UNIT – II Basic Design

Introduction – Basics of Embedded systems design

As computer technology advances, so its technology becomes embedded in more and more electronic products. The capabilities provided by embedded systems enable electronic equipment to have far greater capabilities than would be possible if only hardware techniques were used.

As a result, embedded systems are found in all manner of electronic equipment and gadgets. From small amounts of processing in items like electronic timers, to far more complicated embedded systems like gaming consoles and even major factory and other industrial systems.

The technique gains its name from the fact that the software is embedded into it for a particular application. The embedded system is not like a PC or other computer that can run a variety of programmes and fulfil a whole host of tasks, but instead the item using embedded software is focussed on one task or application. To meet this need, the item using an embedded techniques has its software preloaded, although updates may be undertaken from time to time.

Embedded systems basics

It may be asked what an embedded system is. With many processor-based systems and computers it is useful to define what an embedded system is. A convenient definition for an embedded system is: An embedded system is any computer system contained within a product that is not described as a computer.

Using this embedded system definition, it is possible to understand the various basic characteristics one. Typically, they are:

- Embedded systems are designed for a specific task. Although they use computer techniques, they cannot be used as a general-purpose computer using a variety of different programmes for different task. In this way their function can be focussed onto what they need to do, and they can accordingly be made cheaper and more efficiently.
- The software for embedded systems is normally referred to as firmware. Rather than being stored on a disc, where many programmes can be stored, the single programme for an embedded system is normally stored on chip and it is referred to as firmware.

Embedded systems contain two main elements:

- ***Embedded system hardware:*** As with any electronic system, an embedded system requires a hardware platform on which to run. The hardware will be based around a microprocessor or microcontroller. The embedded system hardware will also contain other elements including memory, input output (I/O) interfaces as well as the user interface, and the display.
- ***Embedded system software:*** The embedded system software is written to perform a particular function. It is typically written in a high-level format and then compiled down to provide code that can be lodged within a non-volatile memory within the hardware.

Embedded systems hardware

When using an embedded system there is a choice between the use of a microcontroller or a microprocessor.

- ***Microcontroller based systems:*** A microcontroller is essentially a CPU, central processor unit, or processor with integrated memory or peripheral devices. As fewer external components are needed, embedded system using microcontrollers tend to be more widely used
- ***Microprocessor based systems:*** Microprocessors contain a CPU but use external chips for memory and peripheral interfaces. As they require more devices on the board, but they allow more expansion and selection of exact peripherals, etc, this approach tends to be used for the larger embedded systems.

Whatever type of processor is used in the embedded system, it may be a very general-purpose type of one of the many highly specialised processors intended for a particular application. In some cases, custom designed chips may be viable for a particular application if quantities are sufficiently high. One common example of a standard class of dedicated processor is the digital signal processor, DSP. This type of processor is used for processing audio and image files. Processing is required very quickly as they may be used in applications such as mobile phones and the like.

Embedded systems software

One of the key elements of any embedded system is the software that is used to run the microcontroller.

There is a variety of ways that this can be written:

- ***Machine code:*** Machine code is the most basic code that is used for the processor unit. The code is normally in hex code and provides the basic instructions for each operation of the processor. This form of code is rarely used for embedded systems these days.
- ***Programming language:*** Writing machine code is very laborious and time consuming. It is difficult to understand and debug. To overcome this, high level programming languages are often used. Languages including C, C++, etc are commonly used.

The code for the embedded system will typically be stored on a form of non-volatile memory held on the processor board. The code is called firmware - the idea is that it is not updated in the same way that software is, being held in the embedded system and it cannot be changed by the user. Often it is possible to update the software, but this can mean changing the memory card on which the firmware is held, or by updating it in another way.

Often additional tools may be used to help with the development of the firmware. Often programmes can become complicated, and it is necessary to ensure the firm ware for the embedded system operates correctly.

Embedded systems design tools

Many embedded systems are complicated and require large levels of software for them to operate. Developing this software can be timing consuming, and it must be very accurate for the embedded system to operate correctly. Coding in embedded systems is one of the main areas where faults occur. To help simplify the process, software development tools are normally used. These help the software developer to programme more quickly, and more accurately.

Microcontroller MCU in embedded systems:

when developing embedded system hardware there is a choice of using a microprocessor or a microcontroller - when using a microcontroller what are the best approaches. When developing an embedded system, one of the options is to base the computational hardware around a microcontroller, MCU rather than a microprocessor, MPU.

Both approaches have their attractions, but generally they will be found in different applications. Typically, the microcontroller, MCU, is found in applications where size, low power and low cost are key requirements. The MCU, microcontroller is different to a microprocessor in that it contains more elements of the overall processing engine within the one chip.

Bringing most of the processing engine components onto a single chip reduces size and cost. This enables it to become economical viable to digitally control even more devices and processes. Also, it is found that mixed signal microcontrollers are being increasingly used, integrating analogue components needed to control non-digital electronic systems.

Microcontroller basics

Microcontrollers comprise the main elements of a small computer system on a single chip. They contain the memory, and IO as well as the CPU one the same chip. This considerably reduces the size, making them ideal for small embedded systems, but means that there are compromises in terms of performance and flexibility.

As microcontrollers are often intended for low power and low processing applications, some microcontrollers may only use 4-bit words and they may also operate with very low clock rates - some 10 kHz and less to conserve power. This means that some MCUs may only consume a milliwatt or so and they may also have sleep consumption levels of a few nanowatts. At the other end of the scale some MCUs may need much higher levels of performance and may have very much higher clock speeds and power consumption.

Microcontroller advantages & disadvantages

As with any device of system approach their various advantages and disadvantages of microcontrollers need to be considered when undertaking a new design.

MICROCONTROLLER ADVANTAGES & DISADVANTAGES SUMMARY

ADVANTAGES	DISADVANTAGES
• Lower cost because many elements of the processor are contained within the one chip resulting in lower chip cost and board cost. • Lower power consumption. • Integrating all components onto one chip enables processor to be optimised for a given application.	• Less flexibility because all components are integrated into the one chip. • Limited performance because the size of memory is limited by what can be accommodated on the chip. • MCUs tend to be application specific so the choice may be limited.

Embedded Processing & Digital / Programmable Logic

Digital or logic circuits have been inextricably linked with programmable logic and embedded and computer processing these days. Digital technology has made major advances. Whereas once all electronic circuits were based around analogue techniques, nowadays digital approaches tend to dominate. Not only are there digital or logic circuits, but programmable logic in the form of field programmable gate arrays and other forms of logic circuit are also available. Of course, digital technology forms the platform used for embedded and processors of all kinds. In this way digital or logic technology is inextricably linked with programmable logic, and software.

Embedded OS

What is an embedded operating system?

An embedded operating system is a specialized operating system (OS) designed to perform a specific task for a device that is not a computer. The main job of an embedded OS is to run the code that allows the device to do its job. The embedded OS also makes the device's hardware accessible to software that is running on top of the OS.

An embedded OS often works within an embedded system. An embedded system is a computer that supports a machine. It performs one task in the bigger machine. Examples include computer systems in cars, traffic lights, digital televisions, ATMs, airplane controls, point of sale (POS) terminals, digital cameras, GPS navigation systems, elevators and Smart meters.

Networks of devices containing embedded systems make up the internet of things (IoT). The embedded systems perform basic operations inside IoT devices, such as transferring data over a network without human interaction

How does an embedded OS work?

An embedded OS enables an embedded device to do its job within a larger system. It communicates with the hardware of the embedded system to perform a specific function. For example, an elevator might contain an embedded system, such as a microprocessor or microcontroller, that lets it understand which buttons the passenger is pressing. The embedded software that runs on that system is the embedded OS.

In contrast to an OS for a general-purpose computer, an embedded OS has limited functionality. Depending on the device in question, the system may only run a single embedded application. However, that application is likely crucial to the device's operation. Given that, an embedded OS must be reliable and able to run with constraints on memory and processing power.

In the case of a Raspberry PI system on a chip, an SD card acts as the device's hard drive and contains the code that runs on the device. The SD card is removable, so its contents can be modified on demand. Various operating systems can run on Raspberry PI devices. The embedded OS makes the device's hardware -- such as USB and HDMI ports -- accessible to the application running on top of the OS.

Examples of embedded OS devices

Some examples of devices with embedded OSes include the following:

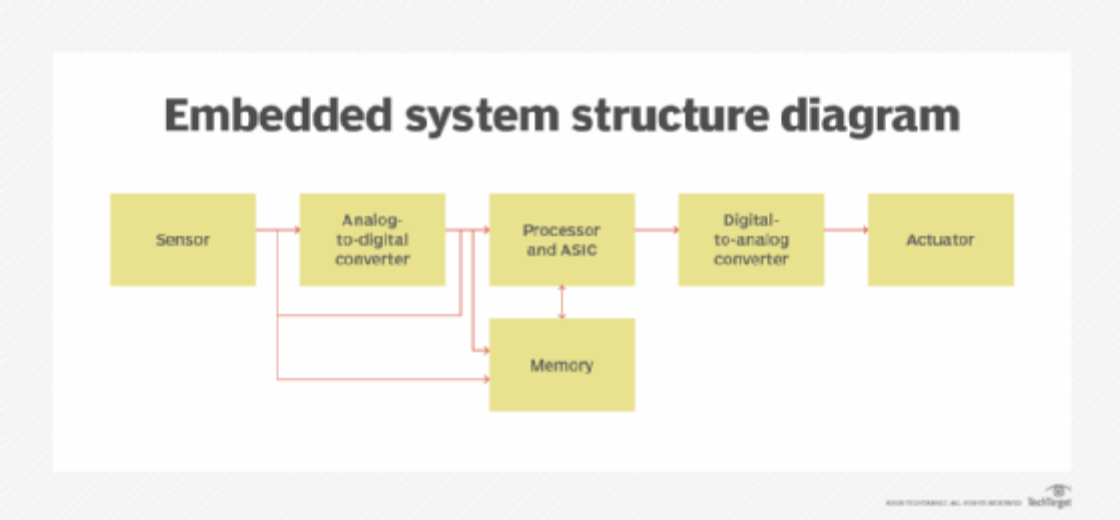
- ATMs
- cellphones
- electric vehicles
- industrial control systems (ICS)
- Arduino-based devices

Arduino is an open source platform with a microcontroller that processes simple inputs, such as temperature or pressure, and turns them into outputs. These devices have a basic embedded OS that acts like a boot loader and a command interpreter. An example of an Arduino-based device is a remote-control car. The Arduino reads inputs from the car's controller and sends output information and commands to other components, such as the brakes.

Common uses of embedded OSes

Embedded OSes are put to a variety of uses, including the following:

- **ATMs.** ATMs have basic OSes that enable the machine to read a user's debit card and personal identification number input and perform bank account functions like withdrawal or checking balances. The OS does little else but react to user inputs and communicate with the ATM hardware.
- **Cellphones.** Cellphones require an OS like Android or iOS to boot the phone and enable applications to communicate with other phone hardware.
- **Electric vehicles.** Microcontrollers host embedded OSes that handle functions like braking or pressure sensing. For example, a certain amount of pressure on the front bumper may cause the airbag to go off. This type of function is known as reactive operation because it reacts to an input.
- **Industrial control systems.** Sensors are used in industrial control systems to measure factory conditions and send alerts if they become dangerous. Sensors contain an embedded OS that enable them to perform these tasks.
- **Traffic lights.** Embedded OSes enable a traffic light to cycle through different signals at programmed intervals.
- **Basic input/output system.** In some cases, BIOS could be considered an embedded OS because it is the firmware that enables a desktop computer's more complex OS to interact with the computer hardware.



Types of embedded OSes

Embedded OS are designed for the task they will perform. The various types of operating systems include the following:

- **Multitasking operating system.** A multitasking OS can perform several tasks at once. It uses job scheduling to perform basic tasks. For example, a cellphone OS divides up CPU resources among multiple tasks.
- **Real-time operating system.** A real-time OS is designed to be reactive. It processes inputs when they are received and responds within a specific timeframe. If the response time falls outside of the specified time period, the system could fail. Real-time OSes sometimes use rate monotonic scheduling, which assigns priorities to tasks.
- **Single loop control system.** This type of embedded OS exercises control over a single variable. An example would be temperature control in a smart home. A smart thermostat measures the temperature in the house and if it exceeds the limit set by the user, turns off the heat.

Embedded vs. non-embedded OSes: What's the difference?

An embedded OS may reside on a chip within an electronic device. They are often limited in the scope of what they can do.

In contrast, a non-embedded OS runs from a hard disk or a solid-state drive. Non-embedded OSes, such as Windows 10 or Mac OS, are configurable and upgradable. They are designed for general-purpose use. Another difference between embedded and non-embedded OSes is in how the operating system is coded. Embedded OSes are usually contained in a single executable image and execute one task. Desktop operating systems and network operating systems contain many applications. Embedded OSes also have a minimal or no user interface (UI). Embedded OSes, on the other hand, have a more dynamic UI.

Design constraints for Mobile applications - Both Hardware and Software related

1. Screen size, sensors, and interactions

Screen sizes are much smaller when designing mobile software. You have a limited canvas, and your design should be simple with enough space for users to touch and interact with different elements. Users may use your software with one hand or two hands or use any of the supported gestures to interact with your software.

Depending on your content, you could choose a spatial format (map view), a list format, a block design, or other ways to display your content. If you are building for iOS or Windows Phone, there are fixed screen sizes and resolutions to plan for. If you're building for Android, you have more variations to keep in mind.

You also have a variety of sensors and enablers that can help you design interactions. While many of them are great enablers for design, they also come with their constraints (eg: GPS when used indoors with a spotty data connection may not return a location. How will your software handle this?)

2. Storage and cache sizes

Depending on what you're building, you might have options to download and store/cache content for offline usage. This could help reduce the data transfer for online transactions, making the product feel more responsive. Think about whether the storage is available on a memory card (removable storage) or internal memory. With a memory card, you must tackle states where the user might delete/modify content on the memory card, pull out the memory card while interacting with the software or for memory card corruption. With internal software, you may have limited allotment for storage, and the possibility of cached data getting overwritten.

3. Latencies

It's a good idea to time operations and load times. Users on a mobile expect an instant response for something they do.

A lot of this depends on your product architecture. You may be able to do some architectural jugglery to deliver a phased experience. Think of how images load in your browser on a slow connection – a lighter pixelated version loads first, and then the sharper image loads. The intermediate stage keeps the user occupied till the final image loads.

Even if you can't eliminate all latencies, you could plan to tackle the intermediate wait stages with engaging messages or cached content. First time use is a special case. When the app is used for the very first time, there are usually a lot of background setup activities that need to be done. One way of mitigating the wait during this time is to have a front-end tutorial for users to engage with, while your app performs setup or bookkeeping in the background. On subsequent usage, you could use data/states cached to deliver a faster experience.

4. Network issues

Any mobile product must contend with network latencies and failure points. Plan for these earlier on, or you may find you're losing users because they keep staring at a 'Try again later' or 'Connecting…' message. Try to avoid blocking spinners – one that does not let the user do anything while he waits. See if your platform and design permit loading partial elements and cached data to keep him occupied when he waits.

Depending on the platform you develop for, you may also have cases where the user can pull out the SIM card during an operation and put it back in or walk in and out of Wi-Fi zones. This affects not just the software on the phone, but also the backend that supports it. Finally: try identifying failure points and use that as avenues to show your product's personality. Twitter's fail-whale is a great example of how a failure point can become an icon.

5. Data use requirements

In addition to just data connection, the SIM card also provides information like MCC (Mobile Country Code) and MNC (Mobile Network Code) that you may have used to identify which country the user is and what network he is connected to. You can get info on whether he is on a roaming network and home network and adjust the amount of data required accordingly. For example, you may choose to download and cache a lot more data when the user is connected to Wi-Fi or a home network but take a download-as-required approach when he is on roaming.

6. Fonts, language, and tone of voice

The fonts and language you use in the product makes a big difference to the user experience. Language and tone of voice reflect your product's personality. The tone can be informal (Howdy!), formal (Welcome), friendly (let's get started), impersonal (press this button to continue) or any other variation that suits your product. It's important to realize what tone your customers are comfortable with and stick to one tone through the product.

Point to note: if you're building a global product, be careful about the tone of voice you choose. What comes across as informal in one culture may seem blasé in another.

Languages may bring their own issues – getting translations, checking that terms do not get curtailed, etc. You may also have to plan for right-to-left languages like Arabic.

7. Corner cases for product usage

There are many use cases that you may not be able to test during development. For example, it might be difficult for you to test 'international roaming' use cases unless you procure an international SIM or send an Indian SIM to a tester/friend sitting abroad.

For many of these, it is usually possible to visualize multiple user journeys and scenarios when you're designing the product and think of which cases you want to implement.

You may decide that there are cases that are too extreme to develop for, which is fine. If you have identified many use cases like this, you could at least plan for these with engaging error/fail messages or take them as opportunities to get feedback (eg: this product was not designed to work in international roaming scenarios. Send us a quick email if you think we should be focusing on this – with an option for the user to mail quick feedback from the mobile client.

Design of your mobile application.

1.1 USER REQUIREMENTS AND ORGANIZATIONAL CONSTRAINTS

a. Application Functionality Requirements – which user interactions and features need to be supported by the application? Which mobile libraries, SDK's, and API's need to be adopted?

b. Hardware requirements of the application – E.g. whether it requires access to global positioning system (GPS) or a camera or the user's address book or emails etc. This also impacts the hardware device that can be chosen for deployment of the application.

1.2 WHAT IS RUNNING IN THE BACKEND

The backend server technology and DB schema hugely determine the flexibility of the actions and functionality available on the client application. E.g. whether metadata updates can be easily communicated to the client device of not, which data transfer protocol to use?

1.3 WHETHER TO BUILD A RICH CLIENT, A THIN WEB CLIENT, OR RICH INTERNET APPLICATION (RIA) ?

a. Rich Client – used if application requires local processing and must work in an occasionally connected scenario. These are more complex to install and maintain.

b. Thin client – used if application can depend on server processing and will always be fully connected.

c. RIA Client – used if application requires a rich user interface (UI), limited access to local resources, and must be portable to other platforms.

1.4 WHAT ALL DEVICE TYPES TO SUPPORT

Hardware and software platforms should be chosen on basis of how their speed, power consumption, memory and CPU resources compare with the application's requirements (and not just on basis of the ease of programming). For each device type, look at resource constraints – storage memory, CPU, process restrictions, UI – screen size, resolution (DPI).

1.5 SDK AND THE DEVELOPMENT TOOL ENVIRONMENTS AVAILABLE:

a. Original Company specific - All major mobile platforms (like Android, iPhone, Windows)

release company specific SDK's along with best practices and specifications. These should be followed and used to build the most comprehensive and good-looking application for that platform.
b. Cross Platform / Cross Hardware - A lot of cross-hardware and cross platform frameworks are also now available and gaining popularity. Whether or not these can be used for the construction of your application depends on the functionality and the native hardware features it wishes to expose.

1.6 MOBILE CONSTRAINTS

Security, Battery Life, Network Bandwidth, Limited CPU, Memory Storage capacity, Security (no corporate firewall but public internet), Performance and Authentication. Out of these Battery Life is the most limiting aspect on the mobile

1.7 PHYSICAL NUANCES IN HARDWARE

E.g. presence of the back button on the Blackberry and Nokia phones vs. the home key and the back button on the screen on the iPhone.

1.8 CONNECTIVITY –fully connected, occasionally connected or limited-bandwidth scenarios need to be considered.

1.9 USABILITY (USER INTERFACE)

"Mobile optimized experiences produce on average 75% higher rate of engagements per visit for mobile user." - Source – Ed Hewett from Omniture

a. Has to be smaller, simpler, and more concise for the mobile platform.
b. Should not take a long time to load, should consume little bandwidth.
c. The navigation, layout, interaction, and flow should be appropriate for the small screen.

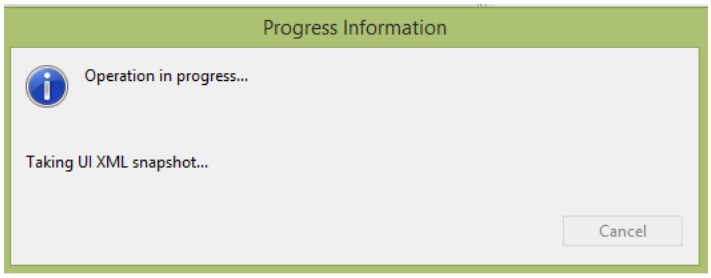
Android - UI Design

UI screen components

A typical user interface of an android application consists of action bar and the application content area.

- Main Action Bar
- View Control
- Content Area
- Split Action Bar

These components have also been shown in the image below –



Understanding Screen Components

The basic unit of android application is the activity. A UI is defined in an xml file. During compilation, each element in the XML is compiled into equivalent Android GUI class with attributes represented by methods.

View and ViewGroups

An activity is consisting of views. A view is just a widget that appears on the screen. It could be button e.t.c. One or more views can be grouped together into one GroupView. Example of ViewGroup includes layouts.

Types of layouts

There are many types of layouts. Some of which are listed below −

- Linear Layout
- Absolute Layout
- Table Layout
- Frame Layout
- Relative Layout

Linear Layout

Linear layout is further divided into horizontal and vertical layout. It means it can arrange views in a single column or in a single row. Here is the code of linear layout(vertical) that includes a text view.

```
<?xml version=”1.0” encoding=”utf-8”?>
<LinearLayout xmlns:android=”http://schemas.android.com/apk/res/android”
android:layout_width=”fill_parent”
android:layout_height=”fill_parent”
android:orientation=”vertical” >
<TextView
android:layout_width=”fill_parent”
android:layout_height=”wrap_content”
android:text=”@string/hello” />
</LinearLayout>
```

Absolute Layout

The Absolute Layout enables you to specify the exact location of its children. It can be declared like this.

```
<AbsoluteLayout
android:layout_width=”fill_parent”
android:layout_height=”fill_parent”
xmlns:android=”http://schemas.android.com/apk/res/android” >
<Button
android:layout_width=”188dp”
android:layout_height=”wrap_content”
```

```
android:text=”Button”
android:layout_x=”126px”
android:layout_y=”361px” />
</AbsoluteLayout>
```

Table Layout

The Table Layout groups views into rows and columns. It can be declared like this.

```
<TableLayout
xmlns:android=”http://schemas.android.com/apk/res/android”
android:layout_height=”fill_parent”
android:layout_width=”fill_parent” >
<TableRow>
<TextView
android:text=”User Name:”
android:width =”120dp”
/>
<EditText
android:id=”@+id/txtUserName”
android:width=”200dp” />
</TableRow>
</TableLayout>
```

Relative Layout

The Relative Layout enables you to specify how child views are positioned relative to each other.It can be declared like this.

```
<RelativeLayout
android:id=”@+id/RLayout”
android:layout_width=”fill_parent”
android:layout_height=”fill_parent”
xmlns:android=”http://schemas.android.com/apk/res/android” >
</RelativeLayout>
```

Frame Layout

The Frame Layout is a placeholder on screen that you can use to display a single view. It can be declared like this.

```
<?xml version=”1.0” encoding=”utf-8”?>
<FrameLayout
android:layout_width=”wrap_content”
android:layout android:layout_alignLeft=”@+id/lblComments”
android:layout_below=”@+id/lblComments”
android:layout_centerHorizontal=”true” >
<ImageView
android:src = “@drawable/droid”
```

```
android:layout_width=”wrap_content”
android:layout_height=”wrap_content” />
</FrameLayout>
```

Apart form these attributes, there are other attributes that are common in all views and ViewGroups. They are listed below −

Sr.No	View & description
1	**layout_width** Specifies the width of the View or ViewGroup
2	**layout_height** Specifies the height of the View or ViewGroup
3	**layout_marginTop** Specifies extra space on the top side of the View or ViewGroup
4	**layout_marginBottom** Specifies extra space on the bottom side of the View or ViewGroup
5	**layout_marginLeft** Specifies extra space on the left side of the View or ViewGroup
6	**layout_marginRight** Specifies extra space on the right side of the View or ViewGroup
7	**layout_gravity** Specifies how child Views are positioned
8	**layout_weight** Specifies how much of the extra space in the layout should be allocated to the View

Units of Measurement

When you are specifying the size of an element on an Android UI, you should remember the following units of measurement.

Sr.No	Unit & description
1	**dp** Density-independent pixel. 1 dp is equivalent to one pixel on a 160 dpi screen.
2	**sp** Scale-independent pixel. This is similar to dp and is recommended for specifying font sizes
3	**pt** Point. A point is defined to be 1/72 of an inch, based on the physical screen size.
4	**px** Pixel. Corresponds to actual pixels on the screen

Screen Densities

Sr.No	Density & DPI
1	**Low density (ldpi)** 120 dpi
2	**Medium density (mdpi)** 160 dpi
3	**High density (hdpi)** 240 dpi
4	**Extra High density (xhdpi)** 320 dpi

Optimizing layouts

Here are some of the guidelines for creating efficient layouts.
• Avoid unnecessary nesting
• Avoid using too many Views
• Avoid deep nesting

Gestures and Touch Events

Gesture recognition and handling touch events is an important part of developing user interactions. Handling standard events such as clicks, long clicks, key presses, etc are very basic and handled in other guides. This guide is focused on handling other more specialized gestures such as:
• Swiping in a direction
• Double tapping for zooming
• Pinch to zoom in or out
• Dragging and dropping
• Effects while scrolling a list

Usage

Handling Touches

At the heart of all gestures is the onTouchListener and the onTouch method which has access to MotionEvent data. Every view has an onTouchListener which can be specified:

```
myView.setOnTouchListener(new OnTouchListener() {
@Override
public boolean onTouch(View v, MotionEvent event) {
// Interpret MotionEvent data
// Handle touch here
return true;
}
});
```

Each onTouch event has access to the MotionEvent which describe movements in terms of an action code and a set of axis values. The action code specifies the state change that occurred such as a pointer going down or up. The axis values describe the position and other movement properties:

• **getAction()** - Returns an integer constant such as MotionEvent.ACTION_DOWN , MotionEvent.ACTION_MOVE , and MotionEvent.ACTION_UP

• **getX()** - Returns the x coordinate of the touch event

• g**etY()** - Returns the y coordinate of the touch event

Note that every touch event can be propagated through the entire affected view hierarchy. Not only can the touched view respond to the event but every layout that contains the view has an opportunity as well.

Handling Multi Touch Events

Note that getAction() normally includes information about both the action as well as the pointer index. In single-touch events, there is only one pointer (set to 0), so no bitmap mask is needed. In multiple touch events (i.e pinch open or pinch close), however, there are multiple fingers involved and a non-zero pointer index may be included when calling getAction() . As a result, there are other methods that should be used to determine the touch event:

• getActionMasked() - extract the action event without the pointer index

• getActionIndex() - extract the pointer index used

The events associated with other pointers usually start with MotionEvent.ACTION_POINTER such as MotionEvent.ACTION_POINTER_DOWN and MotionEvent.ACTION_POINTER_UP

The getPointerCount()

on the MotionEvent can be used to determine how many pointers are active in this touch sequence.

Gesture Detectors

Within an onTouch event, we can then use a GestureDetector to understand gestures based on a series of motion events. Gestures are often used for user interactions within an app. Let's take a look at how to implement common gestures.

For easy gesture detection using a third-party library, check out the popular Sensey library which greatly simplifies the process of attaching multiple gestures to your views.

Double Tapping

You can enable double tap events for any view within your activity using the OnDoubleTapListener. First, copy the code for OnDoubleTapListener into your application and then you can apply the listener with:

```
myView.setOnTouchListener(new OnDoubleTapListener(this) {
@Override
public void onDoubleTap(MotionEvent e) {
Toast.makeText(MainActivity.this, "Double Tap", Toast.LENGTH_SHORT).show();
}
```

Achieving quality constraints

Mobile Application Quality Attributes

Quality Attribute	Applicability	Realization Plan	Priority
Maintainability	Maintainability is important so that future developers can work on the application.	The application will use javascript based frameworks consistently, so that all applications developed use the same technologies.	HIGH
Scalability	Scalability of the application is relatively low priority. Being released as a mobile application will allow it to be released to many users at once. Scalability is much more import in the recognition server.	The application will be released using standard mobile application platforms. Work that does not scale well will be offloaded to the recognition server.	LOW
Performance	Performance of the application is very important. Image recognition performance will be handled on the image recognition server, so the performance that is important is the application's responsiveness.	Currently not handled by the Rock Raiders Team. Public Beta testing is expected to be performed by the RocReadaR main team.	HIGH
Usability	Usability is incredibly important to the customers, as it will be used by the users directly. This is of ever increasing importance with the growing expectations of mobile applications.	Currently not handled by the Rock Raiders Team. Public Beta testing is expected to be performed by the RocReadaR main team.	HIGH
Security	Currently not handled by the Rock Raiders Team.	Currently not handled by the Rock Raiders Team.	UNKNOWN

Availability	Availability of the application is relatively low priority. Being released as a mobile application will allow it to be available almost 100% of the time.	The application will be released using standard mobile application platforms.	HIGH

Management Portal Quality Attributes

Quality Attribute	Applicability	Realization Plan	Priority
Maintainability	Maintainability is important so that future developers can work on the application.	The application will use javascript based frameworks consistently, so that all applications developed use the same technologies.	HIGH
Scalability	Scalability of the management portal is relatively low priority. Image recognition however, will require high scalability.	The application will be hosted in a cloud platform. Scaling to more instances will remain an option in case it is required.	LOW
Performance	Performance is of medium importance. The application should be very usable, but it is not terrible if the publishers experience minor delays.	The application will be hosted in a cloud platform. Scaling to more instances will remain an option in case it is required. Formal usability testing with both publishers and non-publishers will be performed in the middle of development, and slow sections will be identified.	MEDIUM
Usability	Usability is incredibly important to the customers, as it will be used by the publishers directly.	Formal usability testing with both publishers and non-publishers will be performed in the middle of development.	HIGH
Security	Security is somewhat important, however it is not expected to be an issue. The	Standard security practices regarding user accounts and web technologies will be practiced. User	MEDIUM