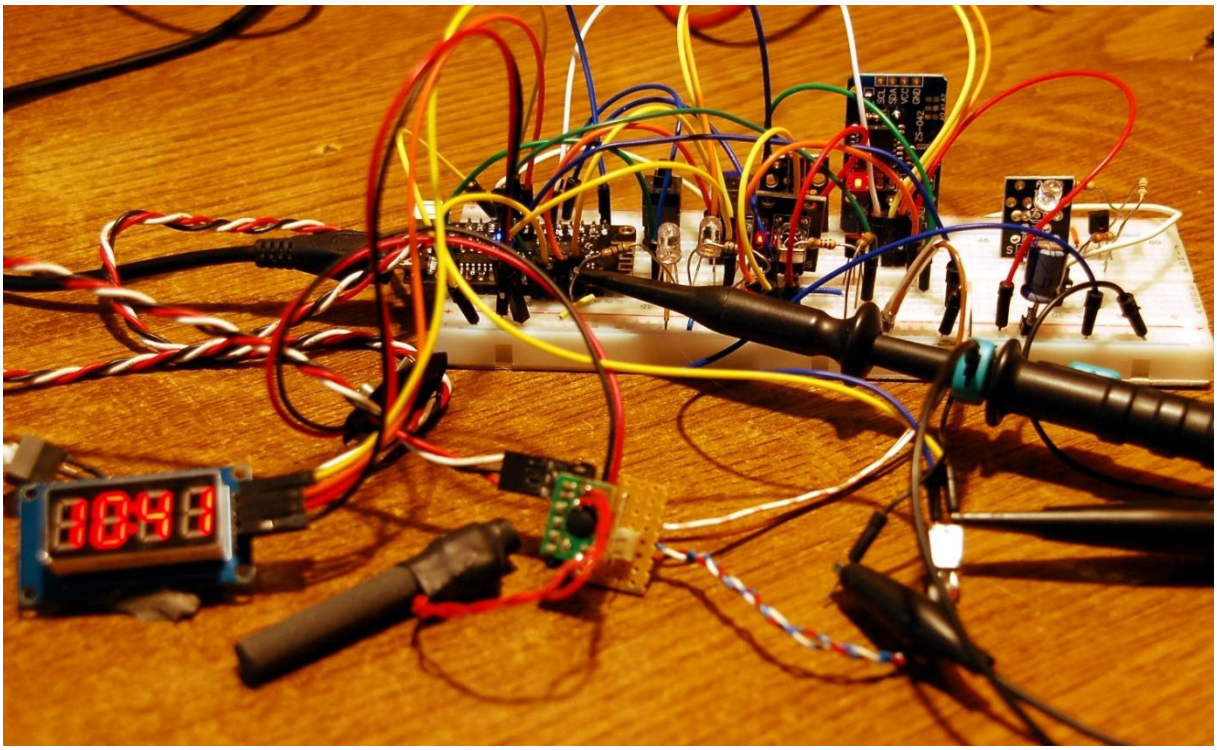


DCF77-Empfangsmodul am ESP32

Diesen Beitrag gibt es auch als PDF-Dokument zum Download.

Willkommen zu einer neuen Folge von

MicroPython auf dem ESP32 und ESP8266

heute

Der Funkwecker nimmt Kontakt mit dem Handy auf.

Unser RC5-Wecker besitzt selbst keine Bedienelemente, die braucht er auch nicht, denn wir werden ihn heute mit einer App ansprechen, die auf einem Android-Handy oder -Tablet läuft. Auf diese Weise kann das Layout des Bedienfeldes jederzeit verändert oder an persönliche Wünsche angepasst werden, ohne an einem Gehäuse rumpfriemeln zu müssen. Weil die Funktionen des Weckers software-defined sind, gilt das gleiche auch für die Schaltung selbst. Über den Parser lassen sich leicht weitere Befehle einbauen, die mittels hinzugefügter Routinen zusätzliche Schaltaktivitäten umsetzen können. Denkbar ist ebenfalls der Einsatz von Sensoren in unserer SDAC (Software-Defined-Alarm-Clock), deren Messdaten vom ESP32 ans Handy weitergegeben werden können. Hier eine kleine Zusammenfassung dessen, was bisher im Rahmen des Projekts behandelt wurde.

In der ersten Folge haben wir dem ESP32 beigebracht, eine RC5-IR-Fernsteuerung auszulesen. Im zweiten Beitrag, sendet der Controller selbst RC5-IR-Code . Dann lernte der ESP32, wie er eine PS/2-Tastatur, abfragen kann. Eine RTC, mit guter Ganggenauigkeit bekam der ESP32 in einer weiteren Folge spendiert, nachdem die

bordeigene Real Time Clock (RTC) alles andere als exakt läuft. Ein großes LED-Display kam in Folge 5. Dann sorgten wir mit einem DCF77-Modul für den Kontakt zum Zeitzeichensender der PTB (Physikalisch Technische Bundesanstalt). Zuletzt haben wir die einzelnen Programmteile zum Wecker zusammengefügt. Ein Parser kann die empfangenen Kommandos in Aktionen umsetzen.

Im Folgenden finden Sie die Auflistung der Hard- und Software. Die Beschreibung der einzelnen Teile finden Sie in den oben verlinkten PDF-Dateien.

Hardware

Da zur Hardware keine weiteren Teile hinzugekommen sind, habe ich die Liste von der letzten Folge übernommen.

1	ESP32 Dev Kit C unverlötet oder ESP32 NodeMCU Module WLAN WiFi Development Board oder NodeMCU-ESP-32S-Kit
1	KY-022 Set IR Empfänger
1	KY-005 IR Infrarot Sender Transceiver Modul
1	0,91 Zoll OLED I2C Display 128 x 32 Pixel
1	Breadboard Kit - 3x Jumper Wire m2m/f2m/f2f + 3er Set MB102 Breadbord kompatibel mit Arduino und Raspberry Pi - 1x Set
1	KY-004 Taster Modul
diverse	Jumper Wire Kabel 3 x 40 STK
1	Real Time Clock RTC DS3231 I2C Echtzeituhr
1	TM1637 4 Digit 7-Segment LED-Display Modul
1	KY-018 Foto LDR Widerstand Photo Resistor Sensor
1	DCF77-Empfänger-Modul
2	NPN-Transistor BC337 oder ähnlich
1	Widerstand 1,0 kΩ
1	Widerstand 10 kΩ
1	Widerstand 330 Ω
1	Widerstand 47Ω
1	Widerstand 560Ω
1	LED (Farbe nach Belieben)
1	Adapter PS/2 nach USB oder PS/2-Buchse
1	Logic Analyzer
1	PS/2 - Tastatur

Abbildung 1 zeigt die Schaltung mit allen Teilen.

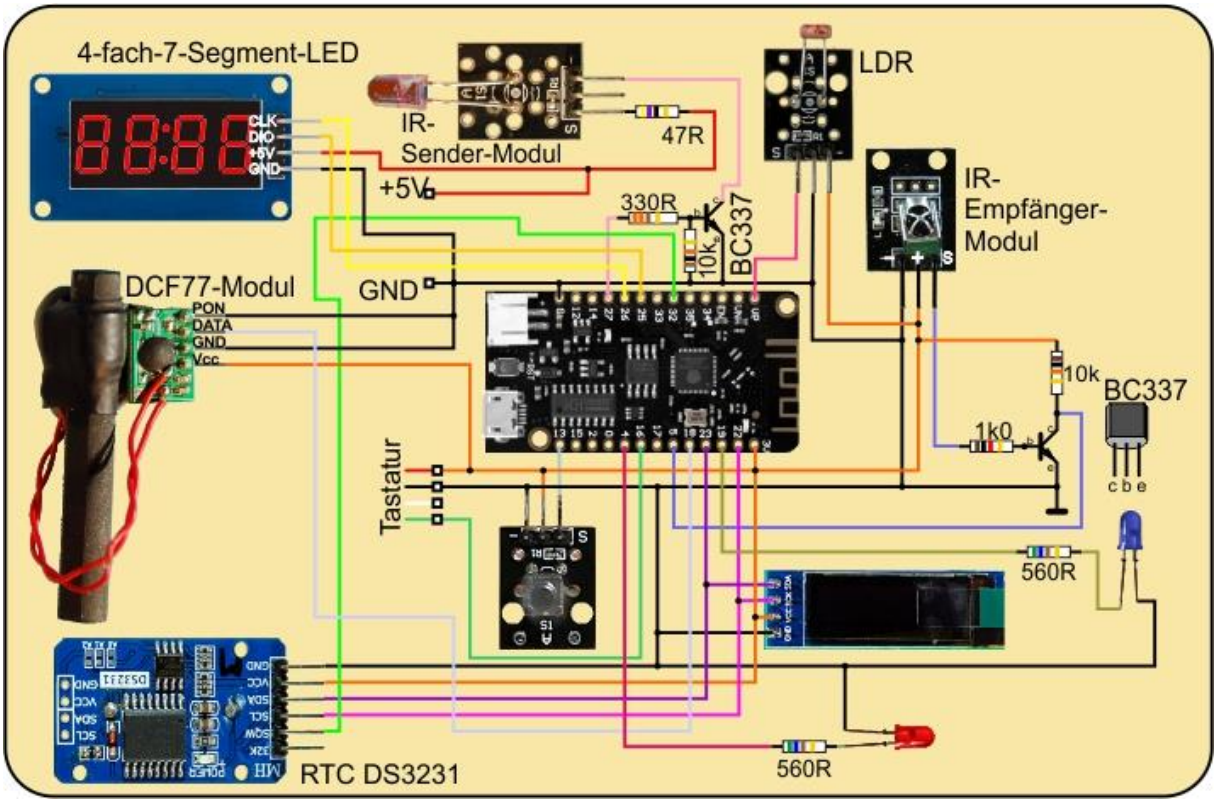


Abbildung 1: Alles zusammen = Funkuhr mit IR-RC-Ambitionen

Die Software

Fürs Flashen und die Programmierung des ESP32:

Thonny oder
µPyCraft
Betriebs-Software Logic 2 von SALEAE
packetsender.exe: Netzwerkterminal für TCP und UDP

Für das Handy

AI2-Companion aus dem Google Play Store.

Für die Handy-App

http://ai2.appinventor.mit.edu
https://ullisroboterseite.de/android-AI2-UDP/UrsAI2UDP.zip
mit app_Inventor_2_ger.pdf Anleitung für den AI2 und die Extension von Ullis Roboterseite

Verwendete Firmware für den ESP32:

v1.19.1 (2022-06-18) .bin

Verwendete Firmware für den ESP8266:

v1.19.1 (2022-06-18) .bin

Die MicroPython-Programme zum Projekt:

tm1637_4.py: API für die 4- und 6-stellige 7-Segment-Anzeige mit dem TM1637
ds3231.py: Treiber-Modul für das RTC-Modul
oled.py: OLED-API
ssd1306.py: OLED-Hardware-Treiber
buttons.py: Modul zur Tastenabfrage
dcf77.py: Treiber für das DCF77-Modul
ir_rx-small.zip: Paket zum IR-Empfangs-Modul
irsend.py: IR-Sende-Modul
timeout.py: Softwaretimer
sync_it.py: Programm zum Synchronisieren mit DCF77
sekundenalarm.py: Demoprogramm zum Alarm auslösen
lern.py: Auslesen von RC5-Codes

MicroPython - Sprache - Module und Programme

Zur Installation von Thonny finden Sie hier eine ausführliche Anleitung (english version). Darin gibt es auch eine Beschreibung, wie die Micropython-Firmware (Stand 18.06.2022) auf den ESP-Chip gebrannt wird.

MicroPython ist eine Interpretersprache. Der Hauptunterschied zur Arduino-IDE, wo Sie stets und ausschließlich ganze Programme flashen, ist der, dass Sie die MicroPython-Firmware nur einmal zu Beginn auf den ESP32 flashen müssen, damit der Controller MicroPython-Anweisungen versteht. Sie können dazu Thonny, µPyCraft oder esptool.py benutzen. Für Thonny habe ich den Vorgang hier beschrieben.

Sobald die Firmware geflasht ist, können Sie sich zwanglos mit Ihrem Controller im Zwiegespräch unterhalten, einzelne Befehle testen und sofort die Antwort sehen, ohne vorher ein ganzes Programm kompilieren und übertragen zu müssen. Genau das stört mich nämlich an der Arduino-IDE. Man spart einfach enorm Zeit, wenn man einfache Tests der Syntax und der Hardware bis hin zum Ausprobieren und Verfeinern von Funktionen und ganzen Programmteilen über die Kommandozeile vorab prüfen kann, bevor man ein Programm daraus strickt. Zu diesem Zweck erstelle ich auch gerne immer wieder kleine Testprogramme. Als eine Art Makro fassen sie wiederkehrende Befehle zusammen. Aus solchen Programmfragmenten entwickeln sich dann mitunter ganze Anwendungen.

Autostart

Soll das Programm autonom mit dem Einschalten des Controllers starten, kopieren Sie den Programmtext in eine neu angelegte Blankodatei. Speichern Sie diese Datei unter boot.py im Workspace ab und laden Sie sie zum ESP-Chip hoch. Beim nächsten Reset oder Einschalten startet das Programm automatisch.

Programme testen

Manuell werden Programme aus dem aktuellen Editorfenster in der Thonny-IDE über die Taste F5 gestartet. Das geht schneller als der Mausklick auf den Startbutton, oder über das Menü **Run**. Lediglich die im Programm verwendeten Module müssen sich im Flash des ESP32 befinden.

Zwischendurch doch mal wieder Arduino-IDE?

Sollten Sie den Controller später wieder zusammen mit der Arduino-IDE verwenden wollen, flashen Sie das Programm einfach in gewohnter Weise. Allerdings hat der ESP32/ESP8266 dann vergessen, dass er jemals MicroPython gesprochen hat. Umgekehrt kann jeder Espressif-Chip, der ein kompiliertes Programm aus der Arduino-IDE oder die AT-Firmware oder LUA oder … enthält, problemlos mit der MicroPython-Firmware versehen werden. Der Vorgang ist immer so, wie hier beschrieben.

Wir bauen die App

Der MIT App-Inventor 2

AI2 ist eine kostenlose Anwendung des MIT (Massachusetts Institute of Technology), die im Browser läuft. Das Wort bauen beschreibt den Vorgang der App-Entwicklung sehr treffend. Nachdem die grafische Oberfläche der App im AI2-**Designer** erstellt ist, geht es im Fenster **Blocks** um das Aufeinanderstapeln von Programmbausteinen, den Blocks.

An dieser Stelle sollte auf dem Handy bereits der AI2-Companion laufen, damit Sie alles, was im App-Inventor passiert auf dem Handy live mitverfolgen können. Starten Sie die App und klicken Sie in der Menüleiste in Inventor-Fenster auf **Connect – AI Companion**. Über den QR-Code verbindet sich das Handy mit dem PC. Dazu ist ein lokales WLAN erforderlich.

Starten Sie jetzt einen Browser, Firefox, Chrome, Opera… Dort stellen Sie die App-Oberfläche mit den Werkzeugen aus der linken Spalte, der **Palette**, zusammen. Die Elemente werden einfach in den **Viewer**, das mittlere Fenster, gezogen und dort an der Zielposition abgesetzt. In **Components** erscheinen dann die vorläufigen Bezeichner der Objekte, die Sie am besten gleich mit "sprechenden" Namen ersetzen, das geht über den Button **Rename**. Ich habe mir dafür ein System ausgedacht, das sowohl den Typ als auch die Aufgabe der Objekte verrät. Das erleichtert später beim Zusammenbau des Programms das Auffinden und Zuordnen. Drei Kleinbuchstaben kennzeichnen den Typ. Der erste Buchstabe des darauffolgenden Namens wird großgeschrieben.

In der **Properties**-Spalte können die Eigenschaften der Objekte eingestellt werden. Da ich schon öfter Apps für meine Projekte erstellt habe, und nicht jedes Mal knapp 20 Seiten Basisanleitung in den Post mit einbauen möchte, habe ich das alles in das Dokument mit app_Inventor_2_ger.pdf hineingepackt. Dort erfahren Sie, wie sie mit AI2 arbeiten, woher Sie die App AI2-Companion für das Handy und die UDP-Erweiterung von Ullis Roboterpage bekommen. Wenn Sie bisher noch nichts mit dem AI2 gebastelt haben, lege ich Ihnen dringend nahe, diese Anleitung durchzulesen.

Die grafische Oberfläche

Abbildung 2: App-Oberfläche

Wir gehen jetzt die einzelnen Elemente des Designs durch und beginnen mit dem **Screen1**. Das ist der Container, in dem alle weiteren Zutaten landen werden. Eigenschaften, Properties, die nicht verändert werden führe ich in der Beschreibung nicht auf. Die Hintergrundfarbe ändern wir schon mal. Nach einem Klick auf das Farbenfeld bei **BackgroundColor** wählen wir **Custom** und stellen die Farbe ein.

Abbildung 3: Hintergrundfarbe ändern

Abbildung 4: Hintergrundfarbe einstellen

Die Ausrichtung von Elementen auf dem Screen regeln diese beiden Felder.

Abbildung 5: Ausrichtung von Elementen

Der Inhalt des Screens wird recht umfangreich. Damit wir an alle Elemente herankommen, machen wir den Schirm **Scrollable**.

Abbildung 6: Rollen erlauben

Der Text in **Title** wird als Anwendungsname angezeigt, und unter **RC5_Control** wird die App in AI2 gespeichert.

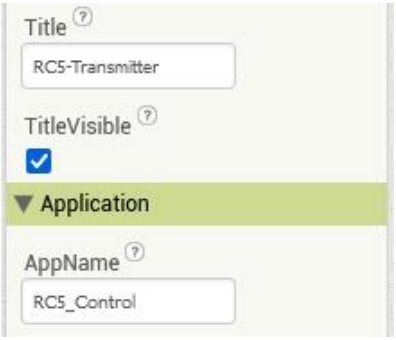


Abbildung 7: Anwendungsname

Um Elemente gruppieren zu können, verwenden wir sogenannte Arrangements, die wir im Ordner Layout finden.

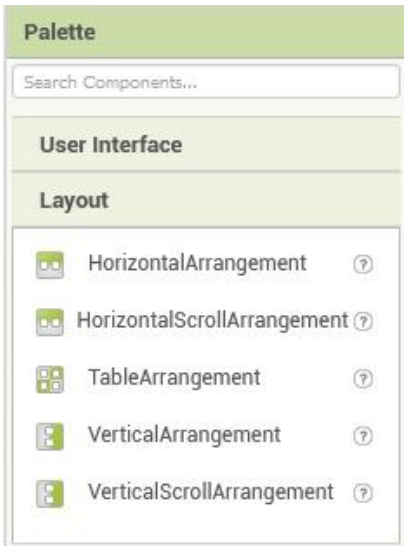


Abbildung 8: Sub-Container

Wir ziehen ein **HorizontalArrangement** in den Viewer, taufen es **harReceiving** und machen es unsichtbar. Damit es aber im Viewer angezeigt wird, setzen wir dort bei **Display hidden Components in Viewer** einen Haken.

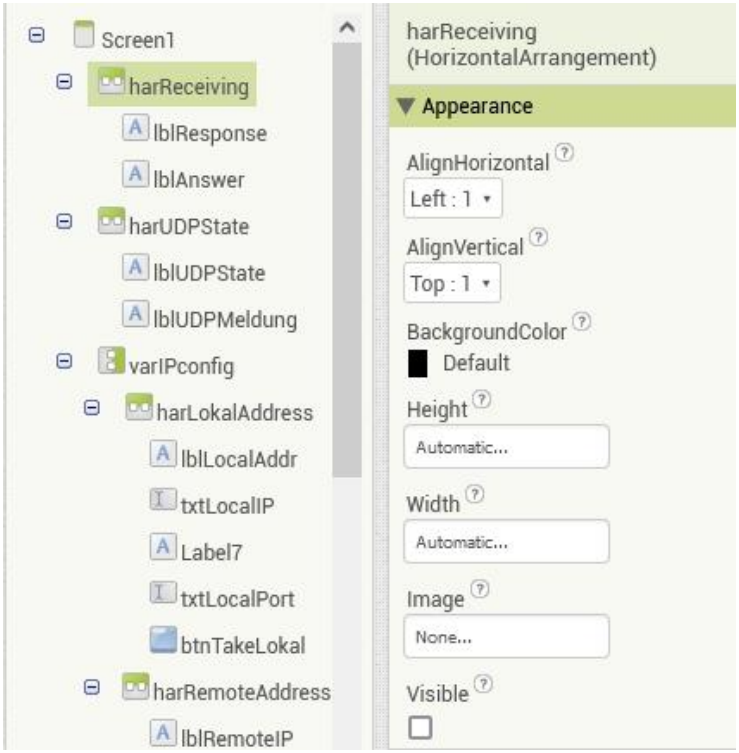


Abbildung 9: Die ersten Elemente

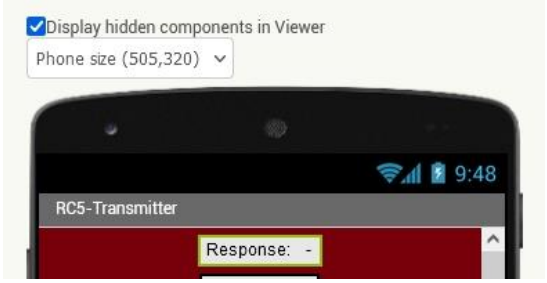


Abbildung 10: Unsichtbare Elemente im Viewer anzeigen

Das horizontale Arrangement befüllen wir mit zwei Labeln aus dem Ordner User Interface, **lblResponse** und **lblAnswer**.

lblResponse hat folgende Eigenschaften.

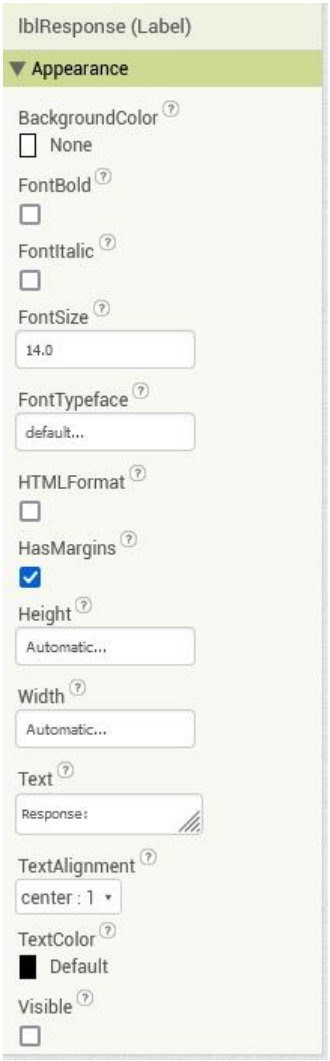


Abbildung 11: Eigenschaften von lblResponse

lblAnswer hat das **TextAllignment** left: 0, und erhält als **Text** "-".

Das horizontale Arrangement **harUDPState** enthält ebenfalls zwei Labels. Deren Eigenschaften sind mit Ausnahme des Textes die gleichen wie bei **lblAnswer**.

Jetzt folgt ein vertikales Arrangement, **varIPconfig**, dem wir zwei horizontale Arrangements implantieren, **harLokalAddress** und **harRemoteAddress**.

varIPconfig:
BackgroundColor: custom (#c66c21ff)

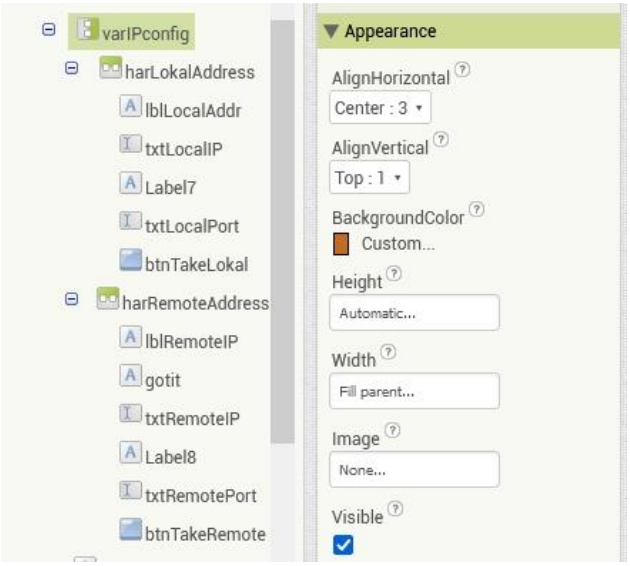


Abbildung 12: Dialog der Verbindungsdaten

harLokalAddress wird mit folgenden Elementen gefüttert, deren Vorgabewerte wie folgt angepasst wurden:

Label **lblLocalAddr:**
Fontsize: 14
Width: 70 pixels
Text: Lokal \nIP-Addr.
TextAlignment: right: 2

Textfeld **txtLocalIP**:
Fontsize: 14
Width: 90 pixels
Text: 10.0.1.65
TextAlignment: center: 1

Label **Label7**:
Fontsize: 14
Text: ":"
TextAlignment: center: 1

Textfeld **txtLocalPort**:
Fontsize: 14
Width: 60 pixels
Text: 9091
TextAlignment: center: 1
NumbersOnly: ja

Button **btnTakeLokal**:
Fontsize: 16
Text: OK

harRemoteAddress ist genauso aufgebaut enthält aber noch das zusätzliche Label **gotit**, das wir zunächst unsichtbar schalten.

Das Label **lblHeadline** leitet den RC5-Block ein.
FontBold: ja
Fontsize: 18
Width: Fill Parent…
Text: RC5-CONTROLLER
TextAlignment: center: 1

Das Tastenfeld der RC5-Steuerung wohnt in einem vertikalen Arrangement. Es besteht aus der Abfolge des **OFF**-Buttons, dem drei horizontale Arrangements folgen, die jeweils drei Zifferntasten enthalten. Die 0-Taste steht wieder alleine auf weiter Flur darunter, gefolgt vom horizontalen Arrangement mit den Tasten zur Lautstärkesteuerung. Zwischen den Steuerelementen sind "namenlose" Labels eingeschoben, um die Buttons auseinanderzurücken. Gleiches gilt für die horizontalen Arrangements.

Abbildung 13: Anordnung der Tasten

varKeypad:
AllignHorizontal: Center: 3
AllignVertical: Center: 2
Width: Fill Parent…
BackgroundColor: custom (#c66c21ff)

btnOff:
BackgroundColor: Orange
FontBold: ja
Fontsize: 30
Height: 55
Text: OFF
Shape: Oval

btn7: (stellvertretend für alle weiteren Tasten)
BackgroundColor: Orange
FontBold: ja
Fontsize: 30
Height: Automatic
Width: 50
Text: 7
Shape: Rounded
TextAlignment: center: 1

Das Label **lblAlarmSettings** ist die Überschrift des Wecker-Blocks. Die Properties entsprechen denen von **lblHeadline**

Das vertikale Arrangement **varAlarm** dient zur Steuerung der Alarmzeit.

Das Angabe der Alarmzeit passiert über das Textfeld **txtAlarmTime**. Mit dem Button **SET** wird der Steuertext an den ESP32 gesendet. **OFF** schaltet den Alarm unscharf. **CANCEL ALARM** schaltet den Alarm aus, jedoch ohne ihn zu deaktivieren.

varAlarm:
AllignHorizontal: Center: 3
AllignVertical: Center: 2
Height: 150 pixels…
BackgroundColor: custom (#c66c21ff)

Darin enthalten sind das horizontale Arrangement **harAlarm** und der Button **btnCancelAlarm**:
Height: 50 pixels…
BackgroundColor: Orange
Fontbold: ja
Text: CANCEL ALARM
Fontsize: 24
TexAllignment: center 1
Shape: Oval

harAlarm:
AllignHorizontal: Left 1
AllignVertical: Top 1

Darin sind verpackt:
btnAlarmOff / btnAlarmSet
Height: 50 pixels…
BackgroundColor: Red /Green
Fontbold: ja
Text: OFF / SET
Fontsize: 30
TexAllignment: center 1
Shape: Oval

txtAlarmTime:
Height: 50 pixels…
Text: 2:35
Fontsize: 30
TexAllignment: center 1
Hint: Weckzeit

Abschließend brauchen wir noch einen Timer und für den Funkverkehr via UDP die Extension von Ullis Robotterseite. Den Timer **Clock1** holen wir aus dem Ordner Sensors. Wie Sie die Extension einrichten, das finden Sie in dem schon eingangs empfohlenen PDF-Tutorial. Die im Ordner **Extensions** befindlichen UDP-Blöcke ziehen wir als Letztes in den Viewer. Der Timer und die UDP-Blöcke sind unsichtbare Elemente. Sie erscheinen nicht im Display des Handys. Im Viewer erscheinen sie dennoch, aber unterhalb des Handybildes.

Das Programm

Jetzt ist es an der Zeit, dass wir **packetsender.exe** auf dem PC starten. Damit können wir die von unserer App ausgelösten Aktionen verfolgen. Aus diesem Grund geben wir für die Remote-IP-Adresse auch erst einmal die IP des PCs an. Die App sendet also nicht an den ESP32, sondern zunächst an den PC mit Packetsender als Empfänger. Wenn alles wunschgemäß übermittelt wird, stellen wir die IP auf den ESP32 um.

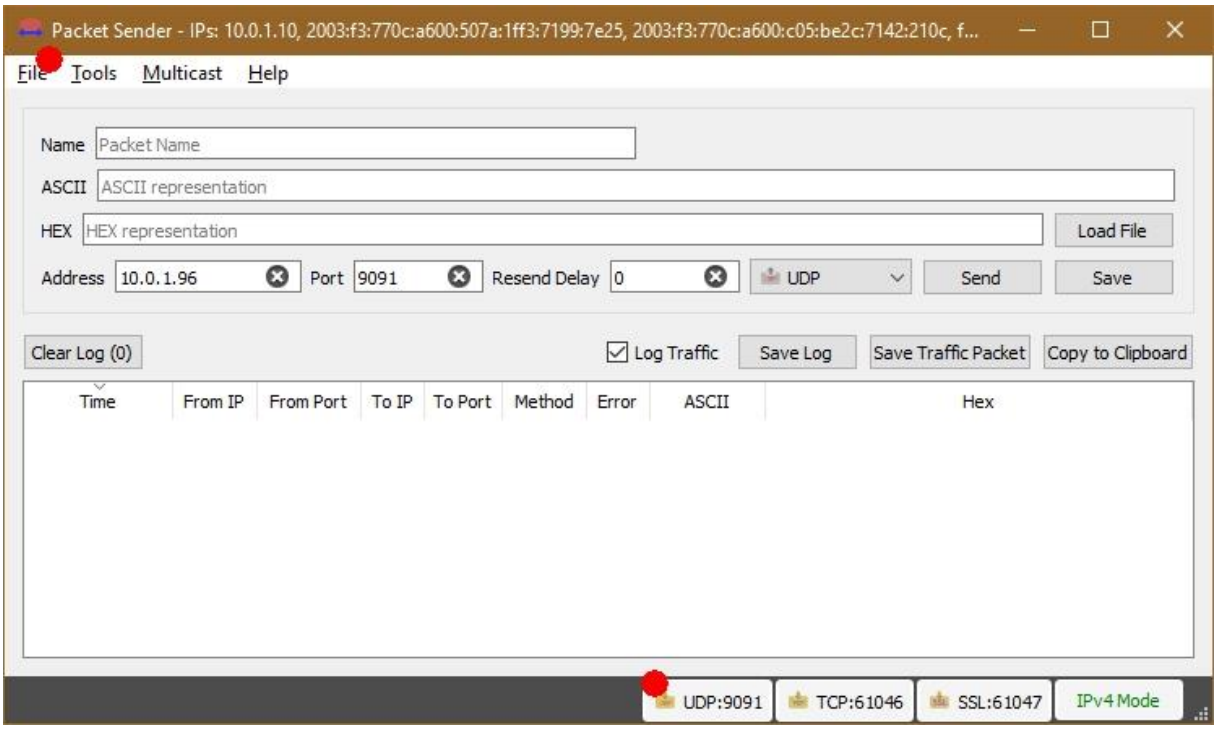


Abbildung 14: Packetsender - Startfenster

Über das **File**-Menü rufen wir die **Settings** auf, um die Portadresse von Packetsender anzupassen.

Abbildung 15: UDP-Portadresse einstellen

Wir wechseln im AI2-Fenster jetzt in den Ordner **Blocks**, dazu klicken wir auf den Button **Blocks** ganz rechts in der grünen Titelzeile.

Wie in MicroPython beginnen wir mit der Deklaration der globalen Variablen.

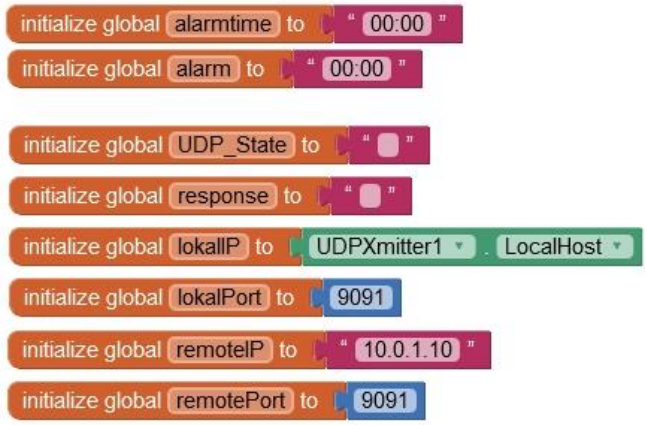


Abbildung 16: Variablendeklaration

Aus dem Ordner **Blocks.Built-in** ziehen wir **initialize global**-Blocks in den Viewer.

Abbildung 17: Bausteine für die Arbeit mit Variablen

Abbildung 18: Andocken

Wir vergeben einen Namen für die Variable und holen aus **Blocks.Built-in.Text** einen Block für eine Textkonstante und geben den Text "00:00" ein. Dann docken wir den Textblock an den Variablenblock an.

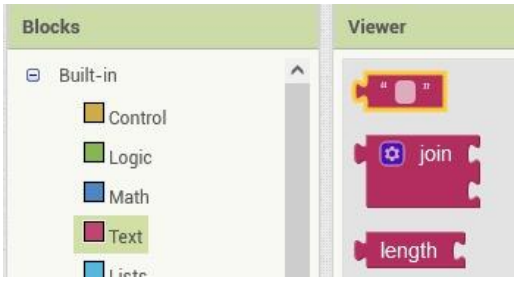


Abbildung 19: Stringvariablen belegen

Für numerische Werte finden wir die Entsprechung in **Blocks.Built-in.Math**.

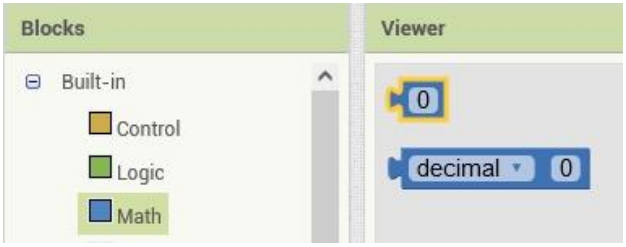


Abbildung 20: Numerische Konstanten festlegen

Die IP-Adresse des Handys fragen wir über die Eigenschaft **UDPXmitter1.LocalHost** ab. Die finden wir in **Blocks.UDPXmitter1**.

Abbildung 21: UDP-Xmitter-Property abfragen

Programme, die mit dem AI2 erstellt werden, sind, wie unser MicroPython-Weckerprogramm, ereignisgesteuert. Es läuft also eine Hauptschleife, die nur darauf wartet, dass ein bestimmtes Ereignis eintritt, um dann darauf zu reagieren.

Das erste Ereignis ist der Bildschirmaufbau. Darin verpacken wir all das, was zur Initialisierung unseres Programms nötig ist. Vergleichbar ist das mit den Vorbereitungen in MicroPython oder dem setup-Block in der Arduino-Umgebung.

Was ist also zu tun?

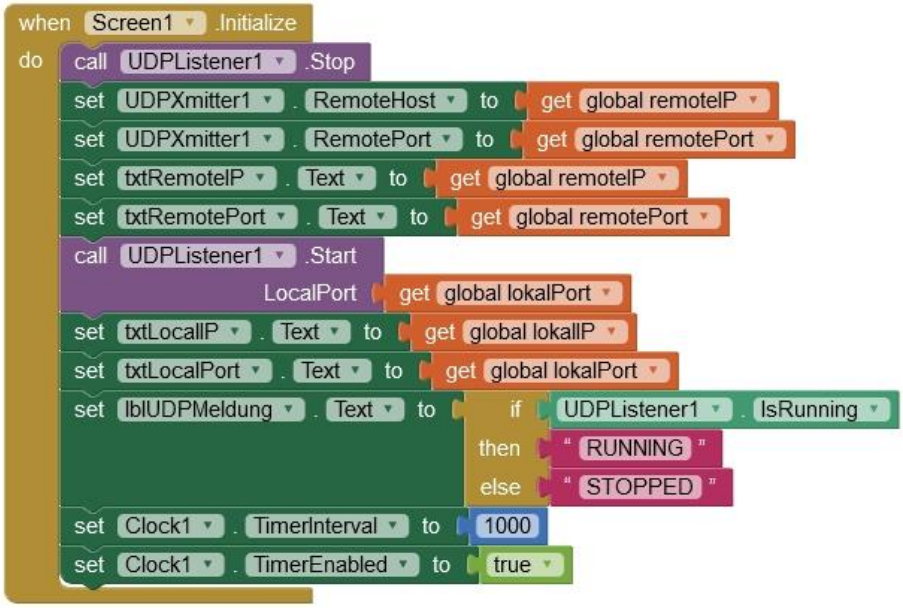


Abbildung 22: Initialisierung der Anwendung

Zum Aufbau einer funktionierenden UDP-Verbindung halten wir den Empfänger erst einmal an, damit wir den UDP-Socket konfigurieren können. Um die WLAN-Verbindung kümmert sich Android selber, da müssen wir nichts unternehmen.

Woher Sie die Blocks bekommen, sehen Sie anhand der Blocknamen. Steuerelemente sind beige, Eigenschaften grün, Referenzen auf Variablen orange, Logikelemente hellgrün und Textelemente pink gefärbt. Eine weitere Orientierung erhalten Sie durch die Farbkästchen in **Blocks.Built-in**.

Wir füttern erst einmal den Socket mit den Remote-Daten und weisen die IP und die Portnummer unseren Textfeldern aus **harRemoteAddress** zu. Dann starten wir den Socket erneut mit der von uns vergebenen lokalenPortnummer. Anschließend füllen wir die Text-Attribute der Felder in **harLokalAddress**, Stellen den Timer auf 1000ms und starten ihn. Die entsprechende ISR wird den Zustand der UDP-Verbindung überprüfen und melden.

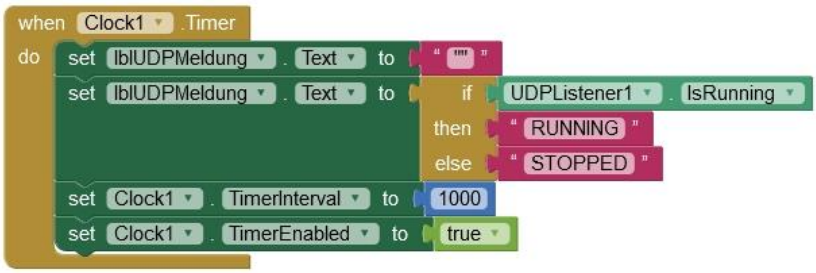


Abbildung 23: Timer-ISR

when Clock1.Timer wird aufgerufen, wenn der Timer abgelaufen ist. Wir löschen den Text in **lblUDPMeldung** und belegen das Attribut in Abhängigkeit des Zustands des UDP-Sockets. Dann starten wir den Timer erneut.

Die lokalen Verbindungsdaten in **harLokalAddress** werden übernommen, wenn der Button **btnTakeLokal** angetippt wird.

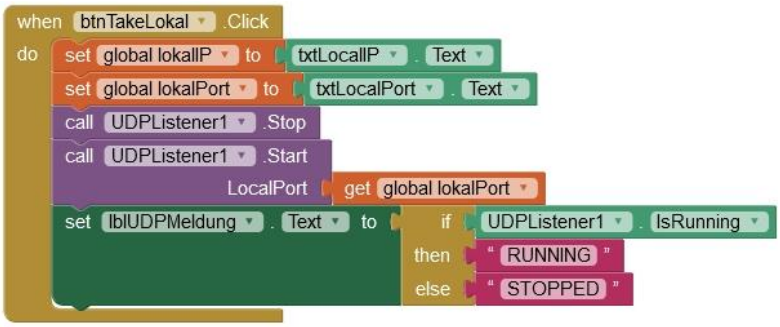


Abbildung 24: Lokale Verbindungsdaten übernehmen

Die globalen Variablen erhalten die neuen Werte, der Socket wird gestoppt und mit der lokalen Portnummer wieder gestartet. Über das Label **lblUDPMeldung** erfahren wir den Zustand des UDP-Interfaces. Ähnlich arbeitet ein Klick auf den Button **btnTakeRemote** für die Remote-Daten.

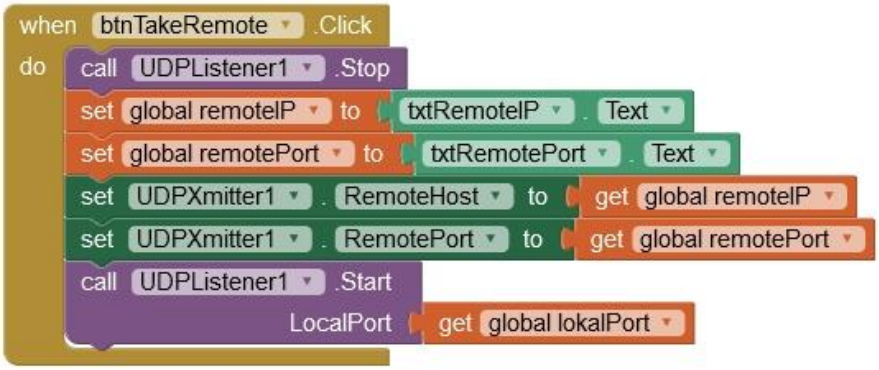


Abbildung 25: Remote-Verbindungsdaten übernehmen

Auftretende UDP-Verbindungsfehler kann man mit **when UDPListerner1.ListenerFailure** abfangen.

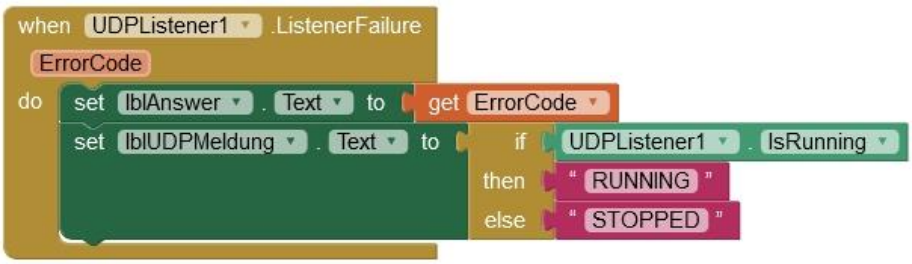


Abbildung 26: Reaktion auf UDP-Verbindungsfehler

Die Referenz auf den Fehlercode bekommen wir aus der Liste **ErrorCode**.

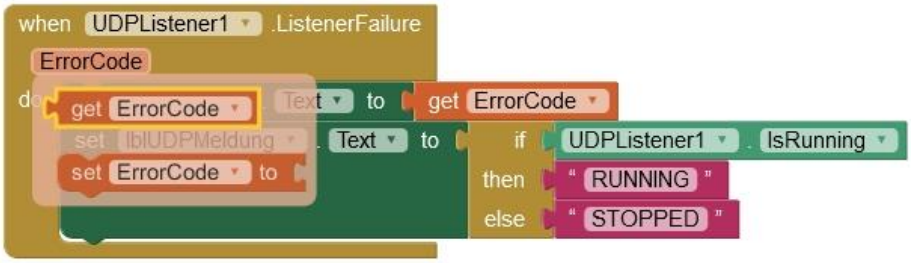


Abbildung 27: Die Referenz uf ErrorCode

Alle Buttons in **varKeypad** arbeiten alle nach demselben Prinzip. Beim Antippen müssen sie die Nachricht an den ESP32 senden, die wir im Programm **wecker.py** in der Funktion **parse**() ab Zeile 157 festgelegt haben.

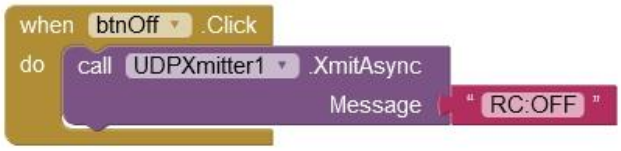


Abbildung 28: Reaktion auf Button-Klicks

Im Gegensatz zum Aufbau der App-Oberfläche im Designer können wir hier mit Copy&Paste arbeiten, um die nötigen Events zu programmieren. Ein Rechtsklick auf den fraglichen Block ruft das Kontextmenü auf. **Duplicate** erzeugt eine Kopie des Blocks, in dem wir nur noch das Buttonobjekt und den zu sendenden Text ändern müssen. Das Button-Objekt wählen wir aus der Liste, die wir durch Klick auf das kleine, abwärts gerichtete Dreieck erhalten. Auf diese Weise erstellen wir auch die Behandlungsroutinen für die Buttons **btnAlarmOff** und **btnCancelAlarm**.

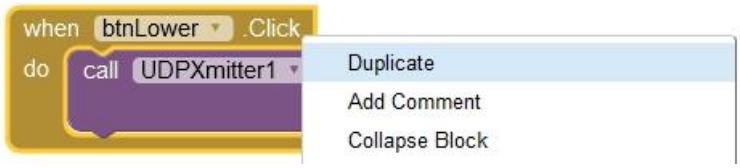


Abbildung 29: Blocks kopieren

Die folgende Tabelle stellt die Verbindung zwischen Buttonbezeichner und Sendetext her.

Button-Objekt	Nachricht
btnLower	RC: L-
btnHigher	RC: L+
btnMute	RC: MUTE
btnOff	RC: OFF
btn0	RC: 0
btn1	RC: 1
btn2	RC: 2
btn3	RC: 3
btn4	RC: 4
btn5	RC: 5
btn6	RC: 6
btn7	RC: 7
btn8	RC: 8
btn9	RC: 9
btnAlarmOff	RC: OFF
btnCancelAlarm	RC: CANCELL

Abbildung 30: Korrelation Button zu Nachricht

Für den Button **btnAlarmOff** fügen wir noch eine Ergänzung ein. Dadurch wird die Hintergrundfarbe auf hellrot gesetzt, wenn der Alarm deaktiviert wird.

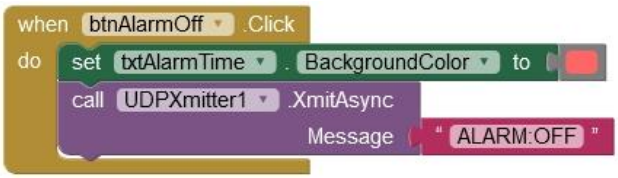


Abbildung 31: Alarm deaktivieren - Ergänzung

Jetzt fehlt uns nur noch das Setzen der Alarmzeit.

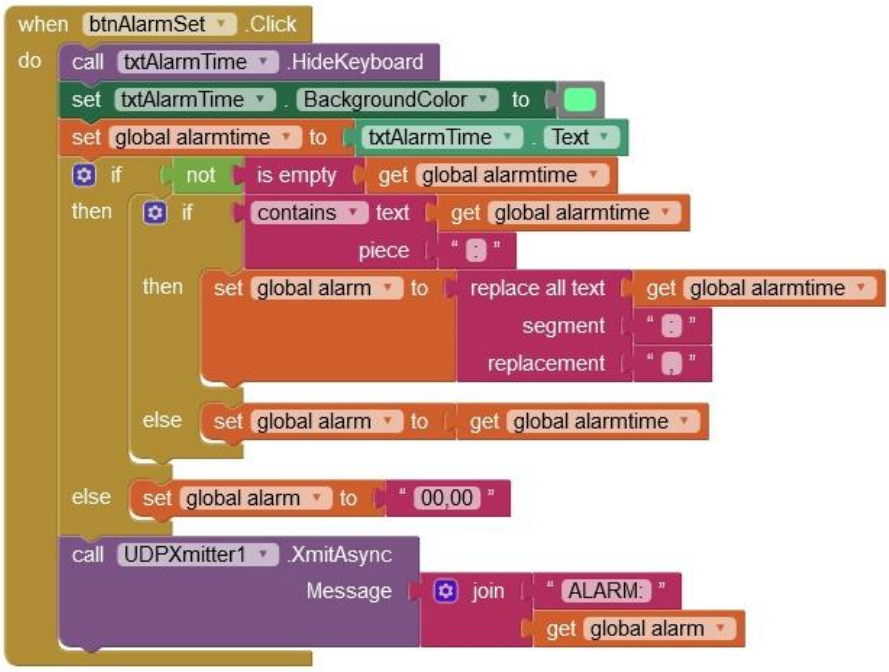
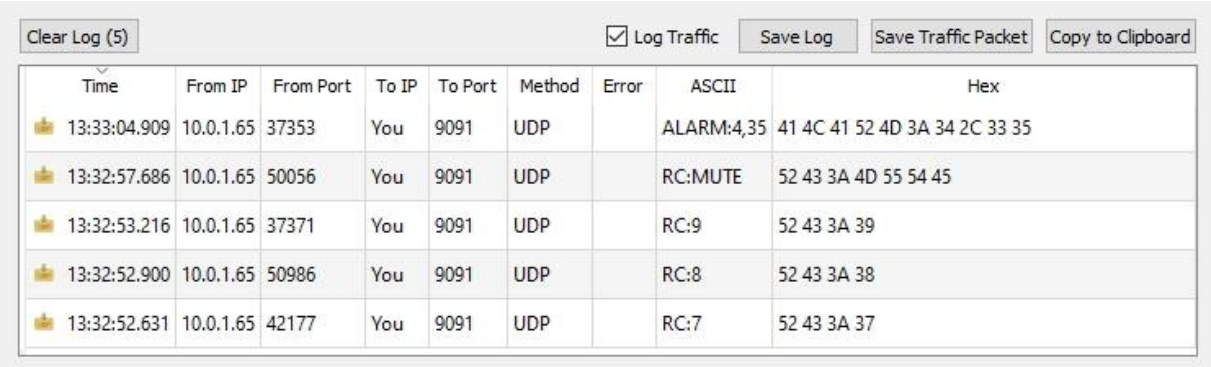


Abbildung 32: Alarmzeit setzen

Mit dem Drücken des Buttons soll zuerst die Tastatur ausgeblendet werden, die mit dem Antippen des Eingabefelds erscheint. Als Zeichen für das Setzen einer Alarmzeit wird die Hintergrundfarbe des Textfelds auf hellgrün gesetzt. Die globale Variable **alarmtime** übernimmt den Wert von **txtAlarmTime.text**. Ein leeres Eingabefeld wird abgefangen und die globale Variable **alarm** auf "00:00" gesetzt. Nichtleere Eingaben werden erst einmal auf einen Doppelpunkt untersucht. Wird einer gefunden, dann ersetzen wir den Doppelpunkt (normales Trennzeichen bei Zeitangaben) durch ein Komma (syntaktisches Trennzeichen unserer Nachrichten). **alarm** wird jetzt auf **alarmtime** gesetzt und schließlich mit dem Kommando **ALARM** an den ESP32 gesendet. Weil wir hierbei den Doppelpunkt als Trenner zwischen Kommando- und Daten-Teil verwenden, mussten wir ihn oben ersetzen, damit beim Parsen beim ESP32 keine Konfusion entsteht.

Wenn jetzt in Packetsender die korrekten Nachrichten erscheinen, wenn am Handy ein Button angetippt wird und wenn die Alarmzeit richtig übermittelt wird, dann haben Sie gewonnen.



Time	From IP	From Port	To IP	To Port	Method	Error	ASCII	Hex
13:33:04.909	10.0.1.65	37353	You	9091	UDP		ALARM:4,35	41 4C 41 52 4D 3A 34 2C 33 35
13:32:57.686	10.0.1.65	50056	You	9091	UDP		RC:MUTE	52 43 3A 4D 55 54 45
13:32:53.216	10.0.1.65	37371	You	9091	UDP		RC:9	52 43 3A 39
13:32:52.900	10.0.1.65	50986	You	9091	UDP		RC:8	52 43 3A 38
13:32:52.631	10.0.1.65	42177	You	9091	UDP		RC:7	52 43 3A 37

Abbildung 33: Empfang von der Handy-App

Jetzt bleibt nur noch das Ändern der IP-Adresse, damit die Handy-Nachrichten nicht an den PC sondern an den ESP32 gesendet werden. Der sollte nun auf die Kommandos reagieren. Jetzt ist auch der richtige Zeitpunkt, um gegebenenfalls den ESP32 vom Routerbetrieb auf den eigenen Accesspoint umzustellen. Alle IP-Adressen dazu passend abzuändern.

Abbildung 34: Geänderte IP-Adresse

Damit die App unabhängig vom AI-Companion funktioniert, müssen wir sie kompilieren. Das geschieht über den Menüpunkt **Build.Android App** (apk).

Die apk-Datei können Sie auf zwei Arten auf dem Handy installieren, per Download oder über den PC. Wie das im Einzelnen abläuft, lesen Sie hier ab Seite 16.

Die komplette apk-Datei finden Sie auch als Download, ebenfalls die Quelle dazu, die Sie via **Projects.Import project (.aia) from my computer** im AI2-Inventor importieren können.