Handbuch Programmierung/ Programming Manual

Programmieranleitung für ACD Geräte mit Android™*/ Programming Instructions for ACD Devices with Android™*

Version: 1.18

Inhaltsübersicht/Content

Deutsch

English

1 Übersicht

1.1 Ziel und Umfang dieses Handbuchs

Dieses Dokument ist ein Handbuch für Programmierer, Systemadministratoren und -konfiguratoren der ACD Geräte mit Android.
Es soll Einsichten in den Funktionsumfang und die Möglichkeiten der betreffenden Geräte bieten und eine erste Anlaufstelle für Fragen rund um die Themen Programmierung, Konfiguration und Administration darstellen.
Aufgrund der Komplexität moderner Betriebssyteme und deren Programmierung geht dieses Dokument dabei nicht in beliebiger Tiefe auf alle Komponenten ein. Für einige Themen existieren daher eigene Dokumentationen, auf die im Text verwiesen wird. Diese können bei Bedarf von ACD angefordert werden.

1.2 ACD Geräte mit Android

Die folgende Tabelle liefert eine Übersicht welche ACD Geräte mit Android betrieben werden können.

Gerät	Produktversion … und höher	Betriebssystem	API-Level
M270	1.10	Industrial+ for Android	23
M270SE	2.00.1	Industrial+ for Android	30
M260TE	1.8	Industrial+ for Android	23
M266SE	1.7	Industrial+ for Android	23
M2Smart®	1.6	Industrial+ for Android	23
M2Smart®SE	2.01	Industrial+ for Android	30
M2Smart®SE 10 Zoll	1.20	Industrial+ for Android	28

Tabelle 1: ACD Geräte mit Android

2 ACD ScanService

Der ACD ScanService bietet Zugriff auf den Scanner, um diesen zu konfigurieren und gescannte Barcodes weiterzuleiten. Er hört außerdem die Scannertasten ab und kann den Scanner auslösen. Gescannte Barcodes werden vom ACD ScanService empfangen und je nach Konfiguration weitergeleitet.

Wird das Standardverhalten beim Lesen eines Barcodes nicht geändert, so wird der gelesene Barcode als Tastaturereignis ins System gemeldet.
Dies kann verwendet werden, um Daten in generische Applikationen zu scannen, die nicht an die Kommunikation mit dem ACD ScanService angepasst werden können. Die App bemerkt in diesem Fall keinen Unterschied zwischen Scan oder Verwendung einer Tastatur.

Für selbst entwickelte Anwendungen wird empfohlen, diese an den ACD ScanService zu binden. Dadurch erkennt Android, dass der ACD ScanService für die Anwendung benötigt wird und hält ihn aktiv.

2.1 Empfang von Barcodes

Es gibt mehrere Möglichkeiten, um einen Barcode von dem Dienst in die Zielanwendung auszuliefern. Sie können in den Voreinstellungen oder über die API des Dienstes von einer App gesetzt werden. Siehe SET_TARGET)

2.1.1 Tastatureinschleifung

Dabei schleift der Dienst den Barcode in das Tastatursystem von Android ein. Dies ist eher mit einer externen Tastatur zu vergleichen, jedoch nicht mit dem Konzept der virtuellen Tastatur.
Dieser Mechanismus kann Daten in jedes aktive Eingabefeld eintragen. So kann beispielsweise ein WLAN-Passwort in das Eingabefeld gescannt statt getippt werden.
Die Einschleifung hat allerdings ihre Grenzen. Das zugrundeliegende Systemprogramm von Android kommt nur mit einem Basis Zeichensatz zurecht, sodass neben den Buchstaben A - Z (Groß und Klein) und den Ziffern 0 - 9 kaum weitere Zeichen akzeptiert werden. Das Verhalten hängt, von im System existierenden, Tastaturlayouts ab und kann somit nicht für alle denkbaren Barcodezeichen garantiert werden.

2.1.2 Versand eines Broadcast

Der Dienst sendet einen Broadcast Intent, mit dem Namen „de.acdgruppe.scanservice.SCANBROADCAST“. Wenn die Dienstversion älter als V1.22 ist, heißt dieser Broadcast Intent „de.acdgruppe.scanservice:SCANBROADCAST“. In den „extra“ Feldern SCANNERDATA_BARCODE und SCANNERDATA_BARCODETYPE enthält dieser Intent den Barcode und dessen Typ als String. In dem Feld SCANNERDATA_TYPECODE wird der Typ des Barcodes als Integer übergeben.
Eine App, die sich als BroadcastReceiver registriert, kann den Barcode auf diese Weise empfangen und nach deren Bedürfnissen verarbeiten. Insbesondere kann damit bei Feldern ohne wahlweise Tastatureingabe die Bildschirmtastatur entfallen, was das Eingabefenster übersichtlicher hält. Diese Methode eignet sich für Anwendungsfälle in denen der Scanner fest konfiguriert wird und die App somit einen unkomplizierten Empfangsmechanismus bereitstellen kann. Es ist allerdings zu beachten, dass jede Anwendung den Broadcast empfangen kann.

2.1.3 Verwendung des Messengers

Eine Anwendung, die sich an den Dienst bindet, kann sich über die Bindung einen „Messenger“ erstellen und über diesen mit dem Dienst kommunizieren. Der Name des Dienstes lautet „`de.acdgruppe.scanservice.AcdScannerService`“. Wird der Dienst für den Versand von Barcodes per Messenger konfiguriert, so werden die Daten gezielt an die Anwendung versandt, die sich als letztes beim Dienst registriert hat. Die Bindung sorgt außerdem dafür, dass Android den Dienst während der Laufzeit der App nicht beendet.

Durch die Konfigurationsmöglichkeit über den Messenger kann die Anwendung den Scannerdienst an den jeweiligen Ablaufschritt anpassen. Für Ablaufschritte ohne Scannernutzung kann sie die automatische Auslösung des Scanners deaktivieren und die Scannertasten für andere Zwecke verwenden.[1]
Die Konfiguration der erlaubten Barcodetypen ist hilfreich, wenn die Ablaufschritte verschiedene Barcodetypen nutzen und kann Fehlscans vermeiden.

2.1.4 Verwendung des Messengers für das M2Longrange Modul

Ab Version 1.40 der App ACD ScanService stellt diese einen zweiten Dienst für die Anbindung des M2Longrange Moduls an das M2Smart®SE bereit. Der Name des Dienstes lautet „`de.acdgruppe.scanservicelr.AcdScannerServiceLR`“. Es treffen alle Informationen des vorhergehenden Absatzes in gleicher Weise zu.

[1] Die Tasten werden der Activity durch die gewohnten Events (onKeyDown etc.) gemeldet, unabhängig davon ob der Scandienst den Scanner auslöst oder nicht.

2.1.5 Beispiele einer Bindung

Hinweis: ein Detailierteres Beispiel für die gesamte Kommunikation ist in Kapitel 2.4 zu finden.

```
Intent scanServiceIntent = new Intent();
ComponentName scanServiceName =
  new ComponentName("de.acdgruppe.scanservice", "de.acdgruppe.scanservice.AcdScannerService");
scanServiceIntent.setComponent(scanServiceName);
if (getApplicationContext().bindService(scanServiceIntent,
  scannerConnection, Context.BIND_AUTO_CREATE)) { .... }

Intent scanServiceM2LRIntent = new Intent();
ComponentName scanServiceM2LRName =
  new ComponentName("de.acdgruppe.scanservice", "de.acdgruppe.scanservicelr.AcdScannerServiceLR");
scanServiceM2LRIntent.setComponent(scanServiceM2LRName);
if (getApplicationContext().bindService(scanServiceM2LRIntent,
  scannerM2LRConnection, Context.BIND_AUTO_CREATE)) { .... }
```

2.2 Die API des Dienstes

Das Konzept der Bindung an den Dienst wird als bekannt vorausgesetzt und soll nur kurz skizziert werden.

1. Die App ruft an geeigneter Stelle bindService( ) auf.
2. Im Ereignis onServiceConnected( ) kann die App ein Messenger Objekt erstellen, das sie zum Senden von Nachrichten an den Dienst verwendet. Die App konfiguriert den Dienst je nach Bedürfnis.
3. Ein weiterer Messenger empfängt vom Dienst kommende Nachrichten. Barcodes, die über den Messenger empfangen werden, leitet die App an den zugehörigen Programmcode weiter.
4. An geeigneter Stelle ruft die App unbindService( ) auf und gibt den Dienst wieder frei.

Bei Verwendung mehrerer Activities in der App ist es hilfreich, den Dienst global, also für die gesamte Lebenszeit der App gebunden zu halten, statt ihn in jeder Activity separat zu binden.

Der Dienst wird durch Messages gesteuert. Ein Message Objekt enthält die drei Zahlenfelder what, arg1 und arg2. what enthält den Nachrichtentyp, arg1 und arg2 enthalten mögliche weitere Details zur Nachricht.

Nachricht	Beschreibung	Nachricht Nr. (what-msg)
OK	Wird vom Dienst als Antwort auf eine Anfrage der Clientapp versendet	1
HELLO	Der Dienst antwortet auf HELLO mit OK. Darüber hinaus findet kein Login statt.	2
CONFIGURE	Nachricht nimmt Einstellungen am Dienst vor	3
SCAN	Scanstrahl aus der Anwendung an- und ausstellen	4
SET_TARGET	Auswahl der Barcodeübertragung	5
BARCODE	Nachricht enthält Barcodeinformationen	6
KEY_ENABLE	Scan auslösen beim Betätigen der Scannertasten	7
AIM	Zielstrahl an- oder ausstellen	8

Tabelle 2: API ScanService

2.2.1 Nachrichtentypen

Für „what“ gibt es die folgenden Möglichkeiten:

OK

Wird vom Dienst als Antwort auf eine Anfrage der Clientapp versandt.

what	arg1	arg2
1	Rückgabe der angefragten what Bsp: ‘2‘ für HELLO	0 = Konnte nicht bearbeitet werden 1 = Rückmeldung ist okay

HELLO

Der Dienst antwortet auf HELLO mit OK. Die Antwort auf HELLO enthält Daten über den Scanner. Darüber hinaus findet kein Login oder sonstiger Informationsaustausch statt.

what	arg1	arg2
2	Keine Funktion	Keine Funktion

Die OK Nachricht als Antwort auf HELLO enthält im Bundle-Feld der Nachricht Informationen über den Scanner.

Es gibt die Einträge:

- ENGINECODE: Der Code der Scanengine.
- SWREVISION: Die Softwarerevision des Scanners.
- NAME: Name des Scanners (*).
- MODULENR: Die Nummer des Scanmoduls (*).

Die mit (*) markierten Felder werden aus dem Enginecode ermittelt und sind leer, falls ein dem Scandienst unbekannter Scanner eingebaut ist.

CONFIGURE

Diese Nachricht nimmt Einstellungen am Dienst vor. Aktuell ist dies noch ohne Funktion.

what	arg1	arg2
3	Keine Funktion	Keine Funktion

SCAN

Der Client kann über diese Nachricht den Scanstrahl auslösen oder beenden. Hierbei ist zu beachten, dass der Scanner nach einer gewissen Zeit automatisch abschaltet, wenn kein erfolgreicher Scan erfolgt ist.

what	arg1	arg2
4	0 = Scanstrahl beenden 1 = Scanstrahl auslösen	Keine Funktion

SET_TARGET

Über diese Nachricht wählt der Client, wie er einen gescannten Barcode erhalten möchte. Die Optionen der Nachricht werden im nächsten Kapitel erläutert.

what	arg1	arg2
5	Ziel	Keine Funktion

Die Nachricht SET_TARGET steuert das Verhalten des Dienstes, wenn er erfolgreich einen Barcode dekodiert hat. Es gilt die jeweils zuletzt vorgenommene Einstellung, unabhängig von der momentan aktiven App. Je nach Einsatzgebiet und Wunsch des Betreibers sollte die App zumindest vor dem Abkoppeln (unbind) den Dienst auf die gewünschten Standardeinstellungen zurücksetzen, beispielsweise die Tastatureinschleifung.

Das Feld arg1 kann bei der Nachricht SET_TARGET folg-ende Werte annehmen:

- NONE
  Der Dienst leitet einen gescannten Barcode nicht weiter. Dieser Wert existiert der Vollständigkeit halber, ein Ablaufschritt ohne Scanvorgang sollte die Scannertasten deaktivieren.
  arg1 = 0

- INJECT
  Der Dienst schleift den Barcode in das Tastatursystem ein. Dies ist die Voreinstellung des Dienstes. Nach Beenden der App (bzw. dem Pausieren) sollte dieses Ziel wiedereingestellt werden. Siehe: „**Fehler! Verweisquelle konnte nicht gefunden werden.** Tastatureinschleifung“
  arg1 = 1

- BROADCAST_INTENT
  Der Barcode wird über einen, per Broadcast, verschickten Intent weitergegeben.
  Siehe: „2.1.2 Versand eines Broadcast“
  arg1 = 2

- MESSENGER
  Der Dienst sendet den Barcode an das Messenger Objekt, das dieses Ziel festgelegt hat. Das versendete Bundle enthält die Felder:
  - „barcode“ vom Typ String: Beinhaltet den Barcode
  - „type“ vom Typ String: Beinhaltet den Barcodetyp
  - „typecode“ vom Typ int: Beinhaltet den Typecode

  Siehe: „2.1.3 Verwendung des Messengers“
  arg1 = 3

BARCODE

Über diese Nachricht sendet der Dienst einen gescannten Barcode an den Client, sofern der Client diese Versandart gesetzt hat. Der Barcode wird an dasjenige Messenger Objekt geschickt, über das es die Versandart gewählt hat. Es wird nur an das jeweils zuletzt gesetzte Messenger Objekt gesandt.

what	Rückgabe
6	Enthält Bundle mit den Barcodeinformationen

KEY_ENABLE
Dadurch legt der Client fest, ob der Dienst bei Bedienung der Scannertasten den Scanner automatisch auslöst oder nicht.

what	arg1	arg2
7	0 = Nicht auslösen 1 = Auslösen	Keine Funktion

AIM
Der Client legt fest, ob der Zielstrahl des Scanners ausgelöst werden soll.
Diese Funktion ist mit dem 2D-Longrange-Scanner nicht verfügbar.

what	arg1	arg2
8	0 = Zielstrahl aus 1 = Zielstrahl ein	Keine Funktion

2.2.2 Barcodetypen

Der ScanService liefert mit jedem gescannten Barcode einen Typecode sowie den Barcodetyp zurück.

Barcodetyp	Typecode
Code 39	01
Code 39 Full ASCII	13
Trioptic Code 39	15
Code 32	20
Code 93	07
Codabar	02
NW 7	18
Code 128	03
ISBT 128	19
ISBT 128 Concatenation	21
Discrete 2 of 5	04
IATA 2 of 5	05
Interleaved 2 of 5	06
Matrix 2 of 5	71
Chinese 2 of 5	72
Korean 3 of 5	73
UPC A	08
UPC A with 2 Supplementals	48
UPC A with 5 Supplementals	88
UPC E	09
UPC E with 2 Supplementals	49
UPC E with 5 Supplementals	89
UPC E1	10
UPC E1 with 2 Supplementals	50
UPC E1 with 5 Supplementals	90
EAN 8	0A
EAN 8 with 2 Supplementals	4A
EAN 8 with 5 Supplementals	8A
EAN 13	0B
EAN 13 with 2 Supplementals	4B
EAN 13 with 5 Supplementals	8B

Barcodetyp	Typecode
4State US	34
4State US4	35
Scanlet Webcode	37
Cue CAT Code	38
French Lottery	2F
GS1 128	0F
GS1 DataBar 14	30
GS1 DataBar Limited	31
GS1 DataBar Expanded	32
GS1 DataBar Expanded Coupon	B4
GS1 Data Matrix	C1
PDF 417	11
Macro PDF 417	28
Micro PDF	1A
Micro PDF CCA	1D
Macro Micro PDF	9A
Data Matrix	1B
GS1 Data Matrix	C1
QR Code	1C
Micro QR Code	2C
Macro QR Code	29
Maxicode	25
Aztec	2D
Aztec Rune Code	2E
Han Xin	B7
Signature	69
OCRB	A0
TLC 39	5A
Composite CC-A GS1-128	51
Composite CC-A EAN-13	52
Composite CC-A EAN-8	53

Barcodetyp	Typecode
Bookland EAN	16
Code 11	0C
Code 49	0D
MSI	0E
Code 16K	12
UPCD	14
Coupon Code	17
Postnet US	1E
Planet US	1F
Postal Japan	22
Postal Australia	23
Postal Dutch	24
Postbar CA	26
Postal UK	27

Barcodetyp	Typecode
Composite CC-A GS1-DB Expanded	54
Composite CC-A GS1-DB Limited	55
Composite CC-A GS1-DataBar 14	56
Composite CC-A UPC-A	57
Composite CC-A UPC-E	58
Composite CC-C GS1-128	59
Composite CC-B GS1-128	61
Composite CC-B EAN-13	62
Composite CC-B EAN-8	63
Composite CC-B GS1-DB Expanded	64
Composite CC-B GS1-DB Limited	65
Composite CC-B GS1-DataBar 14	66
Composite CC-B UPC-A	67
Composite CC-B UPC-E	68

Tabelle 3: Typecode Scanner außer 2D-LR

Ist ein 2D-Longrange-Scanner verbaut, werden andere Typecodes zurückgemeldet.

Barcodetyp	Typecode
Code 39	105
Codabar	101
Codablock	102
Australian Post	97
Canada Post	100
Dutch Post	110
Japan Post	120
Sweden Post	73
Aztec	98
BP0	99
Code11	104
Code 128/GS1-128	107
DataMatrix	109
EAN/UPC	111
Infomail	118

Barcodetyp	Typecode
Matrix 2 of 5	121
MaxiCode	122
PDF417/MicroPDF417	66
Planet	68
Plessey Code	69
Postnet	70
QR Code	71
Standard 2 of 5	72
Telepen	74
TLC39	75
Interleaved 2 of 5	119
Code 93	106
MSI Code	65
GS1 DataBar	117
GS1 Composite	115

Tabelle 4: Typecode Scanner 2D-LR

2.3 Scannerkonfiguration

Die ACD ScanConfig App ist auf den ACD Geräten vorinstalliert und wird dazu verwendet Scanner-einstellungen vorzunehmen. Unter anderem können über die App folgende Einstellungen vorgenommen werden:

- Konfiguration von Scannerparametern (erlaubte Barcodes, …)
- Konfiguration der Scannerdaten (Anhängen von Prefix, Suffix, …)
- Benachrichtigungston beim Scan eines Barcodes
- Etc.

Die Konfigurationsdaten der ACD ScanConfig werden auf dem Gerät unter */sdcard/ACD/scanner* abgelegt. Der ACD ScanService verwendet die Einstellungen, die in der ACD ScanConfig vorgenommen werden, um den im Gerät verbauten Scanner zu konfigurieren.

2.4 Beispiel Scanner Kommunikationsklasse

```
public class MainActivity extends AppCompatActivity {

    private Messenger _serviceMessenger = null;
    private boolean _boundToService = false;
    private TextView txtbarcode;

    public static final int WHAT_OK = 1;
    public static final int WHAT_HELLO = 2;
    public static final int WHAT_SCAN = 4;
    public static final int WHAT_SET_TARGET = 5;
    public static final int WHAT_BARCODE = 6;
    public static final int WHAT_KEY_ENABLE = 7;
    public static final int TARGET_MESSAGE = 3;


    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        txtbarcode = findViewById(R.id.txtbarcode);
        bindToService(this);
    }

    public void bindToService(Context context) {
        Intent intent = new Intent();
        intent.setComponent(new ComponentName("de.acdgruppe.scanservice", "de.acdgruppe.scanservice.AcdScannerService"));
        // for Longrange use: new ComponentName("de.acdgruppe.scanservicelr", "de.acdgruppe.scanservicelr.AcdScannerServiceLR")

        try {
            // Use applicationContext because the activity can change during rotation and it is not
            // obvious what happens to the service if bound to the activity.
            if (!context.bindService(intent, _connection, Context.BIND_AUTO_CREATE))
                return;
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private ServiceConnection _connection = new ServiceConnection() {

        public void onServiceConnected(ComponentName className, IBinder service) {
            // This is called when the connection with the service has been established, giving us the
            // service object we can use to interact with the service.
            _serviceMessenger = new Messenger(service);

            // Send a hello to the service. Next steps follow when the response arrives.
            _boundToService = true;
            sendMessage(WHAT_HELLO, 0, 0);
        }

        public void onServiceDisconnected(ComponentName className) {
            // This is called when the connection with the service has been
            // unexpectedly disconnected, its process crashed.
            _serviceMessenger = null;
            _boundToService = false;
        }
    };

    public void sendMessage(int what, int arg1, int arg2) {
        if (_serviceMessenger == null) {
            // The messenger should never be null, there's one not-changing instance during the app's life.
            String text = String.format("Service messenger is null! (Message=%d)", what);
            return;
        }

        if (!_boundToService) {

            return;
        }

        try {
            Message msg = Message.obtain(null, what, arg1, arg2);
            msg.replyTo = _receiverMessenger;
            _serviceMessenger.send(msg);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }

    private Messenger _receiverMessenger = new Messenger(new Handler(new Handler.Callback() {
        @Override
        public boolean handleMessage(Message message) {
            switch (message.what) {
                case WHAT_OK:
                    // arg1 contains the reason for this OK message
                    switch (message.arg1) {

                        case WHAT_HELLO:
                            // We are connected => set the service to send barcode messages via messenger
                            sendMessage(WHAT_SET_TARGET, TARGET_MESSAGE, 0);
                            // Enable the scankey trigger
                            sendMessage(WHAT_KEY_ENABLE, 1, 0);
                            break;
                    }
                    break;

                case WHAT_BARCODE:
                    txtbarcode.setText(message.getData().getString("barcode"));
                    break;
            }
            return true;
        }
    }));
}
```

3 ACD System Service

Der ACD System Service bietet Zugriff auf spezielle Android Funktionen, die ansonsten nur mit Root-Rechten ausgeführt werden können. Der ACD System Service läuft als Service im Hintergrund und kann von anderen Apps mithilfe von Messages angesprochen werden.

3.1 Die API des Dienstes

Das Konzept der Bindung an den Dienst wird als bekannt vorausgesetzt und soll nur kurz skizziert werden.

1. Die App ruft an geeigneter Stelle bindService( ) auf.
2. Im Ereignis onServiceConnected( ) kann die App ein Messenger Objekt erstellen, das sie zum Senden von Nachrichten an den Dienst verwendet.
3. Ein weiterer Messenger empfängt vom Dienst kommende Nachrichten.
4. An geeigneter Stelle ruft die App unbindService( ) auf und gibt den Dienst wieder frei.

Bei Verwendung mehrerer Activities in der App ist es hilfreich, den Dienst global, also für die gesamte Lebenszeit der App gebunden zu halten, statt ihn in jeder Activity separat zu binden.

Der Dienst wird durch Messages gesteuert. Ein Message Objekt enthält die drei Zahlenfelder „what“, „arg1“ und „arg2“. „what“ enthält den Nachrichtentyp, „arg1“ und „arg2“ enthalten mögliche weitere Details zur Nachricht. Um größere Datenmengen mit einem Message Objekt zu übertragen, können Bundle-Felder verwendet werden.

Nachricht	Beschreibung	Nachricht Nr. (what-msg)
ERROR	Anfrage konnte nicht ausgeführt werden.	0
OK	Wird vom Dienst als Antwort auf eine Anfrage versendet.	1
HELLO	Der Dienst antwortet auf HELLO mit OK. Darüber hinaus findet kein Login oder sonstiger Informationsaustausch statt.	2
REBOOT	Diese Nachricht startet das Gerät sofort, ohne Hinweis, neu.	3
SET_TIME	Mit dieser Nachricht kann die Uhrzeit des Geräts eingestellt werden.	4
SHUTDOWN	Diese Nachricht fährt das Gerät sofort, ohne Hinweis, herunter.	8
OTA_UPDATE	Mit dieser Nachricht kann ein OTA Update (over-the-air), ein Firmwareupdate oder ein Modulupdate angestoßen werden.	9

Tabelle 5: API SystemService

3.2 Nachrichtentypen

Für „what“ gibt es die folgenden Möglichkeiten:

ERROR

Wird von dem Dienst als Antwort auf eine Anfrage versandt, die nicht erfolgreich ausgeführt werden konnte.

<table>
<tr><th>what</th><th>arg1</th><th colspan="2">arg2</th><th>Bundle</th></tr>
<tr><td rowspan="3">0</td><td rowspan="3">Rückgabe der angefragten, fehlgeschlagenen what
Bsp: ‘3‘ für REBOOT</td><td>Wenn „arg1“ 1 - 8 ist:</td><td>Keine Funktion</td><td>Nicht enthalten</td></tr>
<tr><td rowspan="2">Wenn „arg1“ 9 ist:</td><td>0: Update läuft nicht mehr</td><td rowspan="2">Enthält Informationen zum Ergebnis des Updatevorgangs</td></tr>
<tr><td>1: Update wird noch ausgeführt</td></tr>
</table>

Wenn „arg1“ 9 ist (OTA_UPDATE), können aus dem Bundle folgende Felder extrahiert werden:

- RESULT (String): Meldung die Informationen zum Ergebnis des Updates enthält.
- SUCCESS (Boolean): Gibt an ob das Update erfolgreich war oder nicht.

OK

Wird vom Dienst als Antwort auf eine Anfrage der Clientapp versandt.

what	arg1	arg2	Bundle
1	Rückgabe der angefragten what Bsp: ‘2‘ für HELLO	Keine Funktion	Enthält Infos zum System Service

Aus dem Bundle können folgende Felder extrahiert werden:

- NAME (String)
- VERSION_NAME (String)
- VERSION_CODE (INT)

HELLO

Der Dienst antwortet auf HELLO mit OK. Darüber hinaus findet kein Login oder sonstiger Informationsaustausch statt.

what	arg1	arg2
2	Keine Funktion	Keine Funktion

REBOOT

Diese Nachricht startet das Gerät sofort ohne Hinweis neu.

what	arg1	arg2
3	Keine Funktion	Keine Funktion

SET_TIME

Mit dieser Nachricht kann die Uhrzeit des Geräts eingestellt werden. Im Bundle-Feld wird dabei die Uhrzeit als Millisekunden seit der UNIX-Epoche übergeben.

what	arg1	arg2	Bundle
4	Keine Funktion	Keine Funktion	Enthält Uhrzeit

In das Feld “MILLIS“ muss die Uhrzeit als Long übergeben werden.

SHUTDOWN

Diese Nachricht fährt das Gerät sofort, ohne Hinweis, herunter.

what	arg1	arg2
8	Keine Funktion	Keine Funktion

OTA_UPDATE

Mit dieser Nachricht kann ein OTA Update (over-the-air), Firmwareupdate oder Modulupdate angestoßen werden.
Sobald die Nachricht erhalten wurde, wird das Paket überprüft und wenn dies erfolgreich war sofort installiert. Dieser Vorgang kann nicht mehr abgebrochen werden. Auf allen Geräten können OTA Pakete, auf Geräten mit Modulsteckplatz auch Firmware- und Modulupdatepakete installiert werden. Alle Pakete werden von ACD zur Verfügung gestellt.

Achtung:
Der ACD System Service überprüft im Moment nicht den Akkustand des Geräts, dieser sollte mindestens 30 % betragen und von der Clientapp geprüft werden.

what	arg1	arg2	Bundle
9	0 = Keine Benachrichtigung	0 = Updatepaket behalten	URI die den Speicherpfad des Updatepaketes enthält
	1 = Benachrichtigung anzeigen	1 = Updatepaket löschen	

Im Bundle muss die URI in das Feld „PATH“ als String eingetragen werden. Eine Antwort erfolgt nur im Fehlerfall oder bei erfolgreichem Modulupdate bei Geräten mit Modulsteckplatz.

4 Akkulaufzeit

Die Laufzeit des Akkus hängt von diversen Faktoren ab, die durch den Programmierer einer Anwendung beeinflusst werden können.
Die Möglichkeiten zum Energiesparen hängen vom Nutzungsverhalten und Einsatzgebietes des Geräts sowie von der entsprechenden Anwendung ab. Dieses Kapitel gibt Denkanstöße und Anregungen, um möglichst sparsame und trotzdem komfortabel bedienbare Anwendungen zu erstellen.

4.1 Bedienpausen erkennen

Wenn das Gerät während des Arbeitsprozesses nur in bestimmten Schritten bedient wird, kann die Anwendung während der Bedienpausen das Display dimmen oder das WLAN abstellen.

Über den den Lagesensor kann je nach Anwendungsfall eine Pause erkannt werden, sofern die jeweilige Haltung keine typische Bedienposition ist.
Beispiele

- Das Gerät liegt für längere Zeit flach auf dem Tisch, evtl. mit dem Display nach unten.
- Das Gerät hängt kopfüber an einer Umhängeschlaufe, Wiederaufnehmen beendet die Bedienpause.
- Das Gerät befindet sich in einer Halterung in der es nicht bedient wird, bzw. gedimmtes Display ermöglicht eine denkbare Bedienung (Display wird dann trotzdem wieder hell).

4.2 Datenübertragung

Ein aktives WLAN Modul hat einen deutlichen Anteil an der Laufzeit des Geräts. Sofern die Anwendung nur zu bestimmten Zeitpunkten Daten überträgt, kann sie WLAN während eines Großteils der Laufzeit deaktivieren.

- WLAN abstellen, wenn nicht erforderlich. Rechtzeitig wieder einschalten, um Zeit zur Einbuchung ins Netz zu lassen.
- Die Anwendung wenn möglich so konzipieren, dass nur zu bestimmten Zeitpunkten Daten übertragen werden müssen.
- WLAN-Ausleuchtung berücksichtigen: In einer Umgebung mit schlechtem Empfang (etwa in Regalreihen oder Außenbereichen) auf WLAN verzichten, stattdessen ausgewählte Orte für die Datenübertragung vorsehen.

4.3 Auslastung der CPU

Die CPU nimmt, je nach Prozessorlast, verschiedene Leistungsstufen an. Eine Anwendung kann dies kontrollieren, indem sie aufwendige Abläufe nur zu den nötigen Zeitpunkten durchführt.

- Polling von Daten vermeiden, bzw. nur in genügend großen Intervallen pollen (Offline bevorzugen).
- Echtzeitvorgänge (z. B. Aktualisierung der Anzeige) vermeiden, bzw. auf bestimmte Momente einschränken in denen der Nutzer sie tatsächlich beachten kann (Bedienpausen identifizieren).

4.4 Display

Das Display hat ebenfalls eine bedeutende Leistungsaufnahme. Es sollte nicht unnötig hell eingestellt werden.

- Helligkeit der Umgebung anpassen, nicht zu hell einstellen.
- Im oberen Bereich ist die sichtbare Helligkeitszunahme minimal, der Stromverbrauch steigt aber weiterhin exponentiell an.
- Display in Bedienpausen dimmen.

5 NFC/RFID API

Um NFC Tags oder RFID Tags in einer eigenen App zu lesen oder zu schreiben, kann die offizielle Android API verwendet werden: https://developer.android.com/guide/topics/connectivity/nfc
Optional kann auch die TapLinx API von NXP verwendet werden, die das Auslesen und Schreiben von Tags vereinfacht und eine Vielzahl von Funktionen bereit stellt. Die Verwendung der API erfordert allerdings eine Registrierung bei NXP. Alle Infos zu der TapLinx API: https://www.mifare.net/en/products/tools/taplinx/

6 Vibrationsmotor

Um den Vibrationsmotor im **M2Smart®SE 10 Zoll** in einer eigenen App anzusprechen, kann die offizielle Android API verwendet werden: https://developer.android.com/reference/android/os/Vibrator

7 Action Secret Code

Android bietet einen Mechanismus, um versteckte Funktionen in Anwendungen durch sogenannte Secret Codes auszulösen. Davon wird sowohl im Android Betriebssystem, als auch von ACD spezifischen Diensten und Anwendungen Gebrauch gemacht. Auch Kunden können mit ihren eigenen Anwendungen neue Secret Codes einführen, die eine Funktion in einer App auslösen können. Um Überschneidungen zwischen Betriebssystem, ACD spezifischen Diensten/Anwendungen und Kundenanwendungen zu vermeiden, wird empfohlen kundenspezifische Secret Codes mit einem Präfix im Bereich von **900 - 999** zu verwenden.

Siehe auch: ACTION_SECRET_CODE

8 Verzeichnisse

8.1 Tabellenverzeichnis

8.2 Stichwortverzeichnis

9 Overview

9.1 Goal and scope of this manual

This document is a manual for programmers, system administrators, and configurators of ACD devices with Android.
It should offer insights into the functional scope and possibilities of the devices in question and serve as a starting point for questions relating to programming, configuration, and administration.
Due to the complexity of modern operating systems and their programming, this document does not go into depth for all components. For some topics therefore, there is individual documentation, to which reference is made in the text. It can be requested from ACD if necessary.

9.2 ACD devices with Android

The following table provides an overview of which ACD devices can be operated with Android.

Device	Product version ... and higher	Operating system	API level
M270	1.10	Industrial+ for Android	23
M270SE	2.00.1	Industrial+ for Android	30
M260TE	1.8	Industrial+ for Android	23
M266SE	1.7	Industrial+ for Android	23
M2Smart®	1.6	Industrial+ for Android	23
M2Smart®SE	2.01	Industrial+ for Android	30
M2Smart®SE 10 inch	1.20	Industrial+ for Android	28

Table 1: ACD devices with Android

10 ACD ScanService

The ACD ScanService offers access to the scanner in order to configure it and forward scanned bar codes. It also intercepts the scanner keys and can trigger the scanner. Scanned bar codes are received by the ACD ScanService and forwarded depending on configuration.

If the default behavior is not changed when reading a bar code, then the read bar code is reported to the system as a keyboard event.
This can be used in order to scan data into generic applications that cannot be adapted for communication with the ACD ScanService. In this case, the app notices no difference between a scan and the use of a keyboard.

For self-developed applications, it is recommended that you bind these to the ACD ScanService. This way, Android will recognize that the ACD ScanService is required for the application and keep it active.

10.1 Reception of bar codes

There are several possibilities for delivering a bar code from the service to the target application. These can be set in the default settings or set by an app via the API of the service. See SET_TARGET.

10.1.1 Keyboard looping in

Here, the service loops the bar code into the Android keyboard system. This can sooner be compared with an external keyboard, however not with the concept of the virtual keyboard.
This mechanism can enter data in each active input field. Thus, for example, a WLAN password can be scanned into the input field instead of typed.

However, the looping in has its limits. The underlying Android system program can only handle a basic character set, so that in addition to the letters A - z (upper- and lower-case) and the numbers 0 - 9, hardly any other characters are accepted. The behavior depends on existing keyboard layouts in the system and can thus not be guaranteed for all conceivable bar code characters.

10.1.2 Sending a broadcast

The service sends a broadcast intent with the name "de.acdgruppe.scanservice.SCANBROADCAST".
If the service version is older than 1.22, the broadcast intent name is "de.acdgruppe.scanservice:SCANBROADCAST". In the "extra" fields SCANNERDATA_BARCODE and SCANNERDATA_BARCODETYPE, this intent includes the bar code and its type as string. The type of the barcode is transferred as an integer in the SCANNERDATA_TYPECODE field.

An app that registers itself as BroadcastReceiver can receive the bar code this way and process according it to its needs. In particular, for fields without optional keyboard input, the screen keyboard can be omitted, which keeps the input window clearer. This method is suitable for application cases where the scanner is configured permanently and can thus provide the app with an uncomplicated reception mechanism. However, it must be noted that any application can receive the broadcast.

10.1.3 Use of the messenger

An application that binds itself to the service can create a "messenger" via the binding and use it to communicate with the service. If the service is configured for the sending of bar codes via messenger, the data is sent directly to the application that was the last one to register with the service. The binding also ensures that Android does not exit the service during the app's runtime.

Thanks to the configuration possibility via the messenger, the application can adjust the scanner service to the appropriate process step. For process steps without scanner use, it can deactivate the automatic triggering of the scanner and use the scanner keys for other purposes.[2]
The configuration of the permitted bar code types is helpful if the process steps use different bar code types and it can prevent mistaken scans.

10.1.4 Using the Messenger for the M2Longrange Module

As of version 1.40 of the ACD ScanService app, it provides a second service for connecting the M2Longrange module to the M2Smart®SE. The name of the service is „`de.acdgruppe.scanservicelr.AcdScannerServiceLR`". All the information in the previous paragraph applies equally.

[2] The keys are reported to the activity by the usual events (onKeyDown etc.), regardless of whether or not the scan service triggers the scanner.

10.1.5 Examples of Connecting

Hint: a full communication example can be found in chapter 8.4

```
Intent scanServiceIntent = new Intent();
ComponentName scanServiceName =
  new ComponentName("de.acdgruppe.scanservice", "de.acdgruppe.scanservice.AcdScannerService");
scanServiceIntent.setComponent(scanServiceName);
if (getApplicationContext().bindService(scanServiceIntent,
  scannerConnection, Context.BIND_AUTO_CREATE)) { .... }

Intent scanServiceM2LRIntent = new Intent();
ComponentName scanServiceM2LRName =
  new ComponentName("de.acdgruppe.scanservice", "de.acdgruppe.scanservicelr.AcdScannerServiceLR");
scanServiceM2LRIntent.setComponent(scanServiceM2LRName);
if (getApplicationContext().bindService(scanServiceM2LRIntent,
  scannerM2LRConnection, Context.BIND_AUTO_CREATE)) { .... }
```

10.2 The API of the service

It is assumed that you are familiar with the concept of binding to the service, so that will be sketched only briefly here.

1. The app calls bindService() in the appropriate place.
2. In the event onServiceConnected() the app can create a messenger object that it uses to send messages to the service. The app configures the service as it needs to.
3. Another messenger receives incoming messages from the service. The app sends bar codes that are received via the messenger to the associated program code.
4. In an appropriate place, the app calls unbindService() and releases the service again.

With use of several activities in the app, it is helpful to keep the service bound globally, that is, for the entire lifetime of the app, instead of binding it separately to each activity.

The service is controlled by messages. A message object includes the three numeric fields what, arg1 and arg2. what contains the message type, arg1 and arg2 contain possible other details about the message.

Message	Description	Message No (what-msg)
OK	Sent by the service as response to a request from the client app	1
HELLO	The service responds to HELLO with OK. Beyond that there is no login.	2
CONFIGURE	Message makes settings to the service	3
SCAN	Turn scan beam on and off from the application	4
SET_TARGET	Selection of barcode transfer	5
BARCODE	Message contains barcode information	6
KEY_ENABLE	Trigger scan when the scanner keys are pressed	7
AIM	Turn aiming beam on or off	8

Table 2: API ScanService

10.2.1 Message types

For what, there are the following possibilities:

OK

Sent by the service as a response to a request from the client app.

what	arg1	arg2
1	Return of the requested what Example: '2' for HELLO	0 = Not able to process 1 = Response is OK

HELLO

The service responds to HELLO with OK. The answer to HELLO contains data about the scanner. No login or other information exchange takes place.

what	arg1	arg2
2	No function	No function

The OK message as response to HELLO contains information about the scanner in the bundle field of the message.
The following entries are present:

- ENGINECODE: The code of the scanengine.
- SWREVISION: The software revision of the scanner.
- NAME: Name of the scanner (*).
- MODULENR: The number of the scan module (*).

The fields marked with (*) are determined from the enginecode and are empty if a scanner unknown to the scanner service is installed.

CONFIGURE

This message makes settings for the service. Currently this has no function yet.

what	arg1	arg2
3	No function	No function

SCAN

The client can trigger the scan stream or end it with this message. Note that the scanner switches off automatically after a certain time if there is no successful scan.

what	arg1	arg2
4	0 = End scan beam 1 = Trigger scan beam	No function

SET_TARGET

The client uses this message to select how it would like to receive a scanned bar code. The options of the message will be explained in the next section.

what	arg1	arg2
5	Destination	No function

The message SET_TARGET controls the behavior of the service if it has successfully decoded a bar code. The last setting made applies regardless of the currently active app. Depending on the area of application and the operator's preference, the app should reset the service to the desired default setting at least before unbinding (unbind), for example the keyboard looping.

The field arg1 can take on the following values for the message SET_TARGET:

- NONE
  The services does not forward a scanned bar code. This value exists for the sake of completeness. A process step without a scan procedure should deactivate the scanner keys.
  arg1 = 0

- INJECT
  The service loops the bar code into the keyboard system. This is the default setting for the service. After the app terminates (or a pause), this line should be restored. See: „0

  Keyboard looping in"
  arg1 = 1

- BROADCAST_INTENT
  The bar code is forwarded via an intent sent by broadcast. See: „10.1.2 Sending a broadcast“
  arg1 = 2

- MESSENGER
  The service sends the bar code to the messenger object that has specified this target. The sent bundle contains the fields:
  - "barcode" of type String: Contains the barcode
  - "type" of type String: Contains the barcode type
  - "typecode" of type int: Contains the type code

  See: „10.1.3 Use of the messenger2.1.3“
  arg1 = 3

BARCODE

By means of this message, the service sends a scanned bar code to the client, provided the client has set this send type. The bar code is sent to the messenger object by which it selected the send type. It is only sent to the last messenger object to be set.

what	Return value
6	Contains bundle with barcode information

KEY_ENABLE

This way, the client specifies whether or not the service triggers the operation of the scanner automatically with operation of the scanner keys.

what	arg1	arg2
7	0 = Do not trigger scan 1 = Trigger	No function

AIM

The client determines whether or not the aiming beam of the scanner will be triggered.
This function is available with the 2D longrange scanner.

what	arg1	arg2
8	0 = Aiming beam off 1 = Aiming beam on	No function

10.2.2 Barcode types

The ScanService returns a type code and the barcode type with every scanned barcode.

Barcode type	Typecode
Code 39	01
Code 39 Full ASCII	13
Trioptic Code 39	15
Code 32	20
Code 93	07
Codabar	02
NW 7	18
Code 128	03
ISBT 128	19
ISBT 128 Concatenation	21
Discrete 2 of 5	04
IATA 2 of 5	05
Interleaved 2 of 5	06
Matrix 2 of 5	71
Chinese 2 of 5	72
Korean 3 of 5	73
UPC A	08
UPC A with 2 Supplementals	48
UPC A with 5 Supplementals	88
UPC E	09
UPC E with 2 Supplementals	49
UPC E with 5 Supplementals	89
UPC E1	10
UPC E1 with 2 Supplementals	50
UPC E1 with 5 Supplementals	90
EAN 8	0A
EAN 8 with 2 Supplementals	4A
EAN 8 with 5 Supplementals	8A
EAN 13	0B
EAN 13 with 2 Supplementals	4B
EAN 13 with 5 Supplementals	8B

Barcode type	Typecode
4State US	34
4State US4	35
Scanlet Webcode	37
Cue CAT Code	38
French Lottery	2F
GS1 128	0F
GS1 DataBar 14	30
GS1 DataBar Limited	31
GS1 DataBar Expanded	32
GS1 DataBar Expanded Coupon	B4
GS1 Data Matrix	C1
PDF 417	11
Macro PDF 417	28
Micro PDF	1A
Micro PDF CCA	1D
Macro Micro PDF	9A
Data Matrix	1B
GS1 Data Matrix	C1
QR Code	1C
Micro QR Code	2C
Macro QR Code	29
Maxicode	25
Aztec	2D
Aztec Rune Code	2E
Han Xin	B7
Signature	69
OCRB	A0
TLC 39	5A
Composite CC-A GS1-128	51
Composite CC-A EAN-13	52
Composite CC-A EAN-8	53

Barcode type	Typecode
Bookland EAN	16
Code 11	0C
Code 49	0D
MSI	0E
Code 16K	12
UPCD	14
Coupon Code	17
Postnet US	1E
Planet US	1F
Postal Japan	22
Postal Australia	23
Postal Dutch	24
Postbar CA	26
Postal UK	27

Barcode type	Typecode
Composite CC-A GS1-DB Expanded	54
Composite CC-A GS1-DB Limited	55
Composite CC-A GS1-DataBar 14	56
Composite CC-A UPC-A	57
Composite CC-A UPC-E	58
Composite CC-C GS1-128	59
Composite CC-B GS1-128	61
Composite CC-B EAN-13	62
Composite CC-B EAN-8	63
Composite CC-B GS1-DB Expanded	64
Composite CC-B GS1-DB Limited	65
Composite CC-B GS1-DataBar 14	66
Composite CC-B UPC-A	67
Composite CC-B UPC-E	68

Table 3: Typecode Scanner without 2D-LR

If a 2D Longrangescanner is installed, other type codes are reported as well.

Barcode type	Typecode
Code 39	105
Codabar	101
Codablock	102
Australian Post	97
Canada Post	100
Dutch Post	110
Japan Post	120
Sweden Post	73
Aztec	98
BP0	99
Code11	104
Code 128/GS1-128	107
DataMatrix	109
EAN/UPC	111
Infomail	118

Barcode type	Typecode
Matrix 2 of 5	121
MaxiCode	122
PDF417/MicroPDF417	66
Planet	68
Plessey Code	69
Postnet	70
QR Code	71
Standard 2 of 5	72
Telepen	74
TLC39	75
Interleaved 2 of 5	119
Code 93	106
MSI Code	65
GS1 DataBar	117
GS1 Composite	115

Table 4: Typecode Scanner 2D-LR

10.3 Scanner configuration

The ACD ScanConfig app is pre-installed on the ACD devices and will make scanner settings for this. Among other things, the app can be used to make the following settings:

- Configuration of scanner parameters (permitted bar codes, etc.)
- Configuration of the scanner data (attachment of prefix, suffix, etc.)
- Notification tone when scanning a bar code
- Configuration of the scanner keys
- Etc.

The configuration data of the ACD ScanConfig will be stored on the device under */sdcard/ACD/scanner*.

The ACD ScanService uses the settings that are made in the ACD ScanConfig in order to configure the scanner installed in the device.

10.4 Communication class example for the ACD ScanService

```
public class MainActivity extends AppCompatActivity {

    private Messenger _serviceMessenger = null;
    private boolean _boundToService = false;
    private TextView txtbarcode;

    public static final int WHAT_OK = 1;
    public static final int WHAT_HELLO = 2;
    public static final int WHAT_SCAN = 4;
    public static final int WHAT_SET_TARGET = 5;
    public static final int WHAT_BARCODE = 6;
    public static final int WHAT_KEY_ENABLE = 7;
    public static final int TARGET_MESSAGE = 3;


    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        txtbarcode = findViewById(R.id.txtbarcode);
        bindToService(this);
    }

    public void bindToService(Context context) {
        Intent intent = new Intent();
        intent.setComponent(new ComponentName("de.acdgruppe.scanservice", "de.acdgruppe.scanservice.AcdScannerService"));
        // for Longrange use: new ComponentName("de.acdgruppe.scanservicelr", "de.acdgruppe.scanservicelr.AcdScannerServiceLR")

        try {
            // Use applicationContext because the activity can change during rotation and it is not
            // obvious what happens to the service if bound to the activity.
            if (!context.bindService(intent, _connection, Context.BIND_AUTO_CREATE))
                return;
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private ServiceConnection _connection = new ServiceConnection() {

        public void onServiceConnected(ComponentName className, IBinder service) {
            // This is called when the connection with the service has been established, giving us the
            // service object we can use to interact with the service.
            _serviceMessenger = new Messenger(service);

            // Send a hello to the service. Next steps follow when the response arrives.
            _boundToService = true;
            sendMessage(WHAT_HELLO, 0, 0);
        }

        public void onServiceDisconnected(ComponentName className) {
            // This is called when the connection with the service has been
            // unexpectedly disconnected, its process crashed.
            _serviceMessenger = null;
            _boundToService = false;
        }
    };

    public void sendMessage(int what, int arg1, int arg2) {
        if (_serviceMessenger == null) {
            // The messenger should never be null, there's one not-changing instance during the app's life.
            String text = String.format("Service messenger is null! (Message=%d)", what);
            return;
        }

        if (!_boundToService) {

            return;
        }

        try {
            Message msg = Message.obtain(null, what, arg1, arg2);
            msg.replyTo = _receiverMessenger;
            _serviceMessenger.send(msg);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }

    private Messenger _receiverMessenger = new Messenger(new Handler(new Handler.Callback() {
        @Override
        public boolean handleMessage(Message message) {
            switch (message.what) {
                case WHAT_OK:
                    // arg1 contains the reason for this OK message
                    switch (message.arg1) {

                        case WHAT_HELLO:
                            // We are connected => set the service to send barcode messages via messenger
                            sendMessage(WHAT_SET_TARGET, TARGET_MESSAGE, 0);
                            // Enable the scankey trigger
                            sendMessage(WHAT_KEY_ENABLE, 1, 0);
                            break;
                    }
                    break;

                case WHAT_BARCODE:
                    txtbarcode.setText(message.getData().getString("barcode"));
                    break;
            }
            return true;
        }
    }));
}
```

11 ACD System Service

The ACD System Service provides access to special Android features that can otherwise only be run with root rights. The ACD System Service runs as a background service and can be addressed by other apps using messages.

11.1 The API of the service

Understanding of attachment to the service is assumed and will only be briefly outlined.

1. The app calls bindService ( ) at a suitable point.
2. In the onServiceConnected ( ) event, the app can create a messenger object that it uses to send messages to the service.
3. Another messenger receives messages coming from the service.
4. At an appropriate point, the app calls unbindService ( ) and releases the service again.

When using multiple activities in the app, it is helpful to keep the service global, i.e. for the lifetime of the app rather than binding it separately in each activity.

The service is controlled by messages. A message object contains the three number fields "what", "arg1" and "arg2". "What" contains the message type, "arg1" and "arg2" contain possible further details about the message. Bundle fields can be used to transfer larger amounts of data with a message object.

Message	Description	Message No (what-msg)
ERROR	Unable to perform request.	0
OK	Sent by the service as response to a request.	1
HELLO	The service responds to HELLO with OK. No login or other information exchange takes place.	2
REBOOT	This message restarts the device immediately and without notice.	3
SET_TIME	This message can be used to set the time of the device.	4
SHUTDOWN	This message shuts down the device immediately and without notice.	8
OTA_UPDATE	This message can trigger an OTA update (over-the-air update), a firmware update or a module update.	9

Table 5: API SystemService

11.2 Message types

For "what" there are the following options:

ERROR

Dispatched by the service in response to a request that has failed.

what	arg1	arg2		Bundle
0	Return of the requested, failed what Example: ‘3‘ for REBOOT	If „arg1“ is 1 - 8:	No function	Not included
		If „arg1“ is 9:	0: Update no longer running	Contains information about the result of the update process
			1: Update is still to be performed	

If "arg1" is 9 (OTA_UPDATE), the following fields can be extracted from the bundle:

- RESULT (string): Message containing information about the result of the update.
- SUCCESS (Boolean): Indicates whether or not the update was successful.

OK

The service sends the client app in response to a request.

what	arg1	arg2	Bundle
1	Return of the requested, what Example: ‘2‘ for HELLO	No function	Contains information about system service

The following fields can be extracted from the bundle:

- NAME (String)
- VERSION_NAME (String)
- VERSION_CODE (INT)

HELLO

The service responds to HELLO with OK. No login or other information exchange takes place.

what	arg1	arg2
2	No function	No function

REBOOT

This message restarts the device immediately and without notice.

what	arg1	arg2
3	No function	No function

SET_TIME

This message can be used to set the time of the device. In the bundle field, the time is passed as milliseconds since the UNIX epoch.

what	arg1	arg2	Bundle
4	No function	No function	Contains the time

The time must be transferred as Long to the “MILLIS“ field.

SHUTDOWN

This message shuts down the device immediately and without notice.

what	arg1	arg2
8	No function	No function

OTA_UPDATE

This message can trigger an OTA update (over-the-air update), a firmware update or a module update. Once the message has been received, the package is checked and if this was successful instantly installed. This process cannot be canceled**.** OTA packages can be installed on all devices, while firmware and module update packages can be installed only on devices with module sockets. All packages are made available by ACD.

Warning:
The ACD System Service does not currently check the device's battery level, which should be at least 30 % and checked by the client app.

what	arg1	arg2	Bundle
9	0 = No notification	0 = Updatepaket behalten	URI containing the path where the update file is saved
	1 = Show notification	1 = Updatepaket löschen	

The URI must be entered in the bundle as a String in the "PATH" field. A response is given only in case of error or after a successful module update for devices with module sockets.

12 Rechargeable battery runtime

The runtime of the rechargeable battery depends on various factors that can be influenced by the programmer of an application.
The possibilities for saving energy depend on usage behavior and the area of application of the device and on the corresponding app. This chapter provides food for thought and inspiration for creating applications that are as economical and yet comfortable as possible.

12.1 Detecting operating pauses

If the device is operated during the work process only in particular steps, the app can dim the display during operating pauses or switch off the WLAN.

Using the position sensor, depending on the application case, a pause can be detected if the position is not a typical operating position.
Examples

- The device lies flat on the table for a longer time, possibly with the display facing down.
- The device is hanging upside-down on a hanging loop; picking it up again ends the operating pause.
- The device is in a holder in which it is not operated or dimmed display enables conceivable operation (display then brightens nevertheless).

12.2 Data transfer

An active WLAN module occupies a significant share of the runtime of the device. Insofar as the application only transmits data at particular points in time, it can deactivate WLAN during a majority of the runtime.

- Switch off WLAN if not required. Switch on in timely fashion in order to leave time to log onto the network.
- If possible, design the application so that data must only be transferred at particular points in time.
- Consider WLAN illumination: In an environment with poor reception (for example in rows of shelves or outdoor areas), forego WLAN; instead, provide selected locations for the data transfer.

12.3 Utilization of the CPU

Depending on the processor load, the CPU assumes various power levels. An application can control this by executing time-consuming processes only at the necessary times.

- Avoid polling of data or only poll at sufficiently large intervals (prefer offline).
- Avoid realtime processes (e.g. Updating of the display) or limit to particular moments in which the user can actually heed them (identify operating pauses).

12.4 Display

The display also has significant power consumption. It should not be set unnecessarily bright.

- Adjust brightness to the environment, do not set too bright.
- In the upper area, the visible brightness increase is minimal, however the current consumption increases exponentially.
- Dim display in operating pauses.

13 NFC/RFID API

The official Android API can be used to read or write NFC tags or RFID tags in a separate app: https://developer.android.com/guide/topics/connectivity/nfc
It is also possible to use the TapLinx API of NXP, which makes it easier to read and write tags and provides numerous functions. However, registration with NXP is required to use the API. Complete information about the TapLinx API: https://www.mifare.net/en/products/tools/taplinx/

14 Vibration motor

To address the vibration motor in the **M2Smart®SE 10 Inch** in an own app, the official Android API can be used: https://developer.android.com/reference/android/os/Vibrator

15 Action Secret Code

Android provides a mechanism to trigger hidden functions in applications by so-called Secret Codes. This is used in the Android operating system as well as by ACD specific services and applications. Customers can also introduce new secret codes with their own applications that can trigger a function in an app. To avoid overlaps between operating system, ACD specific services/applications and customer applications, it is recommended to use customer specific Secret Codes with a prefix in the range of **900 - 999**.

See also: ACTION_SECRET_CODE

16 Directories

16.1 List of tables

16.2 List of keywords