Projekt Software Defined Radio Demodulation von DCF-77 via WebSDR

David Radtke

12. Juli 2017

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Das Projekt Demodulation von *DCF77* via WebSDR wurde im Rahmen des Wahlpflichtfaches „Software Defined Radio“ durchgeführt. Hierbei wurde das Signal des Langwellensenders *DCF77* aus Mainflingen bei Frankfurt am Main bei 77,5 kHz in Form einer .wav Datei via WebSDR aufgezeichnet. Anschließend wurde das Protokoll analysiert und mithilfe der Open-Source Software „GNU-Radio“ demoduliert. *DCF77* wird von Funk-Uhren und Wetterstationen verwendet um Informationen der aktuellen Wetterlage, sowie das korrekte Datum und die Uhrzeit zu erhalten.

2 Grundlagen

2.1 Software Defined Radio

Als „Software Defined Radio“, werden Funkübertragungssysteme bezeichnet, bei denen wesentliche Anteile der Verarbeitung des OSI-Schichtenmodells mittels Software erfolgen. Dabei können Empfänger und Sender oder jeweils nur der Empfänger oder der Sender nach diesem Prinzip aufgebaut sein. Darunter gibt es Unterschiede zwischen unidirektionaler und bidirektionaler Signalübertragung. *Abbildung 2.1* kann das grundlegende Prinzip entnommen werden. [4] [3]

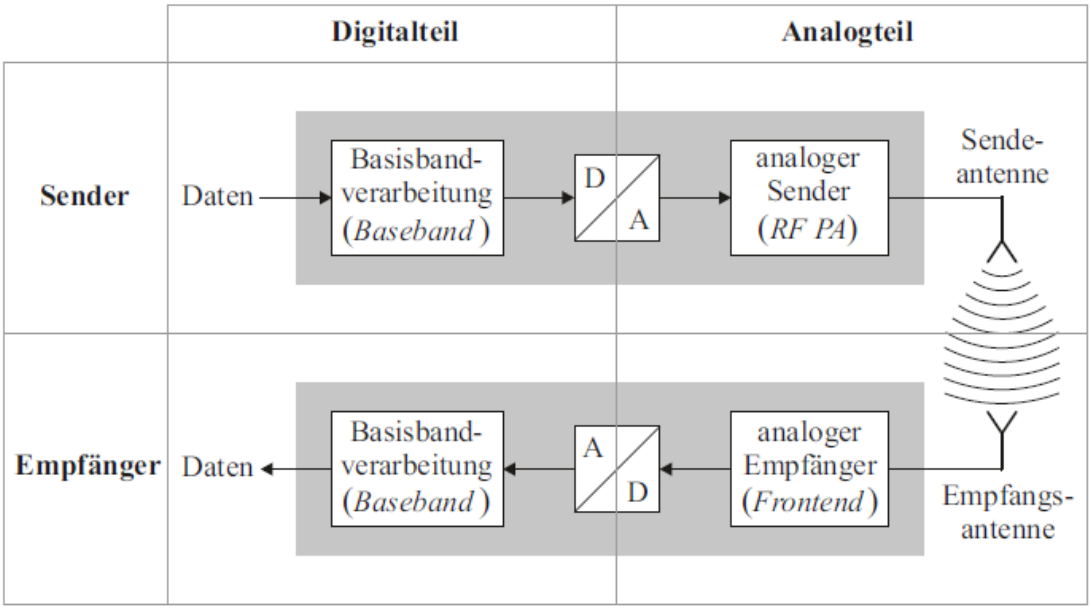

Abbildung 2.1: Aufbau eines Software Defined Radio-Systems [1]

Ein Beispiel für Software Defined Radio sind beispielsweise die Mobilfunk Übertragungstechnologien wie GSM, WCDMA, LTE oder Zeitinformationen durch *DCF77*.

[1] [4] Heuberger, Gamm 2017 - Aufbau eines Software Defined Radio-Systems

Die Vorteile bei der Verwendung von *Software Defined Radio* liegen bei:

- **Interoperabilität**
- **Effiziente Nutzung der Ressourcen**
- **Verwendung ungenutzter Frequenzen**
- **Updatefähigkeit des Systems**
- **geringere Kosten**

[3]

2.2 GNU-Radio

GNU Radio ist ein Open-Source Werkzeug zur Anwendung und Implementierung von Software Defined Radio. Dies wird realisiert, indem Signalverarbeitungsblöcke zu einer Signalverarbeitungskette, auch Flowgraph genannt, zusammengefügt werden. Hierbei werden Die Blöcke aus einer Bibliothek in den aktuellen Workspace gezogen und parametrisiert. Dabei können verschiedene Arten von Signalquellen und Analysetools verwendet werden. Ein Beispiel für einen Flowgraphen kann *Abbildung 2.2* entnommen werden. Des weiteren können eigene Signalverarbeitungsblöcke mithilfe von der Programmiersprachen Python oder C++ programmiert werden. [2]

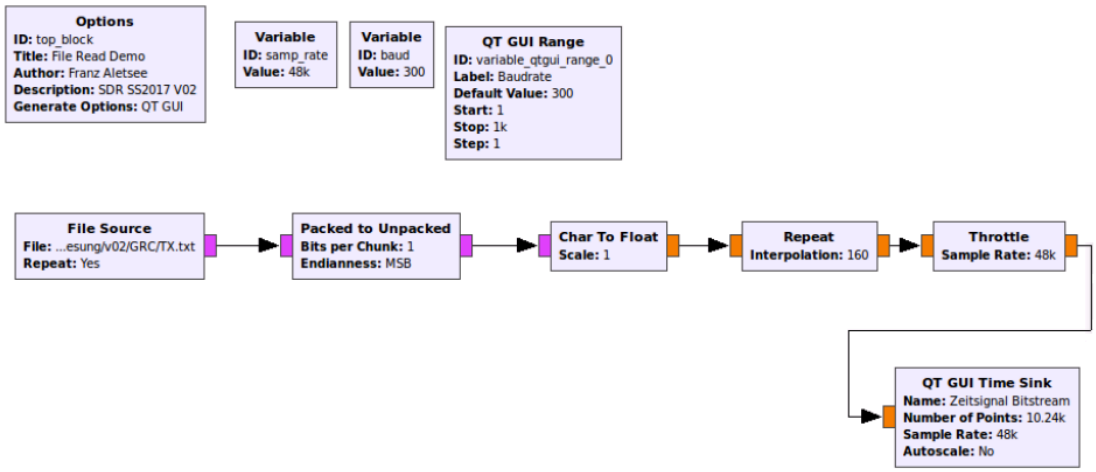

Abbildung 2.2: Beispiel eines Flowgraphen [2]

[2] Aletsee 2017 - Skript Software Defined Radio

3 DCF77

3.1 Allgemeines

DCF77 ist ein Langwellensender in Mainflingen bei Frankfurt am Main, welcher auf einer Frequenz von 77,5kHz sendet. DCF77 wird von der Physikalisch-Technischen Bundesanstalt (PTB) gesteuert. Das Signal wird in weiten Teilen Europas empfangen und wird verwendet um detaillierte und genaue Zeit und Wetter-Informationen zu übertragen. DCF77 findet meist Anwendung in privaten Funkuhren. Daneben wird DCF77 auch in öffentlichen Systemen wie die Zeitdienstsysteme bei der Bahn, Telekommunikation, Informationstechnologie, Rundfunk- und Fernsehanstalten, Tarifschaltuhren bei Energieversorgungsunternehmen, Ampelschaltungen und der Kalibrierung von Normalfrequenzgeneratoren verwendet. [5]

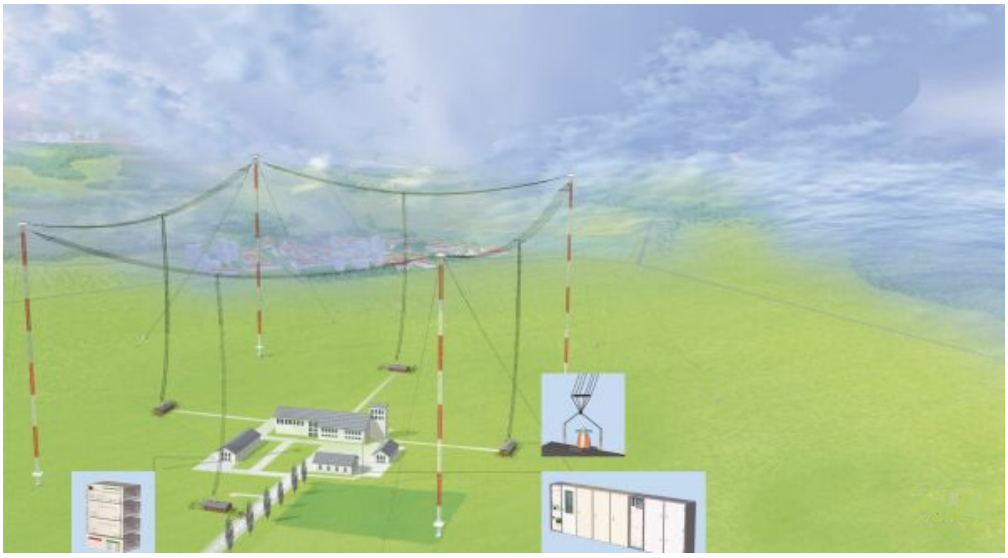

Abbildung 3.1: Schematische Darstellung der Sendefunkstelle Mainflingen, als Einsatzbilder die DCF77 Steuereinrichtungen (links), die Einspeisung in die Antenne auf dem Dach eines Antennenhauses und der Transistorsender [3]

[3] [5] Physikalisch-Technische Bundesanstalt

3.2 Technische Grundlagen

Die Trägerfrequenz von DCF77 beträgt 77,5 kHz. Die Zeitinformation wird am Standort von einer Atomuhr mit einer sehr geringen Abweichung abgeleitet und über die Sendeanlage gesendet. Die Daten werden mithilfe von Amplituden Shift Keying (ASK) übertragen. Im Gegensatz zum OOK (on off Keying) werden mehrere diskrete Stufen zugelassen. Bei DCF77 wird somit zu Beginn jeder Sekunde die Trägeramplitude des Signals auf 15% herabgesenkt. In *Abbildung 3.2* kann gut erkannt werden, das DCF77 das Modulationsverfahren Amplitudenumtastung *(Amplitude Shift Keying)* verwendet. [5] [2]

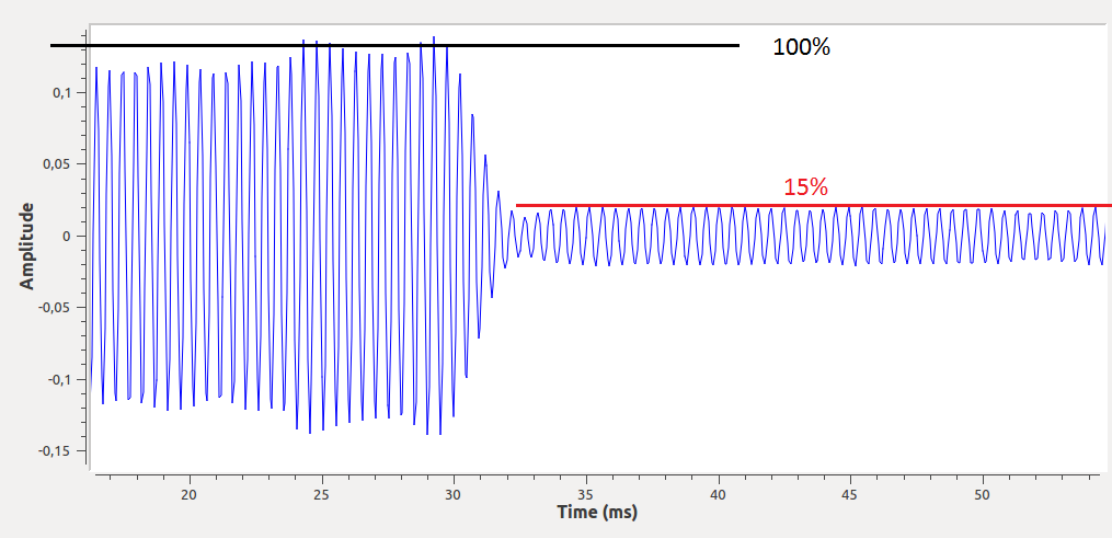

Abbildung 3.2: Verlauf der Aufzeichnung eines DCF77 - Signals, man erkennt die Absenkung der Amplitude auf 15%

Um eine Synchronisation von Sender und Empfänger zu ermöglichen erfolgt beim Beginn einer Minute, zwischen Sekunde 59 und Sekunde 60 (0), keine Absenkung der Trägeramplitude. Im Falle einer Schaltsekunde enthält die Sekunde 59 eine Absenkung und es wird eine weitere Sekunde eingefügt. Die eigentliche Datenübertragung erfolgt über binäre Signale. Die Dauer der der Amplitudensenkung auf 15% steht jeweils für ein binäres Signal. Eine Absenkung, welche 100ms andauert entspricht einer „0“, eine 200ms andauernde Absenkung eine „1“.

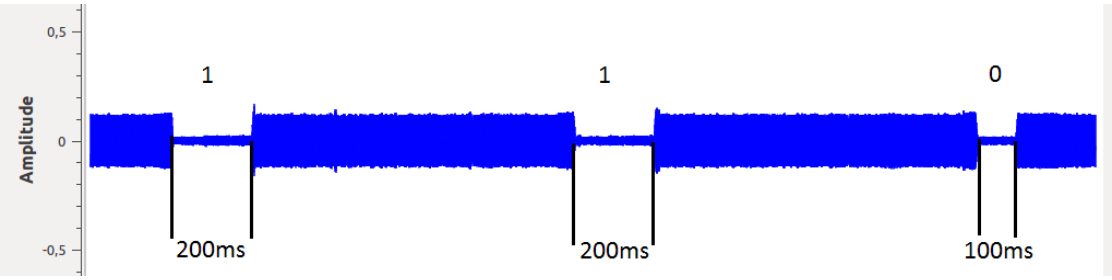

Abbildung 3.3: Übertragung der einzelnen Bits durch DCF77, 100ms Absekung entspricht einer 0; 200ms Absenkung einer 1

Durch die 59-malige Absenkung der Amplitude, stehen 59 Bits innerhalb einer Minute zur Datenübertragung zur Verfügung. Lediglich die Bits 20 bis 59 werden für die Übertragung der Zeitinformation verwendet. Die Bits 1 -14 werden als Wetterinformation der Firma „MeteoTime“ und Informationen für den Katastrophenschutz verwendet. Für die Entschlüsselung der Daten wird daher eine Lizenz benötigt. Für die Codierung der Zeitinformationen siehe *Anhang 1.*

4 Demodulation von DCF77

4.1 Händische Demodulation durch Audacity

Um eine Aufzeichnung eines DCF77 Signales zu erhalten, wurde mithilfe der online verfügbaren WebSDR[4] Seite der *Universität Twente (Niederlande)* eine .wav Audiodatei aufgezeichnet und gespeichert. Hierbei wurde das Signal bei 77,5 kHz und einer Bandbreite von 0,26 kHz aufgezeichnet. Anschließend konnte die .wav Datei mithilfe des Audiotools Audacity analysiert werden und händisch demoduliert werden. Dabei wurde die Erste Sekunde abgesucht (keine Absenkung der Amplitude) und die einzelnen Bits aufgezeichnet und mithilfe der Übersicht *Codierung DCF77 (siehe Anhang 1)* manuell dekodiert.

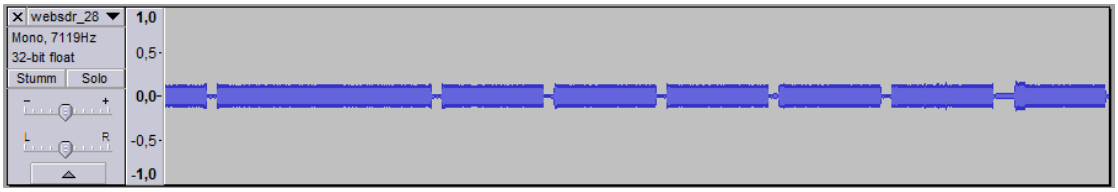

Abbildung 4.1: DCF77 Signal als .wav Datei in Audacity, man kann die erste Sekunde erkennen (keine Absenkung der Amplitude)

4.2 Demodulation durch GNU-Radio

Im Gegensatz zur händischen Demodulation wird nun in GNU-Radio ein Flowgraph entwickelt, welcher das Signal aus einer Datei einliest und die .wav Datei in Bits umwandelt. Anschließend wird mithilfe eines Python-Blocks die Zeitinformation über das Ausgabeterminal ausgegeben. Der *Abbildung 4.2* kann der Flowgraph entnommen werden. Die Funktionsweise wird durch die einzelnen Signalverarbeitungsblöcke veranschaulicht. Dabei wird auf die wichtigsten Blöcke eingegangen. Blöcke wie *Throttle* oder *QT-TimeSink* sind für die Funktion irrelevant.

[4][1] Amateur Radio Club ETGD - Universität Twente Faculty for Electrical Engineering, Mathematics and Computer Science

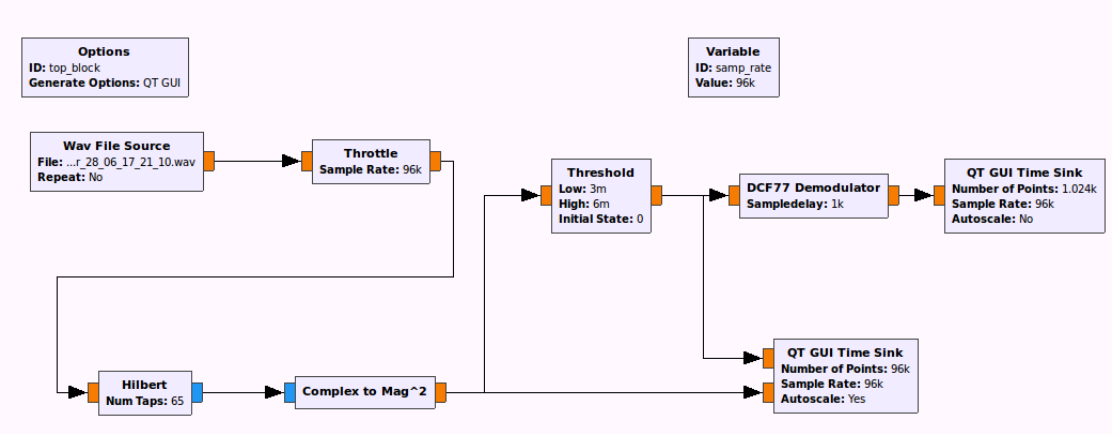

Abbildung 4.2: Flowgraph zur Demodulation von DCF77

4.2.1 File Source

Der erste Block der Signalverarbeitungskette ist die Wav File Source. Wie der Name schon sagt, handelt es sich hierbei um die Signalquelle (.wav – Signalquelle). In diesem Block kann eingestellt werden, ob das Signal permanent wiederholt wird oder nicht (Repeat). Für die Auswertung von DCF77 wurde Repeat: No eingestellt.

4.2.2 Hilbert Transformation und Complex to Mag2

Die Hilbert Transformation ist zur Demodulation eines Amplituden Shift Keying modulierten Signales essentiell. Dabei wird das reelle Eingangssignal in ein analytisches Ausgangssignal umgewandelt. Die mathematische Definition der Hilbert-Transformation lautet:

$$\hat{x}(t) = H\{x(t)\} \tag{4.1}$$

Hierbei werden die negativen Frequenzanteile des Eingangssignales mit $+j$ und die positiven Frequenzanteile des Signales mit $-j$ multipliziert. Um nun das analytische Signal $x_a(t)$ zu erhalten wird die Hibert Transformation wie folgt angewendet:

$$x_a(t) = x(t) + j \cdot H\{x(t)\} = x(t) + j \cdot \hat{x}(t) \tag{4.2}$$

Dadurch erhält man das komplexe Signal $x(t) + j \cdot \hat{x}(t)$. Die Fourier Transformierte $X_a(f)$ für positive Frquenzen des Signales lautet:

$$X_a(f) = X(f) + j \cdot \hat{X}(f) = X(f) + j \cdot (-j) \cdot X(f) = 2 \cdot X(f) \tag{4.3}$$

Die Fourier Transformierte $X_a(f)$ für negative Frquenzen lautet:

$$X_a(f) = X(f) + j \cdot \hat{X}(f) = X(f) + j \cdot j \cdot X(f) = 0 \tag{4.4}$$

Und für die Frequenz gleich 0:

$$X_a(f) = X(f) + j \cdot \hat{X}(f) = X(f) + j \cdot 0 \cdot X(f) = X(f) \tag{4.5}$$

Somit werden alle negativen Frequenzteile elemeniert. Der Betrag des analytischen Signales entspricht folglich der Hüllkurve des reellen Signales. Aus der Sinusschwingung wird also mit der Hilbertransformation und mit Complex to Magnitude2 die Hüllkurve generiert, mit der nun logische Zustände interpretiert werden können. [2]

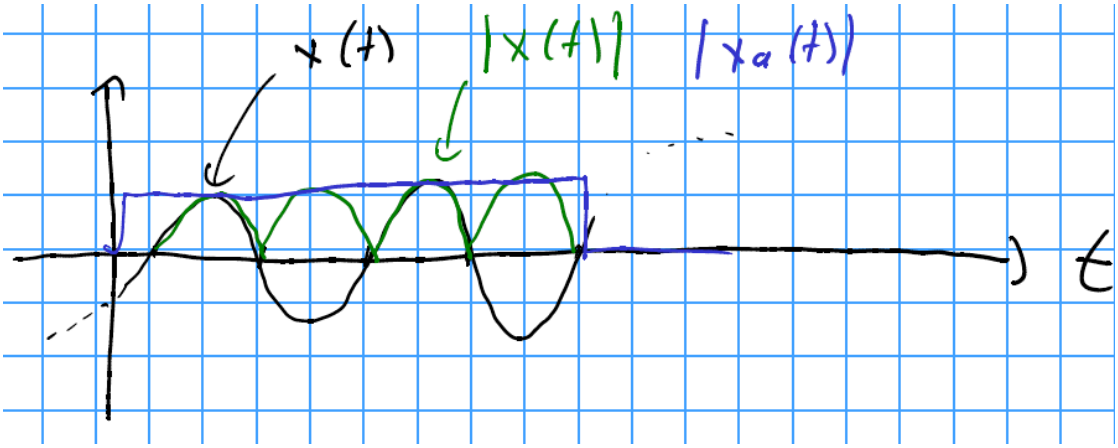

Abbildung 4.3: Hüllkurve aus Sinusschwingung [2]

4.2.3 Threshold

Der Threshold Signalverarbeitungsblock wandelt das Signal nun in logische Zustände 1 und 0 um. Hierzu wird das Ausgangssignal bei Unterschreiten des definierten Low-Wertes der Amplitude auf 0 gesetzt und bei Überschreiten des High-wertes auf 1 gesetzt. Dabei werden Überschwingungen eliminiert und der Pegel angepasst. Die Pegelanpassung ist wichtig, da nach der Hilbert Transformation und Complex to Mag2 das Signal eine sehr geringe Amplitude besitzt.

[2] [2] Aletsee 2017 - Skript Software Defined Radio

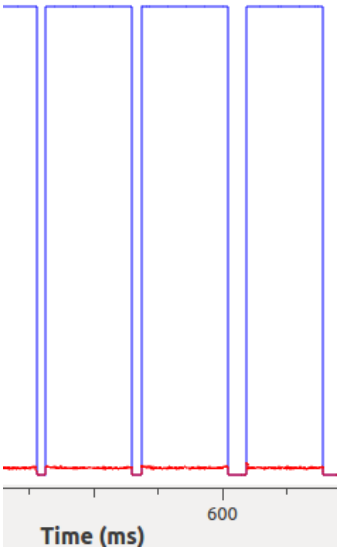

Abbildung 4.4: Ausgangs - und Eingangssignal nach Threshhold Block

4.2.4 Python Block DCF77 Demodulator

Die Demodulation der Einzelnen Bits und die Ausgabe der Uhrzeit erfolgt mit dem selbst programmierten Signalverarbeitungsblock *DCF77 Demodulator (siehe Anhang 2)*. Dazu werden aus der GNU-Radio Python-Bibliothek Klassen abgeleitet und definiert. Dabei stehen unterschiedliche Klassen wie der sync-Block oder der basic-Block zur Verfügung. Grundsätzlich liest der Block jeden Eingangs-Sample input_items[0] einzeln ein und verarbeitet diesen und gibt, falls definiert, Ausgangs-Samples output_items[0] aus.

Mit

```
class blk(gr.basic_block):  # basic Block
```

wird die basic_block Klasse der GNU-Radio Bibliothek aufgerufen. Hierbei handelt es um einen Signalverarbeitungsblock, bei dem die Anzahl der Ausgangs und Eingangssamples ungleich ist. Nun wird die Funktion

```
def general_work(self, input_items, output_items):
```

definiert. Die Funktion übernimmt die eigentliche Verarbeitung und Ausgabe der Samples.

Eine fallende Flanke wird durch folgende Abfrage erkannt:

```
if (in0[k] - in0[k-1] < 0):
```

Hierbei wird der Eingangssample mit dem vorherigen Sample verglichen und durch die Differenz der zwei Samples kann eine fallende Flanke erkannt werden, und die Variablen

```
self.state=1
self.counter=self.SampleDelay
```

werden inittialisiert. Nun wird der Zustand des Signales 1000 Samples *(SampleDelay=1000)* nach der fallenden Flanke abgefragt. 1000 Samples entsprechen einer Zeitdauer von ca 120ms. Wenn nun eine 0 übertragen wird, hat der Eingangssample in0[k] 1000 Samples (ca. 120ms) nach der fallenden Flanke den Wert 1 (Absenkung der Amplitude auf 15% dauert nur 100ms an). Wird eine logische 1 übertragen, ist die Amplitude noch abgesenkt und der Eingangssample hat 1000 Samples nach der fallenden Flanke den Wert 0.

```
# counter wurde von 1000 auf 0 gezaehlt
if self.state==1 and self.counter == 0:
.
.
        self.state=0  #state = 0, um erneut Flanke zu erkennen
.
#Logische 1
        if in0[k] == 0 and self.Bitcounter >= 0:
        .       # Verarbeitung der Bits
        .       # Verarbeitung der Bits
        .       # Verarbeitung der Bits
                # Bitcounter inkrementieren
                self.Bitcounter = self.Bitcounter + 1
# logische 0
        if in0[k] == 1:
                # Bitcounter inkrementieren
                self.Bitcounter = self.Bitcounter + 1
```

Wichtig für die Zuordnung der Bits ist der Bitcounter. Dieser gibt das aktuelle Bit an, mit dem die Zeitinformation bestimmt wird. Bei einer logischen 1 wird dann der Wert des aktuellen Bits zu den Variablen (Jahr, Monat, Stunde...) hinzugerechnet.

```
#Logische 1, Start erst wenn Minute erkannt wurde (Bitcounter >= 0)
        if in0[k] == 0 and self.Bitcounter >= 0:
                if self.Bitcounter == 54:
                        self.Jahr = self.Jahr + 10
                if self.Bitcounter == 53:
                        self.Jahr = self.Jahr + 8
                if self.Bitcounter == 52:
                        self.Jahr = self.Jahr + 4
```

Um die Nummer des aktuellen Bits zuordnen zu können, und den Bitcounter zu initialisieren, muss der Beginn der ersten Sekunde (Bit 0) bestimmt werden. Hierzu wird, analog zur Erkennung einer logischen 1 oder 0, der Wert des Eingangssamples nach einer negativen Flanke abgefragt. Dabei wird der Wert des Signales 7400 Samples (ca 1050 ms) nach der fallenden Flanke geprüft. Liegt dieser Wert noch bei 1 erfolgte keine Absenkung der Amplitude und die erste Sekunde der Minute wurde gefunden und der Bitcounter wird auf 0 gesetzt.

```
# Erstes Bit erkannt (Keine Absenkung auf 15% der Amplitude)
        if self.state==0 and self.counter < -7400:
                .
                .
                self.Bitcounter = 0
```

Die Ausgabe der Datums erfolgt Abschließend mit der Funktion print(). Dabei wurden die Variablen mit '%0.2d' formatiert, um bei einstelligen Werten eine führende 0 auszugeben.

```
if self.Bitcounter == 59:                    #Uhrzeit ausgeben
        print( '%0.2d' %(self.Stunde) + ..... + str(self.Jahr) )
        self.Bitcounter == -1                # Letztes Bit erreicht
```

Ausgabe:

```
Using Volk machine: avx_64_mmx_orc
Start/Stop
Zeit/Datum:  17:11,  Dienstag der 11.07.17
```

Abbildung 4.5: Ausgabe über Ausgabeterminal [2]

Literatur

[1] Amateur Radio Club ETGD - Universität Twente Faculty for Electrical Engineering, Mathematics and Computer Science. URL: `http://websdr.ewi.utwente.nl:8901`.

[2] Franz Aletsee. *Skript Software Defined Radio, Sommersemester 2017*. 2017.

[3] Eugene Grayver. *Implementing software defined radio*. New York, NY: Springer, 2013. ISBN: 978-1-4419-9331-1.

[4] Albert Heuberger und Eberhard Gamm. *Software Defined Radio-Systeme für die Telemetrie: Aufbau und Funktionsweise von der Antenne bis zum Bit-Ausgang*. Berlin, Heidelberg und s.l.: Springer Berlin Heidelberg, 2017. ISBN: 978-3-662-53233-1.

[5] Physikalisch-Technische Bundesanstalt. Hrsg. von Physikalisch-Technische Bundesanstalt. URL: `http://www.ptb.de/cms/ptb/fachabteilungen/abt4/fb-44/ag-442/verbreitung-der-gesetzlichen-zeit/dcf77.html`.

5 Anhang

Anhang 1, Codierung DCF77

Bit-Nummer		Bedeutung
0		Start einer neuen Minute (ist immer „0“)
1-14		Bis Mai 1977: Differenz UT1−UTC als vorzeichenbehaftete Zahl Bis November 2006: Betriebsinformationen der PTB (meist alle 14 Bits null) Seit Ende 2006: Wetterinformationen der Firma MeteoTime sowie Informationen des Katastrophenschutzes
15		Rufbit (bis Mitte 2003 Reserveantenne)
16		„1“: Am Ende dieser Stunde wird MEZ/MESZ umgestellt.
17		„1“: MESZ („0“: MEZ)
18		„1“: MEZ („0“: MESZ)
19		„1“: Am Ende dieser Stunde wird eine Schaltsekunde eingefügt.
20		Beginn der Zeitinformation (ist immer „1“)
21	**Minuten (Einser)**	Bit für 1
22		Bit für 2
23		Bit für 4
24		Bit für 8
25	**Minuten (Zehner)**	Bit für 10
26		Bit für 20
27		Bit für 40
28		ParitätsBit Minute
29	**Stunde (Einer)**	Bit für 1
30		Bit für 2
31		Bit für 4
32		Bit für 8
33	**Stunde (Zehner)**	Bit für 10
34		Bit für 20
35		ParitätsBit Stunde
36	**Kalendertag (Einer)**	Bit für 1
37		Bit für 2
38		Bit für 4
39		Bit für 8
40	**Kalendertag (Zehner)**	Bit für 10
41		Bit für 20
42	**Wochentag**	Bit für 1
43		Bit für 2
44		Bit für 4
45	**Monatsnummer (Einer)**	Bit für 1
46		Bit für 2
47		Bit für 4
48		Bit für 8
49	**Monatsnummer (Zehner)**	Bit für 10

Anhang 1, Codierung DCF77

50	**Jahr (Einer)**	Bit für 1
51		Bit für 2
52		Bit für 4
53		Bit für 8
54	**Jahr (Zehner)**	Bit für 10
55		Bit für 20
56		Bit für 40
57		Bit für 80
58		ParitätsBit Datum

```
#  ANHANG 2


"""
Embedded Python Blocks:

Each time this file is saved, GRC will instantiate the first class it finds
to get ports and parameters of your block. The arguments to __init__  will
be the parameters. All of them are required to have default values!
"""

import numpy as np
from gnuradio import gr


class blk(gr.basic_block):  # basic Block -> input ungleich output


    def __init__(self, SampleDelay=1000):
        """arguments to this function show up as parameters in GRC"""
        gr.basic_block.__init__(
            self,
            name='DCF77 Demodulator',
            in_sig=[np.float32],
            out_sig=[np.float32],
        )


        self.SampleDelay = SampleDelay

    self.set_history(2)
    self.state=0
    self.counter=0
    self.Bitcounter = -1    # -1, da erst bei Start einer Minute wieder 0, Zeigt aktuelles Bit
    self.Jahr = 0
    self.Monat = 0
    self.Wochentag = 0
    self.Kalendertag = 0
    self.Stunde = 0
    self.Minute = 0
    self.day = 0
    def general_work(self, input_items, output_items):
        """example: multiply with constant"""

    in0=input_items[0]
    out0=output_items[0]
    outcounter=0
    for k in range(1,len(in0)):


        if (in0[k] - in0[k-1] < 0):                 #Fallende Flanke erkannt?


            self.state=1                    #state = 1, Fallende Flanke erkannt
            self.counter=self.SampleDelay           #Counter starten

        if self.state==1 and self.counter == 0:         #Status nach fallender Flanke

            if self.Bitcounter == 59:           #Bitcounter zurcksetzen
                self.Bitcounter = 0

            self.state=0

            #Logische 1 von DCF77, Start erst wenn Minute erkannt wurde (Bitcounter >= 0)
            if in0[k] == 0 and self.Bitcounter >= 0:

                # Beginn Zuweisung/Errechnen des Datums/Uhrzeit
                if self.Bitcounter == 54:
                    self.Jahr = self.Jahr + 10
```

```
                if self.Bitcounter == 53:
                    self.Jahr = self.Jahr + 8
                if self.Bitcounter == 52:
                    self.Jahr = self.Jahr + 4
                if self.Bitcounter == 51:
                    self.Jahr = self.Jahr + 2
                if self.Bitcounter == 50:
                    self.Jahr = self.Jahr + 1
                if self.Bitcounter == 49:
                    self.Monat = self.Monat + 10
                if self.Bitcounter == 48:
                    self.Monat = self.Monat + 8
                if self.Bitcounter == 47:
                    self.Monat = self.Monat + 4
                if self.Bitcounter == 46:
                    self.Monat = self.Monat + 2
                if self.Bitcounter == 45:
                    self.Monat = self.Monat + 1
                if self.Bitcounter == 44:
                    self.Wochentag = self.Wochentag + 4
                if self.Bitcounter == 43:
                    self.Wochentag = self.Wochentag + 2
                if self.Bitcounter == 42:
                    self.Wochentag = self.Wochentag + 1
                if self.Bitcounter == 41:
                    self.Kalendertag = self.Kalendertag + 20
                if self.Bitcounter == 40:
                    self.Kalendertag = self.Kalendertag + 10
                if self.Bitcounter == 39:
                    self.Kalendertag = self.Kalendertag + 8
                if self.Bitcounter == 38:
                    self.Kalendertag = self.Kalendertag + 4
                if self.Bitcounter == 37:
                    self.Kalendertag = self.Kalendertag + 2
                if self.Bitcounter == 36:
                    self.Kalendertag = self.Kalendertag + 1
                if self.Bitcounter == 34:
                    self.Stunde = self.Stunde + 20
                if self.Bitcounter == 33:
                    self.Stunde = self.Stunde + 10
                if self.Bitcounter == 32:
                    self.Stunde = self.Stunde + 8
                if self.Bitcounter == 31:
                    self.Stunde = self.Stunde + 4
                if self.Bitcounter == 30:
                    self.Stunde = self.Stunde + 2
                if self.Bitcounter == 29:
                    self.Stunde = self.Stunde + 1
                if self.Bitcounter == 27:
                    self.Minute = self.Minute + 40
                if self.Bitcounter == 26:
                    self.Minute = self.Minute + 20
                if self.Bitcounter == 25:
                    self.Minute = self.Minute + 10
                if self.Bitcounter == 24:
                    self.Minute = self.Minute + 8
                if self.Bitcounter == 23:
                    self.Minute = self.Minute + 4
                if self.Bitcounter == 22:
                    self.Minute = self.Minute + 2
                if self.Bitcounter == 21:
                    self.Minute = self.Minute + 1



                self.Bitcounter = self.Bitcounter + 1

                out0[outcounter] = 1

            if in0[k] == 1:      # logische 0 -> DCF77

                out0[outcounter] = 0
```

```
142                 self.Bitcounter = self.Bitcounter + 1
143             outcounter=outcounter+1
144
145         if self.state==0 and self.counter < -7400:  # Erstes Bit erkannt (Keine
            Absenkung auf 15% der Amplitude)
146
147             print("Start/Stop")
148             self.counter=0
149             out0[outcounter]=-1
150             outcounter=outcounter+1
151             self.Bitcounter = 0         # Erstes Bit
152
153         self.counter=self.counter-1
154
155     self.consume(0,len(in0))
156     if self.Wochentag == 1:
157         self.day = 'Montag'
158     elif self.Wochentag == 2:
159         self.day = 'Dienstag'
160     elif self.Wochentag == 3:
161         self.day = 'Mittwoch'
162     elif self.Wochentag == 4:
163         self.day = 'Donnerstag'
164     elif self.Wochentag == 5:
165         self.day = 'Freitag'
166     elif self.Wochentag == 6:
167         self.day = 'Samstag'
168     elif self.Wochentag == 7:
169         self.day = 'Sonntag'
170
171
172     if self.Bitcounter == 59:       #Uhrzeit ausgeben
173         print("Zeit/Datum:  "
174               + '%0.2d' %(self.Stunde) + ":" + '%0.2d' %(self.Minute) + ",  " +
                  str(self.day) + " der " +
175         '%0.2d' %(self.Kalendertag) + "." + '%0.2d' %(self.Monat) + "." +
            str(self.Jahr) )
176
177
178         self.Bitcounter == -1       # Letztes Bit erreicht
179
180     return outcounter
181
182
183
184     def forecast(self, noutput_items, ninput_items_required):
185         """
186        forecast is only called from a general block
187        this is the default implementation
188        """
189         ninput_items_required[0] = noutput_items
190     return
191
192
193         return
194
```