CHALMERS

Gemensam kodbas i C för mobila plattformar

Examensarbete inom Högskoleingenjörsprogrammet i Datateknik

OLLE SVENSSON
ROBIN TÖRNQUIST

Institutionen för data- och informationsteknik
CHALMERS TEKNISKA HÖGSKOLA
Göteborg, Sverige 2014

Gemensam kodbas i C för mobila plattformar

Olle Svensson, Robin Törnquist

© OLLE SVENSSON, ROBIN TÖRNQUIST, 2014

Institutionen för data- och informationsteknik
Chalmers tekniska högskola
412 96 Göteborg
Tel: 031-772 1000
Fax: 031-772 3663

Institutionen för data- och informationsteknik
Göteborg, 2014

Abstract

Development of smartphone applications involves different programming languages for different platforms. Because of this, there is an interest in solutions where code written in one language is used on multiple platforms. This report examines the possibility of a mutual core of code in the programming language C as a solution to the problem. The purpose of the examination has been to determine suitable functionality for the mutual core. The basis for the examination has been three different areas of functionality. Problems identified during the examination in combination with the purpose of the mutual core were used to develop guidelines for suitable functionality. If the purpose of the mutual core is to reduce the time of development, a mutual core was considered as not suitable. Otherwise, a mutual core was considered as suitable if standard functions offered by the C language were used exclusively.

Sammanfattning

Vid utveckling av mobilapplikationer används olika programmeringsspråk för olika plattformar. På grund av detta finns ett intresse av lösningar där samma kod kan användas till flera plattformar. Rapporten utreder möjligheten av en gemensam kodbas i programmeringsspråket C som en lösning till problemet. Syftet med utredningen har varit att undersöka vilken typ av funktionalitet som är lämplig att ha i den gemensamma kodbasen. Tre skilda implementationsområden har stått till grund för utredningen som gjorts. Utifrån de problem som identifierats i utredningen tillsammans med syftet för den gemensamma koden har generella riktlinjer tagits fram för vilken typ av funktionalitet som är lämplig. Om syftet med den gemensamma kodbasen är att minska utvecklingstiden bedömdes en gemensam kodbas som olämplig. Om syftet är ett annat bedömdes en gemensam kodbas som lämplig i de fall då lösningen endast använder sig av den standardfunktionalitet som C erbjuder.

Beteckningar

ADT	Android Developer Tools
API	Application Programming Interface
BLE	Bluetooth Low Energy
Eclipse	Utvecklingsmiljö för bland annat Java, Android, C och C++.
Git	Ett versionshanteringssystem.
Git-repo	Git repository. En samling filer som versionshanteras med hjälp av Git.
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
JNI	Java Native Interface
NDK	Native Development Kit
SDK	Software Development Kit
SSL	Secure Sockets Layer
VM	Virtual Machine
Xcode	Utvecklingsmiljö för Apple-produkter.

Innehåll

1. Inledning

1.1 Bakgrund

Det finns idag flera plattformar för mobiltelefoner, varav Android och iOS är de två mest använda. I takt med att marknaden för mobilapplikationer växer ökar intresset för en lösning där en och samma källkod kan användas för en applikation till flera plattformar. Intresset av gemensam kod kan vara intressant av flera olika anledningar. En anledning kan vara att minska utvecklingstiden genom att endast behöva skriva en gemensam kod som går att använda till flera plattformar. Anledningen kan också vara att det finns ett särskilt värde i att samma kod körs på flera plattformar. Ett exempel är när det finns ett krav på att koden ska verifieras.

Ur ett bredare perspektiv kan det även vara intressant med en gemensam kod för att uppmuntra utvecklare att lansera en applikation till flera plattformar och därmed nå ut till fler användare. Detta ökar i sin tur möjligheten för nya plattformar att ta sig in på marknaden då utbudet av applikationer är en viktig faktor för att göra en plattform till ett tänkbart alternativ för användare.

Det finns redan ett antal tjänster som låter utvecklare skriva en källkod för en applikation som kan kompileras till flera plattformar. Problemet är att tjänsterna oftast inte erbjuder all funktionalitet som finns hos de olika plattformarna och att det ger känslan av att använda en webb-applikation. Dessutom genererar många av tjänsterna i slutändan kod i standardspråket för den specifika plattformen, vilket gör att dessa tjänster inte är ett alternativ i de fall då det finns ett värde av att samma kod körs.

En alternativ lösning till tjänsterna ovan är att skriva gemensam kod i programspråket C, som är ett väletablerat språk med stöd hos många plattformar. Lösningen ger utvecklaren möjligheten att välja vad som ska vara gemensam kod och vad som behöver utvecklas separat för olika plattformar. Dessutom kommer det för denna lösning att vara samma C-kod som kompileras och körs för de olika plattformarna.

Projektet innebär en undersökning av vilken typ av kod som är lämplig att ha i en gemensam kodbas i C vid utveckling av mobilapplikationer. Underlaget för undersökningen är en medicinsk applikation vars funktionalitet innefattar en algoritm, Internetkommunikation och BLE-kommunikation.

1.2 Syfte

Projektet syftar till att undersöka vilken typ av funktionalitet som är lämplig att ha i en gemensam kodbas skriven i C vid utveckling av mobilapplikationer för flera plattformar. Resultatet av undersökningen ska ligga till grund för en Android-applikation som utvecklas i samband med projektet. Applikationen ska hämta ett värde från en BLE-enhet som behandlas med en algoritm för att sedan laddas upp till en databas. Projektet ska också ge grundläggande kunskap inom hur programspråket C kan användas vid utveckling av mobilapplikationer som slutligen sammanställs i en manual.

1.3 Avgränsningar

Projektet undersöker möjligheten av en gemensam kodbas endast för Android och iOS. Utöver Android-applikationen, med den funktionalitet som nämns i kapitel 1.2, kommer applikationer för Android och iOS utvecklas för att undersöka ett specifikt område. De områden som undersöks är en enkel algoritm, Internetkommunikation och BLE-kommunikation.

Projektet undersöker inte möjligheten av en gemensam kodbas för att hantera applikationernas grafiska gränssnitt. Det som undersöks är möjligheten till gemensam kod för delar av logiken.

1.4 Precisering av frågeställning

Vilken typ av funktionalitet är lämplig att ha i en gemensam kodbas skriven i C vid utveckling av mobila applikationer för flera plattformar?

2. Metod

2.1 Förarbete

Projektet inleddes med ett förarbete som innefattade att undersöka hur Android och iOS är uppbyggt och hur utveckling sker för respektive plattform.

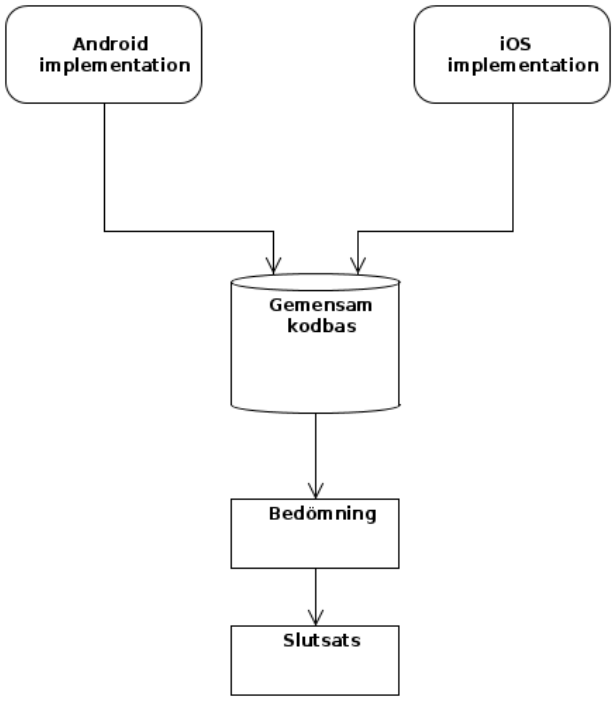
När utvecklingsmiljöer och verktyg var fastställda utreddes tillvägagångssättet för att implementera C-kod i de två plattformarna.

Vidare utreddes användandet av BLE och kommunikation via Internet för båda plattformarna.

2.2 Implementation och bedömning

Utvecklingen innefattade implementation för tre olika testområden: algoritm, Internetkommunikation och BLE-kommunikation.

När förarbetsperioden var avklarad implementerades kod för de tre olika områdena enligt flödesschemat i figur 2.1. Funktionaliteten för det aktuella området implementerades först för både Android och iOS. Utifrån de två olika implementationerna gjordes en bedömning av vilken kod som potentiellt skulle kunna lyftas ut och implementeras gemensamt i C-kod.



Figur 2.1 – Implementeringsstegen.

Den gemensamma C-koden bedömdes sedan i sin tur utifrån aspekterna modularitet, livscykel och användarvänlighet. Utifrån resultatet av denna bedömning drogs till sist en slutsats om huruvida en implementering i C-kod för det aktuella området är lämplig eller ej.

Eclipse och Xcode är två kända utvecklingsmiljöer för Android respektive iOS och var därför ingångsvinkeln i undersökningen av verktyg som skulle användas. För att dela kod användes versionshanteringssystemet Git.

2.3 Slutgiltig Android-applikation och manual

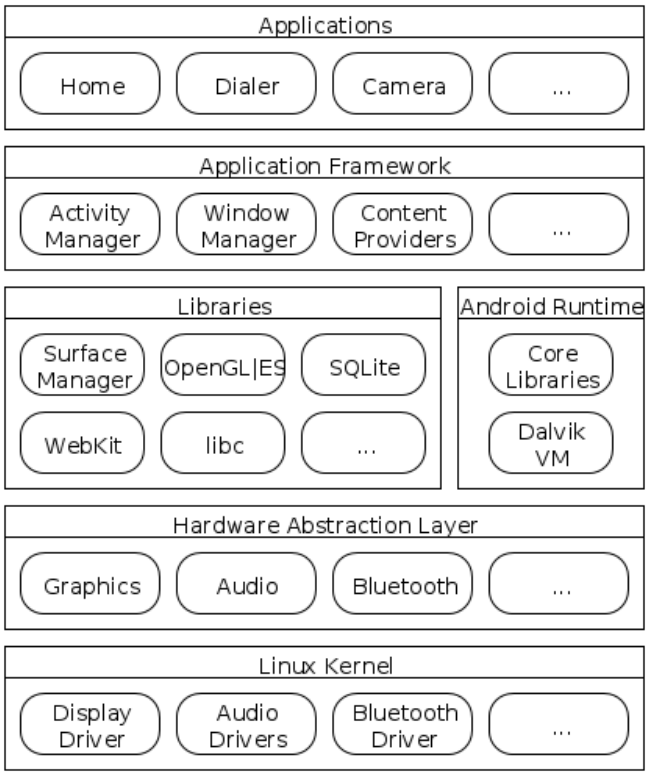
Utifrån bedömningen utvecklades slutligen en fullständig Android-applikation med implementerad funktionalitet för alla tre områden. Med kunskapen som inhämtats under projektets gång skrevs en manual som finns bifogad som bilaga A.

3. Teknisk bakgrund

3.1 Arkitektur - Android

Androids arkitektur består i grunden av en Linux-baserad kärna tillsammans med en rad olika C/C++-bibliotek. Funktionaliteten inom dessa bibliotek presenteras i sin tur genom ett ramverk som tillhandahåller tjänster samt sköter hantering av operativsystemet och dess applikationer [1].

En mer detaljerad bild över arkitekturen och dess lager finns illustrerad i figur 3.1. Följande underrubriker ger en beskrivning över varje lager och dess huvudsakliga syfte:



Figur 3.1 - Android-stacken.

Linux Kernel
Android bygger på Linux-kärnan som innehåller funktionalitet för t.ex. minne- och processhantering [1]. I lagret finns också drivrutiner för hårdvaran.

Hardware Abstraction Layer
I lagret abstraheras hårdvaran ännu ett steg genom att definiera ett flertal gränssnitt skrivna i C/C++. Gränssnitten beskriver den funktionalitet som ska implementeras i drivrutinerna för hårdvaran. Anledningen till att gränssnitten finns är för att ett standardgränssnitt ska finnas till hårdvaran [2]. Det innebär att kopplingen till hårdvaran blir ännu lägre eftersom själva Android-implementationen kan använda hårdvarans funktionalitet på samma sätt, oberoende av hårdvaran.

Libraries
Här återfinns C/C++-bibliotek för att utföra uppgifter som t.ex. rendering av 2D- och 3D-grafik, SSL-kommunikation, hantering av SQLite-databas samt ljud- och videouppspelning. Lagret innehåller även en anpassad version av libc, standardbiblioteket för C, som är optimerad för Android-enheter [3].

Trots att mycket av den funktionalitet som Android erbjuder finns i de nämnda biblioteken ovan, används dessa sällan direkt vid utveckling av Android-applikationer. Istället används något av de många Java-baserade API:er som Android även tillhandahåller och som i sin tur använder sig av C/C++-biblioteken. En direkt användning av biblioteken är dock möjlig genom Android Native Development Kit (NDK) som tillsammans med Java Native Interface (JNI) erbjuder möjligheten att anropa C/C++-metoder direkt ifrån Java-kod.

Android Runtime
Om man utgår ifrån det lägsta lagret som grund och rör sig uppåt i hierarkin är det först i denna del som stacken går från ren Linux-implementation till att innehålla implementation som är specifik för Android.

Lagret innehåller Googles virtuella maskin Dalvik VM tillsammans med grundläggande bibliotek för Android.

Reto Meier [1] beskriver det hela på följande sätt:

"The run time is what makes an Android phone an Android phone rather than a mobile Linux implementation. Including the core libraries and the Dalvik VM, the Android run time is the engine that powers your applications and, along with the libraries, forms the basis for the application framework."

Android Runtime - Core Libraries
De grundläggande biblioteken för Android kan delas in i tre kategorier. Den första kategorin innehåller bibliotek för att kommunicera direkt med den virtuella maskinen. Det är dock sällsynt att man behöva göra just detta vid utveckling av Android-applikationer. Trots att de flesta Android-applikationer utvecklas i Java exekveras de inte i Javas VM utan i Googles Dalvik VM.

Den andra kategorin innehåller bibliotek som utgör en delmängd av de grundläggande biblioteken för Java men som är utformade och anpassade för applikationer som körs inom en Dalvik VM [3]. Det är dessa bibliotek som används när man vid Android-utveckling använder sig av enkla Java-operationer för t.ex. stränghantering.

Om man beskriver den förra kategorin som Java-bibliotek anpassade för Android-utveckling är den tredje en uppsättning Java-baserade bibliotek utvecklade specifikt för Android-utveckling. Biblioteken erbjuder funktionalitet för t.ex. rendering av grafik och hantering av databaser. Det är dock viktigt att notera att majoriteten av klasserna i dessa bibliotek utför lite av det faktiska arbetet. Istället sker i de flesta fall ett anrop till något av de C/C++-bibliotek som beskrivits tidigare och som i sin tur använder sig av Linux-kärnan för att utföra de önskade uppgifterna [3].

Android Runtime - Dalvik VM
Dalvik VM är Androids motsvarighet till Java VM som används för att exekvera Java-klassfiler, vilket är kompilerade Java-filer. Till skillnad från Java VM exekverar Dalvik VM dex-filer, vilket är en komprimerad form av Javas klassfil. Dalvik VM är utvecklad av Google och är optimerad för enheter med begränsad minnesstorlek. Den är också specialanpassad för att mobila enheter effektivt ska kunna köra flera instanser parallellt av den [3].

Varje enskild applikation som körs på en Android-enhet exekveras i en egen Dalvik VM, som i sin tur är en egen process som körs av Linux-kärnan [3].

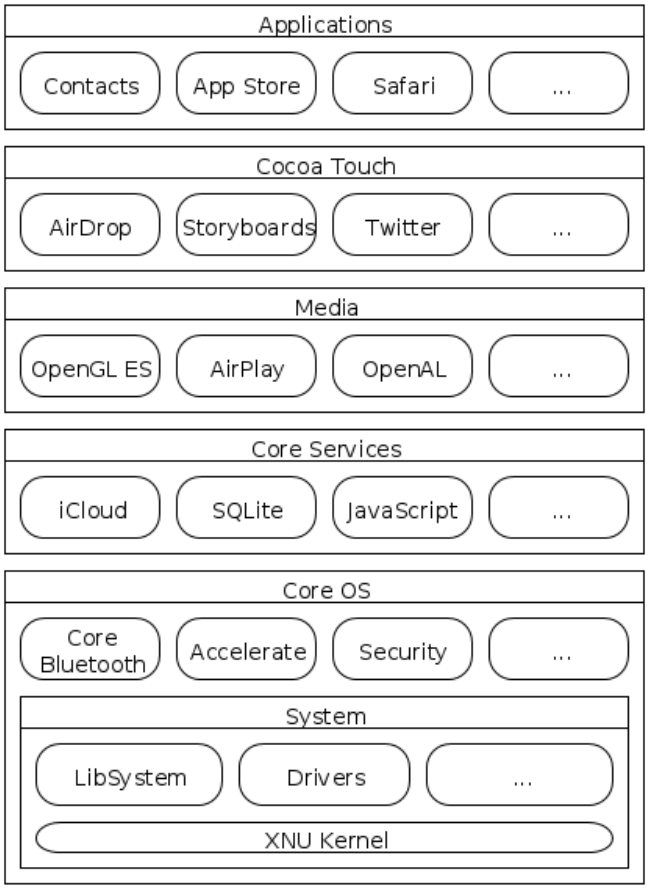
Application Framework
Här finns Java-klasser som används för att utveckla applikationer till specifikt Android. I lagret finns funktionalitet för att komma åt och använda hårdvara [1].

Applications
Lagret innehåller alla applikationer som finns på enheten, till exempel applikationer för att ringa, SMS:a och fotografera. Applikationerna i lagret är i huvudsak skrivna i Java och det är också hit tredjepartsapplikationer hör [3].

3.2 Arkitektur - iOS

Vid en jämförelse av figur 3.1 och figur 3.2 kan man se att det finns många likheter mellan arkitekturen för iOS och arkitekturen för Android. Båda modellerna utgår ifrån en kärna med ett antal lager ovanpå, där varje lager innehåller olika bibliotek och ramverk.

I arkitekturen för iOS innehåller de lägre lagren den mest grundläggande funktionaliteten medan lager högre upp i arkitekturen använder sig av funktionaliteten hos de lägre lagren för att erbjuda mer avancerade tjänster och funktioner. De högre lagren kan även erbjuda förenklade gränssnitt för annars komplexa användningsområden som t.ex. sockets och trådhantering [4].



Figur 3.2 - iOS-stacken.

Core OS

Detta lager ligger närmast kärnan och innehåller den lågnivå-funktionalitet som många av de övre lagren använder. Här finns t.ex. ramverk för att kommunicera med extern hårdvara och kommunikation via Bluetooth.

Den del inom Core OS-lagret som kallas System innefattar allt som har med XNU-kärnan att göra. XNU-kärnan är den kärna som både Mac OS X och iOS använder sig av. Själva XNU-kärnan ansvarar för bland annat trådhantering, filsystem och nätverks-kommunikation [5]. I System-delen finns även de drivrutiner som utgör gränssnittet mellan hårdvarukomponenter och ramverk hos operativsystemet. På grund av säkerhetsskäl är åtkomst till XNU-kärnan och drivrutiner begränsad. Den åtkomst som är tillåten nås genom LibSystem-biblioteket, som innehåller C-baserade gränssnitt för åtkomst till exempelvis filsystemet [4].

Core Services

Här finns t.ex. möjlighet för utvecklaren att låta sin applikation lagra data, kommunicera med HTTP- och FTP-servrar och använda sig av kontaktlistan på telefonen. Applikationer kan enkelt lagra information direkt på telefonen med hjälp av SQLite eller genom att använda sig av iCloud, Apples tjänst för att lagra filer i molnet [4].

Media

I medialagret finns teknologi inom grafik, ljud och video. Lagret förser applikationsutvecklaren med t.ex. funktionalitet för att rendera 2D- och 3D-grafik med OpenGL ES och ett ramverk för uppspelning av

standardformaten för ljud och bild. I lagret finns stöd för utvecklare att implementera funktionalitet som möjliggör streaming av ljud och bild till AirPlay-enheter, som t.ex. Apple TV [4].

Cocoa Touch
Lagret kan jämföras med Android-stackens "Applications Framework" och innehåller ramverk för att bygga iOS-applikationer. Exempel på funktionalitet är AirDrop som kan användas för att dela filer med närliggande enheter och Storyboards som används för att designa det grafiska gränssnittet [4].

Applications
Applikationslagret innehåller de applikationer som finns installerade på enheten. Vid utveckling av iOS-applikationer används vanligtvis programmeringsspråket Objective-C.

3.3 Utvecklingsspråk

3.3.1 C
Programmeringsspråket C utvecklades i början av 1970-talet av Dennis Ritchie vid Bell Laboratories för att skriva om Unix som innan skrivits med hjälp av assembler och programspråket B [6]. C är ett flexibelt språk som kan användas till att t.ex. utveckla operativsystem, men också för att utveckla programmen som körs på ett operativsystem. C är brett använt och kan kompileras och exekveras på nästan alla dagens datorer [7]. Jämfört med maskinkod och assembler är tanken med C att korrekt skriven kod ska fungera på samma sätt i olika miljöer [6]. Dock krävs att C-koden kompileras för den specifika plattformen för att det ska fungera.

3.3.2 Objective-C
I början av 1980-talet skapade Brad Cox och Tom Love programmeringsspråket Objective-C. Språket är en påbyggnad till C, vilket innebär att Objective-C-kod kan innehålla vanlig C-kod. Till skillnad från C är Objective-C ett objektorienterat språk med idéer från programspråket Smalltalk [8].

3.3.3 Java
Java är idag ett mycket väletablerat språk runtom i världen. Det släpptes ursprungligen 1995 av Sun Microsystems men köptes 2009 av Oracle som nu står för vidareutveckling av språket. Språket har till stor del inspirerats av framgångsrika föregångare som t.ex. C samtidigt som man har försökt förbättra de brister och tillkortakommanden som fanns hos föregångarna.

En viktig skillnad är att Java precis som Objective-C är ett objektorienterat språk. En annan skillnad är att kompilerad Java-kod exekveras i en virtuell maskin, vilket gör språket plattformsoberoende. Det går alltså att köra samma kod på olika plattformar förutsatt att det finns en implementationen av den virtuella maskinen för varje plattform. I Java har man även ersatt direkta pekare till adresser i minnet med referenser till objekt [9].

3.4 Implementation av C-kod i Android-applikationer
Till skillnad från Objective-C, som i grund och botten är C, är Java ett helt nytt språk. Detta gör att implementation av C-kod i Objective-C är mycket enkel, medan den är mer komplicerad i Java. För att anropa C-kod från Java används ett ramverk som heter JNI (Java Native Interface). Det är även detta ramverk som används för att implementera C-kod i Android-applikationer.

Figur 3.3 illustrerar de olika delar som krävs för att implementera C-kod i ett Android-projekt. Momenten kan mer eller mindre utföras i valfri ordning. Beroende på hur problemet ser ut finns det olika strategier för vilket steg man börjar med. Ett fall är att C-koden redan finns och att man vill gå från en implementering i C till en implementering i Java. Finns däremot ingen C-kod, men det är motiverat att använda C-kod, kan utvecklaren istället välja att börja med Java-deklarationen först och implementera C-koden sist.

Det här kapitlet beskriver först Android NDK, som är en samling verktyg för att implementera C-kod i Android-applikationer, följt av de olika momenten för att sedan beskriva två verktyg som förenklar de två idéerna ovan.

3.4.1 NDK

Android Native Development Kit kan användas som en komplettering till Android SDK för att erbjuda möjligheten att implementera delar av en applikation i C och C++ istället för Java. Själva innehållet i NDK:n består av header-filer för olika C-bibliotek samt verktyg för att kompilera C-kod till en Android-plattform från en Linux-, Mac OS X- eller Windows-plattform [10].

För att underlätta implementationen innehåller NDK:n även verktyg som sköter kompileringen av C-kod till rätt plattform utifrån två make-filer: Android.mk och Application.mk [10].

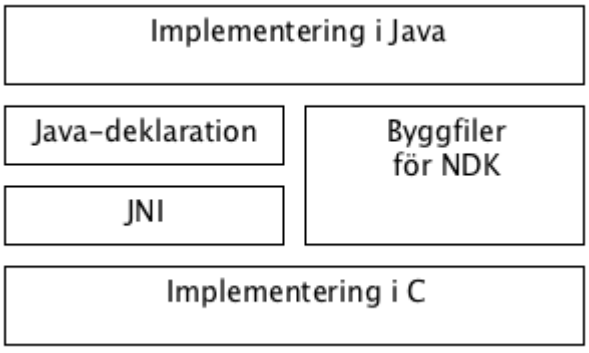
Den första make-filen måste finnas i alla Android-projekt som använder sig av NDK:n för att projektet ska gå att kompilera. Den innehåller dels de C-filer som skall användas av applikationen samt sökvägar till nödvändiga header-filer. Utöver det kan man också ange färdigkompilerade bibliotek som ska inkluderas. Här anges även det modulnamn som skall användas efter att NDK:n har kompilerat de angivna C-filerna till ett enskilt bibliotek. Med hjälp av modulnamnet kan man sedan ladda in det färdigkompilerade biblioteket till applikationen via ett funktionsanrop i Java.

Inom ramen för detta projekt har den andra make-filen endast använts för att tala om till vilken Android-plattform C-filerna skall kompileras.

3.4.2 De olika momenten

Implementering i C
Här skrivs eller finns redan existerande C-filer som innehåller ren C-kod.

JNI
I det här steget skapas det lager som knyter samman C-funktioner med Java-kod. För att implementera C-kod i Java krävs bland annat att man använder speciella returtyper som Java kan tolka och att funktionsnamnen är uppbyggda på ett speciellt sätt.



Figur 3.3 – C till Android.

Byggfiler för NDK
Här konfigureras de två make-filer som används för att tala om för kompilatorn vilka C-filer som ska kompileras och för vilka plattformar.

Java-deklaration
För att kunna anropa C-funktionerna från Java-kod måste modulen, som kompilerats med hjälp av make-filerna beskrivna ovan, laddas in och funktionerna deklareras i Java-filerna. Detta sker i det här steget.

Implementering i Java
Här används de funktioner som deklarerats i Java-koden. Anropen ser ut som vanliga funktions-anrop i Java. Innan det här steget genomförs bör Java-deklarationen vara avklarad.

3.4.3 javah
För att underlätta implementationen av den kod som används för att länka ihop C-kod med Java-kod finns olika verktyg för autogenerering varav ett är javah. Verktyget körs via terminalen och kan utifrån deklarerade funktioner i en given Java-klassfil generera motsvarande funktionsdeklarationer för C-koden. Med figur 3.3 som referens kan javah utifrån Java-deklarationen generera JNI-delen [11].

Exempel på anrop via terminalen:

```
user@computer:~$ javah –classpath bin/classes com.example.package.ClassName
```

3.4.4 SWIG
SWIG är ett verktyg som automatiserar JNI- och delar av Java-deklarationssteget, enligt figur 3.3. Syftet med SWIG är att underlätta arbetet med att använda kod skriven i C och C++ i andra språk, till exempel Java, Python och Perl. Vad SWIG i stora drag gör är att det utifrån en given header-fil, innehållande funktionsdeklarationer och beroenden, genererar den kod som JNI kräver, samt skapar funktionsdeklarationerna i en given Java-fil [12]. För att använda SWIG skriver utvecklaren en i-fil, som innehåller information om vilken header-fil som ska användas och vilka funktioner som SWIG ska exportera. Utvecklaren måste dock skriva Java-koden som laddar in modulen som genereras vid kompileringen.

Exempel på anrop via terminalen:

```
user@computer:~$ swig -java -package com.example.package -outdir
src/com/example/package jni/InterfaceFile.i
```

När SWIG-kommandot anropats skapas en C-fil och ett antal Java-filer. C-filen innehåller bland annat C-kod som anropar funktionerna som angivits i i-filen, men använder sig av funktionsnamn och typer som är JNI-kompatibla. En av Java-filerna innehåller deklarationer som länkar ihop Java med funktionerna skrivna i C. En annan Java-fil innehåller metoder som anropar metoderna som finns deklarerade i den tidigare nämnda Java-filen. Beroende på innehållet i i-filen kan fler Java-filer tillkomma för att hantera t.ex. structer.

Exempel på i-fil:

```
%module ModuleName

%{
#include "exempel1.h"
#include "exempel2.h"
%}

extern int int_function(int);
extern char* hello_world(void);
```

Vid behov av att använda mer komplexa datatyper som t.ex. fält eller strukturer krävs ytterligare kod. Som exempel kan följande kod användas för att använda ett heltalsfält som inparameter:

```
%include "arrays_java.i"
%apply int[] {int *};
```

En C-struct kan t.ex. skrivas på följande sätt:

```
typedef struct Coord {
	int x;
	int y;
} coord;
```

och används i C-kod enligt följande:

```
coord getCoord() {
	coord c;
	c.x = 0;
	c.y = 0;

	return c;
}
```

För att kunna anropa getCoord-funktionen från Java-kod krävs då att i-filen också innehåller deklarationen för C-structen. Detta kommer att generera en motsvarande Java-klass, med get- och set-metoder.

På SWIGs hemsida finns mycket dokumentation om hur annan funktionalitet kan användas.

3.5 Bluetooth

Bluetooth är en teknologi inom trådlös kommunikation som utvecklades av en grupp ingenjörer på Ericsson år 1994. Idén var att ta fram ett alternativ till den trådbundna kommunikationen som använde sig av kabeltypen RS-232. Sedan år 1998 utvecklas Bluetooth av Ericsson, Intel, Nokia, Toshiba och IBM, som tillsammans bildar Bluetooth Special Interest Group [13].

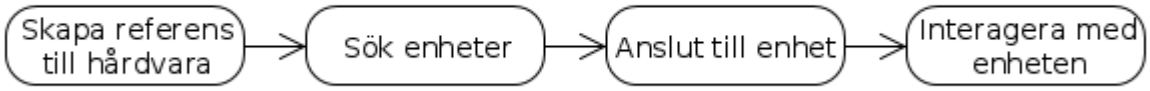
Bluetooth använder sig av radiovågor för att överföra information och har fokuserat på kommunikation vid korta avstånd. För att en enhet ska kunna använda sig av Bluetooth krävs speciell hårdvara och mjukvara [13].

3.6 BLE

År 2011 introducerades Bluetooth Smart som också kallas för BLE [13]. Istället för att fokusera på en hög överföringshastighet, är BLE optimerad för enheter där batteritiden är viktig. Energianvändningen för BLE-tekniken utgör endast 1 till 50 procent av energianvändningen för den klassiska Bluetooth-tekniken [14].

Idag finns inbyggt stöd för BLE på nyare smarttelefoner. En enhet med stöd för att ta emot data via BLE är en "Bluetooth Smart Ready"-enhet. En enhet som erbjuder data via BLE kallas för en "Bluetooth Smart"-enhet [13]. Första Android-versionen med stöd för BLE är 4.3, vilket motsvarar API 18 [15]. Apples första iPhone med BLE-stöd var iPhone 4s [16].

Processen för att interagera med en enhet ser ungefär likadan ut för både Android och Apple och beskrivs övergripligt i figur 3.4. Android och Apple har abstraherat interaktionsprocessen genom att implementera användarvänliga Java- respektive Objective-C-klasser. På så vis slipper utvecklare att tänka på protokoll och hårdvara.



Figur 3.4 - Steg för att interagera med BLE-enhet.

3.7 cURL

cURL är verktyg för att göra t.ex. HTTP-anrop via terminalen. cURL erbjuder också ett C-bibliotek, kallat libcurl, för att göra t.ex. HTTP-anrop i C-kod. Verktyget är öppen källkod vilket innebär att det går att redigera och kompilera koden själv. Både biblioteket och verktyget fungerar på en rad olika plattformar [17].

4. Genomförande

Följande kapitel beskriver genomförandet av undersökningen i projektet. Inledningsvis beskrivs konfigurationen av utvecklingsmiljöer. Därefter följer en redogörelse för de applikationer som utvecklats i samband med undersökningen.

4.1 Initiering

Projektet inleddes med att sätta upp utvecklingsmiljön. För att utveckla applikationer till Android kan Eclipse med några Android-specifika tillägg användas och för att utveckla iOS-applikationer används Xcode.

4.1.1 Eclipse

För att använda Eclipse vid Android-utveckling krävs tillägg för att det ska fungera. Dessa tillägg är ADT (Android Developer Tools) och Android SDK (Software Development Kit). Om det dessutom ska gå att använda C-kod krävs NDK (Native Development Kit).

Om stödet för C inte behövs finns ett färdigt paket att ladda hem från Androids hemsida, men att installera NDK är mer komplicerat.

För att få iordning en utvecklingsmiljö med stöd för C-implementation användes boken “Pro Android C++ with the NDK” av Onur Cinar som manual. I stora drag utförs följande steg:

- Installera Java.
- Ladda hem Android SDK och Android NDK.
- Sätta systemvariabler.
- Ladda hem Eclipse.
- Ladda hem Android ADT via Eclipse.

I boken beskrivs hur ovanstående steg görs i Windows, Mac OS X och Ubuntu.

4.1.2 Xcode

För utvecklingen av iOS-applikationer användes Apples egen utvecklingsmiljö Xcode. Utvecklingsmiljön finns att ladda hem utan kostnad via Apples Mac App Store. För att ladda hem ytterligare tillägg till Xcode krävs att man registrerar sig som utvecklare hos Apple genom ett kostnadsfritt medlemskap.

För att kunna testa implementerad BLE-funktionalitet under projektets gång var möjligheten att köra applikationer på en fysisk iOS-enhet ett krav eftersom iOS-emulatorn i Xcode inte har stöd för BLE. För att kunna göra detta krävs dock ett utökat medlemskap som utvecklare hos Apple som inte längre är kostnadsfritt. I detta projekt bidrog uppdragsgivaren med en licens för utökat medlemskap.

Då Objective-C är en förlängning av språket C krävs inga ytterligare specifika tillägg för att implementera C-kod i applikationer för iOS.

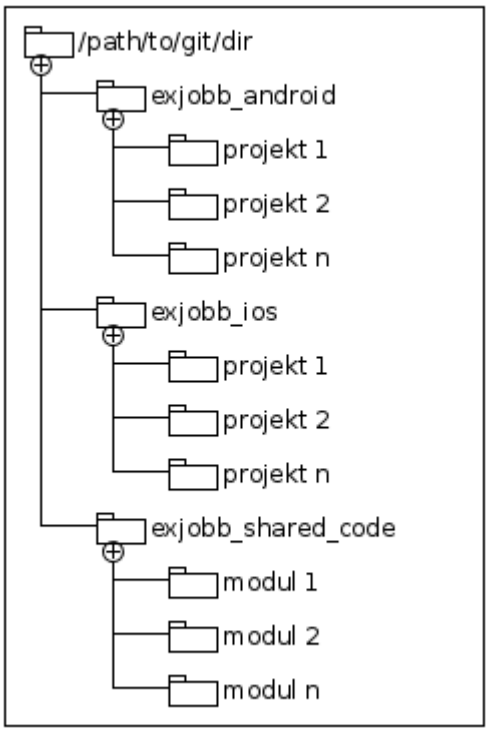
4.1.3 Git

I samband med att utvecklingsmiljöer sattes upp, sattes också tre Git-repon upp. Ett för Android-projekten, ett för iOS-projekten och ett för den gemensamma kodbasen.

4.1.4 Gemensam kodbas

För att lättare underhålla koden länkas den gemensamma C-koden in i respektive projekt istället för att filerna kopieras. På detta sätt har båda projekten alltid samma version av filen. För att åstadkomma detta i Xcode krävs bara att man väljer att filerna ska länkas och inte kopieras när de importeras till ett projekt. För Android behöver man ange sökvägen till de C-filer som ska användas i en av make-filerna för att NDK-kompilatorn ska hitta filerna.

Eftersom länkningen av filer måste fungera på flera datorer bestämdes att mappstrukturen i figur 4.1 skulle användas. Den här mappstrukturen möjliggör relativa sökvägar som fungerar för alla, istället för att varje dator har specifika sökvägar.



Figur 4.1 - Mappstruktur för git-repon.

4.1.5 Testutrustning

Den testutrustning som krävdes för projektet var en Android-, en iOS- och en "Bluetooth Smart"-enhet . Kraven på enheterna var att de måste kunna kommunicera med BLE-teknik.

För att testa Android-applikationerna har en Sony Xperia Z1 Compact använts och för att testa iOS-applikationerna har en iPhone 5 använts.

4.1.6 Övriga verktyg

Eftersom idén med projektet är att skapa en gemensam kodbas skriven i C, innebär det att C-koden skrivs först och sedan implementeras i respektive plattformsspecifikt projekt. Av den anledningen användes verktyget SWIG för att effektivisera detta arbete.

4.2 Exempelprojekt

Som ett konkret mål för att testa att utvecklingsmiljöer installerats korrekt och att de klarade av vad de skulle, gjordes ett exempelprojekt. Målet med exempelprojektet var att skriva två funktioner i C-kod och sedan få ett Android- och iOS-projekt att kunna använda sig av dessa.

4.2.1 C-kod

Följande C-kod skrevs:

dod_test.h:

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

char* string_test (void);
double parameter_test (int, int);
```

dod_test.c:

```
#include "dod_test.h"
char* string_test(void) {
	return "hello from C";
}

double parameter_test (int a, int b) {
	return (double)a / b;
}
```

4.2.2 Android

Först skapades ett Android-projekt. För att kunna använda C i ett Android-projekt krävs att projektet har så kallad “native support”, vilket lades till via Eclipse.

För att testa exempelprojektet skapades ett enkelt GUI bestående av en knapp för att göra anropen till C-funktionerna samt två textfält för att skriva ut resultaten av funktionsanropen.

För att använda funktionerna i “dod_test.c” krävs att JNI och dess specifika typer används. Eftersom C-koden redan var skriven och den skulle användas i Java-koden användes SWIG för att generera den C- och Java-kod som krävs. Följande i-fil skapades:

dod_test.i:

```
%module NativeTest

%{
#include "dod_test.h"
%}

extern double parameter_test(int, int);
extern char* string_test(void);
```

Slutligen inkluderades C-filen med testfunktionerna, den JNI-genererade C-filen samt sökväg till header-filen i Android-projektets make-fil.

4.2.3 iOS

För att testa iOS-utvecklingsmiljön skapades ett iOS-projekt och ett GUI med en knapp och två textfält.

C-filerna länkades sedan in i projektet, men kan också kopieras in i projektet om så önskas. För att använda funktionerna inkluderades “dod_test.h” i den fil där anropen sker. Anropen till C-funktionerna skrevs sedan in där de skulle enligt vanlig C-syntax, vilket fungerar av anledningen att Objective-C är en direkt påbyggnad av C.

4.3 Implementering av Internetkommunikation

För att testa Internetkommunikationsdelen utvecklades små applikationer som gör olika typer av HTTP-anrop. Upplägget var att först utveckla en Android-applikation som endast använder sig av Java, sen en motsvarande iOS-applikation som endast använder sig av Objective-C och slutligen skriva C-kod som kan användas av både Android- och iOS-applikationen.

4.3.1 Android

En Android-applikation utvecklades som, med hjälp av standardklasser, gör ett enkelt GET-anrop. Applikationen har ett inmatningsfält för URL, en “GET”-knapp och ett textfält där responsen från GET-anropet hamnar.

För att inte låsa huvudtråden och därmed det grafiska gränssnittet har Android förhindrat möjligheten att starta en Internetuppkoppling direkt i huvudtråden. Istället måste en ny tråd skapas. I Android finns en standardklass som skapar just en ny tråd på ett enkelt sätt. Utvecklaren anger vad som ska göras i tråden och vad som ska hända när jobbet är färdigt. Denna klass används för att göra GET-anropet.

Applikationen byggdes sedan ut med ett extra inmatningsfält för att kunna skicka med POST-data, vilket innebar ett fåtal ändringar i koden.

4.3.2 iOS

Precis som i Android-fallet utvecklades en iOS-applikation som gör ett enkelt GET-anrop med hjälp av standardklasser. Ett motsvarande GUI utvecklades med ett inmatningsfält, en knapp för att göra GET-anropet och ett textfält för att presentera responsen.

För att starta uppkopplingen finns tre alternativ. Det första och mest enkla alternativet körs direkt i huvudtråden och kommer att låsa applikationen tills dess att metod-anropet är färdigt. Detta alternativ ska dock endast användas när man vill hämta lokal data på enheten som alltid finns direkt tillgänglig.

Det andra alternativet körs i bakgrunden och låser inte applikationen under anropet men erbjuder ingen möjlighet att hantera händelser som kan uppstå medan uppkopplingen är aktiv, utan endast ett resultat när hela anropet är färdigt.

Det tredje och mest avancerade alternativet körs också i bakgrunden, men erbjuder till skillnad från det andra alternativet, även hantering av händelser under uppkopplingen som t.ex. den första responsen från servern, felmeddelanden, slutförd nedladdning eller begäran om inloggning. För att hantera dessa händelser implementeras vissa metoder som sedan skickas med vid metodanropet som görs då GET-anropet utförs.

Som implementation för applikationen valdes det tredje alternativet, eftersom hantering av inloggning skulle kunna bli intressant längre fram i projektet.

Analogt med Android byggdes sedan applikationen ut med ett extra inmatningsfält för att kunna hantera POST-data, även i detta fall med endast ett fåtal ändringar i den ursprungliga koden.

4.3.3 Gemensam kodbas

När det fanns en Android- och en iOS-applikation som kunde utföra GET-anrop undersöktes vad som eventuellt hade kunnat finnas i en gemensam kodbas istället. Det var också nödvändigt att se över vilka alternativ som fanns för Internetkommunikation med C. Mest lovande var att använda C-biblioteket libcurl.

libcurl finns tillgängligt för ett antal olika plattformar, däribland Android och iOS. Dock är libcurl ett tredjepartsbibliotek som inte ingår i varken Android eller iOS från början och det krävs att utvecklaren på något vis kompilerar och lägger till det själv, vilket är möjligt då libcurl är öppen källkod. För iOS fanns libcurl färdigkompilerat, men inte för Android.

För att kompilera libcurl för Android användes en korskompilator, vilket är en kompilator som kompilerar kod för en annan plattform än den plattform som kompilatorn körs på. Med Android NDK följer en skriptfil som kan användas för att skapa en samling verktyg som bland annat innehåller en korskompilator. Skriptfilen kan användas för att skapa de här verktygen för plattformarna Android körs på. För att alla Android-enheter ska kunna exekvera en applikation som innehåller C-kod krävs alltså flera kompilatorer. För att skapa de här samlingarna av verktygen användes en skriptfil som förenklar arbetet.

Vidare användes ytterligare en skriptfil för att kompilera själva libcurl-biblioteket med hjälp av kompileringsverktygen. I skriptfilen angavs även nödvändiga konfigurationsparametrar beroende på plattform. Det färdigkompilerade biblioteket inkluderades sedan i Android-projektet genom att lägga till några rader i en av projektets make-filer.

När libcurl fungerade på Android skrevs C-kod som skulle kunna användas i både Android- och iOS-applikationen. Att göra ett GET-anrop i C med hjälp av libcurl kräver endast ett fåtal rader, men för att kunna returnera svaret som en sträng krävs några extrarader. För att kunna returnera svaret skrevs en funktion som tar hand om den data som mottagits, istället för att den skrivs direkt till standardströmmen för utmatning.

Efter att ha använt SWIG för att generera nödvändiga filer för att kunna använda C-funktionerna från Java var det endast en rad som behövde ändras. Metoden som tidigare gjorde GET-anropet raderades och den rad där metoden tidigare anropats ändrades till att anropa C-funktionen.

C-koden kompletterades senare med en rad för att göra POST-anrop istället för GET-anrop. För att skicka med POST-data lades ett extra inmatningsfält till i Android-applikationen.

Då en fungerande Android-applikation hade utvecklats med hjälp av libcurl och C-kod från den gemensamma kodbasen utvecklades en motsvarande iOS-applikation. I och med att det fanns en färdigkompilerad version av libcurl för iOS via hemsidan för biblioteket laddades den hem och länkades in i ett nytt iOS-projekt. Även nödvändiga C-filer, vilket innefattar header-filer för libcurl och den egenskrivna C-koden, länkades in.

I iOS-projektet som endast använder Objective-C, används en självskriven klass som hanterar de händelser som genereras i samband med GET-anropet. Den här klassen slopades i samband med att C-kod

skulle sköta GET-anropet, vilket innebär att utvecklaren får mindre kontroll över vad som händer med anslutningen. Den här klassen gör också att GET-anropet körs i en egen tråd. För att undvika låsning av huvudtråden skapades en ny tråd och en metod som kan köras av tråden.

Metoden som exekveras i bakgrundstråden gör ett anrop till C-funktionen och visar sedan upp svaret i en textruta.

4.3.4 HTTPS-kommunikation

När HTTP-kommunikation var implementerad i Java, Objective-C och C, gjordes försök att använda HTTPS. För att kunna göra HTTPS-anrop i Android-applikationen som använder sig av Java behövde ett fåtal rader editeras. Beroende på vilken guide eller hur man har valt att implementera sina HTTP-anrop så följer HTTPS med och då behövs inte någon extra editering.

Implementationen som tidigare gjorts för Objective-C krävde ingen förändring för att kunna göra HTTPS-anrop. Inte heller iOS-applikationen med libcurl behövde ändras för att kunna göra HTTPS-anrop, vilket beror på hur libcurl är kompilerad. Den förkompilerade versionen av libcurl, som fanns att ladda hem, hade kompilerats med stöd för SSL och allt det som behövs.

När ett HTTPS-anrop testades på Android-applikationen som använde libcurl stod det att protokollet inte stöds. För att kunna göra HTTPS-anrop krävs SSL, vilket kan läggas till vid kompileringen av libcurl.

Ett fullbordat HTTPS-anrop i Android med hjälp av libcurl lyckades inte implementeras inom tidsramen för projektet. Med den tid som fanns kompilerades OpenSSL och libcurl kompilerades om med stöd för SSL. Det återstående problemet är att det inte går att verifiera certifikaten som används hos en server.

4.4 Implementering av algoritm

För att utvärdera när och om det är lämpligt att ha en algoritm i en delad kodbas implementerades en algoritm som tar en matris av heltal som inparameter. Heltalen kan vara både positiva och negativa. Algoritmen returnerar den största summan som kan fås av en submatris. Både Android- och iOS-applikationen använder sig av ett GUI med en knapp och ett textfält. Då användaren trycker på knappen körs algoritmen och svaret skrivs ut i textfältet. Applikationerna låter inte användaren fylla i en egen matris, istället finns två statiska matriser i koden.

4.4.1 Android

En Java-metod skrevs för att implementera algoritmen. Metoden tar en kvadratisk heltalsmatris av valfri storlek som inparameter och returnerar ett heltal med den största summan. Jämfört med Internetkommunikationen som inte kan köras på huvudtråden kan algoritmen det. Dock valdes att låta en bakgrundstråd exekvera algoritmen.

4.4.2 iOS

Metoden som skrivits i Java skrevs om i Objective-C, dock med några förändringar i strukturen. I C och Objective-C kan en metod inte ta en heltalsmatris av godtycklig storlek som inparameter. En metod kan ta en matris där en dimension är bestämd och en obestämd, men inte två obestämda. Ett annat sätt att lösa problemet, och samtidigt bibehålla flexibiliteten, är att se en matris som ett långt fält.

I det här fallet valdes att bibehålla flexibiliteten och se en matris som ett långt fält istället. Detta medförde också att gradtalet måste anges eftersom algoritmen måste veta när en ny rad skulle ha börjat.

Som i den motsvarande applikationen för Android valdes att exekvera algoritmen i en bakgrundstråd.

4.4.3 Gemensam kodbas

Efter att ha gjort en applikation för både Android och iOS gjordes bedömningen att hela algoritmen kunde lyftas ut för att skrivas i en gemensam kodbas. I och med att problemet med obestämd storlek på en matris uppenbarade sig under implementationen för iOS valdes att använda iOS-koden som utgångspunkt för C-koden snarare än Java-implementationen.

När en implementation av algoritmen fanns skriven i C användes verktyget SWIG för att generera de filer som krävs för att kunna anropa C-koden i Android-applikationen. Då ett fält används som inparameter till C-funktionen krävdes mer kod i i-filen som SWIG använder för att generera filerna. I i-filen adderades dels inkludering av en i-fil som möjliggör användandet av fält, samt en specifikation om vilken typ av fält som används.

Då det enkla fallet att använda ett heltalsfält gjorde att i-filen behövde kompletteras, vilket krävde mer tid, togs beslutet att även testa hur man går tillväga för att använda C-structer som in- och utparametrar.

En enkel C-struct, kallad “matrix_info”, innehållande min- och maxvärde, samt största summan för en submatris skapades. För att testa structen skrevs en funktion som tar emot ett heltalsfält, längden på fältet och sen returnerar en matrix_info-struct.

För att kunna använda C-structen krävs att structen även deklareras i i-filen. Då SWIG-kommandot körs från terminalen skapas en motsvarande Java-klass som används i Java-koden.

Med Android-applikationen skriven enbart i Java som referenspunkt raderades den Java-implementerade algoritmen och raden där algoritmen tidigare hade anropats editerades så att C-funktionen anropas istället. Eftersom C-funktionen tar ett heltaltsfält och ett gradtal som inparametrar istället för en heltalsmatris, kompletterades applikationen med kod som tar ut gradtalet på matrisen och gör om matrisen från en tvådimensionell representation till ett endimensionellt heltalsfält som sedan skickas med till C-funktionen.

Då Android-applikationen fungerade gjordes motsvarande steg för att göra en iOS-applikation som använder C-funktionen.

4.5 Implementering av BLE-kommunikation

Den här delen i projektet inleddes med att leta efter existerande C-bibliotek som skulle kunna användas på både Android och iOS, vilket inte hittades. Det närmaste som hittades var BlueZ, som bland annat erbjuder ett C-bibliotek för BLE-kommunikation anpassat för Linux-kärnor. Biblioteket skulle därför eventuellt kunna användas för Android-enheter, som körs på en Linux-kärna, men inte för iOS-enheter eftersom dessa körs på en annan typ av kärna.

En implementation i C bedömdes inte vara möjlig då existerande C-bibliotek med stöd för båda plattformarna ej hittades. Trots detta utvecklades en applikation för både Android och iOS som använde sig av BLE-kommunikation för att undersöka om det fanns någon typ av kod som skulle kunna skrivas gemensamt i C, men som nödvändigtvis inte har med BLE-kommunikationen att göra.

4.5.1 Android

Android-applikationen utvecklades med en vy innehållandes två knappar, en för att ansluta till BLE-enheten och en för att läsa data från enheten när man väl är ansluten. I vyn finns även en logg där den lästa datan presenteras tillsammans med händelser som har med BLE-anslutningen att göra. De händelser som skrivs ut är t.ex. när applikationen söker efter BLE-enheter, när en enhet har hittats och när man har lyckats ansluta till en enhet.

För att sköta BLE-kommunikationen används en Android-klass som är lämplig när en applikation ofta behöver utföra arbete i bakgrunden. En instans A av denna klass länkas ihop med en instans B av den klass som ansvarar för det grafiska gränssnittet. B kan sedan genom en referens till A göra anrop för att starta bakgrundsarbete. Därefter praktiseras en typ av observer-mönstret där A kan sända ut händelser som B tar emot och i detta fall skriver ut i loggen.

För själva BLE-kommunikationen används det ramverk som Android erbjuder. I stora drag skapas en referens till hårdvaran i form av ett objekt. Med hjälp av detta objekt utförs operationer mot enheten som t.ex. sökning, anslutning och läsning. För att ta emot information från en ansluten enhet används ytterligare ett objekt där olika metoder körs beroende på vilken typ av händelse som skett.

4.5.2 iOS

För iOS-applikationen utvecklades vyn med en knapp för att både ansluta och läsa data från BLE-enheten. Händelser relaterade till BLE-anslutningen presenteras i ett textfält som uppdateras när nya händelser inträffar. När data har lästs från enheten presenteras detta i ett eget textfält.

Till skillnad från Android-applikationen används ingen specifik instans för att hantera BLE-kommunikationen utan detta sköts av samma instans som ansvarar för det grafiska gränssnittet. För BLE-kommunikationen används det ramverk som erbjuds av iOS. Proceduren för att ansluta och läsa data från en enhet ser i princip likadan ut som för Android. Operationer mot en ansluten enhet sker genom en referens till hårdvaran i form av ett objekt och händelser från enheten hanteras i olika metoder hos ytterligare ett objekt.

4.5.3 Gemensam kodbas

När två enkla applikationer utvecklats för Android och iOS konstaterades att de anrop och händelser som är involverade i kommunikationen är mycket lika för de båda plattformarna. En idé var till en början att det vid en gemensam händelse skulle ske ett anrop till en gemensam funktion i C som hanterade händelsen.

Vidare konstaterades att händelserna går att dela in i två kategorier. I den ena kategorin har en händelse skett som gör att ett nytt anrop mot enheten sker. Till exempel görs ett anrop för att ansluta när en enhet har hittats. I den andra kategorin har en händelse skett som gör att man vill uppdatera det grafiska gränssnittet, t.ex. när applikationen har tagit emot data som ska presenteras i ett textfält.

Eftersom anropet i den första kategorin sker med hjälp av ramverket för respektive plattform finns ingen möjlighet till en gemensam C-kod i detta fall. Då det grafiska gränssnittet sköts av respektive plattform finns inte heller här något intresse i att ha gemensam kod skriven i C.

5. Bedömning av implementationsområden

Kapitlet innehåller en bedömning av C-implementationen för de tre undersökta områdena. Bedömningen baseras på resultatet av projektets genomförande utifrån aspekterna modularitet, livscykel och användarvänlighet.

5.1 Övergripande bedömning

5.1.1 Fördelar

Då en och samma kod används blir koden enkel att underhålla och små förändringar blir smidiga att implementera för flera plattformar samtidigt. Enligt avgränsningen för detta projekt har en gemensam kodbas i C endast utvecklats för delar av logiken medan grafik fortfarande sköts av plattformens standardspråk. Detta gör att metoder och funktioner får ett tydligt ansvar och att det lätt går att urskilja vad som tillhör logik respektive grafik.

Som nämnts i kapitel 3.3.1, kan C-kod kompileras för en rad olika plattformar och arkitekturer. Dessutom kan C anropas från ett antal olika programmeringsspråk utöver Java och Objective-C, som t.ex. Python och Perl. Även i dessa fall skulle verktyget SWIG kunna användas för att generera nödvändiga deklarationer och filer. Detta tyder på att funktionalitet utvecklad i C enkelt kan användas på en rad andra plattformar och arkitekturer med andra programmeringsspråk utöver det som behandlats i detta projekt.

5.1.2 Nackdelar

Eftersom all kod inte är skriven i samma språk och alla filer inte hamnar på samma ställe, då en gemensam kodbas i C används, kan projektet upplevas som oorganiserat. Filer som ligger utanför projektet måste länkas in med sökvägar vilket kräver en konsekvent mappstruktur för att länkningen ska kunna ske på ett smidigt sätt när projekt utvecklas parallellt av flera utvecklare.

Programspråket C lämnar mycket ansvar till utvecklaren, vilket medför högre krav på utvecklaren. Till exempel ansvarar utvecklaren för att frigöra arbetsminne som programmet allokerar under exekvering.

Att använda C i Android innebär en del arbete utöver själva utvecklingen. Till exempel måste alla utvecklare installera Android NDK, utöver standardverktyget Android SDK, vilket är en engångskostnad tidsmässigt. Det är också en fördel att lära sig verktyget SWIG för att effektivisera användandet av C från Java. Vid underhåll av kod kan SWIG behöva användas för att generera nya funktionsdeklarationer, beroende på vad som ändras i C-koden. Om det bara handlar om förändringar i en funktion behövs inga nya deklarationer, men om inparametrar eller returtyper ändras krävs nya deklarationer. SWIG behöver dock inte användas om en hjälpfunktion läggs till som bara används någon annanstans i C-koden, men om en funktion ska kunna anropas från Java-kod krävs en deklaration för funktionen.

Då C-kod ska användas i iOS krävs inget speciellt gränssnitt, vilket behövs för Android, men istället kan returtyper och inparametrar behöva omvandlas mellan C och Objective-C.

Både Android och iOS erbjuder ett stort antal API:er för att exempelvis använda Internetkommunikation, något standardbiblioteken i C inte tillhandahåller. Detta gör att en C-implementation kan komma att bli beroende av tredjepartsbibliotek vilket i vissa fall kan vara svåra att använda.

5.2 Individuella bedömningsaspekter

För att bedöma de olika områdena har ett antal aspekter tagits fram.

5.2.1 Modularitet

Projektet har som syfte att undersöka vilken typ av kod som kan vara i en gemensam kodbas, skriven i C. Huruvida detta är möjligt och lämpligt beror till stor del på hur modulär koden är.

5.2.2 Livscykel

Bedömning av livscykeln har delats upp i tre delar: initiering, utvecklingsarbete och underhåll. Vikten av hur lätt eller svårt till exempel underhållsarbetet är kan tänkas väga tungt för ett beslut om hur koden ska implementeras. Initiering innebär saker som behöver göras innan utvecklingsarbetet påbörjas, t.ex. att inhämta ny kunskap. Utvecklingsarbetet innefattar de delmoment under utvecklingsperioden som är relaterade till den gemensamma C-koden. Med underhåll menas de moment som kan komma att bli nödvändiga vid eventuella uppdateringar.

5.2.3 Användarvänlighet

Under rubriken användarvänlighet bedöms hur användarvänliga de olika alternativen är. En sådan faktor kan till exempel vara om det finns gott om dokumentation eller hur lättförståeligt konceptet är. Om något är svårt att förstå kan utvecklingstiden vara längre eller inlärningstiden vara lång.

5.3 Bedömning: implementation av algoritm

5.3.1 Modularitet

All C-kod som utvecklades för algoritmen använder endast standardbiblioteken för C och har inga övriga beroenden. Eftersom C-kod kan kompileras för en rad olika plattformar och arkitekturer, speciellt om endast standardfunktionalitet används, bedöms lösningen som modulär.

5.3.2 Livscykel

Initiering

För att påbörja utvecklingsprocessen krävdes inte något förarbete.

Utvecklingsarbete

Under arbetets gång utvecklades den algoritm som tidigare bestämts. Utvecklingen fick ta hänsyn till att C inte kan hantera matriser med helt obestämt gradtal. C kan hantera matriser där ena eller båda dimensionerna är bestämda, men inte matriser där ingen dimension är bestämd.

I Android-fallet användes SWIG för att genera nödvändiga filer. I och med att fält och en struct användes behövdes mer kod, jämfört med att funktionerna endast använder enkla datatyper, i filen som SWIG använder för att generera de nödvändiga filerna.

Underhåll

Implementationen av algoritmen använder sig uteslutande av standardbibliotek. Detta gör underhållet enkelt eftersom det merarbete som kan tillkomma vid uppdateringar endast är relaterat till SWIG och byggfiler för NDK:n.

5.3.3 Användarvänlighet
Standardfunktionaliteten för C är väl dokumenterad. Dock medförde användandet av fält och structer direkt att mer tid behövde läggas på att få SWIG att generera de filer som behövs. Typkonvertering mellan Objective-C och C användes där det behövdes.

5.4 Bedömning: implementation av Internetkommunikation

5.4.1 Modularitet
Implementationen gjord i C använde sig av tredjepartsbiblioteket libcurl. Biblioteket är brett använt och finns för ett stort antal plattformar. Kod som använder sig av libcurl är i den meningen flyttbar och modulär. Det som krävs är att biblioteket finns kompilerat på enheten.

5.4.2 Livscykel
Initiering
Kompilering av libcurl krävdes i Android-fallet, men inte för iOS. Detta innebar bland annat att kunskap om kompilering av libcurl för Android-plattformar behövde inhämtas.

Utvecklingsarbete
Exempel på hur libcurl används följdes och kod för att göra GET- och POST-anrop utvecklades. För Android användes SWIG för att generera nödvändiga filer som krävs för att göra C-anrop från Java-kod. I vissa fall behövdes C-typer omvandlas till och från Objective-C-typer i iOS-projektet.

Underhåll
Då det kommer en ny version av libcurl kommer libcurl att behöva omkompileras. Det är inte heller säkert att det alltid kommer att finnas en färdigkompilerad version av libcurl för iOS, vilket kan ge kompileringsarbete även där. I övrigt är koden enkel att underhålla.

Applikationerna som utvecklades för Internetkommunikation använde sig endast av HTTP-anrop. Om HTTPS-anrop ska göras tillkommer mer kompileringsarbete av dels libcurl, men också av bibliotek som möjliggör användning av SSL, t.ex. OpenSSL.

5.4.3 Användarvänlighet
C-biblioteket libcurl är väl dokumenterat och det finns många exempel att följa. Dock finns det inte mycket dokumentation som på ett bra sätt beskriver hur libcurl kompileras för Android. Vad det gäller kompilering av libcurl för iOS finns dokumentation men då det redan fanns kompilerat har inte detta testats.

5.5 Bedömning: implementation av BLE-kommunikation
I och med att en gemensam kodbas, av flera orsaker, inte bedömdes möjlig i applikationerna som utvecklades för att testa BLE-kommunikation, kan i detta fall ingen utförlig bedömning göras.

6. Resultat

Kapitlet består av en analys där riktlinjer gällande vad som är lämpligt att ha i en gemensam kodbas i C diskuteras. Kapitlet beskriver även den slutgiltiga Android-applikationen som tillämpar dessa riktlinjer.

6.1 Analys av bedömning

Syftet med projektet har varit att undersöka vilken typ av kod som är lämplig att ha i en gemensam kodbas skriven i C. Bedömningen av vad som är lämpligt eller inte beror till största del på vad syftet med den gemensamma koden är. En anledning kan vara att minska utvecklingstiden. I detta fall blir bedömningen förhållandevis enkel. Den typ av kod eller funktionalitet som i en gemensam kodbas bedöms minska utvecklingstiden blir per automatik också lämplig att ha gemensam. Detta förutsätter dock att det går att göra en lämplig uppskattning av utvecklingstiden för fallet med gemensam kodbas jämfört med utan.

En annan anledning är när det finns ett särskilt intresse av att samma kod kompileras och körs på flera plattformar. Här blir en bedömning mer komplicerad. Bedömningen kommer att behöva grundas i en avvägning mellan värdet av den gemensamma koden gentemot nackdelar som ökad utvecklingstid, ökad komplexitet eller försämrade egenskaper när det gäller livscykel. Vad som är lämpligt i detta avseende kommer troligtvis att bero mycket på det specifika fallet.

Som underlag för undersökningen har projektet utvärderat tre olika områden där det för varje område krävts olika förutsättningar för att göra en implementation i C möjlig. Det första området innehåller funktionalitet som endast använder sig av standardbiblioteken i C. Den förutsättning som krävs i detta fall är att det går att göra anrop till C från standardspråket för de olika plattformarna. Det andra området innehåller funktionalitet som kan vara tidsödande och komplicerad att skriva från grunden i C. I detta fall används tredjepartsbibliotek för att undvika arbetet med att skriva all funktionalitet från grunden. Förutsättningen för detta fall är utöver anrop från standardspråken att de tredjepartsbibliotek som används ska finnas kompilerade för varje plattform. Det tredje och sista området innehåller precis som det andra området komplicerad funktionalitet där användandet av ett tredjepartsbibliotek är befogat, men där ett sådant inte finns. Förutsättningen för en gemensam kod i det sista fallet är anrop från standardspråken men också att utvecklaren ansvarar för att implementera hela den önskade funktionaliteten från grunden i C.

I de fall då endast standardbibliotek används är det relativt enkelt att implementera C-kod i både Android- och iOS-projekt. I och med att C är en delmängd av Objective-C krävs inte lika mycket arbete med att omvandla typer mellan språken vid iOS-utveckling jämfört med Android. För Android användes SWIG för att generera den kod som krävs för att anropa C-kod. De genererade filerna använder sig av JNI, vilket innebär att vanliga Java-typer kan användas vid anrop till C-funktioner.

Vid införande av mer komplicerade datastrukturer som t.ex. fält och structer, vilket används i implementationen av algoritmen i C-kod, konstaterades att ytterligare inkluderingar och deklarationer krävdes i i-filen som SWIG använder. Detta innebar i sig en tidsmässig engångskostnad för att ta reda på vilka inkluderingar och deklarationer som krävdes. Vid vidareutveckling kan detta bli en återkommande kostnad tidsmässigt då användandet av ny funktionalitet eller nya datatyper implementeras.

Om SWIG inte är bekant sen tidigare kan det ta tid att skriva en i-fil som är korrekt. Dessutom kan ytterligare tid behövas för att modifiera filen vid vidareutveckling. Med stor sannolikhet sparas inte någon

tid på att implementera logik i C, istället för i plattformens primära språk. Med detta som grund är en C-implementation inte lämplig om anledningen är att minska utvecklingstiden.

En annan anledning är när det finns ett värde i att samma kod kompileras och exekveras på flera plattformar. Ett exempel är om koden måste bevisas korrekt, vilket kan vara en tidskrävande process. Medicinska tillämpningar är ett exempel då detta kan vara nödvändigt. Om det då går att bevisa korrektheten för en källkod istället för två kan det finnas besparingar både ekonomiskt och tidsmässigt.

Bortsett från det merarbete som SWIG kan innebära vid vidareutveckling finns inga egentliga nackdelar vad det gäller modularitet, livscykel eller användarvänlighet. Alltså bedöms en C-implementation som lämplig i det fallet då det finns ett intresse av att samma kod körs.

Vid jämförelse av C, Objective-C och Java är en skillnad att både Objective-C och Java erbjuder en större mängd standard-API:er. För C finns ett antal tredjepartsbibliotek som kan användas för att få motsvarande funktionalitet. Att använda tredjepartsbibliotek kan dock innebära att utvecklaren själv måste kompilera biblioteket. Kompilering kan ibland göras enkelt genom att använda verktyget “make”. Om inget konfigureras sker kompilering för värdplattformen. Konfigurering så att verktyget istället kompilerar för en annan plattform, t.ex. Android eller iOS kan vara krångligt och ta tid om inte kunskapen finns för hur tillvägagångssättet ser ut. Dessutom finns Android tillgängligt för fyra olika arkitekturer vilket innebär att kompilering kan behöva göras flera gånger. För att rätt version sen ska inkluderas för de olika arkitekturerna krävs extra kod i make-filerna som används då Android-applikationen kompileras.

I vissa fall kan bibliotek vara beroende av andra bibliotek, vilket gör problemet ännu mer komplext. Det här problemet visade sig då HTTPS-anrop skulle göras med hjälp av biblioteket libcurl. För att libcurl ska kunna göra HTTPS-anrop krävs att libcurl kan hantera SSL. Den färdigkompilerade versionen av libcurl för iOS hade SSL-stöd, men för Android blev det tvunget att först kompilera OpenSSL för att sen kompilera libcurl med SSL-stöd.

Arbetet med att kompilera tredjepartsbibliotek kan vara komplext och tidsödande. En anledning till att kompileringen kan bli tidsödande är om ingen tidigare kunskap finns om hur kompileringen utförs. Detta eftersom den mängd dokumentation som finns för att beskriva hur kompileringen utförs är begränsad och otydlig. Detta tillsammans med att nya beroenden för fler bibliotek kan uppstå vid vidareutveckling gör att utvecklingstiden blir svår att förutspå. I vissa fall skulle ett nytt beroende av bibliotek också kunna innebära att en lösning blir ohållbar om det inte går att kompilera biblioteket för alla plattformar. Då det inte går att lämna några garantier för hur hållbar lösningen är ur ett livscykel-perspektiv är lösningar som använder sig av tredjepartsbibliotek inte lämpliga att ha i en gemensam kodbas.

Den tredje typen av C-implementation som har undersökts är då specifik hårdvara ska användas. Eftersom kommunikation mellan mjuk- och hårdvara sker via drivrutiner, som sköts av kärnan som används i plattformen, är det inte säkert att det finns ett standardiserat sätt att interagera med hårdvaran från t.ex. C-kod. Detta kan i förlängningen innebära att C-koden kommer att behöva skrivas specifikt för olika plattformar och alltså inte vara intressant för en gemensam kodbas. En alternativ lösning för att ändå kunna använda en gemensam kodbas är om plattformsspecifik kod som abstraherar hårdvaran skrivs för att ge ett gemensamt gränssnitt utåt. Gemensam kod som använder sig av det gemensamma gränssnittet skulle då kunna skrivas.

I projektet har möjligheten av att implementera BLE-kommunikation i C undersökts. Inget C-bibliotek som skulle kunna fungera på flera plattformar hittades och att utveckla ett eget ramverk för BLE skulle innebära ett mycket omfattande arbete som involverar både hårdvara och protokoll. Att det skulle finnas fall där värdet av en gemensam C-kod skulle motivera utveckling av ett eget ramverk bedöms som osannolikt. Det är med andra ord inte lämpligt med en gemensam C-kod om det också innebär att ett eget ramverk måste utvecklas.

6.2 Slutgiltig Android-applikation utifrån analys

Ett syfte med projektet var att utveckla en slutgiltig Android-applikation efter implementering och utvärdering av alla områdena. Utifrån analysen togs beslutet att Android-applikationen skulle implementera algoritmen i C, medan resten av funktionaliteten skulle implementeras i Java.

En applikation med tre aktiviteter utvecklades. Den första aktiviteten som syns är endast en meny bestående av två knappar som antingen tar användaren till en aktivitet för att hämta ett värde från en BLE-enhet eller till en aktivitet för att titta på gamla värden som tidigare laddats upp till en databas.

Om användaren väljer att hämta ett värde från BLE-enheten startas en ny aktivitet som automatiskt försöker hämta ett värde från den första BLE-enheten som hittas. Om hämtning av värdet lyckades kan användaren antingen välja att hämta ett nytt igen, ladda upp värdet till en databas eller backa och inte göra någonting med värdet. Då hämtning av värdet misslyckas kan användaren välja att försöka igen.

Vid valet att ladda upp värdet till en databas startas en ny aktivitet som försöker ladda upp värdet genom att göra ett POST-anrop. Innan värdet laddas upp sker ett anrop till en C-funktion som behandlar värdet. Om POST-anropet lyckades kommer ett GET-anrop att göras som hämtar alla värden som tidigare laddats upp, inklusive det värde som precis laddats upp. I det fall då GET-anropet inte lyckas kommer användaren att kunna välja att försöka igen. Om anropet istället lyckas kommer alla värden att skrivas ut i en text-vy och sen med hjälp av en C-funktion beräkna medelvärdet och skriva ut det.

Den aktivitet som startas då ett värde ska laddas upp är också den aktivitet som startas då användaren väljer att titta på gamla värden, men i detta fall kommer inte ett nytt värde att laddas upp. Istället försöker applikationen hämta värdena direkt.

För att sköta Internetkommunikation och BLE-kommunikation implementerades separata klasser som på ett smidigt sätt uppdaterar aktiviteterna som i sin tur uppdaterar det grafiska gränssnittet.

7. Resultatanalys

Ett argument för en gemensam kodbas är värdet av att samma kod kompileras och körs på flera plattformar. Att samma C-kod kompileras innebär dock inte att en viss indata med 100% säkerhet returnerar samma utdata på alla plattformar och arkitekturer. Dels kan kompilatorer skilja sig åt och dels kan processorer utföra operationer på olika sätt. Det är dock inte särskilt troligt att det ger några problem.

Ett alternativ till att skriva en kod som implementeras på flera plattformar är att låta applikationen för respektive plattform skicka data till en server och låta servern utföra arbetet för att sedan returnera svaret till applikationen. Detta alternativ innebär att samma kompilator, plattform och arkitektur används för alla beräkningar. Beroende på vad applikationen ska använda returvärdet till kan denna lösning diskuteras. Om returvärdet ska lagras i en databas på en server skulle arbetet på datan kunna utföras på servern i samband med uppladdning av värdet. Om returvärdet däremot endast ska befinna sig på enheten är en server som bearbetar data en onödig kostnad både ekonomiskt och för miljön.

Under projektets gång har inte fallet att återanvända tidigare skriven C-kod tagits upp. Då funktionalitet i en existerande C-kod redan finns kan det finnas ett intresse av att återanvända koden istället för att behöva skriva om funktionaliteten i ett eller flera andra programmeringsspråk. Beroende på vad C-koden innehåller kan det vara lämpligt att återanvända koden.

Ett tänkbart sätt att bedöma om det är lämpligt att återanvända existerande C-kod är att bortse från att koden existerar och bedöma om funktionalitet i sig vore lämplig att implementera i C. Om bedömningen är att inte implementera funktionaliteten i C-kod är det förmodligen bättre att implementera funktionaliteten i standardspråket för plattformen även i de fall då existerande C-kod finns. I det här fallet måste också mängden kod tas med som en aspekt. Ju större mängd kod desto större intresse finns det i att använda den redan utvecklade C-koden och en gemensam kod skulle då kunna anses som lämplig trots ökad komplexitet.

Projektet har endast behandlat Android och iOS. Om en applikation ska göras för fler plattformar kan värdet av en gemensam kodbas i C eventuellt vara större, men samtidigt blir det ännu mer arbete med att implementera C-koden i applikationerna.

De flesta motgångarna för att implementera gemensam kod i C har varit i samband med Android. Att använda C-kod tillsammans med iOS har inte varit något problem med tanke på att Objective-C är en direkt påbyggnad av C. Mycket tid under projektet har använts till att kompilera libcurl för Android, varför Android framstått som mer komplext. Hade inte en färdigkompilerad version av libcurl funnits tillgänglig för iOS hade med stor sannolikhet mycket tid också behövts för att kompilera libcurl till iOS. Att ha en gemensam kodbas i C för iOS och någon annan plattform än Android är eventuellt enklare och mer lämpligt. Samtidigt är det inte osannolikt att det skulle uppstå liknande problem vid t.ex. kompilering av tredjepartsbibliotek.

I projektet togs bedömningsaspekter fram efter undersökningen av implementationsområdena. Ett annat sätt som hade kunnat ge en utförligare bedömning hade varit att ta fram bedömningsaspekter först. Därefter hade implementationsområden kunnat identifieras för att testa aspekterna på ett mer utförligt sätt.

De undersökningar som står till grund för de slutsatser som dras i rapporten är implementeringar av C-kod i marknadens två mest använda mobila plattformar. Som en fortsättning på projektet hade antalet

plattformar kunnat utökas för att undersöka hur väl slutsatsen stämmer med verkligheten. Förutom att utöka antalet plattformar, hade en fortsättning kunnat vara att göra implementeringar av C-kod inom fler testområden. Av de två alternativ som nämnts bidrar förmodligen alternativet att öka antalet plattformar mest till ett bättre underlag för slutsatsen. Fler testområden skulle endast bidra till en bättre grund för en undersökning då endast Android och iOS är av intresse, medan ett större antal plattformar skulle bidra till en bättre grund för en undersökning gällande en gemensam kodbas i C generellt sett.

8. Slutsats

En gemensam kodbas i C bedömdes inte minska utvecklingstiden för något av de områden som undersöktes i projektet och därmed är slutsatsen att ingen funktionalitet är lämplig att ha i en gemensam kodbas om syftet är att minska utvecklingstiden.

Om det finns ett annat syfte med den gemensamma koden är funktionalitet som endast använder sig av standardbiblioteken i C lämplig att ha i gemensam kodbas med avseende på modularitet, livscykel och användarvänlighet.

Funktionalitet som använder sig av tredjepartsbibliotek bedöms inte som lämplig i en gemensam kodbas, eftersom lösningen blir komplex och ohållbar ur ett livscykelperspektiv.

Då ytterligare plattformsspecifik C-kod krävs för att göra en gemensam kod i C möjlig, anses inte en gemensam kod lämplig. Detta eftersom det arbete som skulle krävas för att implementera den plattformsspecifika C-koden inte bedöms som motiverat med avseende på tid och komplexitet.

9. Källor

[1] R. Meier, *Professional Android 4 Application Development*, tredje upplagan. Indianapolis, Indiana: John Wiley & Sons, Inc., 2012.

[2] Google Developers, “Anatomy & Physiology of an Android”, *Youtube*. 9 juni, 2008. [Online] Tillgänglig: http://www.youtube.com/watch?v=G-36noTCaiA [Hämtad 20 mars, 2014].

[3] N. Smyth, *Android 4.4 App Development Essentials*. eBookFrenzy, 2014.

[4] Apple, Inc, "iOS Technology Overview", *Apple Developer*, 2013. [Online]. Tillgänglig: https://developer.apple.com/library/ios/documentation/miscellaneous/conceptual/iphoneostechoverview/Introduction/Introduction.html#//apple_ref/doc/uid/TP40007898-CH1-SW1. [Hämtad: 21 mars, 2014].

[5] O. H. Halvorsen och D. Clarke, *OS X and iOS Kernel Programming*. New York: Springer Science+Business Media, 2011.

[6] U. Bilting och J. Skansholm, *Vägen till C*, upplaga 4:1. Lund: Studentlitteratur AB, 2011.

[7] S. Y. Amdani, *‘C’ Programming*. New Delhi: Laxmi Publications Pvt. Ltd., 2009.

[8] J. Dovey och A. Furrow, *Beginning Objective-C*. New York: Springer Science+Business Media, 2012.

[9] J. Skansholm, *Java direkt med Swing*, sjunde upplagan. Lund: Studentlitteratur AB, 2013.

[10] "Android NDK", *Android*. [Online]. Tillgänglig: https://developer.android.com/tools/sdk/ndk/index.html. [Hämtad: 22 april, 2014].

[11] C. Onur, *Pro Android C++ with the NDK*. New York: Springer Science+Business Media, 2012.

[12] "Executive Summary", *SWIG*, 2014. [Online]. Tillgänglig: http://swig.org/exec.html. [Hämtad: 1 april, 2014].

[13] Bluetooth SIG, Inc., "Fast Facts", *Bluetooth*, 2013. [Online]. Tillgänglig: http://www.bluetooth.com/Pages/Fast-Facts.aspx. [Hämtad: 2 april, 2014].

[14] Bluetooth SIG, Inc., "Bluetooth Basics", *Bluetooth*, 2013. [Online]. Tillgänglig: http://www.bluetooth.com/Pages/Basics.aspx. [Hämtad: 2 april, 2014].

[15] "Bluetooth Low Energy", *Android*. [Online]. Tillgänglig: http://developer.android.com/guide/topics/connectivity/bluetooth-le.html. [Hämtad: 2 april, 2014].

[16] Bluetooth SIG, Inc., "Bluetooth Smart Devices", *Bluetooth*, 2013. [Online]. Tillgänglig: http://www.bluetooth.com/Pages/Bluetooth-Smart-Devices.aspx. [Hämtad: 2 april, 2014].

[17] D. Stenberg, "Front Page", *cURL*, 2014. [Online]. Tillgänglig: http://curl.haxx.se. [Hämtad: 22 april, 2014].

A. Manual för att implementera C-kod i Android- och iOS-applikationer

OLLE SVENSSON
ROBIN TÖRNQUIST

1. Android

Det här kapitlet beskriver de olika stegen för att implementera C-kod i Android-projekt med utvecklingsmiljön Eclipse.

1.1 Initiering

För att följa manualen krävs följande verktyg:

- Eclipse
- Android Developer Tools
- Android Software Development Kit
- Android Native Development Kit
- SWIG

Beroende på vilket operativsystem som används ser installationsprocessen för de olika verktygen annorlunda ut. En detaljerad beskrivning av hur installationen ser ut för Windows, Mac OS X och Ubuntu finns i boken *Pro Android C++ with the NDK* av Onur Cinar, under kapitel 1, *Getting Started with C++ on Android* samt kapitel 4, *Auto-Generate JNI Code Using SWIG*.

Denna manual använder sig av Eclipse Kepler. C-filerna som används i exemplet finns beskrivna i kapitel 3.

1.2 Android-applikation med C-implemenation

1.2.1 Skapa projekt

Skapa ett nytt Android-projekt i Eclipse via:

File → New → Other... → Android → Android Application Project

Klicka *Next* och följ därefter anvisningarna.

1.2.2 Native support

Lägg till stöd för C-kod i projektet genom att högerklicka på projektet i *Package Explorer* eller *Project Explorer* och välj:

Android Tools → Add Native Support…

Välj namn på biblioteket för C-koden och klicka på *Finish*.

I samband med det här steget skapas en mapp i projektmappen med namnet *jni*. Mappen innehåller filer som är relaterade till C-implementationen. Bland annat ska *Android.mk* och *Application.mk* finnas i mappen, om någon fil saknas kan den skapas genom att högerklicka på *jni*-mappen och välja:

New → File

Fyll i filnamnet för den fil som saknas och klicka på *Finish*.

1.2.3 SWIG

Nästa steg är att använda verktyget SWIG för att generera de filer som krävs för att anropa C-koden från Java. Börja med att skapa en i-fil i *jni*-mappen på samma sätt som i steg 2 genom att högerklicka på *jni*-mappen och välja:

New → File

Välj filnamn och klicka på *Finish*.

Filen behöver innehålla modulnamn, inkludering av header-filer och funktionsdeklarationer för de funktioner som ska anropas från Java. I detta exempel kallas filen för *manual.i* och innehåller följande:

```
%module Manual

%{
#include "hello.h"
%}

extern char* hello_from_c(void);
```

Från terminalen kan nu följande kommando köras:

```
swig -java -package <paketnamn> -outdir <utmapp> <i-fil>
```

`<paketnamn>` ersätts med det önskade namnet för Java-paketet.
`<utmapp>` ersätts med sökvägen till den mapp där de genererade Java-filerna ska hamna.
`<i-fil>` ersätts med sökvägen till i-filen. I samma mapp hamnar även en genererad C-fil.

Sökvägarna är relativa den aktuella katalogen som terminalen befinner sig i. SWIG-kommandot för det här exemplet blir följande då terminalen utgår från projektkatalogen:

```
swig -java -package com.example.manual -outdir src/com/example/manual
jni/manual.i
```

Notera att hela kommandot skrivs på samma rad och att det ska finnas mellanslag mellan parametrar och sökvägar.

Efter att SWIG-kommandot utförts kan projektet behöva uppdateras. Högerklicka på projektet i *Package Explorer* eller *Project Explorer* och välj *Refresh*. Det ska nu finnas minst två nya Java-filer i *src*-mappen under paketet som angavs samt en ny C-fil i *jni*-mappen.

1.2.4 Konfigurera mk-filer

När alla nödvändiga filer genererats är nästa steg att konfigurera de två mk-filerna som tidigare skapats i *jni*-mappen. Filerna innehåller information som kompilatorn behöver för att bygga projektet. För enkelhetens skull ligger C-filerna för detta exempel i *jni*-mappen.

Filerna för detta exempel ser ut enligt följande:

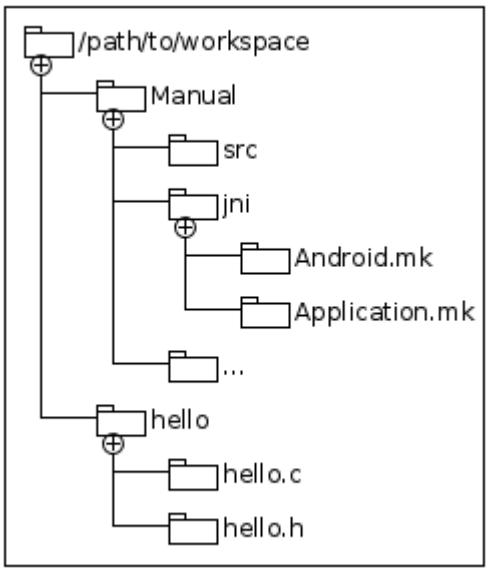
Android.mk

```
1    LOCAL_PATH := $(call my-dir)
2
3    include $(CLEAR_VARS)
4
5    LOCAL_MODULE    := Manual
6    LOCAL_SRC_FILES := manual_wrap.c hello.c
7
8    include $(BUILD_SHARED_LIBRARY)
```

Rad 1, 3 och 8 måste alltid finnas med i *Android.mk*. På rad 5 anges det namn som senare används i Java-filerna för att ladda in modulen. På rad 6 anges de C-filer som ska kompileras och ingå i modulen. I detta exempel består C-filerna av *manual_wrap.c* som är den SWIG-genererade filen samt *hello.c* som innehåller funktionen *hello_from_c* som ska anropas från Java-kod.

Application.mk

```
1    APP_ABI := all
```



I detta exempel består *Application.mk* endast av en rad. Denna rad talar om för kompilatorn vilka arkitekturer modulen ska byggas för. I detta fall ska modulen byggas för alla de arkitekturer som Android-implementationer finns på, nämligen *armeabi, armeabi-v7a, mips* och *x86*.

Som tidigare nämndes, ligger C-filerna i *jni*-mappen för detta exempel. Om C-filerna istället legat i en annan mapp, t.ex. enligt mappstrukturen i figuren till höger hade *Android.mk* sett ut enligt följande:

```
1    LOCAL_PATH := $(call my-dir)
2    LOCAL_C_PATH := ../../hello
3
4    include $(CLEAR_VARS)
5
6    LOCAL_MODULE    := Manual
7    LOCAL_SRC_FILES := manual_wrap.c $(LOCAL_C_PATH)/hello.c
8
9    LOCAL_C_INCLUDES := $(LOCAL_PATH)/$(LOCAL_C_PATH)
10
11   include $(BUILD_SHARED_LIBRARY)
```

Det som har adderats i denna fil är den lokala variabeln *LOCAL_C_PATH* på rad 2 som innehåller den relativa sökvägen från *jni*-mappen till C-filerna samt *LOCAL_C_INCLUDES* på rad 9 som anger ytterligare sökvägar till kataloger med header-filer. I och med att header-filen nu ligger på en annan plats behöver kompilatorn veta att den ska leta efter header-filer även där.

1.2.5 Importera C-modul

För att kunna använda C-funktionerna från Java-kod måste C-modulen laddas in. För att ladda in modulen används följande kod:

```
static {
      System.loadLibrary("Manual");
}
```

Manual är det modulnamn som angetts i *Android.mk*.

I exemplet läggs denna kod in i Java-filen för den aktivitet där C-koden ska anropas enligt följande:

```
public class MainActivity extends Activity {

	static {
		System.loadLibrary("Manual");
	}

	@Override
	protected void onCreate(Bundle savedInstanceState) {
		...
	}
	...
}
```

1.2.6 Anropa C-kod

Anrop till C-funktioner kan nu göras via de statiska metoder som finns i en av Java-filerna som SWIG genererade. I detta exempel heter filen *Manual.java*.

För att på ett enkelt sätt testa att allt nu fungerar som det ska kan returvärdet av en C-funktion skrivas ut i ett logg-meddelande enligt:

```
@Override
protected void onCreate(Bundle savedInstanceState) {

	...

	Log.d("MANUAL", Manual.hello_from_c());
}
```

1.3 Tredjepartsbibliotek

I de fall då tredjepartsbibliotek skrivna i C ska användas krävs ibland att utvecklaren själv ansvarar för kompilering av dessa. För att kompilera ett bibliotek krävs en så kallad *toolchain*. En *toolchain* består av en samling verktyg, som bland annat innehåller en korskompilator som kompilerar C-kod till den arkitektur som koden ska exekveras på. Det här kapitlet beskriver hur en *toolchain* skapas med hjälp av Android NDK. Kapitlet innehåller också ett exempel på hur ett kompilerat tredjepartsbibliotek kan importeras i ett Android-projekt.

1.3.1 Skapa toolchain

För att skapa en *toolchain* i en mapp används skript-filen *make-standalone-toolchain.sh* som går att hitta i NDK-katalogen under *build/tools/*. Då arbetskatalogen i terminalen är *<NDK-katalog>/build/tools/* kan skript-filen anropas från terminalen enligt:

```
./make-standalone-toolchain.sh \
      --platform=<platform> \
      --install-dir=<install_dir> \
      --arch=<arch> \
      --system=<system>
```

Snedstrecken på slutet av raderna innebär att nästa rad är en fortsättning av föregående rad.

`<platform>` ersätts med den version av Android som används i projektet.
`<install_dir>` ersätts med sökvägen till katalogen där filerna ska placeras.
`<arch>` ersätts med namnet på den arkitektur verktygen ska användas för.
`<system>` ersätts med värdsystemets namn.

För en 64-bitars Linux-distribution kan följande anrop från terminalen göras:

```
./make-standalone-toolchain.sh \
      --platform=android-19 \
      --install-dir=/home/user/android/x86_toolchain \
      --arch=x86 \
      --system=linux-x86_64
```

I exemplet ovan kommer en *toolchain* anpassad för API-version 19 och arkitekturen *x86* att skapas och placeras i katalogen */home/user/android/x86_toolchain*.

För Mac OS X kan motsvarande anrop från terminalen göras:

```
./make-standalone-toolchain.sh \
      --platform=android-19 \
      --install-dir=/home/user/android/x86_toolchain \
      --arch=x86 \
      --system=darwin-x86_64
```

Dessa verktyg kan sen användas för att kompilera tredjepartsbibliotek för Android. Eftersom kompileringsprocessen av tredjepartsbibliotek kan se olika ut för olika bibliotek kommer inte en genomgång av detta att beskrivas i den här manualen.

1.3.2 Importera tredjepartsbibliotek i Android-projekt

Då ett tredjepartsbibliotek skrivet i C finns finns kompilerat kan det importeras i Android-projektet genom ytterligare konfiguration i *Android.mk*. Här ges ett exempel på en *Android.mk*-fil där tredjepartsbiblioteket *libcurl* importeras:

```
1   LOCAL_PATH := $(call my-dir)
2
3   include $(CLEAR_VARS)
4   LOCAL_MODULE := static_curl
5
6   # TARGET = ARMEABI
7   ifeq ($(TARGET_ARCH_ABI), armeabi)
8         LOCAL_SRC_FILES := armeabi/libcurl.a
9   endif
10
11  # TARGET = ARMEABI-V7A
12  ifeq ($(TARGET_ARCH_ABI), armeabi-v7a)
13        LOCAL_SRC_FILES := armeabi-v7a/libcurl.a
14  endif
15
16  # TARGET = X86
17  ifeq ($(TARGET_ARCH_ABI), x86)
18        LOCAL_SRC_FILES := x86/libcurl.a
19  endif
20
21  # TARGET = MIPS
22  ifeq ($(TARGET_ARCH_ABI), mips)
23        LOCAL_SRC_FILES := mips/libcurl.a
24  endif
25
26  LOCAL_EXPORT_LDLIBS := -lz
27  include $(PREBUILT_STATIC_LIBRARY)
28
29  include $(CLEAR_VARS)
30  LOCAL_MODULE    := example_module
31
32  LOCAL_SRC_FILES :=    swig_wrap.c \
33                        user_defined.c
34
35  LOCAL_C_INCLUDES := ./curl
36
37  LOCAL_STATIC_LIBRARIES := static_curl
38
39  include $(BUILD_SHARED_LIBRARY)
```

Det som sker i raderna ovan är följande:

- `4:` Anger namnet på den lokala modulen som ska innehålla det statiska tredjepartsbiblioteket.
- `6 - 24:` Väljer det förkompilerade bibliotek som ska användas till den aktuella arkitekturen.
- `26:` Talar om för kompilatorn att *libz*, ett C-kodsbibliotek, ska inkluderas när den lokala modulen byggs (*libcurl* är beroende av *libz*).
- `27:` Talar om för kompilatorn att föregående rader ska tolkas som ett statiskt bibliotek.
- `35:` Anger var header-filer för *libcurl* finns.
- `37:` Talar om för kompilatorn att inkludera modulen som innehåller det statiska bibliotek som definierats på rad 3 - 27.

I ovanstående *Android.mk*-fil är sökvägarna förenklade.

En mer detaljerad beskrivning av hur *Android.mk*-filen ska konfigureras beroende på vad som ska inkluderas finns i boken *Pro Android C++ with the NDK* av Onur Cinar, under kapitel 2, *Exploring the Android NDK*.

2. iOS

Det här kapitlet beskriver de olika stegen för att implementera C-kod i iOS-projekt med utvecklingsmiljön Xcode.

2.1 Initiering

För att följa manualen krävs att Xcode finns installerat. I nuläget finns Xcode endast till Mac OS X. Manualen använder sig av Xcode Version 5.1.1. C-filerna som används i exemplet finns beskrivna i kapitel 3.

2.2 iOS-applikation med C-implementation

2.2.1 Skapa projekt

Skapa ett nytt iOS-projekt i Xcode via:

File → New → Project

Välj *Single View Application*, klicka på *Next* och följ därefter anvisningarna.

2.2.2 Importera C-filer

Importera de C-filer som ska användas i projektet genom att välja:

File → Add Files to "<projektnamn>" ...

Välj vilka filer alternativt vilken mapp som ska importeras och klicka på *Add*.

2.2.3 Anropa C-kod

För att kunna anropa C-koden behöver header-filen för koden inkluderas i implementations-filen där anropet sker. För detta exempel blir inkluderingen följande rad:

```
#import "hello.h"
```

Raden ovan kan antingen skrivas direkt i implementations-filen eller i en annan header-fil som senare inkluderas i implementations-filen.

För att på ett enkelt sätt testa att allt nu fungerar som det ska kan returvärdet av en C-funktion skrivas ut i ett logg-meddelande enligt:

```
@implementation ViewController

- (void)viewDidLoad
{
      ...

      NSLog([NSString stringWithUTF8String: hello_from_c()]);
}
...
@end
```

3. Exempelkod

hello.h

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

char* hello_from_c(void);
```

hello.c

```
#include "hello.h"

char* hello_from_c(void) {
      return "Hello, from C";
}
```