. `c alc_error_t alcp_digest_init(alc_digest_handle_p p_digest_handle);`
4. Update : Both block and stream digests are treated alike, however the update() method allows application to build on previously processed blocks. `c alc_error_t alcp_digest_update(const alc_digest_handle_t* p_digest_handle, Uint8* buf, Uint64 size);`

5. Duplicate : Duplicates the digest handle from 'pSrcHandle' to 'pDestHandle'. The independent duplicated handled can then be used to proceed with the remaining steps in lifecycle c
```
alc_error_t     alcp_digest_context_copy(const alc_digest_handle_p
pSrcHandle,                              const alc_digest_handle_p
pDestHandle);
```
6. Squeeze : Valid only for Shake(SHA3) algorithm for squeezing the digest out. This can be called multiple times to squeeze the digest out in steps. This API cannot be called together with 'alcp_digest_finalize'. c
```
alc_error_t     alcp_digest_shake_squeeze(const
alc_digest_handle_p pDigestHandle,                                Uint8*
pBuff,                              Uint64                    size);
```
7. Finalize: This is the marker for end of sequence and also copies the digest that is computed. Once this is called, its not possible to call update() again. c
```
alc_error_t
alcp_digest_finalize(const alc_digest_handle_t* p_digest_handle,
Uint8*                      digest,                         Uint64
digest_size);
```
8. Finish : This is a cleanup phase, once finish is called the session ends, and the handle is no longer valid. Hence the digest needs to be copied by the application before this step. c
```
void     alcp_digest_finish(const alc_digest_handle_t*
p_digest_handle);
```

Chapter 13

Ciphers

13.1 Symmetric Ciphers

Symmetric ciphers uses the same key for both encryption and decryption, The key types are described in Key Types.

The library supports Symmetric ciphers with GCM, CFB, CTR and XTS modes. Supported ciphers can be checked programatically using `alcp_cipher_available()` function.

Each Algorithm registers itself with algorithm-manager, which keeps a list of currently supported algorithm. The `alcp_cipher_available()` in turn calls the internal function `alcp_algo_available()` function to check if the provided mode / keylength is supported by the algorithm.

Crypto library uses "Factory" design pattern to create and manage the Cipher module. All ciphers are requested using `alcp_cipher_request()` API, which accepts various parameters to determine cipher and exact mode to operate.

```
alc_error_t
alcp_cipher_request(const alc_cipher_mode_t cipherMode,
                    const Uint64            keyLen,
                    alc_cipher_handle_p     pCipherHandle);
```

In the above api, `alc_cipher_mode_t` is described as in `alc_cipher_mode_t`, which describes the cipher type and mode of operation

13.1.1 The `alc_cipher_context_t` structure

The Cipher's context is very specific to a given cipher algorithm. This type is an opaque pointer which is purely internal to the library.

```
typedef void alc_cipher_context_t;
```

13.1.2 The `alc_cipher_mode_t` type

Cipher modes are expressed in one of the following enumerations

```
typedef enum _alc_cipher_mode
{
    ALC_AES_MODE_NONE = 0,

    // aes ciphers
    ALC_AES_MODE_ECB,
    ALC_AES_MODE_CBC,
    ALC_AES_MODE_OFB,
    ALC_AES_MODE_CTR,
    ALC_AES_MODE_CFB,
    ALC_AES_MODE_XTS,
    // non-aes ciphers
    ALC_CHACHA20,
    // aes aead ciphers
    ALC_AES_MODE_GCM,
    ALC_AES_MODE_CCM,
    ALC_AES_MODE_SIV,
    // non-aes aead ciphers
    ALC_CHACHA20_POLY1305,

    ALC_AES_MODE_MAX,

} alc_cipher_mode_t;
```

13.2 AES (Advanced Encryption Standard)

The library supports AES(Advanced Encryption Standard), as part of the Symmetric Cipher module.

13.2.0.1 CFB (Cipher FeedBack)

CFB Mode is cipher feedback, a stream-based mode. Encryption occurs by XOR'ing the key-stream bytes with plaintext bytes. The key-stream is generated one block at a time, and it is dependent on the previous key-stream block. CFB does this by using a buffered block, which initially was supplied as IV (Initialization Vector).

13.3 Message Authentication Codes (MAC) (TODO: WIP)

13.4 AEAD Ciphers (TODO: WIP)

13.5 Key Derivation Functions (KDF) (TODO: WIP)

13.6 Random Number Generator

The AOCL Crypto library supports both PRNG and TRNG algorithms. AMD Zen series of processors provide 'RDRAND' instruction as well as 'RDSEED', however there are speculations on its security. Also it is prone to side-channel attacks.

PRNG's usually requires a seed, and not considered cryptographically secure. The OS-level PRNG(/dev/random) are not desired as well for high-security randomness, as they are known to never produce data more than 160-bits (many have 128-bit ceiling).

However there are cryptographically secure PRNGs (or in other words CRNG) which output high-entropy data.

On Unix like modern operating systems provide blocking `/dev/random` and a non-blocking `/dev/urandom` which returns immediately, providing cryptographical randomness. In theory `/dev/random` should produce data that is statistically close to pure entropy,

Also the traditional `rand()` and `random()` standard library calls does not output high-entropy data.

RNG module will support two modes 'accurate' and 'fast', along with multiple distribution formats. The library also supports 'Descrete' and 'Continuous' distribution formats. RNG type specified - i : Integer based

Descrete Distribution formats:

Type of Distribution	Data Types	RNG	Description
Uniform	i	i	Uniform discrete distribution on the interval [a,b)
	i	i	Uniformly distributed bits in 64-bit chunks

13.6.0.1 Design

Each RNG is represented by the `alc_rng_info_t` structure. The library provides interface to query if a RNG configuration is available using `alcp_rng_supported()`, this provides the option for the application to fall back to different algorithm/configuration when not supported.

As usual with other modules, all the RNG api's return `alc_error_t` and use of `alcp_is_error(ret)` will provide sufficient information to fallback or to abort for the application.

All available RNG algorithms will register with Module Manager with type `ALC_MODULE_TYPE_RNG`, Types of Generator are described by

An RNG generator can be requested using `alcp_rng_request()`, which accepts an `alc_rng_info_t` structure, which has following layout.

```
typedef struct {
    alc_rng_type_t        r_type;
    alc_rng_source_t      r_source;
    alc_rng_distrib_t     r_distrib;
    alc_rng_algo_flags_t  r_flags;
} alc_rng_info_t;
```

```
typedef enum {
    ALC_RNG_TYPE_INVALID = 0,
    ALC_RNG_TYPE_SIMPLE,
    ALC_RNG_TYPE_CONTINUOUS,
    ALC_RNG_TYPE_DISCRETE,

    ALC_RNG_TYPE_MAX,
} alc_rng_type_t ;
```

Random Number source can be selected using following enumeration. The request function

```
typedef enum {
    ALC_RNG_SOURCE_ALGO = 0,  /* Default: select software CRNG/PRNG */
    ALC_RNG_SOURCE_OS,        /* Use the operating system based support */
    ALC_RNG_SOURCE_DEV,       /* Device based off-loading support */

    ALC_RNG_SOURCE_MAX,
} alc_rng_source_t;
```

Random Generation algorithms and their distribution are described by enumeration `alc_rng_distribution_t`.

```
typedef enum {
    ALC_RNG_DISTRIB_UNKNOWN = 0,

    ALC_RNG_DISTRIB_BETA,
    ALC_RNG_DISTRIB_CAUCHY,
    ALC_RNG_DISTRIB_CHISQUARE,
    ALC_RNG_DISTRIB_DIRICHLET,
    ALC_RNG_DISTRIB_EXPONENTIAL,
    ALC_RNG_DISTRIB_GAMMA,
    ALC_RNG_DISTRIB_GAUSSIAN,
    ALC_RNG_DISTRIB_GUMBEL,
    ALC_RNG_DISTRIB_LAPLACE,
    ALC_RNG_DISTRIB_LOGISTIC,
    ALC_RNG_DISTRIB_LOGNORMAL,
    ALC_RNG_DISTRIB_PARETO,
    ALC_RNG_DISTRIB_RAYLEIGH,
```

```
    ALC_RNG_DISTRIB_UNIFORM,
    ALC_RNG_DISTRIB_VONMISES,
    ALC_RNG_DISTRIB_WEIBULL,
    ALC_RNG_DISTRIB_WALD,
    ALC_RNG_DISTRIB_ZIPF,

    ALC_RNG_DISTRIB_BERNOULLI,
    ALC_RNG_DISTRIB_BINOMIAL,
    ALC_RNG_DISTRIB_GEOMETRIC,
    ALC_RNG_DISTRIB_HYPERGEOMETRIC,
    ALC_RNG_DISTRIB_MULTINOMIAL,
    ALC_RNG_DISTRIB_NEGBINOMIAL,
    ALC_RNG_DISTRIB_POISSON,
    ALC_RNG_DISTRIB_UNIFORM_BITS,
    ALC_RNG_DISTRIB_UNIFORM,

    ALC_RNG_DISTRIB_MAX,
} alc_rng_distrib_t;
```

Each algorithm have some flags to further extend/restrict. This may or may not have valid information. For example `ALC_RNG_DISTRIB_POISON` could be selected in multiple format 1. Normal Poison distribution 2. With Varying mean

```
typedef enum {

} alc_rng_algo_flags_t;
```

13.6.0.2 APIs

To support the fallback for applications in cases where the expected RNG support is not available, `alcp_rng_supported()`, returns error not supported. No errors if the given RNG and its Distribution support is available.

```
alc_error_t
alcp_rng_supported(const alc_rng_info_t *tt);
```

An RNG handle can be requested using `alc_rng_request()`, the context(handle) can only be used if the check `if (!alc_is_error(ret))` passes for the call.

```
alc_error_t
alcp_rng_request(const alc_rng_info_t *tt, alc_context_t *);
```

The `alcp_rng_gen_random()` generates random numbers and fills the buffer pointed by `buf` for length specified by `size` in bytes.

```
alc_error_t
alcp_rng_gen_random(alc_context_t *tt,
```

```
                    uint8_t       *buf,  /* RNG output buffer */
                    uint64_t       size  /* output buffer size */
                    );
```

Chapter 14

Random Number Generator (RNG)

14.1 PRNG

14.2 TRNG

Chapter 15

Message Authentication Codes (MAC)

Part VI

Dynamic Loading Feature

Dynamic loading of library is a necessary feature to support allow extensible library. Here the extension is for supporting new devices that may arrive in near future, but library itself doesn't have to have support for immediate use.

Chapter 16

Dynamic library loading

Libraries are built as part of the “Provider Kit”, herein referred to as PK. The PK provides features / functionalities that are not already part of the library. Also it helps reduce the size of the linked library where the specific module, device is not present or not needed.

Libraries are provided both as static library and as dynamic loadable. This section just presents the dynamic loading part.

The cases where libraries are provided as static or archived versions (like as in .a or .lib), the library just needs to be linked at the final stage of compilation.

Static libraries usually hosts a ‘constructor’ functions which gets called at the very beginning of the program execution. The extended module registers itself as part of the library.

Chapter 17

Dynamic Feature/Class loading

Once the library is loaded, one way or another.

Chapter 18

Design

Since each operating systems implements dynamic loading differently, there is a wrapper class present in __dynlib.cc_. the implementation details are present in *impl/dynlib_linux.cc* for Linux or Unix specific loading which makes use of the *libdl.so* APIs *dlopen()*, *dlclose()* and *dlsym()* etc.

Windows specific implementation should be present in *impl/dynlib_win.c*

18.1 DynamicLibrary class

`DynamicLibrary` class supports following functionality: - `load()` - loads a library - `unload()` - unload a previously loaded library - `isLoaded()` - checks if the loading was successful - `getSymbol()` - get a symbol that is part of the symbol table. - `suffix()` - gets the library suffix for a given operating system. - `setSearchPath()` - Sets the search path for loading libraries, usually its just current directory

18.2 ClassLoader class

This class implements loading a class, but for now this portion is not implemented as the final decision on whether to allow C interface or C++ interface for the