PhD Diary

Wei Minn
Singapore Management University

Version 1.0.5
Last compiled: November 23, 2023

Contents

Part I

2023

Chapter 1

November

1.1 November 14, 2023

1.1.1 AOSP

Zygote

Zygote initializes by pre-loading the entire Android framework. Unlike desktop Java, it does not load the libraries lazily; it loads all of them as part of system start up. After completely initializing, it enters a tight loop, waiting for connections to a socket. When the system needs to create a new application, it connects to the Zygote socket and sends a small packet describing the application to be started. Zygote clones itself, creating a new kernel-level process.

Memory is organized into uniformly sized **pages**. When the application refers to memory at a particular address, the device hardware reinterprets the address as an index into a **page table**. Newly cloned Zygote processes for newly started applications are simply clone of Zygote's page table, pointing to the exact same pages of physical memory. Only the pages the new application uses for its own purposes are not shared:

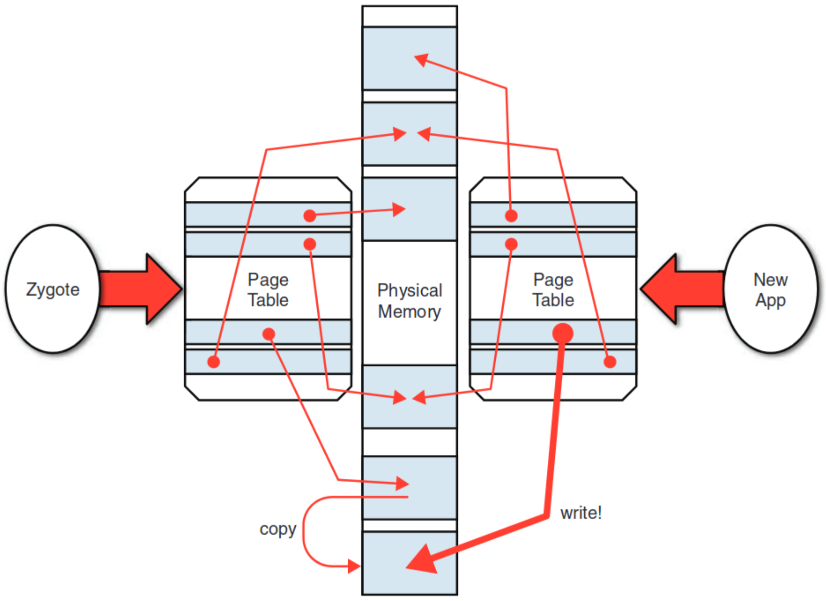

Figure 1.1: Zygote Copy-on Write

Zygote Initialization

Zygote is started by `init`. `ro.zygote` system variable set at platform build time decides which of four types of Zygotes are started and which one is "primary". Both the `init` and Zygote scripts are stored inside `$AOSP/system/core/rootdir`. In the following `init.zygote64_32.rc`, 2 Zygote processes, primary and secondary, are started at 2 different sockets:

```
service zygote /system/bin/app_process64 -Xzygote \
        /system/bin --zygote --start-system-server --socket-name=zygote
    class main
    priority -20
    user root
    group root readproc reserved_disk
    socket zygote stream 660 root system
    socket usap_pool_primary stream 660 root system
    onrestart exec_background - system system -- /system/bin/vdc volume abort_fuse
    onrestart write /sys/power/state on
    onrestart restart audioserver
    onrestart restart cameraserver
    onrestart restart media
    onrestart restart media.tuner
    onrestart restart netd
    onrestart restart wificond
    task_profiles ProcessCapacityHigh MaxPerformance
    critical window=${zygote.critical_window.minute:-off} target=zygote-fatal

service zygote_secondary /system/bin/app_process32 -Xzygote \
        /system/bin --zygote --socket-name=zygote_secondary --enable-lazy-preload
    class main
    priority -20
    user root
    group root readproc reserved_disk
    socket zygote_secondary stream 660 root system
    socket usap_pool_secondary stream 660 root system
    onrestart restart zygote
    task_profiles ProcessCapacityHigh MaxPerformance
    disabled
```

The actual application that is started as user root at the very highest priority by `init` is `/system/bin/app_process64`. The script requests that `init` create a stream socket for the process and catalog it as `/dev/socket/zygote_secondary` which will be used by the system to start new Android applications.

Zygote is only started once during the system startup, by `app_process64` and `app_-process32`, and is simply cloned to start subsequent applications. Zygote initialization sequence is described below:

Method	Description	Source
`init.rc`	Imports the `init.zygote64_32.rc` that contains the script that starts Zygote service.	`$AOSP/system/core/rootdir`
`init.zygote64_-32.rc`	Runs `app_process64` and `app_process32` which will initialize the starting of Zygote service.	`$AOSP/system/core/rootdir`
`app_process`	Creates `AppRuntime`, a subclass of `AndroidRuntime`, that does bookkeeping, naming the process, setting up parameter, and the name of the class to run when not running Zygote, and then calls `AndroidRuntime.start()` to invoke the runtime.	`$AOSP/frameworks/base/cmds/app_-process`
`AppRuntime::start`	Invokes `startVM.startVM` which invokes `JNI_CreateJavaVM`.	`$AOSP/frameworks/base/core/jni/Andro`
`JNI_CreateJavaVM`	Calls `Runtime::Create`.	`$AOSP/art/runtime/jni/java_-vm_ext.cc`
`Runtime::Create`	Initializes the ART runtime, loading the system OAT files and the libraries they contain.	`$AOSP/art/runtime/runtime.cc`

Table 1.1: Zygote Initialization Sequence

The argument that `app_process` passed to `start` is `com.android.internal.os.Zygote.Init`, the source for which is in `$AOSP/frameworks/base/core/java/com/android/internal/os/ZygoteInit.java`. `app_process` is the launcher for all Java programs (not apps!) in the Android system, and Zygote is one example of the programs (system service) to be launched.

Zygote System Service

Zygote has 3 major tasks, on startup:

1. Register the socket to which the system will connect to start new application. Handled by `registerServerSocket` method which creates socket using the named passed as parameter for `init` script.
2. Preload Android resources (classes, libraries, resources and even WebViews) with a call to `preload` method. After `preload` is finished, Zygote is fully initialized and ready to clone to new applications very quickly.
3. Start Android System Server. Thus, `SystemServer` is the first application to be cloned by Zygote.

After it has completed these three tasks, it enters a loop, waiting for connections to the socket.

1.1.2 C++ Primer

Primitive Built-in Types

Includes **arithmetic types** and a special type named **void** which has no associated values and can be used in only a few circumstances, most commonly as the return type for functions that do not return a value.

The arithmetic types are divided into two categories: **integral types** (which include character and boolean types) and floating-point types.

Type	Meaning	Minimum Size
`bool`	boolean (`true` or `false`)	NA
`char`	character	8 bits
`w_char_t`	wide character	16 bits
`char16_t`	Unicode character	16 bits
`char32_t`	Unicode character	32 bits
`short`	short integer	16 bits
`int`	integer	16 bits
`long`	long integer	32 bits
`long long`	long integer	64 bits
`float`	single-precision floating-point	6 significant digits
`double`	double-precision floating-point	10 significant digits

`long double`	extended-precision floating-point	10 significant digits

Table 1.2: Zygote Initialization Sequence

Except for `bool` and extended character types, the integral types may be **signed** (can represent negative or positive numbers) or **unsigned**. By default, `int`, `short`, `long`, `long long` are all signed. To declare unsigned type, prepend `unsigned` to the type. `char` is signed on some machine and `unsigned` on others, and `unsigned int` is abbreviated as `unsigned`.

Conversions happen automatically when we use an object of one type where an object of another type is expected.

```cpp
unsigned u = 10;
int i = -42;
std::cout << i + i << std::endl; // prints -84
std::cout << u + i << std::endl; // if 32-bit ints, prints 4294967264
```

In the above snippet, converting a negative number to `unsigned` will cause the value to "wrap around" because `unsigned` values can never be less than 0. Thus, extra care should be taken if we want to write loops with `unsigned` values and stopping conditions at negative values like the snippet below:

```cpp
// WRONG: u can never be less than 0; the condition will always succeed
for (unsigned u = 10; u >= 0; --u)
    std::cout << u << std::endl;
```

As such it is always advisable to not mix `signed` and `unsigned` types. By default, integer literals (42) are signed, while octal (`024`) and hexadecimal (`0x14`) may be signed or unsigned.

Escape sequences are used as if they were single characters:

```cpp
std::cout << '\n'; // prints a newline
std::cout << "\tHi!\n"; // prints a tab followd by "Hi!" and a newline
```

Variables

Initialization is not assignment. Initialization happens when a variable is given a value when it is created. Assignment obliterates an object's current value and replaces that value with a new one.

Four different ways to initialize:

```cpp
int units_sold = 0;
int units_sold = {0}; // list initialization; does not work for built-in types if
    data loss is likely
int units_sold{0};
int units_sold(0);
```

Variables defined outside any function body are initialized to zero by default. Variables of built-in type defined inside a function are **uninitialized** and therefore undefined. Objects of class type that we do not explicitly initialize have a value that is defined by the class.

A **declaration** makes a name known to the program. A file that wants to use a name defined elsewhere includes a declaration for that name. A **definition** creates the associated entity. A definition involves declaration, allocates storage and may provide the variable with an initial value.

```
extern int i;   // declares but not define j
int j;          // declares and defines j
int k = 12;     // declares, defines and initializes j
extern double pi = 3.14;     // definition
```

Variables must be defined only once but can be declared several times. To use a variable in more than one file requires declarations that are separate from the variable's definition. To use the same variable in multiple files, we must define that in one - and only one - file. Other files that use that variable must declara - but not define - that variable.

1.2 November 15, 2023

1.2.1 C++ Primer

Scopes of Names

Most scopes in C++ are delimited by curly braces.

```
#include <iostream>
int main() {
    int sum = 0;
    // sum values from 1 through 10 inclusive
    for (int val = 1; val <= 10; ++val)
        sum += val; // equivalent to sum=sum+val
    std::cout << "Sum␣of␣1␣to␣10␣inclusive␣is␣"
        << sum << std::endl;
    return 0;
}
```

In above program, `main` - like most names defined outside a function - has **global scope** and thus, is accessible throughout the program. `sum` has **block scope** and is accessible from its point of declaration throughout the rest of the `main` function. `val` is defined in the scope of the `for` statement and can be used in that statement but not elsewhere in `main`.

Names declared in the outer scope can also be redefined in an inner scope although it is always a bad idea:

```
#include <iostream>
// Program for illustration purposes only: It is bad style for a function
// to use a global variable and also define a local variable with the same name
int reused = 42; // reused has global scope
int main() {
    int unique = 0; // unique has block scope

    // output #1: uses global reused;prints 42 0
    std::cout << reused << "␣" << unique << std::endl;

    int reused = 0; // new, local object named reused hides global reused
    // output #2: uses local reused; prints 0 0
    std::cout << reused << "␣" << unique << std::endl;

    // output #3: explicitly requests the global reused; prints 42 0
    std::cout << ::reused << "␣" << unique << std::endl;

    return 0;
}
```

When the scope operator (`::` **operator**) has an empty LHS, it is a request to fetch the name on the RHS from the global scope.

References

A **reference** defines an alternative name for an object. A reference type can be defined by writing a declarator of the form `&d` where `d` is the name being declared:

```
int ival = 1024;
int &refVal = ival; // refVal refers to (is another name for) ival
int &refVal2; // error: a reference must be initialized
```

When we define a reference, instead of copying the initializer's value, we bind the reference to its initializer. Once initialized, a reference remains bound to its initial object. There is no way to rebind a reference to refer to a different object. Because there is no way to rebind a reference, references must be initialized.

A reference is not an object. Instead, a reference is just another name for an already existing object. Thus, *all* operation on that reference are actually operations on the object to which the reference is bound:

```
refVal = 2; // assigns 2 to the object to which refVal refers, i.e., to ival
int ii = refVal; // same as ii=ival
```

Becuase references are not objects, we may not define a reference to a reference. We can define references in a single definition with each identifier that is a reference being preceded by the `&` symbol:

```
int i = 1024, i2 = 2048; // i and i2 are both ints
int &r = i, r2 = i2; // r is a reference bound to i; r2is an int
int i3 = 1024, &ri = i3; // i3 is an int; riis a reference bound to i3
int &r3 = i3, &r4 = i2; // both r3and r4are references
```

Pointers

Like references, pointers are used for indirect access to other objects. Unlike a reference, a pointer is an object in its own right. Pointers can be assigned and copied; a single pointer can point to several different objects over its lifetime. Unlike a reference, a pointer need not be initialized at the time it is defined. Like other built-in types, pointers defined at block scope have undefined value if they are not initialized.

We define a pointer type by writing a declarator of the form *d,whered is the name being defined. The * must be repeated for each pointer variable:

```
int *ip1, *ip2; // both ip1 and ip2 are pointers to int
double dp, *dp2; // dp2 is a pointer to double; dp is a double
```

A pointer holds the address of another object. We get the address of an object by using the address-of operator (& **operator**):

```
int ival = 42;
int *p = &ival; // p holds the address of ival; p is a pointer to ival

double dval;
double *pd = &dval; // ok: initializer is the address of a double
double *pd2 = pd; // ok: initializer is a pointer to double
int *pi = pd; // error: types of pi and pd differ
pi = &dval; // error: assigning the address of a doubletoapointertoint
```

We can use the dereference operator (* **operator**) to access that object:

```
int ival = 42;
int *p = &ival; // p holds the address of ival; p is a pointer to ival
cout << *p; // * yields the object to which p points; prints 42

*p = 0; // * yields the object; we assign a new value to ival through p
cout << *p; // prints 0
```

When we assign to `*p`, we are assigning to the object to which p points. We may dereference only a valid pointer that points to an object.

`void*` is a special pointer type that can hold the address of any object. Its useful for when the type of the object at that address is unknown:

```
double obj = 3.14, *pd = &obj; // ok: void*can hold the address value of any data
    pointer type
void *pv = &obj; // objcan be an object of any type
pv = pd; // pvcan hold a pointer to any type
```

The modifiers * and & do not apply to all variables defined in a single statement:

```
int* p1, p2; // p1 is a pointer to int; p2is an int
int *p1, *p2; // both p1and p2are pointers to int
```

As pointers are objects in memory, they also have addresses of their own. Therefore, we can store the address of a pointer in another pointer:

```
int ival = 1024;
int *pi = &ival; // pi points to an int
int **ppi = &pi; // ppi points to a pointer to an int
```

We indicate each pointer level by its own `*`. Dereferencing a pointer to a pointer yields the pointer. So in this case, you must dereference twice to access the underlying object.

1.2.2 AOSP

Starting Android System Server and Other Apps using Zygote

During its initialization, Zygote will check for `start-system-server` flag, and if set, will bring up `SystemServer` in the following sequence:

Method	Description	Source
`ZygoteInit.forkSystemServer`	Runs after the Zygote process has been initialized. It is hardcoded with System Server classpath[1] as one of the arguments to call `Zygote.forkSystemServer` to spawn `SystemServer` process.	`AOSP/framework/base/core/java/com/android/internal/os/ZygoteInit.java`
`Zygote.forkSystemServer`	`Zygote` class wraps native methods that communicate with Android Runtime, one of whom is `com_android_internal_os_nativeForkSystemServer`.	`AOSP/framework/base/core/java/com/android/internal/os/Zygote.java`
`com_android_internal_os_nativeForkSystemServer`	Calls `zygote::ForkCommon` and `SpecializeCommon` which does the actual forking.	`AOSP/frameworks/base/core/jni/com_android_internal_os_Zygote.cpp`
`SpecializeCommon`	Looks at the flags and Process ID for setting up sandboxing, configuring the correct SE Linux context, and process capabilities. Afterwards, it will call `Zygote` methods for post-fork procedures.	`AOSP/frameworks/base/core/jni/com_android_internal_os_Zygote.cpp`
`Zygote.callPostForkSystemServerHooks`	Calls `ZygoteHooks.java` at the end of specialization procedures. Only applicable for `SystemSever`.	`AOSP/framework/base/core/java/com/android/internal/os/Zygote.java`
`Zygote.callPostForkChildHooks`	Calls `ZygoteHooks.java` at the end of specialization procedures. Applicable to all applications and services including `SystemServer`	`AOSP/framework/base/core/java/com/android/internal/os/Zygote.java`

[1]`com.android.server.SystemServer`, the source for which is stored in `AOSP/frameworks/base/services/java/com/android/server/SystemServer.java`.

Method	Description	Source
ZygoteHooks.postForkSystemServer and ZygoteHooks.postForkSystemServer	Wrappers for ZygoteHooks inside the Android Runtime. They call their respective native code inside the ART via Java Native Interface.	AOSP/libcore/dalvik/src/main/java/dalvik/system/ZygoteHooks.java
ZygoteHooks_nativePostForkSystemServer	Loads the specialized class libraries to start the the System Server.	AOSP/art/runtime/native/dalvik_system_ZygoteHooks.cc
ZygoteHooks_nativePostForkChild	Loads the specialized class libraries to start the service/application.	AOSP/art/runtime/native/dalvik_system_ZygoteHooks.cc
handleSystemServerProcess	The control returns to ZygoteInit, and finish remaining work for the newly forked system server process, and calls ZygoteInit.zygoteInit.	AOSP/framework/base/core/java/com/android/internal/os/ZygoteInit.java
ZygoteInit.zygoteInit	The main function called when started through the zygote process, which calls RuntimeInit.applicationInit	AOSP/framework/base/core/java/com/android/internal/os/ZygoteInit.java
RuntimeInit.applicationInit	Calls the public static void main method of the application	AOSP/framework/base/core/java/com/android/internal/os/RuntimeInit.java
ZygoteServer.runSelectLoop	After forking has finished, the control enters ZygoteServer which starts an endless loop that handles incoming connections with ZygoteConnection.processCommand.	AOSP/framework/base/core/java/com/android/internal/os/ZygoteServer.java
ZygoteConnection.processCommand	Calls Zygote.forkAndSpecialize which is a version of ZygoteInit.forkSystemServer for the masses.	AOSP/framework/base/core/java/com/android/internal/os/ZygoteConnection.java
Zygote.forkAndSpecialize	A version of Zygote.forkSystemServer for the masses.	AOSP/frameworks/base/core/jni/com_android_internal_os_Zygote.cpp

Method	Description	Source
`com_android_internal_os_nativeForkAndSpecialize`	Calls `zygote::ForkCommon` and `SpecializeCommon` which does the actual forking, and returns to `ZygoteInit` and immediately enters `ZygoteServer`.	`AOSP/frameworks/base/core/jni/com_android_internal_os_Zygote.cpp`

Table 1.3: System Server and Applications Initialization Sequence

1.3 November 17, 2023

1.3.1 AOSP Hardware Abstraction Layer

The interface to the hardware is a device drivers which are usually device specific and sometimes proprietary. A single set of C header files describes the functionality that a HAL provides to the Android system. HAL Code for a particular device is the implementation of the API defined by those header files, so that no code above the HAL needs to be changed to port Android to use the new device.

HAL Code Structure

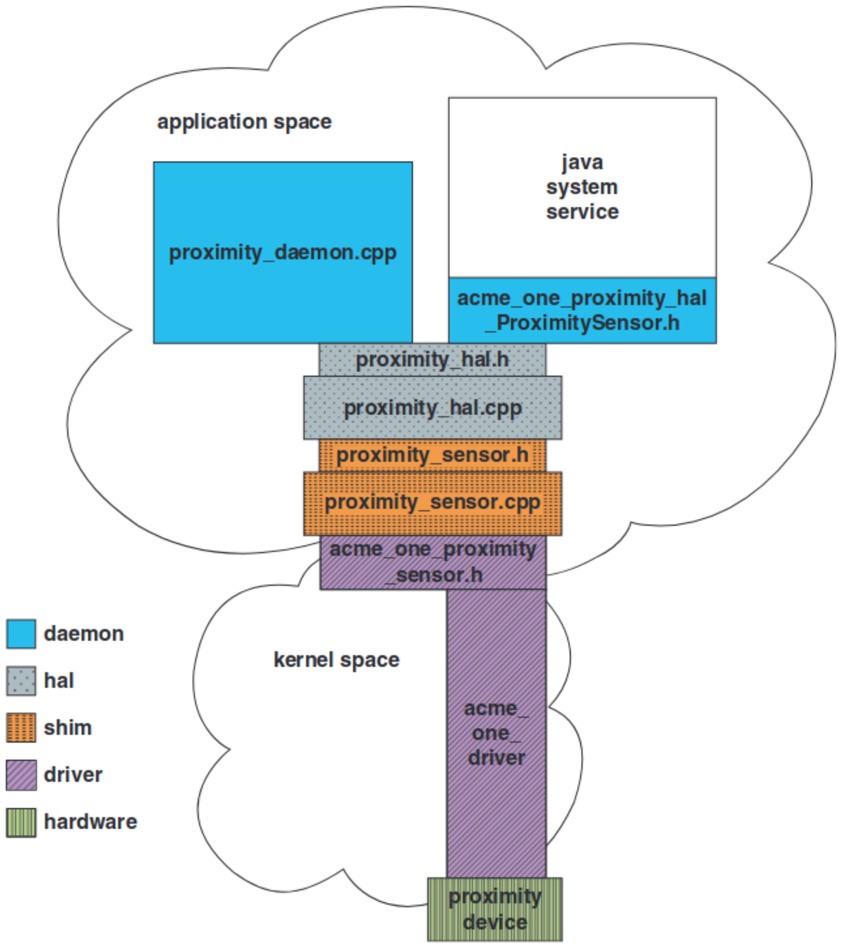

Figure 1.2: HAL Layer Structure

The code consists of four functional components as show in Figure 1.2:

1. **HAL code (dotted boxes):** Abstraction that separates the capabilities of hardware from its specific implementations. The `.h` file defines the HAL interface, and the implementation (`.cpp` file) specializes the Android HAL API for the target hardware.

2. **Shim code (dashed boxes):** Glue code that connects the HAL to a specific device hardware/driver. This code adapts the Android HAL API to the device driver for the hardware.

3. **Daemon (blue):** Stand-alone application that interacts with the hardware through the HAL.

4. **Java System Service (white):** System Service that Android applications will use to access the custom hardware.

The source for those components are structured like in the directory tree below:

```
one
├── app
├── native_daemon
│   └── ...
├── java_daemon
│   └── ...
└── proximity
    ├── include
    │   ├── dev
    │   │   └── ...
    │   └── ...
    ├── dev
    │   └── ...
    ├── hal
    │   └── ...
    └── jni
        └── ...
```

where `one` is the device folder of the AOSP project. All the code implementing the HAL for the proximity sensors goes into a new subdirectory `proximity`.[2]

1.4 November 18, 2023

1.4.1 Configuring AOSP for Acme Device on Ubuntu 23

Repo Manifest

Top-level subdirectory named `.repo` contains the `manifests` repository inside `manifests` subdirectory. The `manifests` repo contain one or more manifest files named as the argument of the `-m` command line option. `.repo/manifest` file controls the structure of the rest of the repository, and includes `.repo/manifests/default.xml`[3] which is a list of git repositories. `repo` program parses `manifest.xml`, and thus `default.xml`, and clone each repository into a location specfied inside `default.xml`.

[2]To be a "real" HAL, the interface `proxmity/include/proximity_hal.h` would have to be promoted from its current directory specifically for the One device, up into the Android source tree to a location that would make it visible to other code that needed to use it. Here, it is only shared by Acme devices, so it is put under the subdirectory of the Acme device directory. If it's visible across device from multiple vendors, it might be promoted into the `device` directory itself.

[3]`https://gerrit.googlesource.com/git-repo/+/master/docs/manifest-format.md`

Each project element in the XML identifies a git repository by its name, relative to some base URL, its remote; and where that repository should be placed in the local workspace, its path. If the full URL for the repository is not specified, repo will use the default remote specified in the default element near the top of the manifest:

```
<default revision="refs/tags/android-13.0.0_r11"
    remote="aosp"
    sync-j="4" />
```

where the remote, aosp, is defined likewise in the top of of the default.xml:

```
<remote  name="aosp"
    fetch=".."
    review="https://android-review.googlesource.com/" />
```

Instead of including a URL as its attribute, it includes the fetch attribute which indicatese the URL for this remote should be derived from the URL used to initialize the workspace (the argument to the -u option).

```
git ls-remote -h https://android.googlesource.com/platform/manifest.git
repo init -u https://android.googlesource.com/platform/manifest -b android-10.0.0
    _r33
git config --global user.email "mg.weiminn@gmail.com"
git config --global user.name "weiminn"
repo init -u https://android.googlesource.com/platform/manifest -b android-10.0.0
    _r33
repo sync -j31
source build/envsetup.sh
lunch sdk_phone_x86_64-userdebug
make -j31
```

Ubuntu 23 does not have the repository for libncurses5 because they already have libncurses6, so you have to add old focal repository to your /etc/apt/sources.list:

```
deb http://security.ubuntu.com/ubuntu focal-security main universe
```

After including the old repo, update the repository index and install libncurses5:

```
sudo apt update
sudo apt install libncurses5
```

1.5 November 19, 2023

1.5.1 Building AOSP HAL

Implementing the HAL

Device driver and its API are usually provided by a third-party hardware provider. Shim code include .h files for one or more device drivers.

New devices can communicate with the processor via USB which is popular. Even without driver, the device can be accessed with generic USB commands. *libusb*[4] is

[4]https://libusb.info

a portable, user-mode, and USB-version agnostic library for using USB devices, and supports Android. If the device has a driver, it is likely to be a specialization of the USB command.

```c
#ifndef PROXIMITY_HAL_H
#define PROXIMITY_HAL_H

#include <hardware/hardware.h>

#define ACME_PROXIMITY_SENSOR_MODULE "libproximityhal"

typedef struct proximity_sensor_device proximity_sensor_device_t;

struct value_range {
    int min; int range;
};

typedef struct proximity_params {
    struct value_range precision;
    struct value_range proximity;
}

proximity_params_t;

struct proximity_sensor_device { hw_device_t common;
    int fd;
    proximity_params_t params;
    int (*poll_sensor)(proximity_sensor_device_t *dev, int precision);
}; #endif // PROXIMITY_HAL_H
```

1.6 November 20, 2023

1.6.1 `const` Qualifier

When we have variables whose value we know cannot be changed, and We want to prevent code from inadvertently giving a new value to the variable, we define the variable's type as `const`:

```c
const int bufSize = 512; // input buffer size
bufSize = 512; // error: attempt to write to const object
```

Because we can't change the value of a const object after we create it, it must be initialized:

```c
const int i = get_size(); // ok: initialized at run time
const int j = 42; // ok: initialized at compile time
const int k; // error: k is uninitialized const
```

`const` is Local to the File

When a `const` object is initialized from a compile-time constant as in:

```c
const int bufSize = 512;
```

the compiler will usually replace uses of the variable with its corresponding value during compilation. Thus, when we split a program to multiple files, every file that uses that const must have a access to its initializer. In order to see the initializer, the variable must be defined in every file that wants to use the variable's value. To support this, we define the const in one file, and declare it in other files that use that object.

```
// file_1.cc defines and initializes a const that is accessible to other files
extern const int bufSize = fcn();

// file_1.h
extern const int bufSize; // same bufSize as defined in file_1.cc
```

Because bufSize is const, we must specify extern in order for bufSize to be used in other files. extern signifies that bufSize is not local to this file and that its definition will occur elsewhere.

References to const

Unlike an ordinary reference, a reference to const cannot be used to change the object to which the reference is bound:

```
const int ci = 1024;
const int &r1 = ci; // ok: both reference and underlying object are const
r1 = 42; // error: r1is a reference to const int &r2 =
ci; // error: nonconstreference to a constobject
```

We can bind a reference to a const to a nonconst object, literals, or a more general expression:

```
int i = 42;
const int &r1 = i; // we can bind a const int& to a plain int object
const int &r2 = 42; // ok: r1 is a reference to const
const int &r3 = r1 * 2; // ok: r3 is a reference to
const int &r4 = r * 2; // error: r4is a plain, nonconst reference
```

It is important to realize that a reference to const restricts only what we can do through that reference. Binding a reference to const to an object says nothing about whether the underlying object itself is const.

Pointers and const

Like a reference to const, a pointer to const may not be used to change the object to which the pointer points. We may store the address of a const object only in a pointer to const, where we can modify the pointer itself (change the reference stored inside) but not the object (value) pointed to:

```
const double pi = 3.14; // pi is const; its value may not be changed
double *ptr = &pi; // error: ptr is a plain pointer
const double *cptr = &pi; // ok: cptr may point to a double that is const
*cptr = 42; // error: cannot assign to *cptr
```

Like a reference to `const`, a pointer to `const` says nothing about whether the object to which the pointer points is `const`, that there is no guarantee that an object pointed to by a pointer to `const` won't change.

We can have a pointer that is itself `const`, and the address it holds cannot by changed. We indicate that the pointer is `const` by putting the `const` by the `const`:

```cpp
int errNumb = 0;
int *const curErr = &errNumb; // curErr will always point to errNumb
const double pi = 3.14159;
const double *const pip = &pi; // pip is a constpointer to a const object

*pip = 2.72; // error: pip is a pointer to const  and cannot be changed as the
    pointer is const
// if the object to which curErrpoints (i.e., errNumb) is nonzero
if (*curErr) {
    errorHandler();
    *curErr = 0; // ok: reset the value of the object to which curErr is bound
        because errNumb is not const
}
```

Constant Expressions

A constant expression is an expression whose value cannot change and that can be evaluated at compile time. A literal is a constant expression. A `const` object that is initialized from a constant expression is also a constant expression:

```cpp
const int max_files = 20; // max_files is a constant expression
const int limit = max_files + 1; // limit is a constant expression
int staff_size = 27; // staff_size is not a constant expression
const int sz = get_size(); // sz is not a constant expression
```

Although `staff_size` is initialized from a literal, it is not a constant expression because it is a plain `int`, not a `constint`. Even though `sz` is a `const`, the value of its initializer is not known until run time, and thus, `sz` is not a constant expression.

We can ask the compiler to verify that a variable is a constant expression by declaring the variable in a `constexpr` declaration. Variables declared as `constexpr` are implicitly `const` and must be initiated by constant expressions:

```cpp
constexpr int mf = 20; // 20 is a constant expression
constexpr int limit = mf + 1; // mf+1 is a constant expression
constexpr int sz = size(); // ok only if size is a constexpr function
```

Because a constant expression is one that can be evaluated at compile time, only literal types (arithmetic, reference, and pointer) types can be defined as constant expressions. Custom classes, library IO and `string` types are not literal types. We can point (or bind) to an object that remains at a fixed address.

`constexpr` declaration applies to the pointer, not the type to which the pointer points:

```cpp
const int *p = nullptr; // pis a pointer to a constint
constexpr int *q = nullptr; // qis a constpointer to int
```

`p` is a pointer to `const` (low-level), whereas `q` is a constant pointer (top-level).

1.6.2 Rudimentary AOSP HAL Application

Create Rudimentary Service source file, `rudi.cpp`, in `AOSP/device/generic/goldfish/app/wei_daemon`

```
#include <unistd.h>
#include <stdio.h>
#include <android/log.h>

#define DELAY_SECS 2
#define ALOG(msg) __android_log_write(ANDROID_LOG_DEBUG, "WEIMINN␣PROJ", msg)

int main(int argc, char *argv[]) {
    ALOG("STARTING␣WEIMINN␣PROJECT");

    int n = 0;
    while (true) {
        sleep(DELAY_SECS);
        n++;

        ALOG("TESTING");
    }
}
```

And create Soong build file, `Android.bp` in the same folder:

```
cc_binary {
    name: "weiminn_daemon",
    relative_install_path: "hw",
    init_rc: ["init.weiminn.rc"],
    header_libs: [
        "liblog_headers",
    ],
    srcs: [
        "weiminn.cpp"
    ],
    shared_libs: [
        "liblog",
        "libcutils",
    ],
    static_libs: [
    ],
    vendor: true,
    proprietary: true,
}
```

Add `wei_rudi_daemon` to the `PRODUCT_PACKAGES+=\` attribute of `AOSP/device/generic/goldfish/vendor.mk`

Add the startup script below to the end of the `init.rc` file:

```
service wei_rudi_daemon /vendor/bin/hw/wei_rudi_daemon
    class main
    user system
    group system
    oneshot
```

But the service cannot start due to SE Policy not being implemented for the service yet:

```
11-21 02:13:54.242  1862  1862 W cp : type=1400 audit(0.0:231): avc: denied {
    getattr } for path="/vendor/bin/hw/wei_rudi_daemon" dev="dm-3" ino=110
    scontext=u:r:shell:s0 tcontext=u:object_r:vendor_file:s0 tclass=file
    permissive=0
```

1.7 November 21, 2023

1.7.1 Fixing SELinux Policy to Start System Service

Add `seclabel` to the startup script:

```
service weiminn_daemon /vendor/bin/hw/weiminn_daemon
    class main
    user system
    group system
    seclabel u:r:ueventd:s0
```

Add `start weiminn_daemon` under `on early-init` right after `start ueventd`.

Got a new permission denied error this time:

```
11-21 09:45:47.732 0 0 E init : cannot execv('/vendor/bin/hw/weiminn_daemon'). See
     the 'Debugging init' section of init's README.md for tips: Permission denied
11-21 09:45:47.733 0 0 I init : Service 'weiminn_daemon' (pid 1402) exited with
    status 127
11-21 09:45:47.733 0 0 I init : Sending signal 9 to service 'weiminn_daemon' (pid
    1402) process group...
```

Adding `device/generic/goldfish/sepolicy/x86/weiminn.te` with following contents:

```
type weiminn, domain;
permissive weiminn;
type weiminn_exec, vendor_file_type, exec_type, file_type;

init_daemon_domain(weiminn)
```

and changed `seclabel` of the startup script to `seclabel u:r:weiminn_exec:s0`. And it reverts to the previous error:

```
11-21 10:19:43.865 0 0 I init : starting service 'weiminn_daemon'...
11-21 10:19:43.866 0 0 F init : cannot setexeccon('u:r:weiminn_exec:s0') for
    weiminn_daemon: Invalid argument
11-21 10:19:43.867 0 0 I init : Service 'weiminn_daemon' (pid 1827) exited with
    status 6
11-21 10:19:43.868 0 0 I init : Sending signal 9 to service 'weiminn_daemon' (pid
    1827) process group...
11-21 10:19:43.868 0 0 I libprocessgroup : Successfully killed process cgroup uid
    1000 pid 1827 in 0ms
```

1.7.2 Memory Dump

Try on Android Emulator

Set up emulator environment by appending the following inside `~/.bashrc`:

```
export ANDROID_SDK_ROOT=~/Android/Sdk
export ANDROID_HOME=~/Android/Sdk
export ANDROID_AVD_HOME=~/.android/avd

PATH=$PATH:$ANDROID_SDK_ROOT/emulator
```

Register the path:

```
source ~/.bashrc
```

Create new AVD, Pixel 7 Pro with Tramisu (Android 13 with Google APIs, not Google Play), via Android Studio.

Run emulator using command:

```
emulator -avd Pixel_7_Pro_API_33
```

Load APK into emulator:

```
adb devices # to get id of running instance
adb root
adb install -r de.drmaxnix.birthdaycountdown.apk
```

Clone Fridump[5]:

```
git clone https://github.com/Nightbringer21/fridump
code fridump
pip install frida frida-tools
```

Download and Run Fridump dependencies:

```
git clone https://github.com/Nightbringer21/fridump
code fridump
adb push frida-server /data/local/tmp
sudo sysctl kernel.yama.ptrace_scope=0 && frida-ps -D emulator-5554
frida-ps -D emulator-5554 | grep Birth # to get process ID of the Birthday app

# inside emulator shell
./data/local/tmp/frida-server
```

Found out that Fridump doesn't support the latest Android 13 and API 34, so download Android 10 with API 29 with Pixel 3a.

Run Fridump:

```
python fridump.py -U -s Birthday\ Countdown
```

and it works now!

1.8 November 22, 2023

1.8.1 C++ Types and Data Structures

Aliases

Traditionally, we use `typedef` for synonym for another type:

[5]https://pentestcorner.com/introduction-to-fridump

```cpp
typedef double wages; // wages is a synonym for double
typedef wages base, *p; // base is a synonym for double, p for double*
```

The new standard introduced a second way to define **alias declaration** type alias:

```cpp
using SI = Sales_item; // SI is a synonym for Sales_item
```

It is also possible to declare "pointer to" alias:

```cpp
typedef char *pstring;
const pstring cstr = 0; // equivalent to char *const cstr = 0;
const pstring *ps; // equivalent to const char *ps;
```

`auto` Type Specifier

We can let the compiler deduce the type from the initializer for us by using the `auto` type specifier:

```cpp
// the type of item is deduced from the type of the result of adding val1 and val2
auto item = val1 + val2; // item initialized to the result of val1+val2
```

The initializaer for all the variables muxt have types that are consistent with each other:

```cpp
auto i = 0, *p = &i; // ok: i is int and p is a pointer to int
auto sz = 0, pi = 3.14; // error: inconsistent types for sz and pi
```

`auto` ignores top-level `const`s, and only low-level initializer are usually kept:

```cpp
const int ci = i, &cr = ci;
auto b = ci; // b is an int (const in ci is dropped)
auto c = cr; // c is an int (cr is an alias for ci whose const is dropped)
auto d = &i; // d is an int* (& of an int object is int*)
auto e = &ci; // e is const int* (& of a const object is low-level const)
const auto f = ci; // deduced type of ci is int; f has type const int
```

Normal initialization rules still apply for reference to the `auto`-deduced type:

```cpp
const int ci = i, &cr = ci;
auto &g = ci; // g is a const int& that is bound to ci
auto &h = 42; // error: we c a n t bind a plain reference to a literal
const auto &j = 42; // ok: we can bind a const reference to a literal
```

When we ask for reference to an `auto`-deduced type, top-level `const`s in the initializer are not ignored. `const`s are not top-level when we bind a reference to an initializer.

`decltype` Type Specifier

We want to define a variable with a type that the compiler deduces from an expression but do not want to use the expression to initialize the variable:

```cpp
decltype(f()) sum = x; // sum has whatever type f returns
```

The compiler does not call `f`, but it uses the type that such a call would return as the type for `sum`.

`decltype` handles top-level `const` and references differs subtly from `auto`

```cpp
const int ci = 0, &cj = ci;
decltype(ci) x = 0; // x has type const int
decltype(cj) y = x; // y has type const int& and is bound to x
decltype(cj) z; // error: z is a reference and must be initialized
```