In [1]:

```
# you might need to pip install the bitstring package. Here, take this:
import sys

!"{sys.executable}" -m pip install bitstring
```

```
Requirement already satisfied: bitstring in c:\users\hexst\scoop\apps\python35
\3.5.4\lib\site-packages (3.1.6)
```

In [2]:

```
import matplotlib.pyplot as plt
import scipy
from scipy import signal
from scipy.signal import chirp
import numpy
import numpy as np
import scipy.io.wavfile
import bitstring
```

PWM GPIO

We're going to attempt to bit-bang a GPIO from the PRU on the BBB. This amounts to turning the GPIO 'on' and 'off' at specific delays to approximate the train of chirps that are expected by the receiving devices on PLC. The methods range from 'dumb' to 'way to clever.' We'll start with the dumb stuff first.

The Chirps

From previous investigations, there are a couple way to create chirps which we've experimented with and found work on both diagnostic tools and ECUs on PLC. We'll define both here for later reference. These are imported from `J2497_common.py`

In [154]:

```
def generate_single_chirp(samp_rate):
    wave = numpy.hstack((
        numpy.tile(numpy.hstack((
                chirp(numpy.linspace(     0,  63E-6,  int(63E-6 * samp_rate))
                        f0=203E3, f1=400E3, t1=63E-6, phi=-90, method='linear
                chirp(numpy.linspace( 63E-6,  67E-6,   int(4E-6 * samp_rate))
                        f0=400E3, f1=100E3, t1=67E-6, phi=-90, method='linear
                chirp(numpy.linspace( 67E-6, 100E-6,  int(33E-6 * samp_rate))
                        f0=100E3, f1=200E3,t1=100E-6, phi=-90, method='linear
        )), 1)
    ))
    target_len = int(100e-6 * samp_rate)
    wave = numpy.append(wave, numpy.zeros(numpy.max([0, target_len - len(wave
    return wave

def generate_single_chirp_alt(samp_rate):
    wave = numpy.hstack((
        numpy.tile(numpy.hstack((
                1.0*chirp(numpy.linspace(     0,  63E-6,  int(63E-6 * samp_ra
                        f0=203E3, f1=394E3, t1=63E-6, phi=-90, method='linear
                1.0*chirp(numpy.linspace( 63E-6,  67E-6,   int(4E-6 * samp_ra
                        f0=400E3, f1=100E3, t1=67E-6, phi=-90, method='linear
                chirp(numpy.linspace( 67E-6, 100E-6,  int(33E-6 * samp_rate))
                        f0=1E3, f1=216E3,t1=100E-6, phi=-30, method='linear')
        )), 1)
    ))
    target_len = int(100e-6 * samp_rate)
    wave = numpy.append(wave, numpy.zeros(numpy.max([0,target_len - len(wave)
    return wave
```

The BBB PRU runs at 200MHz and we have a `__delay_cycles()` function there which lets us execute delays at that resolution. So our PWM sample rate can effictively be 200MHz

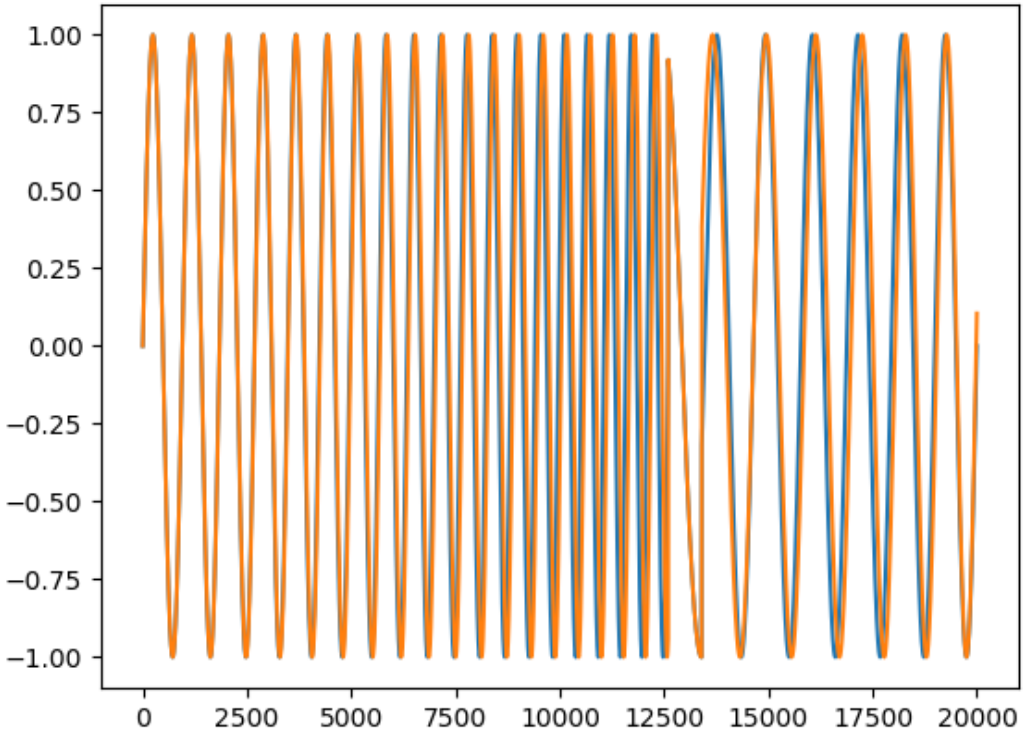
In [155]:

```
pwm_samp_rate = int(200E6)
```

At that sample rate the chirps -- default and alt -- look like this:

In [156]:

```
%matplotlib notebook
plt.plot(generate_single_chirp(pwm_samp_rate))
plt.plot(generate_single_chirp_alt(pwm_samp_rate))
plt.show()
```



The Dumbest Way (First)

The first way to try this is 'bang-bang' control: when the target chirp is above zero turn the GPIO on, when it crosses to negative then turn it off. This is not *good* PWM. But it might be good enough.

Aside: PRU PWM Implementation

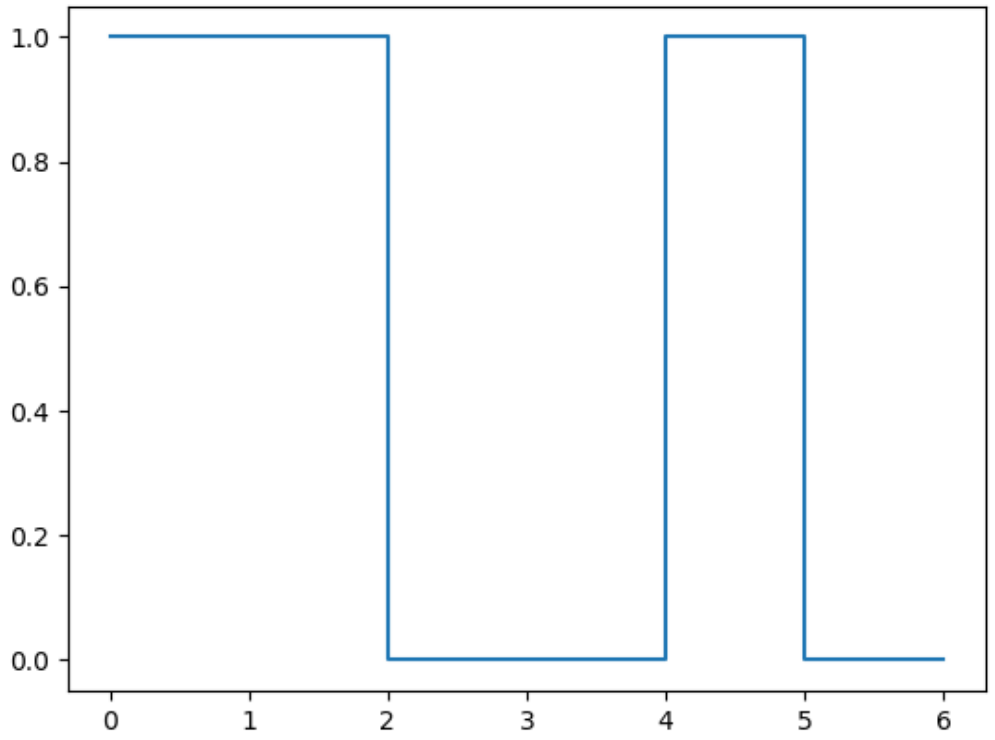
For even the most complicated of PWM synthesis we can think of at present, the PRU GPIO toggling implementation could be realized as a simple lop that sleeps for each subsequent duration in an array and toggles. i.e. The ARM host creates an array of 'sleep values' based on the select PWM synthesis method; the PRU receives this sleep array and processes it by turning-on the GPIO then sleeping for the number of cycles given by the next value in the array, the turning it off and sleeping etc.

```
while ! array_from_ARM_host.empty():
    gpio_on()
    __delay_cycles(array_from_ARM_host.pop())
    gpio_off()
    __delay_cycles(array_from_ARM_host.pop())
```

We will assume the PRU is implemented this way and so let's create a function to visualize the operation of PRU if it were implemeted this way.

In [157]:

```
def plot_viz_pru(array_from_ARM_host):
    times = np.zeros(0, np.uint32)
    vals = np.zeros(0, np.float32)

    val = 1
    time = 0
    vals = np.append(vals, val)
    times = np.append(times, time)

    pos = None
    for pos in range(len(array_from_ARM_host) - 1):
        sleep = array_from_ARM_host[pos]
        time = time + sleep

        vals = np.append(vals, val)
        times = np.append(times, time)

        val = 0 if val != 0 else 1
        vals = np.append(vals, val)
        times = np.append(times, time)

    if not pos is None:
        vals = np.append(vals, val)
        times = np.append(times, time + array_from_ARM_host[pos+1])

    plt.plot(times, vals)

%matplotlib notebook
plot_viz_pru([2,2,1,1])
plt.show()
```

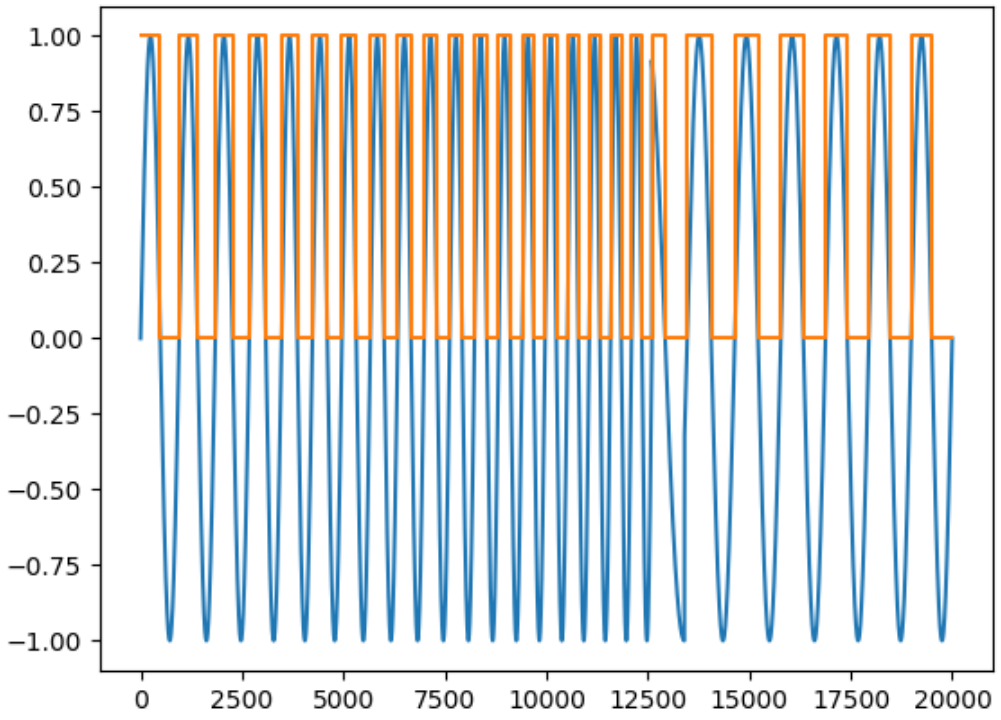


Doing the Dumbest Way

Back to being dumb. We'll walk all the samples we're given and whenever they cross zero, we'll mark a 'sleep' time. At the beginning, if the initial direction is down we will add a zero-sleep as a cheat. Then we'll plot the result.

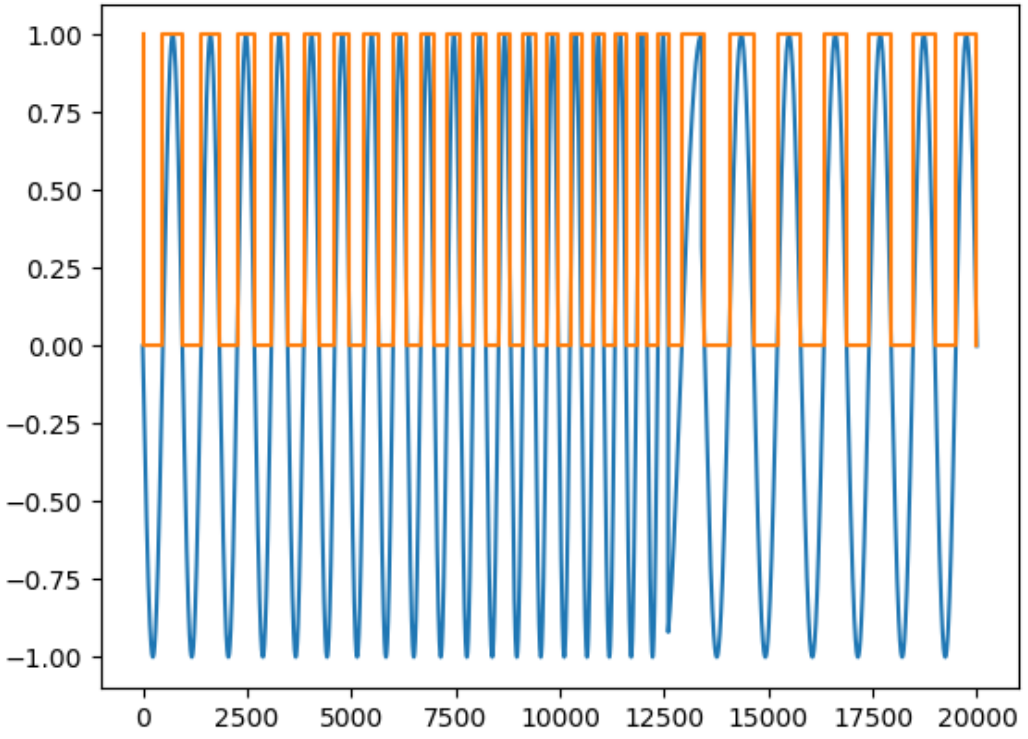
In [158]:

```
def dumbest_pwm(signal):
    sleeps = np.zeros(0, np.uint32)
    if signal[0] < 0:
        sleeps = np.append(sleeps, 0)
        state = 0
    else:
        state = 1

    prev_t = 0
    for t in range(len(signal)):
        if state == 0:
            if signal[t] > 0:
                state = 1
                sleeps = np.append(sleeps, t - prev_t)
                prev_t = t
        else:
            if signal[t] <= 0:
                state = 0
                sleeps = np.append(sleeps, t - prev_t)
                prev_t = t

    sleeps = np.append(sleeps, len(signal) - prev_t)

    if len(sleeps) % 2 == 1:
        sleeps = np.append(sleeps, 0)

    return sleeps

%matplotlib notebook
plt.plot(generate_single_chirp(pwm_samp_rate))
plot_viz_pru(dumbest_pwm(generate_single_chirp(pwm_samp_rate)))
plt.show()
```


1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
2500
5000
7500
10000
12500
15000
17500
20000

In [159]:

```
%matplotlib notebook
plt.plot(-1 * generate_single_chirp(pwm_samp_rate))
plot_viz_pru(dumbest_pwm(-1 * generate_single_chirp(pwm_samp_rate)))
plt.show()
```

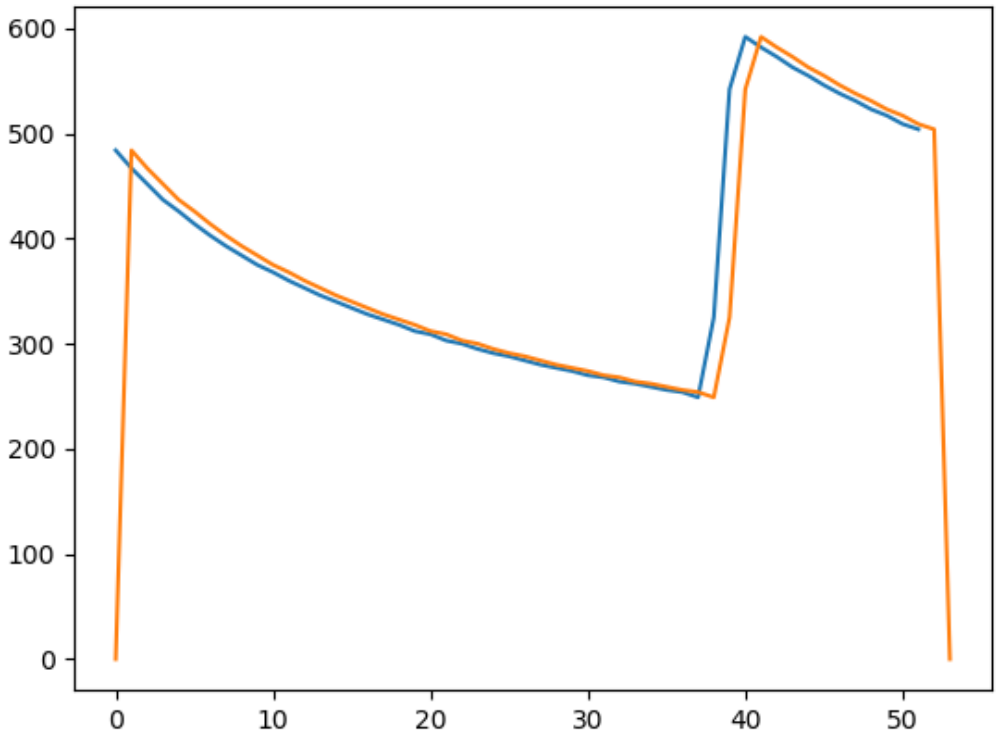


Chirp Trains

When we generate PLC frames we will need to create trains of these chirps. In the preamble there will need to be inter-chirp delays as well.

If both arrays are "the same" the PRU can store one array of sleeps. The inverse symbol array of sleeps is the same as the normal symbol but with a zero-sleep prepended.

In [160]:
```
1 symbol = generate_single_chirp(pwm_samp_rate)
2 %matplotlib notebook
3 plt.plot(dumbest_pwm(symbol))
4 plt.plot(dumbest_pwm(-1 * symbol))
5 plt.show()
```



In [10]:
```
1 np.array_equiv( np.append(np.zeros(1, np.uint32), dumbest_pwm(symbol)),  dumb
```

Out[10]: False

In [161]:
```
1 len(dumbest_pwm(symbol))
```

Out[161]: 52

In [162]:
```
1 len(dumbest_pwm(-1 * symbol))
```

Out[162]: 54

The PRU code can store an array of the sleeps above and 'emit' each while looping over the payload bits. In the preamble only regular symbols or silence is emitted at symbol length of 114 us. In the body both regular and inverted symbols are emitted at 100us symbol length.

The total sleeps in the PWM sleep array calculated for the chirp should be 100 us

In [163]:
```
1 np.sum(dumbest_pwm(symbol))/pwm_samp_rate == 100E-6
```

Out[163]: True

The preamble emitting should be something like this pseudocode:

```
for bit in preamble_bits:
    if bit is True:
        emit_symbol()
        __delay_cycles(14E-6 * 200E6)
    else:
        gpio_off()
        __delay_cycles(114E-6 * 200E6)
```

The body emitting should be something like this pseudocode:

```
for bit in payload_bits:
    if bit is True:
        emit_symbol()
    else:
        -1 * emit_symbol()
#bus access time idle
gpio_off()
__delay_cycles(12 * 114E-6 * 200E6)
```

Let's import some of the J2497 functions for generating the bit streams from the `J2497_common.py` file.

In [164]:

```
def get_preamble_bits(preamble_mid):
    mid_bits = bitstring.BitArray(bytes=preamble_mid)
    mid_bits.reverse()
    mid_bits.prepend(bitstring.ConstBitArray(bin='00'))
    mid_bits.prepend(bitstring.ConstBitArray(bin='0'))
    mid_bits.append(bitstring.ConstBitArray(bin='1'))
    return mid_bits

def get_checksum_bits(payload):
    msg = str(bitstring.ConstBitArray(bytes=payload).bin)
    checksum = 0
    for n in range(0,len(msg),8):
        checksum = checksum + int(msg[n:n+8],2)

    # Two's Complement (10)
    binint = int("{0:b}".format(checksum))             # Convert to binary (1
    flipped = ~binint                                  # Flip the bits (-1011
    flipped += 1                                       # Add one (two's compl
    intflipped=int(str(flipped),2)                     # Back to int (-10)
    intflipped = ((intflipped + (1 << 8)) % (1 << 8))  # Over to binary (246)
    intflipped = '{0:08b}'.format(intflipped)          # Format to one byte (

    checksum_bits = bitstring.BitArray(bin=intflipped)
    return checksum_bits

def get_payload_bits(payload):
    payload_bits = bitstring.BitArray()

    payload_bits.append(bitstring.ConstBitArray(bin='11111'))
    for b_int in bytes(payload):
        b_bytes = bytes([b_int])
        b_bits = bitstring.BitArray(bytes=b_bytes)
        b_bits.reverse()

        payload_bits.append(bitstring.ConstBitArray(bin='0')) # start bit
        payload_bits.append(b_bits) # bit-reversed byte
        payload_bits.append(bitstring.ConstBitArray(bin='1')) # stop bit

    checksum_bits = get_checksum_bits(payload)
    checksum_bits.reverse()

    payload_bits.append(bitstring.ConstBitArray(bin='0'))
    payload_bits.append(checksum_bits)
    payload_bits.append(bitstring.ConstBitArray(bin='1'))

    payload_bits.append(bitstring.ConstBitArray(bin='1111111'))

    return payload_bits
```

Just a couple simple tests to make sure these are operating as expected in the notebook

In [165]:

```
expected = bitstring.ConstBitArray(bin='000010100001')
get_preamble_bits(b'\x0a') == expected
```

Out[165]: True

In [166]:

```
expected = bitstring.ConstBitArray(bin='11111001010000100000000010011011111111
get_payload_bits(b'\x0a\x00') == expected
```

Out[166]: True

We'll also need to be able to create PLC signal waveforms to plot against. Let's define some utility functions -- partly from `J2497_common.py`

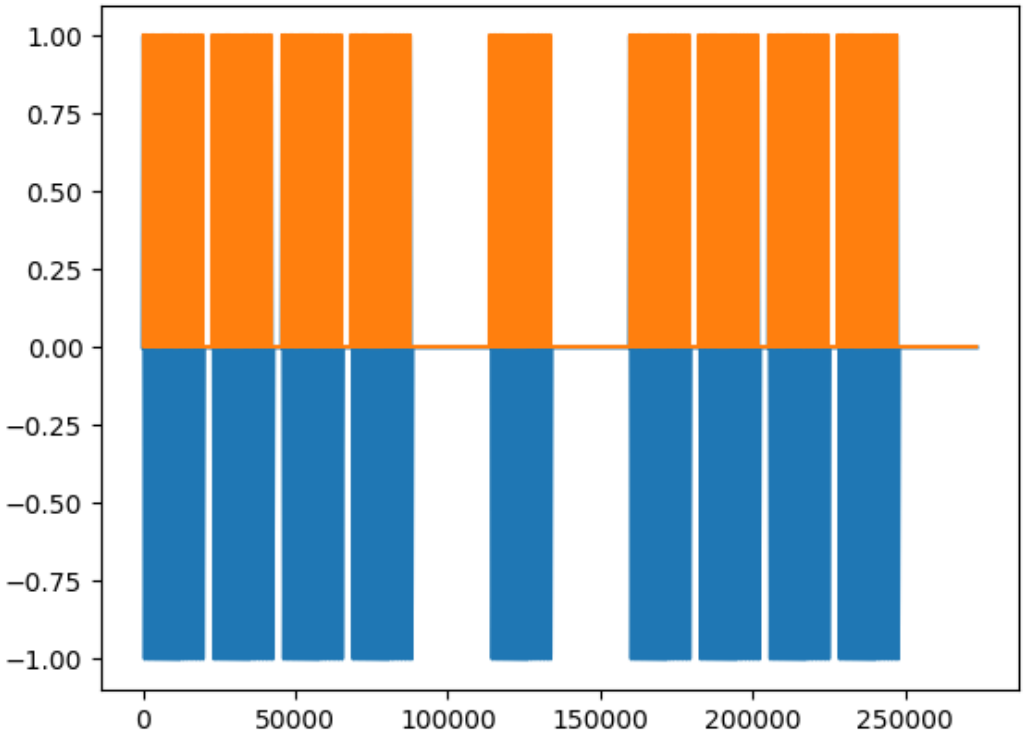
In [167]:

```
def generate_preamble_signal(j1708_message, samp_rate):
    chirp = generate_single_chirp(samp_rate)

    wave = numpy.zeros(0, numpy.float32)

    j2497_preamble = chr(bytes(j1708_message)[0]).encode('latin-1')
    j2497_preamble_bits = get_preamble_bits(j2497_preamble)
    for n in j2497_preamble_bits:
        if not n:
            wave = numpy.append(wave, chirp)
            wave = numpy.append(wave, numpy.zeros(int(samp_rate * 114e-6) - l
        else:
            wave = numpy.append(wave, numpy.zeros(int(samp_rate * 114e-6)))
    return wave

def generate_payload_signal(j1708_message, samp_rate):
    chirp = generate_single_chirp(samp_rate)

    wave = numpy.zeros(0, numpy.float32)

    j2497_payload_bits = get_payload_bits(j1708_message)
    for n in j2497_payload_bits:
        if n:
            wave = numpy.append(wave, chirp)
        else:
            wave = numpy.append(wave, chirp * -1)

    #append a minimum-length bus access time idle period so that returns from
    bus_access_silence = numpy.zeros(int(114e-6 * samp_rate) * 12, numpy.floa
    wave = numpy.append(wave, bus_access_silence)

    return wave

preamble_test = generate_preamble_signal(b'\x0a', pwm_samp_rate)
payload_test = generate_payload_signal(b'\x0a\x00', pwm_samp_rate)
```

Let's define some functions to create the sleeps arrays for preamble and payload

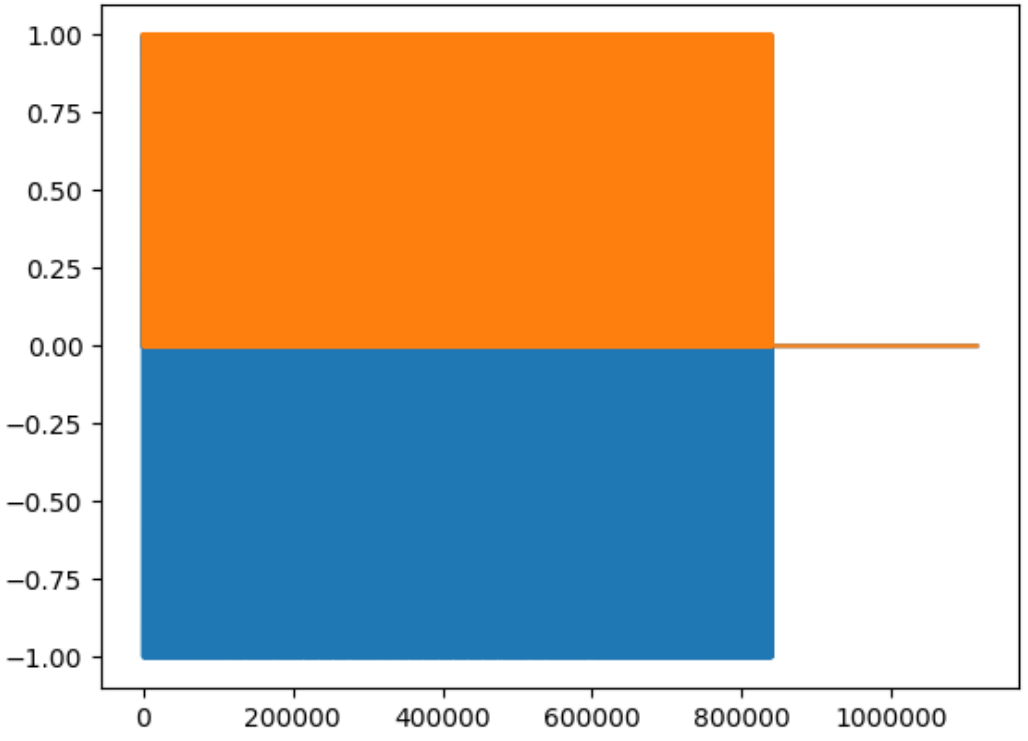
In [168]:

```
def generate_preamble_sleeps(j1708_message, samp_rate):
    chirp_sleeps = dumbest_pwm(generate_single_chirp(pwm_samp_rate))
    sleeps = np.zeros(0, np.uint32)

    j2497_preamble = chr(bytes(j1708_message)[0]).encode('latin-1')
    j2497_preamble_bits = get_preamble_bits(j2497_preamble)
    for n in j2497_preamble_bits:
        if not n:
            sleeps = numpy.append(sleeps, chirp_sleeps)
            sleeps.flat[-1] = sleeps.flat[-1] + int(114e-6 * samp_rate) - np.
        else:
            sleeps.flat[-1] = sleeps.flat[-1] + int(samp_rate * 114e-6) #  ex

    return sleeps

%matplotlib notebook
plt.plot(generate_preamble_signal(b'\x0a\x00', pwm_samp_rate))
plot_viz_pru(generate_preamble_sleeps(b'\x0a\x00', pwm_samp_rate))
plt.show()
```



In [169]:

```
def generate_payload_sleeps(j1708_message, samp_rate):
    symbol = generate_single_chirp(pwm_samp_rate)
    pos_chirp_sleeps = dumbest_pwm(symbol)
    neg_chirp_sleeps = dumbest_pwm(-1 * symbol)

    sleeps = np.zeros(0, np.uint32)

    j2497_payload_bits = get_payload_bits(j1708_message)
    for n in j2497_payload_bits:
        if n:
            sleeps = numpy.append(sleeps, pos_chirp_sleeps)
        else:
            sleeps = numpy.append(sleeps, neg_chirp_sleeps)


    #append a minimum-length bus access time idle period so that returns from
    sleeps.flat[-1] = sleeps.flat[-1] + int(114e-6 * samp_rate) * 12 #  exten

    return sleeps

payload = b'\x0a\x00'

%matplotlib notebook
plt.plot(generate_payload_signal(payload, pwm_samp_rate))
plot_viz_pru(generate_payload_sleeps(payload, pwm_samp_rate))
plt.show()
```



So it seems we have a reasonable plan to chain together any PWM decomposition of the chirps. In this case we were testing with the 'dumbest' PWM.

PRU Implementation

Sending an array of inter-bit delays from the ARM to the PRU could be really big...

Just the preamble is pretty big. It is always the same size and, in number of bits:

In [170]:
```
1 len(get_preamble_bits(b'\xff'))
```

Out[170]: 12

or in number of sleeps:

In [171]:
```
1 len(generate_preamble_sleeps(b'\xff', pwm_samp_rate))
```

Out[171]: 156

This could be *even bigger* if we move to smarter PWM methods.

Since the PWM emitted is a repetition of the same sleeps obtained by analysis of a chirp, we can instead send the bit sequence from the ARM to the PRU. This still let's us keep the 'secret' sauce of creating chirps more hidden in the PRU code as well.

The maximum bit sequence length for preamble is shown above: 12. For the payload the length is variable and could be unlimited if the vehicle is not moving. It's probably safe to assume we would never create something longer than 255 bytes. The existing PRU code uses 42.

At 255 bytes payload, the buffer size needed for the bits represented as bytes would be:

In [172]:
```
1 len(get_payload_bits(b'\xff'*255))
```

Out[172]: 2572

As a stream of bits in an array of bytes this would be 8 times smaller

In [173]:
```
1 len(get_payload_bits(b'\xff'*255)) / 8
```

Out[173]: 321.5

The PRUs have only 8KiB of data memory. We need to fit the sleeps array for the chirp in there as well as the produce-consume ring buffer information and a length field for number of bits of payload and a bitstream for preable bit stream too.

How big does the array of chirp sleeps need to be? Using the dumbest PWM method the maximum sleep value and length of sleeps are:

In [174]:
```
sleeps = dumbest_pwm(generate_single_chirp(pwm_samp_rate))
print(str(np.max(sleeps)))
print(str(len(sleeps)))
```

```
592
52
```

So the chirp sleeps would need to be 16bit unsigned * 52. This means we could fit a maximumringbuffer of depth calculated as follows:

In [175]:
```
import math
```

In [176]:
```
frame_size = 2+ math.ceil(len(get_preamble_bits(b'\xff')) / 8) + math.ceil(le
ring_overhead = 4+4
chirp_sleeps = 1 + 2*len(dumbest_pwm(generate_single_chirp(pwm_samp_rate)))
stack_overhead = 256
(8192-ring_overhead-chirp_sleeps-stack_overhead) / frame_size
```

Out[176]:
```
23.850609756097562
```

, which is great -- we'll only need 16 since that was enough for the J1708 PRU code.

So, the arm host python code will send frames as sequences of bits. The memory structure in the PRU will be something like

```
typedef struct {
    uint16_t volatile bit_length;
    uint8_t volatile packed_bits[322];
} frame_t;

typedef struct {
    uint32_t volatile produce;
    uint32_t volatile consume;
    frame_t volatile frames[16];
} ring_buffer_t;

uint16_t chirp_sleeps[52];
```

Here's the array of sleeps to setup for the chirps

In [177]:
```
dumbest_pwm(generate_single_chirp(pwm_samp_rate))
```

Out[177]:
```
array([484, 467, 452, 437, 426, 414, 403, 393, 384, 375, 368, 360, 353,
       346, 340, 334, 328, 323, 318, 312, 309, 303, 300, 295, 291, 288,
       284, 280, 277, 274, 270, 268, 264, 262, 259, 256, 254, 249, 325,
       542, 592, 582, 573, 563, 555, 546, 538, 531, 523, 517, 509, 504],
      dtype=int64)
```

In [178]:
```
1 dumbest_pwm(-1 * generate_single_chirp(pwm_samp_rate))
```

Out[178]:
```
array([  0, 484, 467, 452, 437, 426, 414, 403, 393, 384, 375, 368, 360,
       353, 346, 340, 334, 328, 323, 318, 312, 309, 303, 300, 295, 291,
       288, 284, 280, 277, 274, 270, 268, 264, 262, 259, 256, 254, 249,
       325, 542, 592, 582, 573, 563, 555, 546, 538, 531, 523, 517, 509,
       504,   0], dtype=int64)
```

But the `__delay_cycles` macro requires an integer constant, so let's generate some code to copy and paste

In [179]:
```
1 def render_pru(sleeps):
2     for i in range(int(len(sleeps)/2)):
3         print('asm("\tset r30, r30, 1");')
4         print('__delay_cycles(%d);' % sleeps[i*2])
5         print('asm("\tclr r30, r30, 1");')
6         print('__delay_cycles(%d);' % sleeps[i*2 + 1])
7
```

For the positive symbol:

In [180]:

```
1 render_pru(dumbest_pwm(generate_single_chirp(pwm_samp_rate)))
```

```
asm("   set r30, r30, 1");
__delay_cycles(484);
asm("   clr r30, r30, 1");
__delay_cycles(467);
asm("   set r30, r30, 1");
__delay_cycles(452);
asm("   clr r30, r30, 1");
__delay_cycles(437);
asm("   set r30, r30, 1");
__delay_cycles(426);
asm("   clr r30, r30, 1");
__delay_cycles(414);
asm("   set r30, r30, 1");
__delay_cycles(403);
asm("   clr r30, r30, 1");
__delay_cycles(393);
asm("   set r30, r30, 1");
__delay_cycles(384);
asm("   clr r30, r30, 1");
__delay_cycles(375);
asm("   set r30, r30, 1");
__delay_cycles(368);
asm("   clr r30, r30, 1");
__delay_cycles(360);
asm("   set r30, r30, 1");
__delay_cycles(353);
asm("   clr r30, r30, 1");
__delay_cycles(346);
asm("   set r30, r30, 1");
__delay_cycles(340);
asm("   clr r30, r30, 1");
__delay_cycles(334);
asm("   set r30, r30, 1");
__delay_cycles(328);
asm("   clr r30, r30, 1");
__delay_cycles(323);
asm("   set r30, r30, 1");
__delay_cycles(318);
asm("   clr r30, r30, 1");
__delay_cycles(312);
asm("   set r30, r30, 1");
__delay_cycles(309);
asm("   clr r30, r30, 1");
__delay_cycles(303);
asm("   set r30, r30, 1");
__delay_cycles(300);
asm("   clr r30, r30, 1");
__delay_cycles(295);
asm("   set r30, r30, 1");
__delay_cycles(291);
asm("   clr r30, r30, 1");
__delay_cycles(288);
asm("   set r30, r30, 1");
__delay_cycles(284);
asm("   clr r30, r30, 1");
__delay_cycles(280);
asm("   set r30, r30, 1");
__delay_cycles(277);
```

```
asm("   clr r30, r30, 1");
__delay_cycles(274);
asm("   set r30, r30, 1");
__delay_cycles(270);
asm("   clr r30, r30, 1");
__delay_cycles(268);
asm("   set r30, r30, 1");
__delay_cycles(264);
asm("   clr r30, r30, 1");
__delay_cycles(262);
asm("   set r30, r30, 1");
__delay_cycles(259);
asm("   clr r30, r30, 1");
__delay_cycles(256);
asm("   set r30, r30, 1");
__delay_cycles(254);
asm("   clr r30, r30, 1");
__delay_cycles(249);
asm("   set r30, r30, 1");
__delay_cycles(325);
asm("   clr r30, r30, 1");
__delay_cycles(542);
asm("   set r30, r30, 1");
__delay_cycles(592);
asm("   clr r30, r30, 1");
__delay_cycles(582);
asm("   set r30, r30, 1");
__delay_cycles(573);
asm("   clr r30, r30, 1");
__delay_cycles(563);
asm("   set r30, r30, 1");
__delay_cycles(555);
asm("   clr r30, r30, 1");
__delay_cycles(546);
asm("   set r30, r30, 1");
__delay_cycles(538);
asm("   clr r30, r30, 1");
__delay_cycles(531);
asm("   set r30, r30, 1");
__delay_cycles(523);
asm("   clr r30, r30, 1");
__delay_cycles(517);
asm("   set r30, r30, 1");
__delay_cycles(509);
asm("   clr r30, r30, 1");
__delay_cycles(504);
```

Then for the negative symbol:

In [181]:
```
1 render_pru(dumbest_pwm(-1 * generate_single_chirp(pwm_samp_rate)))
```

```
__delay_cycles(325);
asm("    set r30, r30, 1");
__delay_cycles(542);
asm("    clr r30, r30, 1");
__delay_cycles(592);
asm("    set r30, r30, 1");
__delay_cycles(582);
asm("    clr r30, r30, 1");
__delay_cycles(573);
asm("    set r30, r30, 1");
__delay_cycles(563);
asm("    clr r30, r30, 1");
__delay_cycles(555);
asm("    set r30, r30, 1");
__delay_cycles(546);
asm("    clr r30, r30, 1");
__delay_cycles(538);
asm("    set r30, r30, 1");
__delay_cycles(531);
asm("    clr r30, r30, 1");
```

To optimize, we are going to avoid 'sending' the predictable prepends and appends of the preamble and payload to the PRU; we'll use a struct like this:

```
#define TX_FRAME_LEN 42
typedef struct  {
    uint8_t volatile length;
    uint8_t volatile preamble; /* all the bits, without prepended 00, 0 or ap
pended 1*/
    uint8_t volatile payload[TX_FRAME_LEN]; /* all the bits, including checks
um, without 11111 prepended or 11111 appended */
} tx_frame_t;

#define TX_RING_BUFFER_LEN 16
typedef struct {
    uint32_t volatile produce;
    uint32_t volatile consume;
    tx_frame_t volatile frames[TX_RING_BUFFER_LEN];
} tx_ring_buffer_t;
```

so here are some functions defined to mimic what we should send to the PRU

In [182]:

```
def get_checksum_bits(payload):
    msg = str(bitstring.ConstBitArray(bytes=payload).bin)
    checksum = 0
    for n in range(0,len(msg),8):
        checksum = checksum + int(msg[n:n+8],2)

    # Two's Complement (10)
    binint = int("{0:b}".format(checksum))             # Convert to binary (1
    flipped = ~binint                                  # Flip the bits (-1011
    flipped += 1                                       # Add one (two's compl
    intflipped=int(str(flipped),2)                     # Back to int (-10)
    intflipped = ((intflipped + (1 << 8)) % (1 << 8))  # Over to binary (246)
    intflipped = '{0:08b}'.format(intflipped)          # Format to one byte (

    checksum_bits = bitstring.BitArray(bin=intflipped)
    return checksum_bits

def get_special_preamble_bits(preamble_mid):
    mid_bits = bitstring.BitArray(bytes=preamble_mid)
    return mid_bits

def get_special_payload_bits(payload):
    payload_bits = bitstring.BitArray()

    for b_int in bytes(payload):
        b_bytes = bytes([b_int])
        b_bits = bitstring.BitArray(bytes=b_bytes)
        b_bits.reverse()

        payload_bits.append(bitstring.ConstBitArray(bin='0')) # start bit
        payload_bits.append(b_bits) # bit-reversed byte
        payload_bits.append(bitstring.ConstBitArray(bin='1')) # stop bit

    checksum_bits = get_checksum_bits(payload)
    checksum_bits.reverse()

    payload_bits.append(bitstring.ConstBitArray(bin='0'))
    payload_bits.append(checksum_bits)
    payload_bits.append(bitstring.ConstBitArray(bin='1'))

    payload_bits.reverse()

    return payload_bits
```

So for example of preamble and payload, we can use

In [183]:

```
p = get_special_preamble_bits(b'\x0a')
print("bits length: %d" % p.len)
print("hex %s" % p[0:32].tobytes().hex())
```

```
bits length: 8
hex 0a
```

In [184]:

```
p = get_special_payload_bits(b'\x0a\x00')
print("bits length: %d" % p.len)
print("padded hex %s" % p[0:32].tobytes().hex())

p = bitstring.BitArray(hex=p.tobytes().hex())
p.byteswap()
print("byte-swapped padded hex %s" % p[0:32].tobytes().hex())
```

```
bits length: 30
padded hex fb200850
byte-swapped padded hex 500820fb
```

Tuning Based on Capture

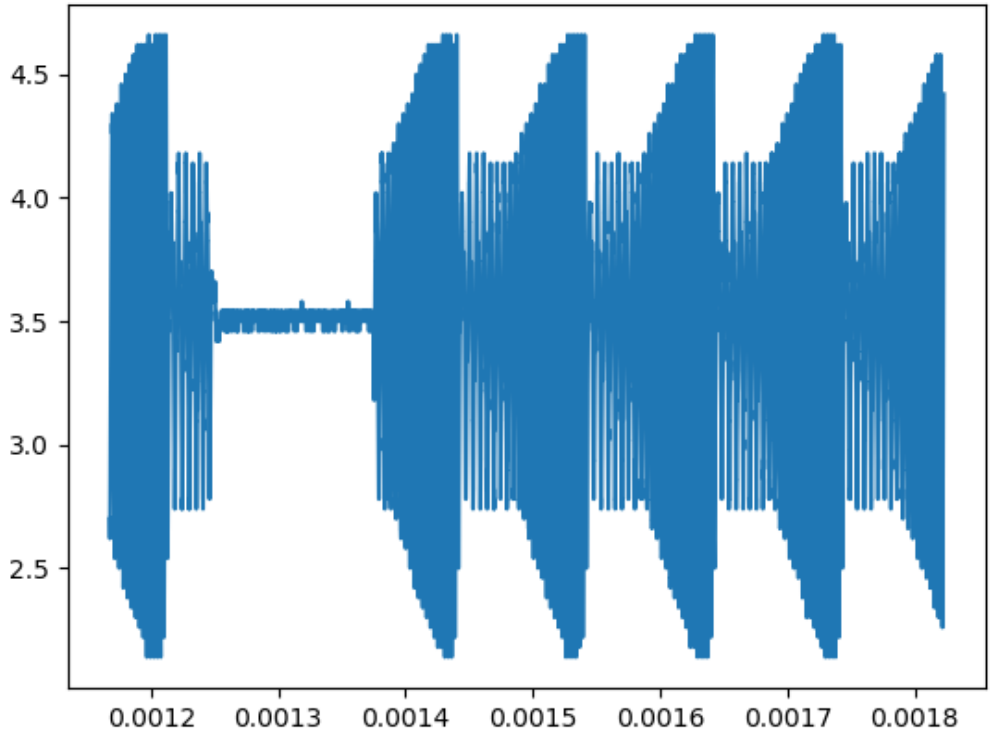
We implemented the above and got an oscilloscope capture at the point after the filters on the DG tech diagnostics adapter.

In [185]:

```
import pandas as pd
```

In [186]:

```
df = pd.read_csv('first_gpiopwm_0a00.csv')

%matplotlib notebook
plt.plot(df['time'], df['voltage'])
plt.show()
```



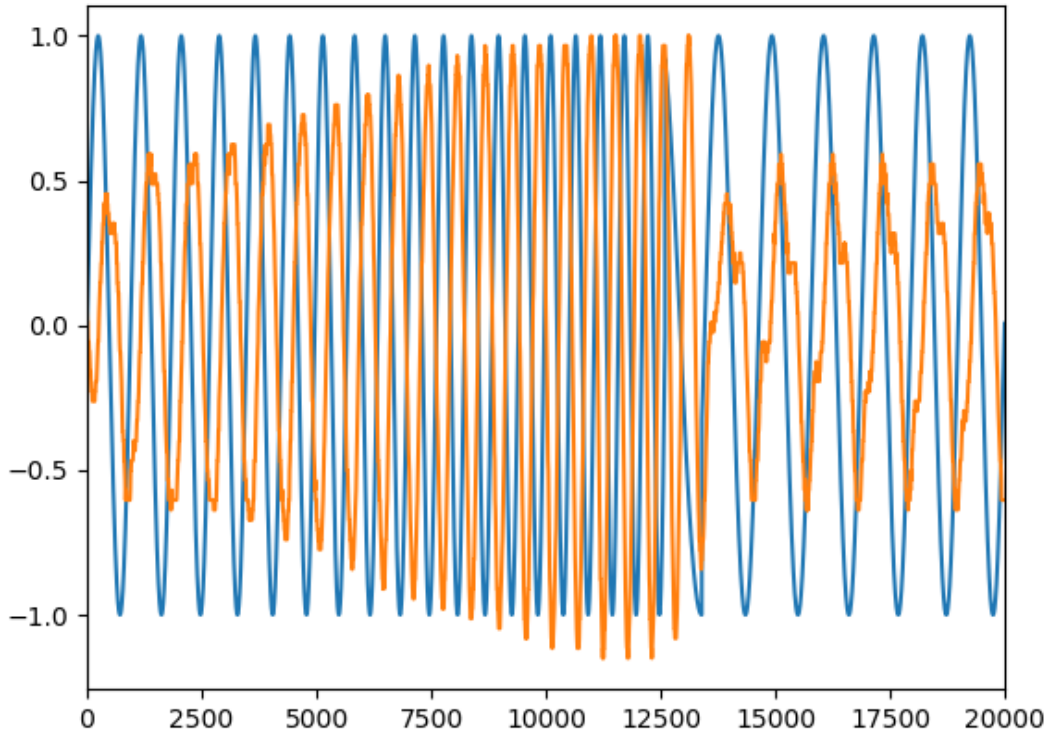
In [37]:

```
def plot_capture1(samp_rate):
    df = pd.read_csv('first_gpiopwm_0a00.csv')
    df = df[ df['time'] >= 0.00137472 ] # manually-selected start of symbol

    #df['voltage'] = df['voltage'] * -1
    df['voltage'] = df['voltage'] - df['voltage'].mean()
    df['voltage'] = df['voltage'] / df['voltage'].max()

    df['time'] = df['time'] * samp_rate
    df['time'] = df['time'] - df['time'].values[0]

    plt.plot(df['time'], df['voltage'])
```

We can overlay one normalized symbol on top of the syntetic symbol

In [187]:

```
%matplotlib notebook
ax1 = plt.plot(generate_payload_signal(payload, pwm_samp_rate))
plot_capture1(pwm_samp_rate)
plt.xlim([0, pwm_samp_rate * 100e-6])
plt.show()
```



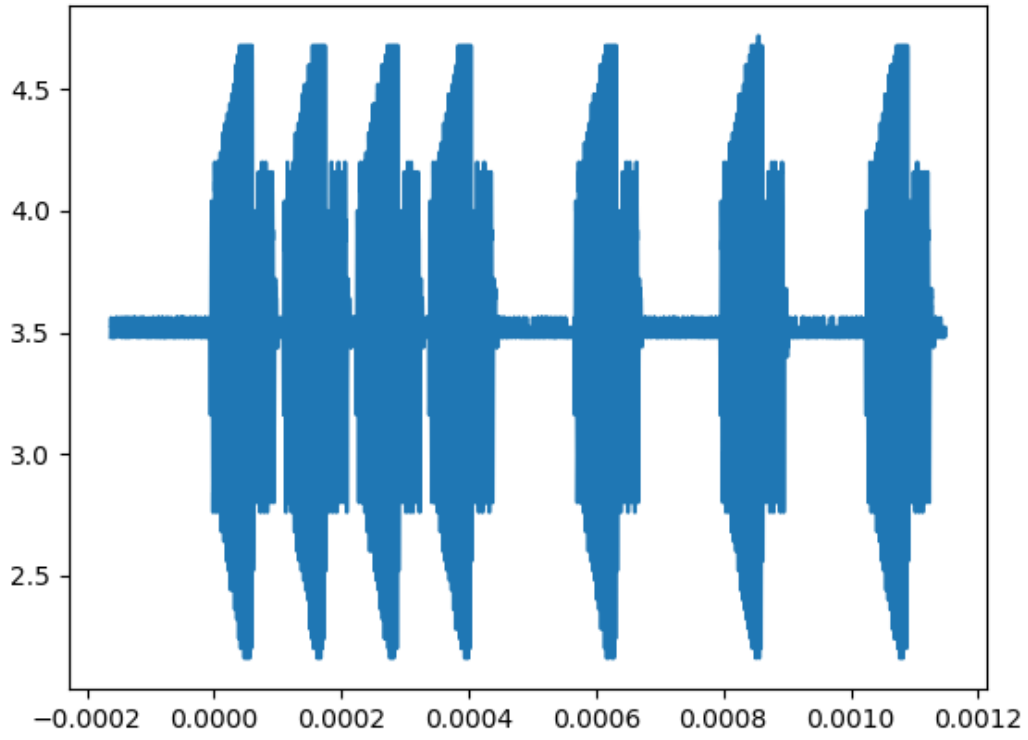
The filtering of the input to the P485 here on the DG tech gives a pretty decent result. We can safely assume that other targets will have filtering roughly as robust.

There does appear to be some 'drift' of the peaks. There is probably a `cycle_delay` and loop overhead that needs to be accounted for in the code.

Tuning via the Preamble

We modified the PRU code to send only the preamble and captured a signal after the DG tech filters as before.

In [188]:
```
df = pd.read_csv('first_gpiopwm_0a00_preambleonly.csv')

%matplotlib notebook
plt.plot(df['time'], df['voltage'])
plt.show()
```

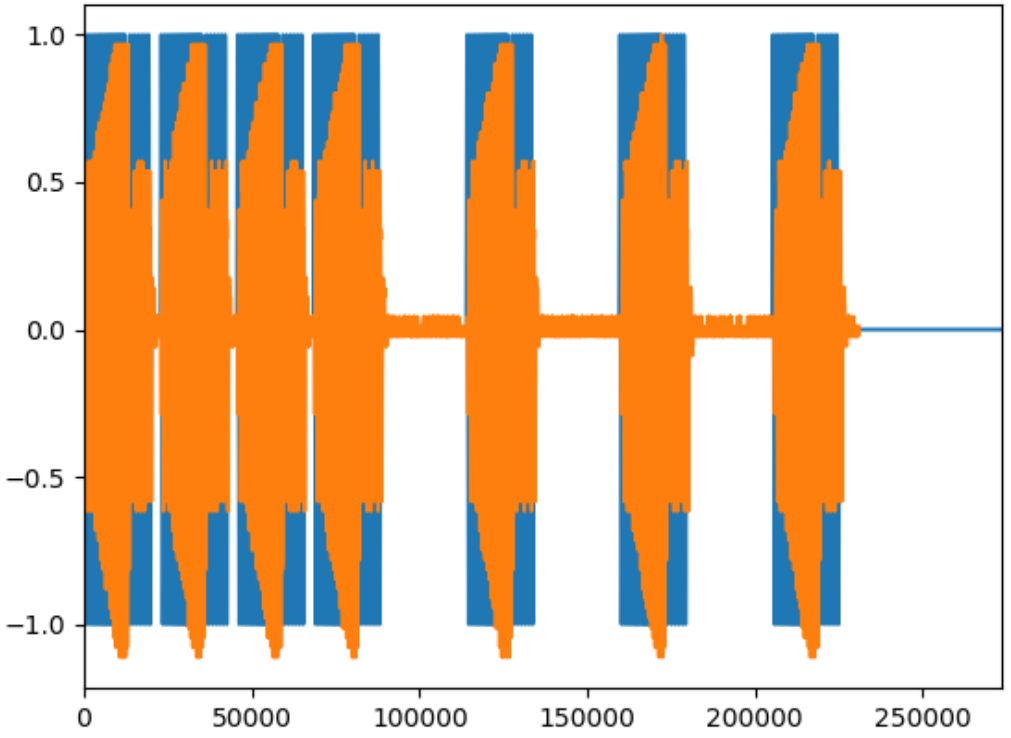


In [40]:
```
def plot_capture2(samp_rate):
    df = pd.read_csv('first_gpiopwm_0a00_preambleonly.csv')
    df = df[ df['time'] >= -6.23963e-6 ] # manually-selected start of symbol

    #df['voltage'] = df['voltage'] * -1
    df['voltage'] = df['voltage'] - df['voltage'].mean()
    df['voltage'] = df['voltage'] / df['voltage'].max()

    df['time'] = df['time'] * samp_rate
    df['time'] = df['time'] - df['time'].values[0]

    plt.plot(df['time'], df['voltage'])
```

We can overlay the capture of preamble-only on-top-of the same synthesized preamble.

In [189]:

```
%matplotlib notebook
plt.plot(generate_preamble_signal(b'\xaa', pwm_samp_rate))
plot_capture2(pwm_samp_rate)
plt.xlim([0, pwm_samp_rate * 114e-6 * 12])
plt.show()
```

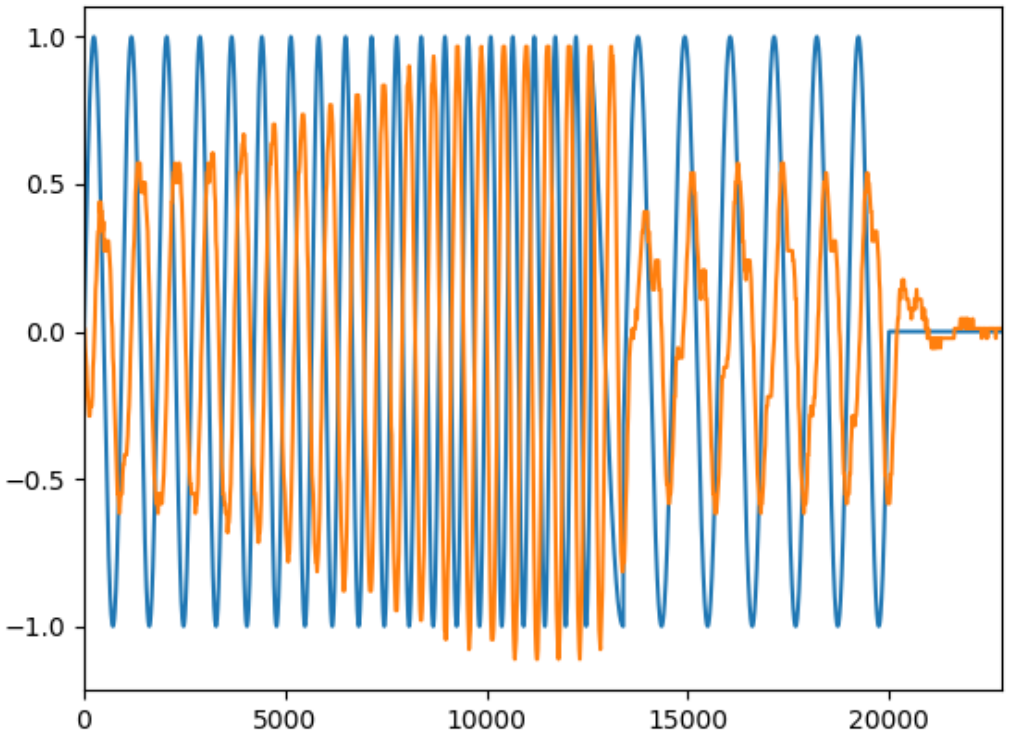


They align pretty well at a high level.

Let's look closer at one single symbol.

In [190]:

```
%matplotlib notebook
plt.plot(generate_preamble_signal(b'\xaa', pwm_samp_rate))
plot_capture2(pwm_samp_rate)
plt.xlim([0, pwm_samp_rate * 114e-6])
plt.show()
```



In [43]:

```
12826-12597
```

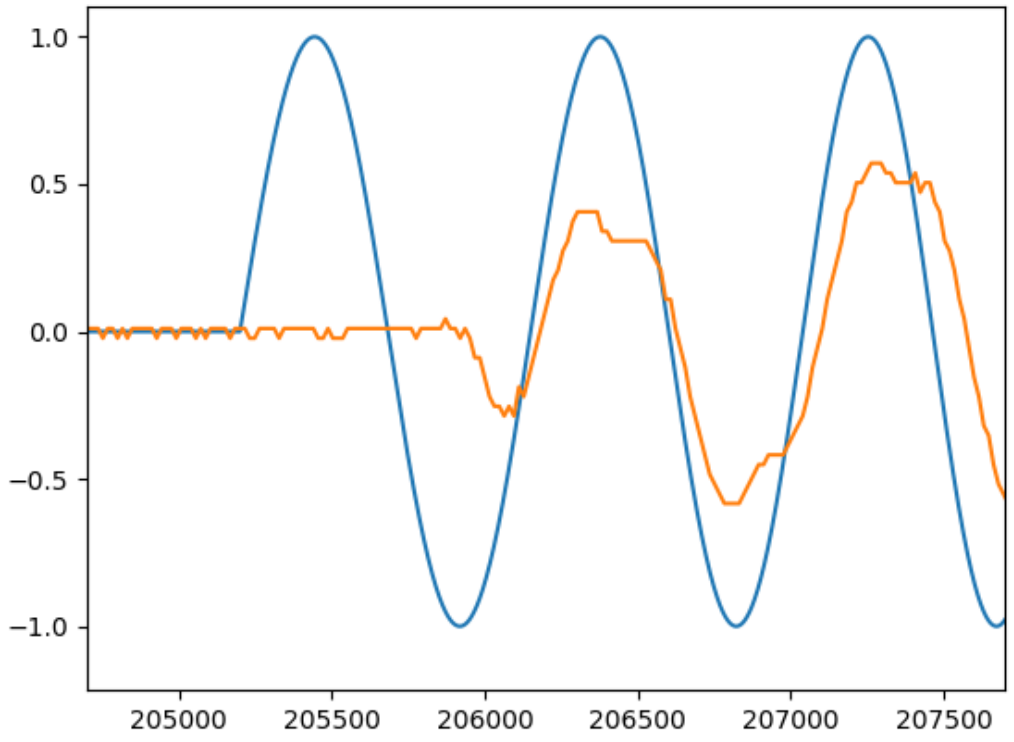
Out[43]: 229

The duration might be a little long.

Let's look at the last (superior symbol)

In [44]:

```
%matplotlib notebook
plt.plot(generate_preamble_signal(b'\xaa', pwm_samp_rate))
plot_capture2(pwm_samp_rate)
plt.xlim([pwm_samp_rate * 114e-6 * 9 - 500, pwm_samp_rate * 114e-6 * 9 - 500
plt.show()
```



By the time we are at the 10th symbol in the preamble, the PWM is slow

In [45]:

```
205920-205201
```

Out[45]: 719

by 719 samples @ 200 MHz

Investigating the Delays

We'll look into the assembly generated by the compiler to see where delays are being introduced that might pile up to ~720 cycles. Things that are potential (and obvious) so should be considered first

1. function call overhead. we hhave judicious use of `__inline` but we did hit a limit where we couldn't inline everything because the chirp emit bitbanging is very long and inlining more than one of those ends up crashing the assembler
2. loop overhead
3. unexpect instruction cycle counts
4. interrupts to the PRU

Let's look at the last thing first. 4) interrupts to the processor should be simple to fix: mask interrupts while emitting PLC symbols.

Looking for 3) unexpected intruction cycle counts: the TI wiki https://processors.wiki.ti.com/index.php/Programmable_Realtime_Unit#Load_.2F_Store_Instructions (https://processors.wiki.ti.com/index.php/Programmable_Realtime_Unit#Load_.2F_Store_Instructions) lists instruction counts for most instructions. Pretty much everything is 1 clock cycle per instruction. There are some exceptions which we will list here

long time instructions	cycle counts
LBBO	`1 + WdCnt` (VBUS) or `2 + WdCnt` (VBUSP)
SBBO	`1 + WdCnt`
LBCO	`1 + WdCnt` (VBUS) or `2 + WdCnt` (VBUSP)
SBCO	`1 + WdCnt`
SCAN	IF fw = fs `2+((fc*fw+3)/4)` ELSE `2+fc` This is a worst case cycle count. Matching scans could take fewer cycles.
SLP	1 to infinity

There is also some unexpected instruction timings when the instructions go off to access DDR as opposed to the 'on-die' PRU memory. c.f. https://groups.google.com/forum/#!topic/beagleboard/6tZfe7kBEOw (https://groups.google.com/forum/#!topic/beagleboard/6tZfe7kBEOw) . In that thread 'Marcelo' offers some measurements of cycle count on instructions when they go out to DDR vs SRAM

long time DRAM or SRAM instructions	Marcelo's cycle counts
LBBO	3
LBCO	3
SBBO	2
SBCO	2

long time DDR instructions	Marcelo's cycle counts
LBBO	43+ (43.3 avg of 1000)

We can use the above cycle count summaries in estimating cycle count of our code.

According to https://github.com/cdsteinkuehler/linuxcnc/blob/MachineKit-ubc/src/hal/drivers/hal_pru_generic/pru_generic.p#L135 (https://github.com/cdsteinkuehler/linuxcnc/blob/MachineKit-ubc/src/hal/drivers/hal_pru_generic/pru_generic.p#L135) any writes to the GPIOs takes 2 cycles but the CPIO logic only updates every 8 cycles. Furthermore, the write instructions to the PGIOs will only stay 2 cycles long for the first 20 in sequence, after that there will be stalls and the write instruction will take 8 cycles. ***We will need to ensure we don't have sleeps tighter than 8 cycles -- this makes the PRU GPIO PWM rate 25 MHz not 200MHz***

As for (2) loop overhead, the PRU compiler implements loops like this:

```
        LDI       r1, 0x02c8            ; [] |507| length

||$C$L6||:
        XXXXX your stuff XXXXX
        SUB       r1, r1, 0x01          ; [] |329| length,length
        QBNE      ||$C$L6||, r1, 0x00   ; [] |329| length
```

So each time through a loop costs 2 cycles and 1 extra cycle for loop setup.

Let's look at function call overhead...

- For a function of , e.g. 42 sized stack

```
        SUB       r2, r2, 0x2a          ; []
        SBBO      &r3.b2, r2, 0, 42     ; []
        XXXXX your stuff XXXXX
        LBBO      &r3.b2, r2, 0, 42     ; []
        ADD       r2, r2, 0x2a          ; []
        JMP       r3.w2                 ; []
```

  it isn't clear why this `emit_pos_symbol` has a stack size of 42 in the first place. Nevertheless the overhead in this case is `1 + 1 + 42/4 + 1 + 1 + 42/4 + 1 + 1`

- For a function of zero stack there is zero overhead

The function above with a 42-deep stack is `emit_pos_symbol` it isn't clear why this function doesn't have zero sized stacked. Neither commenting out some `asm()` statements nor some `delay_cycles` made the function stack smaller.

Then something totally unexpected; the mix of `delay_cycles` and inline asm seems to get `mov` instructions inserted...

```
;------------------------------------------------------------------------
; 191 | __delay_cycles(484);
; 192 | asm("   clr r30, r30, 1");
;------------------------------------------------------------------------
        .newblock
        LDI32    r0, 241
$1:     SUB      r0, r0, 1
        QBNE     $1, r0, 0             ; [] |191|
   clr r30, r30, 1
        MOV       r0.w0, r1.w0          ; []
```

So what to do about it?

A) cycle-count the instructions and adapt the timings to `delay_cycles` so that they are shorter to compensate for this. The problem here could be that occasional DDR accesses end up causing longer than modeled delays

B) Change the code to have wait-sync points where the code sleeps until the cycle counter is at or passed an expected time. We don't know how to code this at present, but it seems possible based on at least forum comments. This alone probably won't be enough when the PRU is emitting the symbols in

the PLC body because the symbols need to be emitted back-to-back. There's no 'big delays' after the symbols so the symbol emitting would need to be written to use wait-sync points throughout.

We need to check the delays for nothing less than 8 cycle sleeps. Otherwise these transitions are lost.

We also need to think about code reuse. We can't inline everything. There needs to be functions. So the functions need to have 'zero delay' in their final statements and callers need to implement the right amount of dealy after their calls depending on where they are called (e.g. in a loop or in straight code).

For B) there is actually an example of how to use the PRU CYCLE register in https://theembeddedkitchen.net/beaglelogic-building-a-logic-analyzer-with-the-prus-part-1/449 (https://theembeddedkitchen.net/beaglelogic-building-a-logic-analyzer-with-the-prus-part-1/449)

```
MOV    R1, CTPPR_0
MOV    R2, 0x00000220      // C28 = 00_0220_00h = PRU0 CFG Registers
SBBO   &R2, R1, 0, 4

LBCO   &R1, C28, 0, 4      // Enable CYCLE counter
SET    R1, 3
SBCO   &R1, C28, 0, 4

LBCO &R1, C28, 0xC, 4     // Load "before" cycle count into R1
// your assembly code here
LBCO &R2, C28, 0xC, 4     // Load "after" cycle count into R2
```

This seems like the ideal timing method but I'm not quite sure how to loop-until value unless we also have a way to jam the cycle counter to 0...

We can't just convert the emit symbol function into a loop so that it can be inlined because `__delay_cycles` is a compiler intrinsic that required the argument be a constant. I think it is the register file being preserved since the `__delay_cycles` emitted by the compiler are using a mix of registers and LDI/LDI32

TODO:

[Y] add a copy-to-stack per frame before emitting symbols

[Y] move toggling to inside of a loop and compensate for loop overhead and possibly compensate for inserted `mov` instructions too

[Y] get to 1-2 cycle function call overhead by zero stack or inlining

[Y] remove the last `delay_cycles` from functions and make the caller wait.

[N] supress interrupts around the symbol emitting

Generate ASM to avoid large stack frame call overhead

AS seen above, functions that make many calls to `delay_cycles` incur a large stack frame overhead as the compiler mixes and matches allocation of registers to the scratch counter for that intrinsic. Let's generate our own assembly function for `emit_pos_symbol` and `emit_neg_symbol` .

The PRU CGT calling convention lists r0,r1,r14-r24 as save-on-call registers. We have some sleeps that are greater than 255 so we will need to use `LDI32` and register from that group that is compatible with that instruction. The following should do for delay cycles:

```
        LDI32    r0, %d
$1:     SUB      r0, r0, 1
        QBNE     $1, r0, 0
```

where we will pass `math.floor(cycles/2 - 1)` for `%d`

NB: whereas http://www.ti.com/lit/ug/spruhv7b/spruhv7b.pdf (http://www.ti.com/lit/ug/spruhv7b/spruhv7b.pdf) clearly stats R0 is a save-on-call register, the assembly generated by function calls in a for loop does not preseve r0 in my testing; r1 seems like a safer choice

In [61]:

```
def render_asm_sleep(cycles):
    print('''\
asm("        .newblock");
asm("        LDI32    r1, %d");
asm("$1:     SUB      r1, r1, 1");
asm("        QBNE     $1, r1, 0");\
    ''' % math.floor(cycles/2 - 1))

def render_asm_emit_pos_symbol(sleeps):
    for i in range(int(len(sleeps)/2)-2):
        print('asm("        SET\tr30, r30, 1");')
        render_asm_sleep(sleeps[i*2] - 1)
        print('asm("        CLR\tr30, r30, 1");')
        render_asm_sleep(sleeps[i*2 + 1] - 1)
    print(    'asm("        SET\tr30, r30, 1");')
    render_asm_sleep(sleeps[len(sleeps) - 2] - 1)
    print('// caller is responsible for %d cycle delay' % sleeps[len(sleeps)

render_asm_emit_pos_symbol(dumbest_pwm(generate_single_chirp(pwm_samp_rate)))

```

```
asm("        QBNE     $1, r1, 0");
asm("        CLR        r30, r30, 1");
asm("        .newblock");
asm("        LDI32    r1, 186");
asm("$1:     SUB      r1, r1, 1");
asm("        QBNE     $1, r1, 0");
asm("        SET        r30, r30, 1");
asm("        .newblock");
asm("        LDI32    r1, 182");
asm("$1:     SUB      r1, r1, 1");
asm("        QBNE     $1, r1, 0");
asm("        CLR        r30, r30, 1");
asm("        .newblock");
asm("        LDI32    r1, 178");
asm("$1:     SUB      r1, r1, 1");
asm("        QBNE     $1, r1, 0");
asm("        SET        r30, r30, 1");
asm("        .newblock");
asm("        LDI32    r1, 175");
asm("$1:     SUB      r1, r1, 1");
```

Testing that through the compiler: it results in many warnings but also gives a zero-stack function and links successfully.

Here's a variant to output the negative symbol

In [62]:

```
def render_asm_emit_neg_symbol(sleeps):
    for i in range(int(len(sleeps)/2)-2):
        print('asm("        CLR\tr30, r30, 1");')
        render_asm_sleep(sleeps[i*2] - 1)
        print('asm("        SET\tr30, r30, 1");')
        render_asm_sleep(sleeps[i*2 + 1] - 1)
    print(    'asm("        CLR\tr30, r30, 1");')
    render_asm_sleep(sleeps[len(sleeps) - 2] - 1)
    print('// caller is responsible for %d cycle delay' % sleeps[len(sleeps)

render_asm_emit_neg_symbol(dumbest_pwm(generate_single_chirp(pwm_samp_rate)))

```

```
asm("        SET        r30, r30, 1");
asm("        .newblock");
asm("        LDI32    r1, 232");
asm("$1:     SUB      r1, r1, 1");
asm("        QBNE     $1, r1, 0");
asm("        CLR        r30, r30, 1");
asm("        .newblock");
asm("        LDI32    r1, 224");
asm("$1:     SUB      r1, r1, 1");
asm("        QBNE     $1, r1, 0");
asm("        SET        r30, r30, 1");
asm("        .newblock");
asm("        LDI32    r1, 217");
asm("$1:     SUB      r1, r1, 1");
asm("        QBNE     $1, r1, 0");
asm("        CLR        r30, r30, 1");
asm("        .newblock");
asm("        LDI32    r1, 211");
asm("$1:     SUB      r1, r1, 1");
asm("        QBNE     $1, r1, 0");
```

Second Attempt

We implemented calls to the above emitting functions and included the other planned actions:

[Y] add a copy-to-stack per frame before emitting symbols

[Y] move toggling to inside of a loop and compensate for loop overhead and possibly compensate for inserted `mov` instructions too

[Y] get to 1-2 cycle function call overhead by zero stack or inlining

[Y] remove the last `delay_cycles` from functions and make the caller wait.

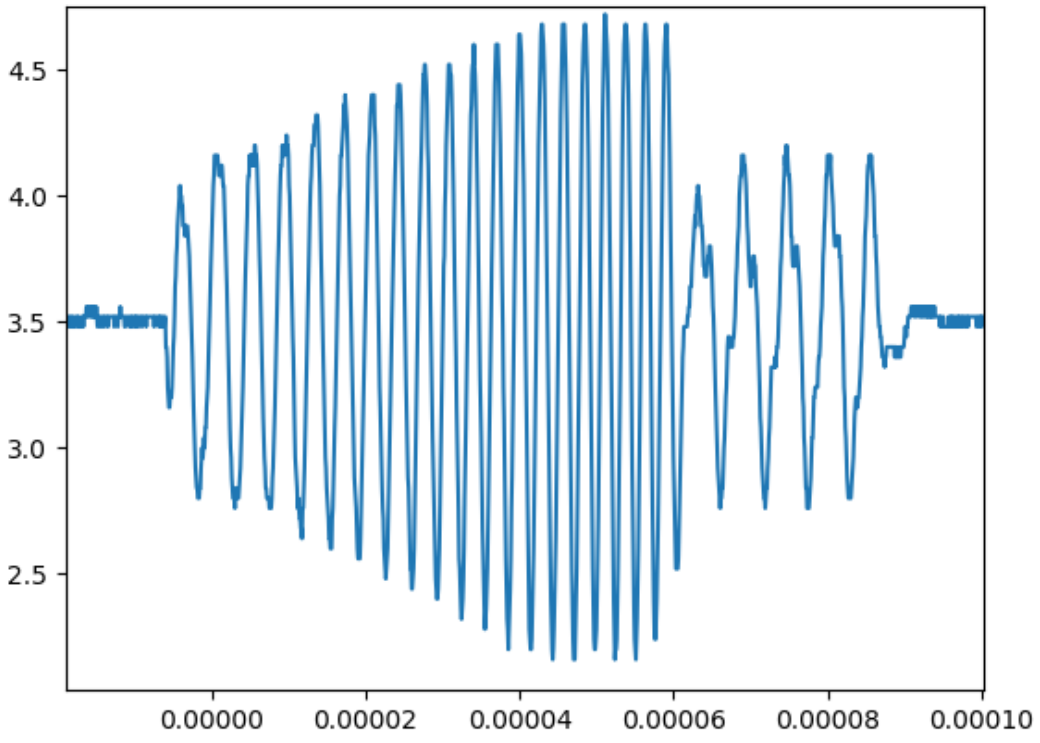
[N] supress interrupts around the symbol emitting

There didn't appear to be a way to mask interrupts for the PRU so this was skipped.

We setup sending of 0xaa preamble only and captured an waveform as before.

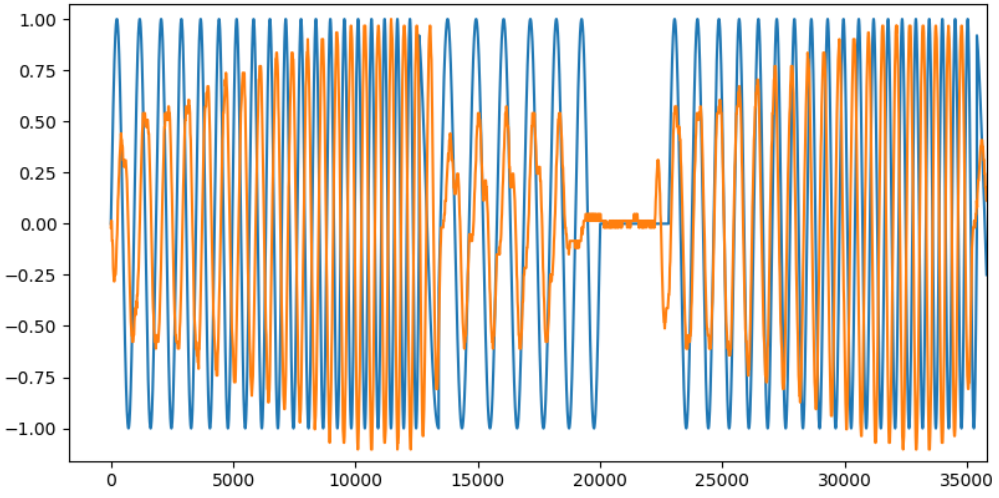
In [66]:

```
df = pd.read_csv('tune1_gpiopwm_0a00_preambleonly.csv')

%matplotlib notebook
plt.plot(df['time'], df['voltage'])
plt.show()
```



In [71]:

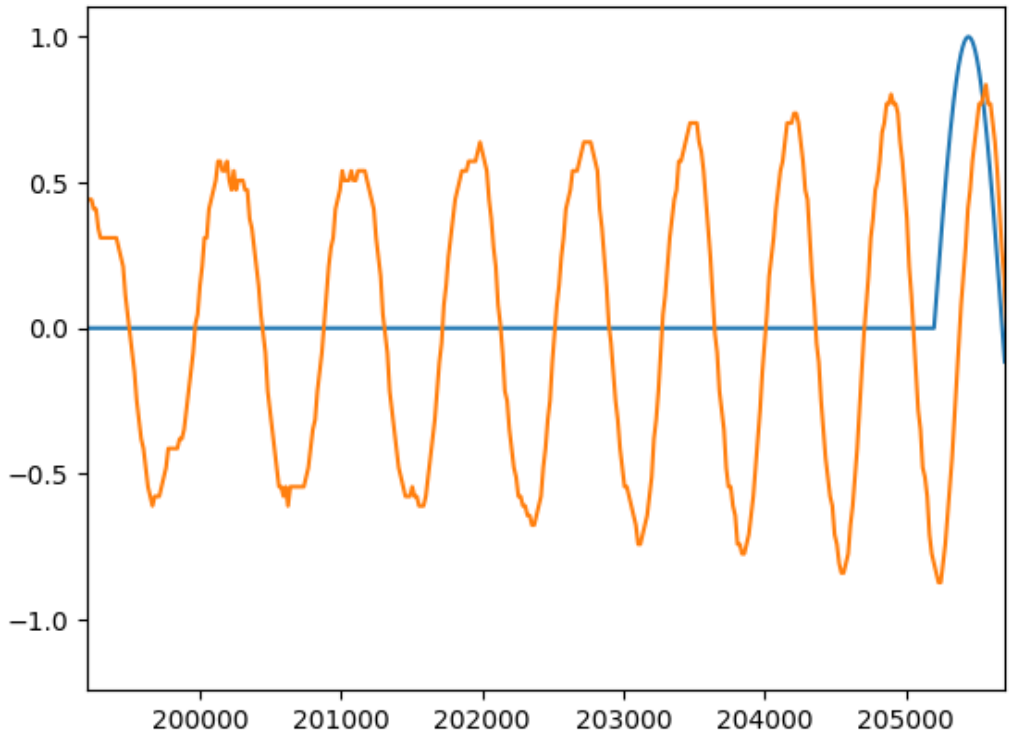
```
def plot_capture3(samp_rate):
    df = pd.read_csv('tune1_gpiopwm_0a00_preambleonly.csv')
    df = df[ df['time'] >= -6.17318e-06 ] # manually-selected start of symbol

    #df['voltage'] = df['voltage'] * -1
    df['voltage'] = df['voltage'] - df['voltage'].mean()
    df['voltage'] = df['voltage'] / df['voltage'].max()

    df['time'] = df['time'] * samp_rate
    df['time'] = df['time'] - df['time'].values[0]

    plt.plot(df['time'], df['voltage'])

%matplotlib notebook
plt.plot(generate_preamble_signal(b'\xaa', pwm_samp_rate))
plot_capture3(pwm_samp_rate)
plt.xlim([0, pwm_samp_rate * 114e-6 * 12])
plt.show()
```



This time, the symbols are far to short...

In [70]:

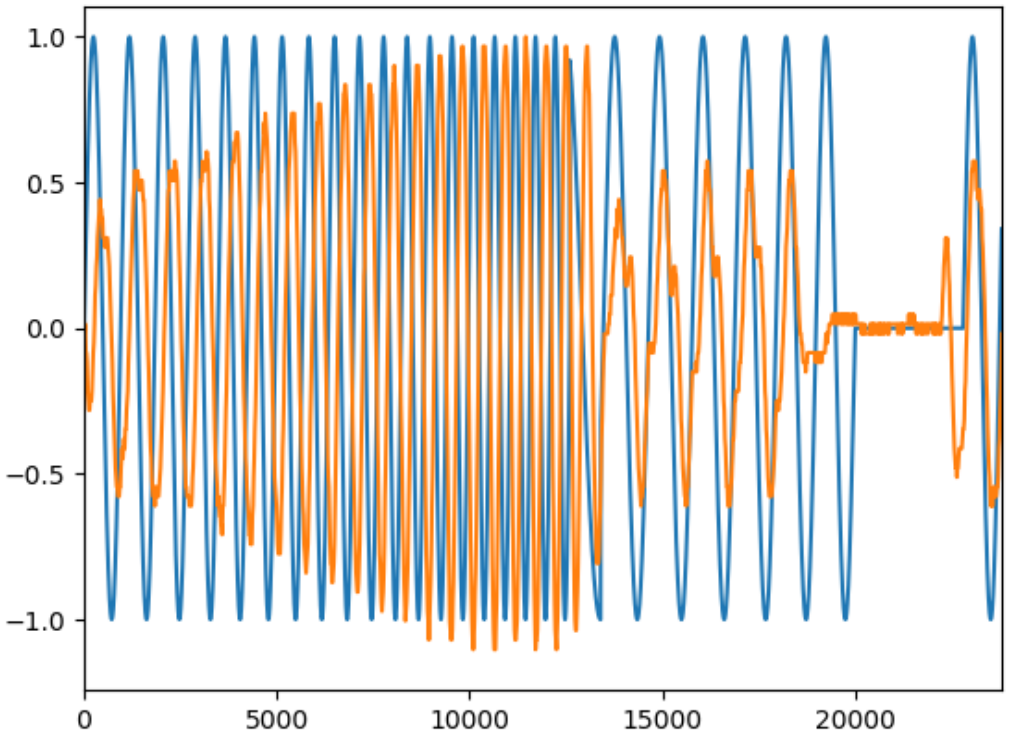
```
%matplotlib notebook
plt.plot(generate_preamble_signal(b'\xaa', pwm_samp_rate))
plot_capture3(pwm_samp_rate)
plt.xlim([pwm_samp_rate * 114e-6 * 9 - 6000, pwm_samp_rate * 114e-6 * 9 - 500
plt.show()
```



It appears as though the symbols are too short. Something wasn't implemented correctly in the emit_neg_symbol() code I think

In [73]:

```
%matplotlib notebook
plt.plot(generate_preamble_signal(b'\xaa', pwm_samp_rate))
plot_capture3(pwm_samp_rate)
plt.xlim([pwm_samp_rate * 0 , pwm_samp_rate * 114e-6 * 1 + 1000])
plt.show()
```



The problem

I think we generated the assembly wrong this time around.

In [77]:

```
def render_asm_sleep(cycles):
    print('''\
asm("        .newblock");
asm("        LDI32    r1, %d ; sleep %d cycles total");
asm("$1:     SUB      r1, r1, 1");
asm("        QBNE     $1, r1, 0");\
    ''' % (math.floor(cycles/2 - 1), cycles) )
```

In [78]:

```
def render_asm_emit_pos_symbol(sleeps):
    def render_pos_toggle(index):
        if index % 2 == 0:
            print('asm("        SET\tr30, r30, 1");')
        else:
            print('asm("        CLR\tr30, r30, 1");')
    for i in range(len(sleeps)-1):
        render_pos_toggle(i)
        render_asm_sleep(sleeps[i] - 1)
    render_pos_toggle(len(sleeps) - 1)
    print('// caller is responsible for %d cycle delay' % sleeps[len(sleeps)

render_asm_emit_pos_symbol(dumbest_pwm(generate_single_chirp(pwm_samp_rate)))
```

```
asm("        SET        r30, r30, 1");
asm("        .newblock");
asm("        LDI32    r1, 240 ; sleep 483 cycles total");
asm("$1:     SUB      r1, r1, 1");
asm("        QBNE     $1, r1, 0");
asm("        CLR        r30, r30, 1");
asm("        .newblock");
asm("        LDI32    r1, 232 ; sleep 466 cycles total");
asm("$1:     SUB      r1, r1, 1");
asm("        QBNE     $1, r1, 0");
asm("        SET        r30, r30, 1");
asm("        .newblock");
asm("        LDI32    r1, 224 ; sleep 451 cycles total");
asm("$1:     SUB      r1, r1, 1");
asm("        QBNE     $1, r1, 0");
asm("        CLR        r30, r30, 1");
asm("        .newblock");
asm("        LDI32    r1, 217 ; sleep 436 cycles total");
asm("$1:     SUB      r1, r1, 1");
```

In [79]:

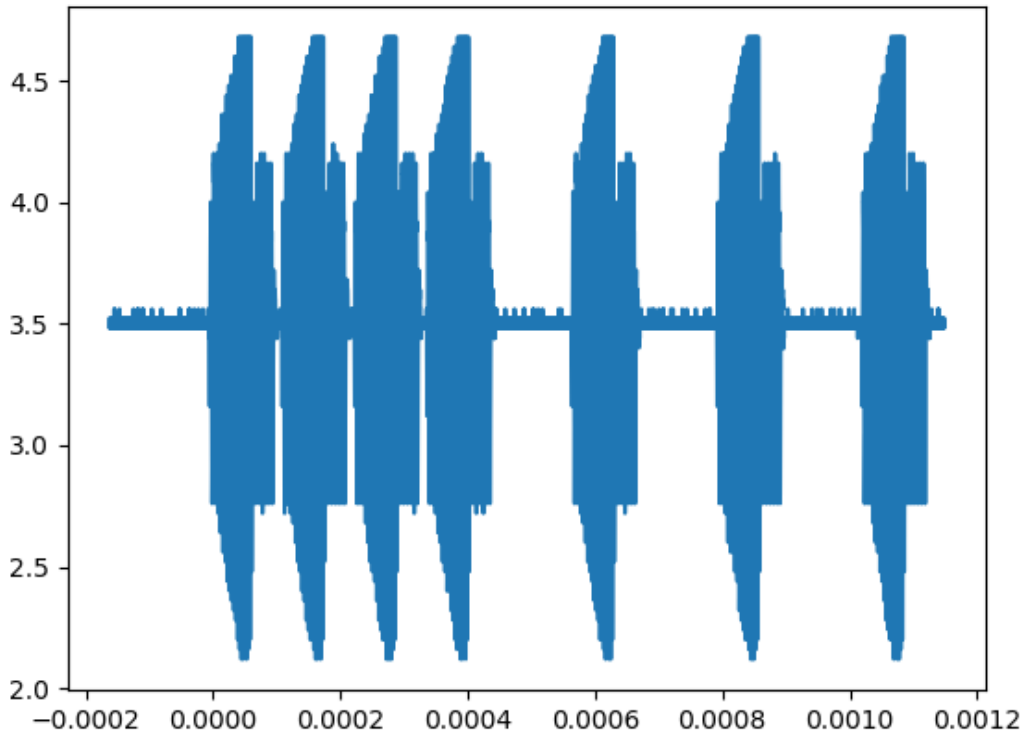
```
def render_asm_emit_neg_symbol(sleeps):
    def render_neg_toggle(index):
        if index % 2 == 0:
            print('asm("        CLR\tr30, r30, 1");')
        else:
            print('asm("        SET\tr30, r30, 1");')
    for i in range(len(sleeps)-1):
        render_neg_toggle(i)
        render_asm_sleep(sleeps[i] - 1)
    render_neg_toggle(len(sleeps) - 1)
    print('// caller is responsible for %d cycle delay' % sleeps[len(sleeps)

render_asm_emit_neg_symbol(dumbest_pwm(generate_single_chirp(pwm_samp_rate)))
```

```
asm("        CLR        r30, r30, 1");
asm("        .newblock");
asm("        LDI32    r1, 240 ; sleep 483 cycles total");
asm("$1:     SUB      r1, r1, 1");
asm("        QBNE     $1, r1, 0");
asm("        SET        r30, r30, 1");
asm("        .newblock");
asm("        LDI32    r1, 232 ; sleep 466 cycles total");
asm("$1:     SUB      r1, r1, 1");
asm("        QBNE     $1, r1, 0");
asm("        CLR        r30, r30, 1");
asm("        .newblock");
asm("        LDI32    r1, 224 ; sleep 451 cycles total");
asm("$1:     SUB      r1, r1, 1");
asm("        QBNE     $1, r1, 0");
asm("        SET        r30, r30, 1");
asm("        .newblock");
asm("        LDI32    r1, 217 ; sleep 436 cycles total");
asm("$1:     SUB      r1, r1, 1");
```

So we used the above rendering of asm functions instead to see.

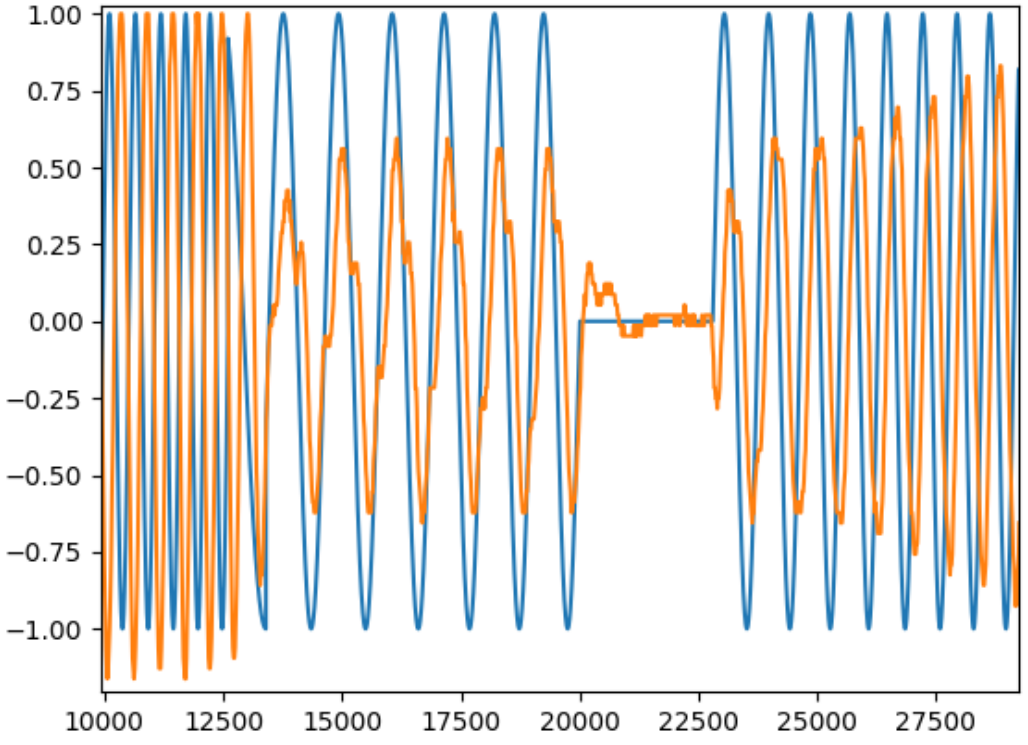
In [88]:

```
df = pd.read_csv('tune2_gpiopwm_0a00_preambleonly.csv')

%matplotlib notebook
plt.plot(df['time'], df['voltage'])
plt.show()
```



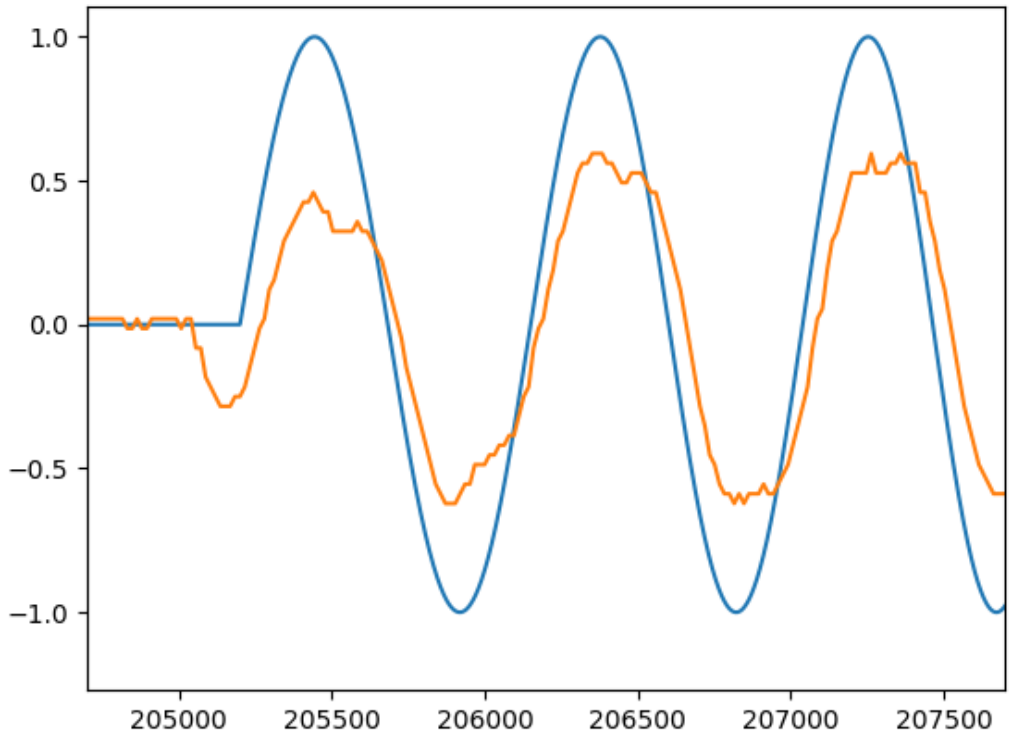
In [127]:

```
def plot_tune2(samp_rate):
    df = pd.read_csv('tune2_gpiopwm_0a00_preambleonly.csv')
    df = df[ df['time'] >= -6.07377e-6 ] # manually-selected start of symbol

    #df['voltage'] = df['voltage'] * -1
    df['voltage'] = df['voltage'] - df['voltage'].mean()
    df['voltage'] = df['voltage'] / df['voltage'].max()

    df['time'] = df['time'] * samp_rate
    df['time'] = df['time'] - df['time'].values[0]

    plt.plot(df['time'], df['voltage'])

%matplotlib notebook
plt.plot(generate_preamble_signal(b'\xaa', pwm_samp_rate))
plot_tune2(pwm_samp_rate)
plt.xlim([0, pwm_samp_rate * 114e-6 * 12])
plt.show()
```



In [92]:

```
%matplotlib notebook
plt.plot(generate_preamble_signal(b'\xaa', pwm_samp_rate))
plot_tune2(pwm_samp_rate)
plt.xlim([pwm_samp_rate * 114e-6 * 9 - 500, pwm_samp_rate * 114e-6 * 9 - 500
plt.show()
```



The last negative symbol in the preamble is now a *little* early by 162 cycles

In [93]:

```
pwm_samp_rate * 114e-6 * 9 - 205038
```

Out[93]: 162.0

The Next Problem

Maybe we're still sleeping too little in the symbols emitted. Let's try some more sleep cycles in the symbol and at ends

In [103]:

```
def render_asm_sleep(cycles):
    print('''\
    asm("        .newblock");
    asm("        LDI32    r1, %d ; sleep %d cycles total");
    asm("$1:     SUB      r1, r1, 1");
    asm("        QBNE     $1, r1, 0");\
    ''' % (math.floor((cycles - 1)/2), cycles) )
```

In [104]:

```
def render_pos_toggle(index):
    if index % 2 == 0:
        print('    asm("        SET\tr30, r30, 1");')
    else:
        print('    asm("        CLR\tr30, r30, 1");')

def render_neg_toggle(index):
    render_pos_toggle(index + 1)

def render_asm_emit_symbol(sleeps, toggler):
    for i in range(len(sleeps)-1):
        toggler(i)
        render_asm_sleep(sleeps[i])
    toggler(len(sleeps) - 1)
    print('    // caller is responsible for %d cycle delay' % sleeps[len(slee

render_asm_emit_symbol(dumbest_pwm(generate_single_chirp(pwm_samp_rate)), ren
```

```
asm("        SET    r30, r30, 1");
asm("        .newblock");
asm("        LDI32    r1, 241 ; sleep 484 cycles total");
asm("$1:     SUB      r1, r1, 1");
asm("        QBNE     $1, r1, 0");
asm("        CLR    r30, r30, 1");
asm("        .newblock");
asm("        LDI32    r1, 233 ; sleep 467 cycles total");
asm("$1:     SUB      r1, r1, 1");
asm("        QBNE     $1, r1, 0");
asm("        SET    r30, r30, 1");
asm("        .newblock");
asm("        LDI32    r1, 225 ; sleep 452 cycles total");
asm("$1:     SUB      r1, r1, 1");
asm("        QBNE     $1, r1, 0");
asm("        CLR    r30, r30, 1");
asm("        .newblock");
asm("        LDI32    r1, 218 ; sleep 437 cycles total");
asm("$1:     SUB      r1, r1, 1");
```

In [105]:

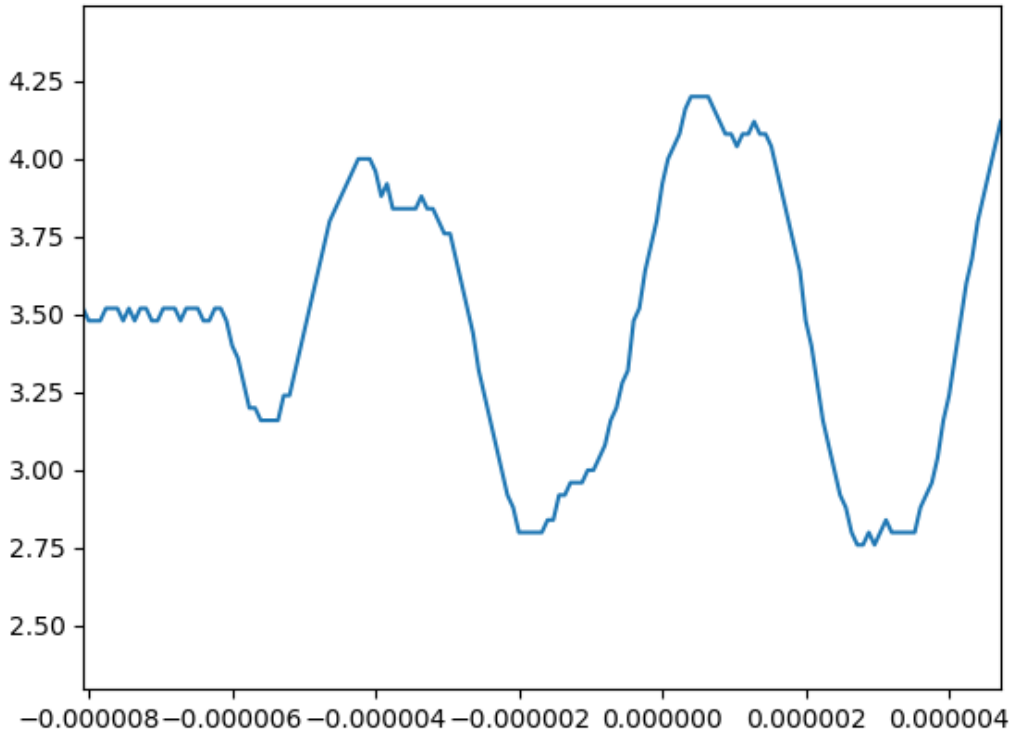
```
render_asm_emit_symbol(dumbest_pwm(generate_single_chirp(pwm_samp_rate)), ren
```

```
asm("        CLR    r30, r30, 1");
asm("        .newblock");
asm("        LDI32    r1, 241 ; sleep 484 cycles total");
asm("$1:     SUB      r1, r1, 1");
asm("        QBNE     $1, r1, 0");
asm("        SET    r30, r30, 1");
asm("        .newblock");
asm("        LDI32    r1, 233 ; sleep 467 cycles total");
asm("$1:     SUB      r1, r1, 1");
asm("        QBNE     $1, r1, 0");
asm("        CLR    r30, r30, 1");
asm("        .newblock");
asm("        LDI32    r1, 225 ; sleep 452 cycles total");
asm("$1:     SUB      r1, r1, 1");
asm("        QBNE     $1, r1, 0");
asm("        SET    r30, r30, 1");
asm("        .newblock");
asm("        LDI32    r1, 218 ; sleep 437 cycles total");
asm("$1:     SUB      r1, r1, 1");
```

We implemented the above to see

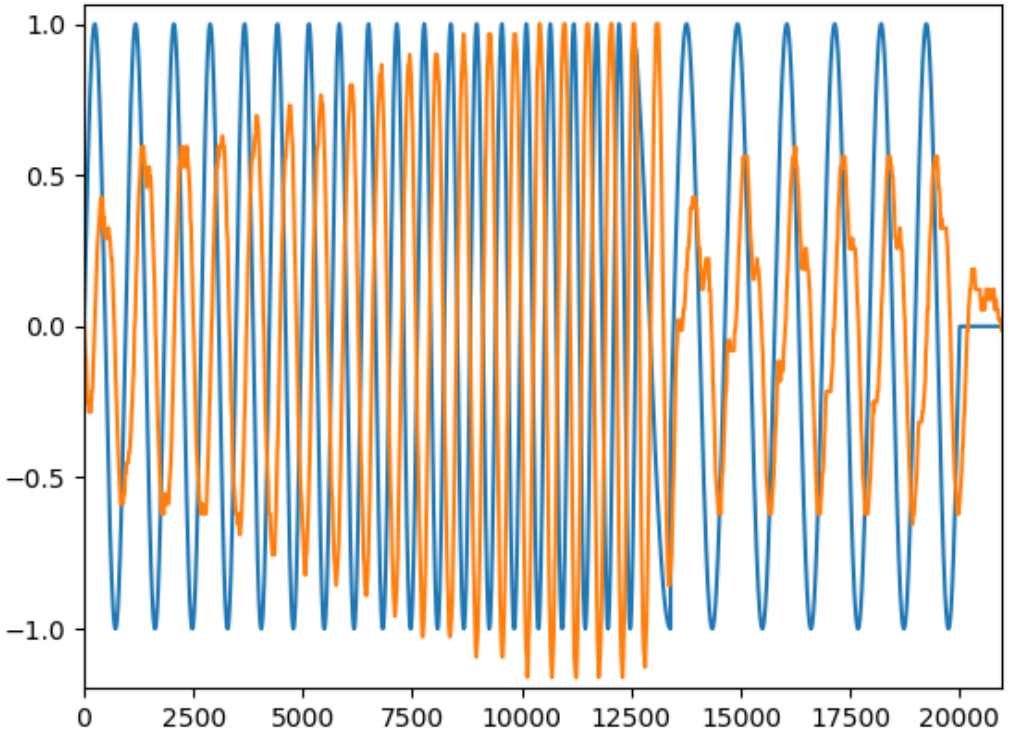
In [106]:

```
df = pd.read_csv('tune3_gpiopwm_0a00_preambleonly.csv')

%matplotlib notebook
plt.plot(df['time'], df['voltage'])
plt.show()
```

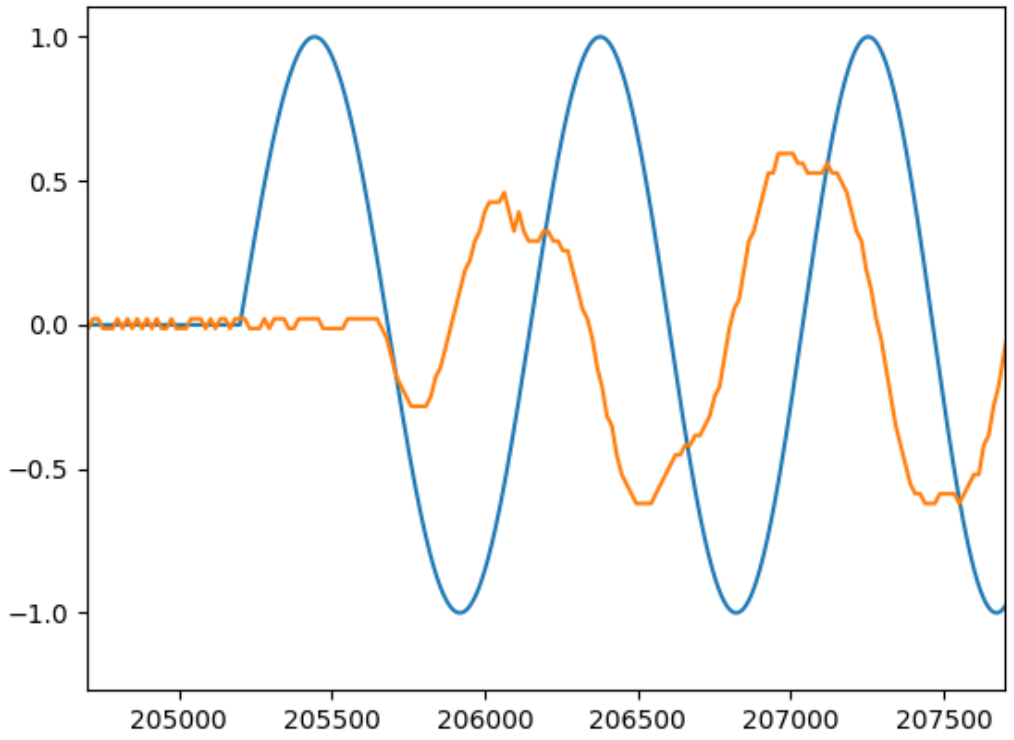


In [111]:

```
def plot_tune3(samp_rate):
    df = pd.read_csv('tune3_gpiopwm_0a00_preambleonly.csv')
    df = df[ df['time'] >= -6.19825e-6 ] # manually-selected start of symbol

    #df['voltage'] = df['voltage'] * -1
    df['voltage'] = df['voltage'] - df['voltage'].mean()
    df['voltage'] = df['voltage'] / df['voltage'].max()

    df['time'] = df['time'] * samp_rate
    df['time'] = df['time'] - df['time'].values[0]

    plt.plot(df['time'], df['voltage'])

%matplotlib notebook
plt.plot(generate_preamble_signal(b'\xaa', pwm_samp_rate))
plot_tune3(pwm_samp_rate)
plt.xlim([0, pwm_samp_rate * 114e-6 * 12])
plt.show()

```



In [109]:
```
%matplotlib notebook
plt.plot(generate_preamble_signal(b'\xaa', pwm_samp_rate))
plot_tune3(pwm_samp_rate)
plt.xlim([pwm_samp_rate * 114e-6 * 9 - 500, pwm_samp_rate * 114e-6 * 9 - 500
plt.show()
```



Which removed too many sleeps since now the signal is 449 cycles too slow on the last negative symbol

In [110]:
```
205649 - pwm_samp_rate * 114e-6 * 9
```

Out[110]: 449.0

The Next Next Problem

That was obviously to many sleeps added. We'll try to use the previous delay_cycles-like LDI32 calculation but not adjust the target number of sleep cycles by the 1-cycle cost of the GPIO toggling instruction.

In [116]:

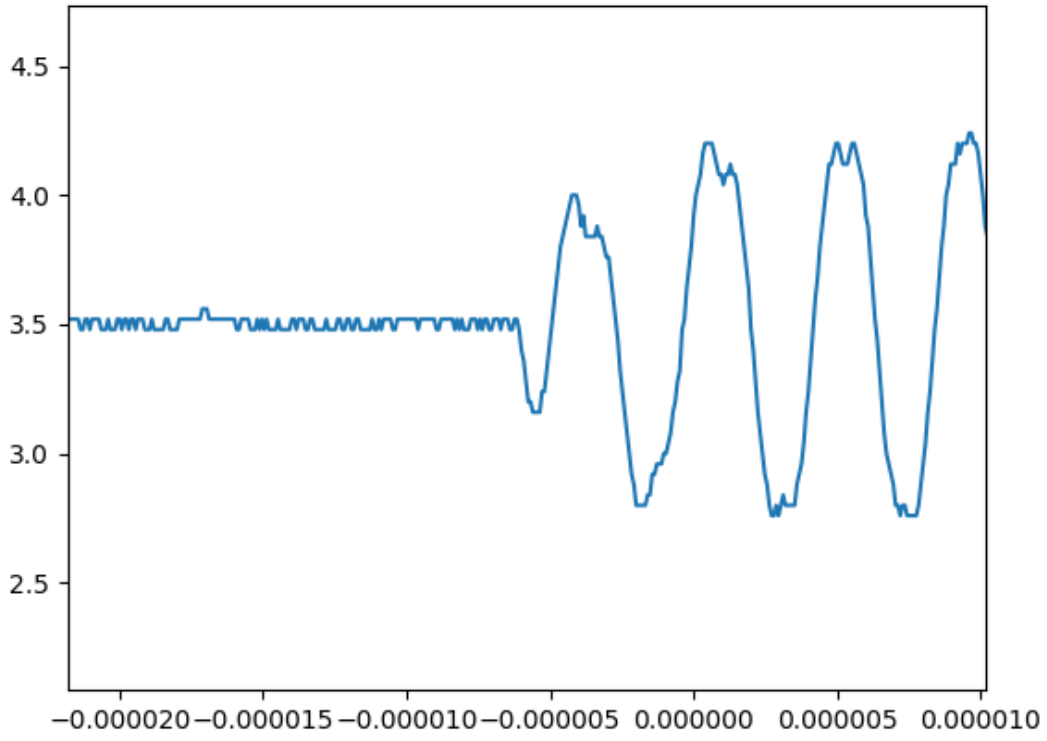
```
def render_asm_sleep(cycles):
    print('''\
    asm("        .newblock");
    asm("        LDI32    r1, %d ; sleep %d cycles total");
    asm("$1:     SUB      r1, r1, 1");
    asm("        QBNE     $1, r1, 0");\
    ''' % (math.floor(cycles/2) - 1, cycles) )

def render_pos_toggle(index):
    if index % 2 == 0:
        print('    asm("        SET\tr30, r30, 1");')
    else:
        print('    asm("        CLR\tr30, r30, 1");')

def render_neg_toggle(index):
    render_pos_toggle(index + 1)

def render_asm_emit_symbol(sleeps, toggler):
    for i in range(len(sleeps)-1):
        toggler(i)
        render_asm_sleep(sleeps[i])
    toggler(len(sleeps) - 1)
    print('    // caller is responsible for %d cycle delay' % sleeps[len(slee

print('void emit_pos_symbol() {')
render_asm_emit_symbol(dumbest_pwm(generate_single_chirp(pwm_samp_rate)), ren
print('}\n')

print('void emit_neg_symbol() {')
render_asm_emit_symbol(dumbest_pwm(generate_single_chirp(pwm_samp_rate)), ren
print('}\n')
```

```
void emit_pos_symbol() {
    asm("        SET    r30, r30, 1");
    asm("        .newblock");
    asm("        LDI32    r1, 241 ; sleep 484 cycles total");
    asm("$1:     SUB      r1, r1, 1");
    asm("        QBNE     $1, r1, 0");
    asm("        CLR    r30, r30, 1");
    asm("        .newblock");
    asm("        LDI32    r1, 232 ; sleep 467 cycles total");
    asm("$1:     SUB      r1, r1, 1");
    asm("        QBNE     $1, r1, 0");
    asm("        SET    r30, r30, 1");
    asm("        .newblock");
    asm("        LDI32    r1, 225 ; sleep 452 cycles total");
    asm("$1:     SUB      r1, r1, 1");
    asm("        QBNE     $1, r1, 0");
    asm("        CLR    r30, r30, 1");
    asm("        .newblock");
    asm("        LDI32    r1, 217 ; sleep 437 cycles total");
```

We implemented the above to see

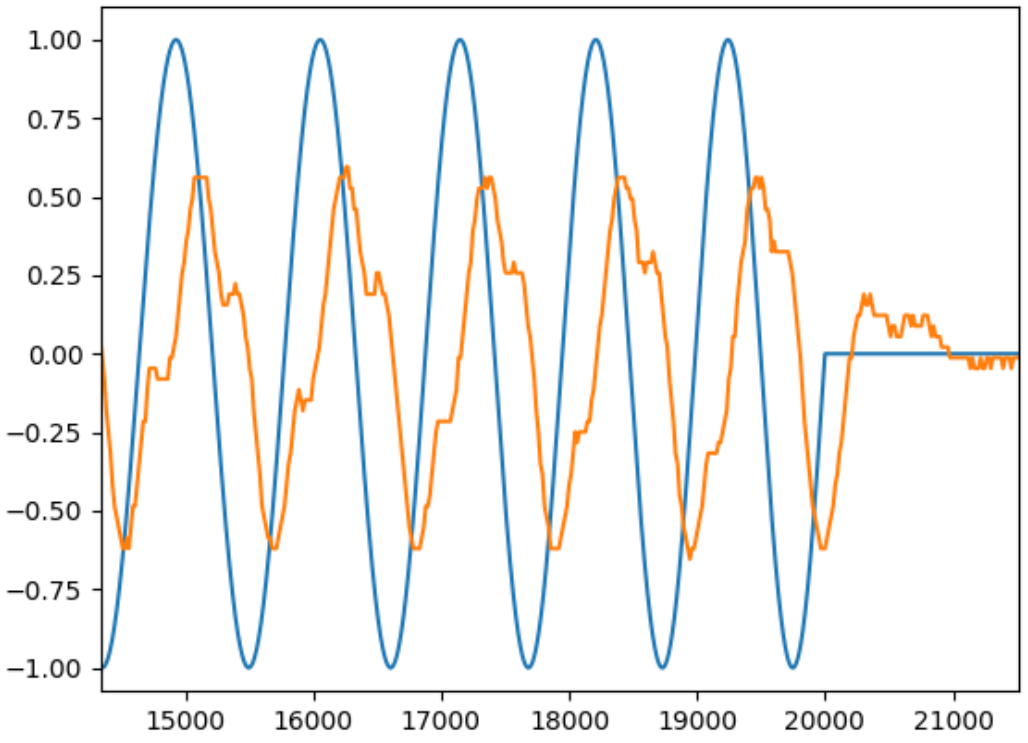
In [117]:

```
df = pd.read_csv('tune4_gpiopwm_0a00_preambleonly.csv')

%matplotlib notebook
plt.plot(df['time'], df['voltage'])
plt.show()
```



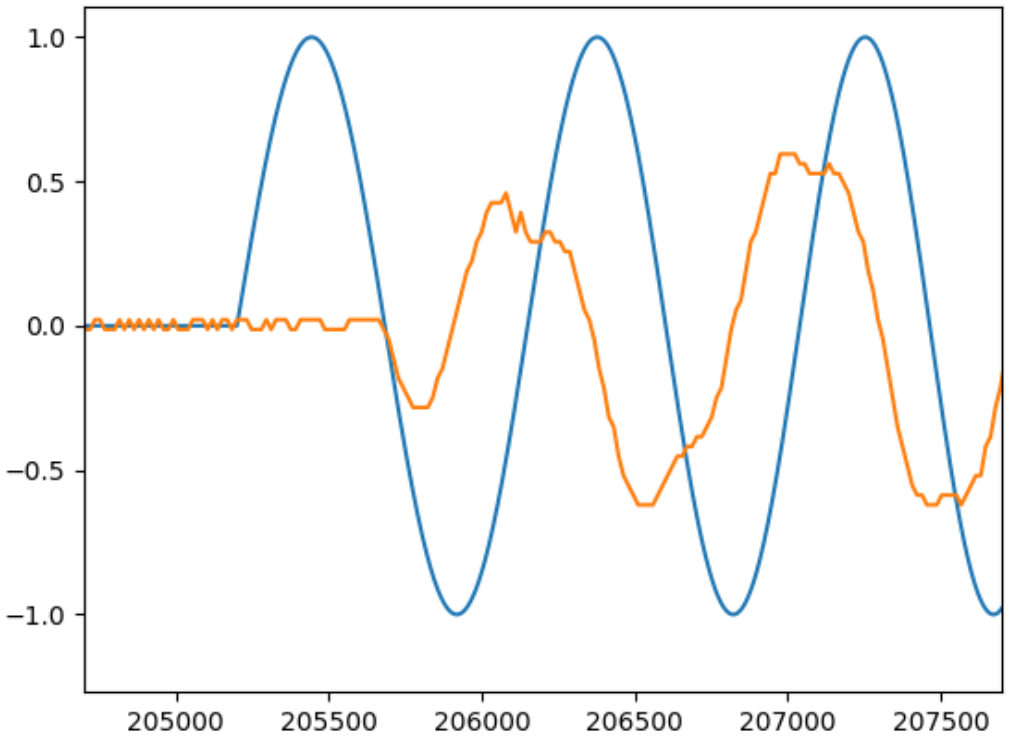
In [125]:

```
def plot_tune4(samp_rate):
    df = pd.read_csv('tune4_gpiopwm_0a00_preambleonly.csv')
    df = df[ df['time'] >= -6.26768e-06] # manually-selected start of symbol

    #df['voltage'] = df['voltage'] * -1
    df['voltage'] = df['voltage'] - df['voltage'].mean()
    df['voltage'] = df['voltage'] / df['voltage'].max()

    df['time'] = df['time'] * samp_rate
    df['time'] = df['time'] - df['time'].values[0]

    plt.plot(df['time'], df['voltage'])

%matplotlib notebook
plt.plot(generate_preamble_signal(b'\xaa', pwm_samp_rate))
plot_tune4(pwm_samp_rate)
plt.xlim([0, pwm_samp_rate * 114e-6 * 12])
plt.show()
```



In [122]:

```
%matplotlib notebook
plt.plot(generate_preamble_signal(b'\xaa', pwm_samp_rate))
plot_tune4(pwm_samp_rate)
plt.xlim([pwm_samp_rate * 114e-6 * 9 - 500, pwm_samp_rate * 114e-6 * 9 - 500
plt.show()
```



Once again too slow by about 500 cycles

Tune 5: Tune Harder...

That last result doesn't make much sense -- unless the cycles to sleep are alway odd so the change had no impact. We might also be observing the impact of the /8 in the GPIO output clocking; i.e. the GPIO outputs are sync'd to a 200MHz/8 clocks = 25MHz with a minimum period of

In [124]:

```
1/25E6
```

Out[124]: 4e-08

```
But that is still small enough to not account for a 450 cycle difference.

```

```
Looking at the end of the first symbol emitted in the 'tune4' attempt we can see
that it is 187 samples too long.

Recall, the length of the PWM 'sleeps' we are emitting is 52
```

In [126]:
```
len(sleeps)
```

Out[126]: 52

So, in this case, there might as much as 3 cycles slept-too-many per PWM sleep...

In [144]:

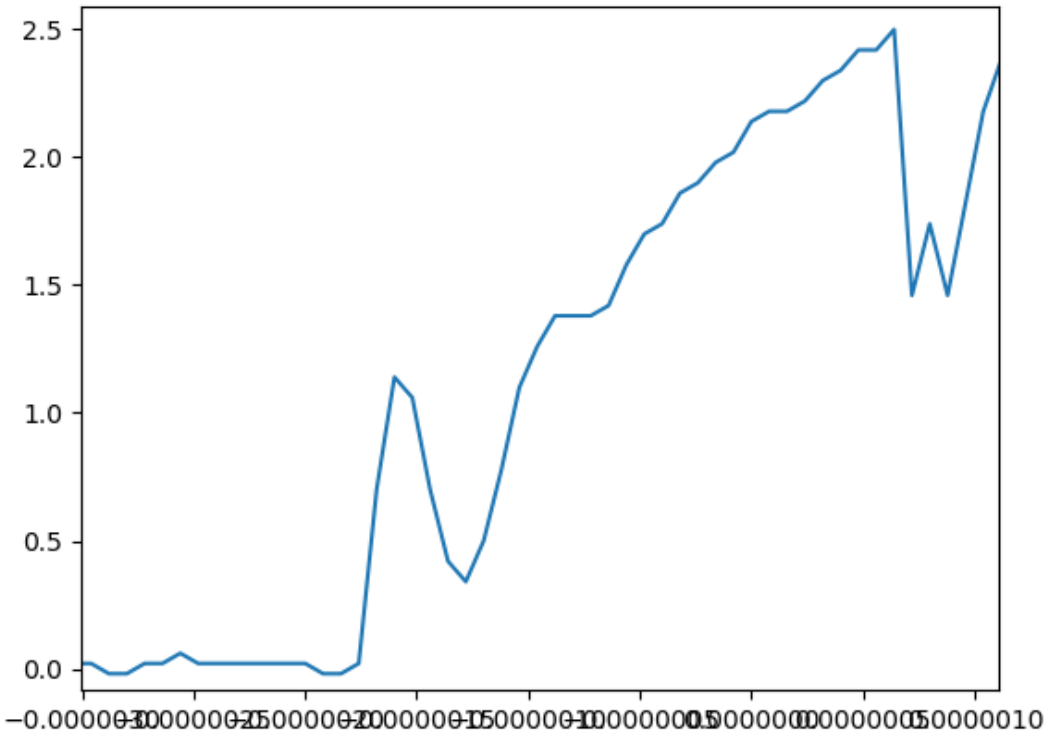
```
def render_asm_sleep(cycles):
    print('''\
    asm("        .newblock");
    asm("        LDI32    r1, %d ; sleep %d cycles total");
    asm("$1:     SUB      r1, r1, 1");
    asm("        QBNE     $1, r1, 0");\
''' % (math.floor((cycles - 2)/2), cycles) )

def render_pos_toggle(index):
    if index % 2 == 0:
        print('    asm("        SET\tr30, r30, 1");')
    else:
        print('    asm("        CLR\tr30, r30, 1");')

def render_neg_toggle(index):
    render_pos_toggle(index + 1)

def render_asm_emit_symbol(sleeps, toggler):
    for i in range(len(sleeps)-1):
        toggler(i)
        render_asm_sleep(sleeps[i]
                         - 0 #custom fudge factor
                        )
    toggler(len(sleeps) - 1)
    print('    // caller is responsible for %d cycle delay' % sleeps[len(slee

print('''
// WARNING: we use r1 here because it appeared to be a safe choice when looki
// the caller. R0 is also listed as a save-on-call register in the calling
// conventions in http://www.ti.com/lit/ug/spruhv7b/spruhv7b.pdf but it turne
// out not to be a good choice. YMMV.\
''')
print('void emit_pos_symbol() {')
render_asm_emit_symbol(dumbest_pwm(generate_single_chirp(pwm_samp_rate)), ren
print('}\n\n#define EMIT_POS_SYMBOL_FINAL_CYCLES 504')

print('''
// WARNING: we use r1 here because it appeared to be a safe choice when looki
// the caller. R0 is also listed as a save-on-call register in the calling
// conventions in http://www.ti.com/lit/ug/spruhv7b/spruhv7b.pdf but it turne
// out not to be a good choice. YMMV.\
''')
print('void emit_neg_symbol() {')
render_asm_emit_symbol(dumbest_pwm(generate_single_chirp(pwm_samp_rate)), ren
print('}\n\n#define EMIT_NEG_SYMBOL_FINAL_CYCLES 504')
```

```
asm("        .newblock");
asm("        LDI32    r1, 268 ; sleep 538 cycles total");
asm("$1:     SUB      r1, r1, 1");
asm("        QBNE     $1, r1, 0");
asm("        SET    r30, r30, 1");
asm("        .newblock");
asm("        LDI32    r1, 264 ; sleep 531 cycles total");
asm("$1:     SUB      r1, r1, 1");
asm("        QBNE     $1, r1, 0");
asm("        CLR    r30, r30, 1");
asm("        .newblock");
asm("        LDI32    r1, 260 ; sleep 523 cycles total");
asm("$1:     SUB      r1, r1, 1");
asm("        QBNE     $1, r1, 0");
asm("        SET    r30, r30, 1");
```

```
asm("        .newblock");
asm("        LDI32    r1, 257 ; sleep 517 cycles total");
asm("$1:     SUB      r1, r1, 1");
asm("        QBNE     $1, r1, 0");
asm("        CLR    r30, r30, 1");
```

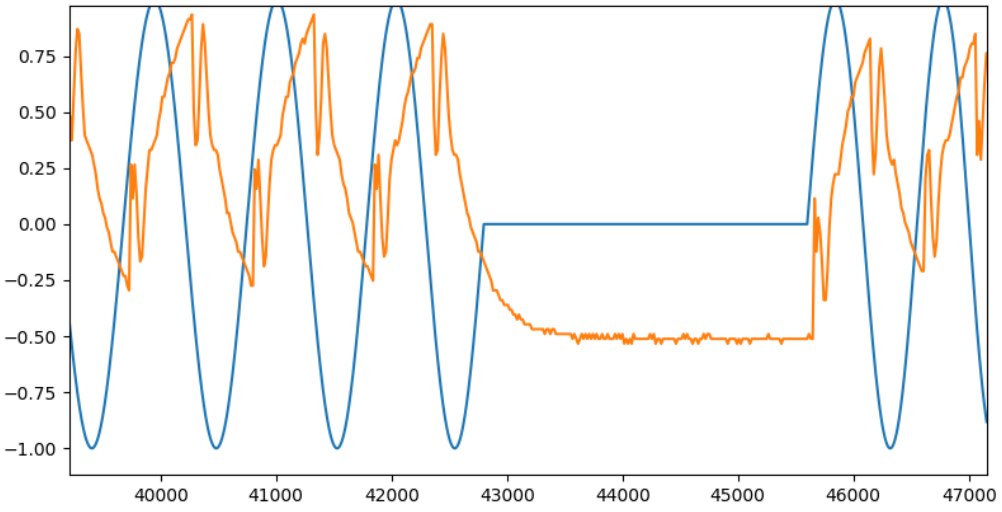
In [147]:

```
df = pd.read_csv('tune5_gpiopwm_0a00_preambleonly.csv')

%matplotlib notebook
plt.plot(df['time'], df['voltage'])
plt.show()
```



In [149]:

```
def plot_tune5(samp_rate):
    df = pd.read_csv('tune5_gpiopwm_0a00_preambleonly.csv')
    df = df[ df['time'] >= -1.76178e-6] # manually-selected start of symbol

    #df['voltage'] = df['voltage'] * -1
    df['voltage'] = df['voltage'] - df['voltage'].mean()
    df['voltage'] = df['voltage'] / df['voltage'].max()

    df['time'] = df['time'] * samp_rate
    df['time'] = df['time'] - df['time'].values[0]

    plt.plot(df['time'], df['voltage'])

%matplotlib notebook
plt.plot(generate_preamble_signal(b'\xaa', pwm_samp_rate))
plot_tune5(pwm_samp_rate)
plt.xlim([0, pwm_samp_rate * 114e-6 * 12])
plt.show()
```



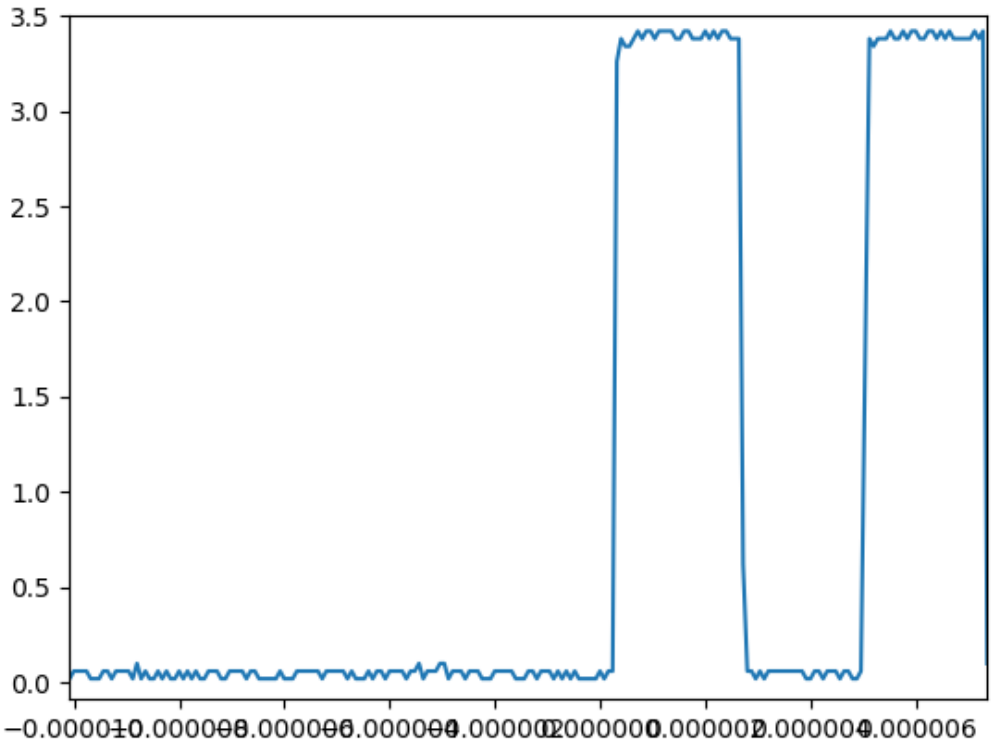
Tune6: Square Tuning

In the last attempt above we got the timings pretty close but because we are looking at the PWM GPIO output when connected to the coupling network we see lots of the filter effects and this is making it hard to adjust the difference between the intra- and inter-symbol timings.

We disconnected the coupling network so that we would have only square waves and thus could better adjust the timings.

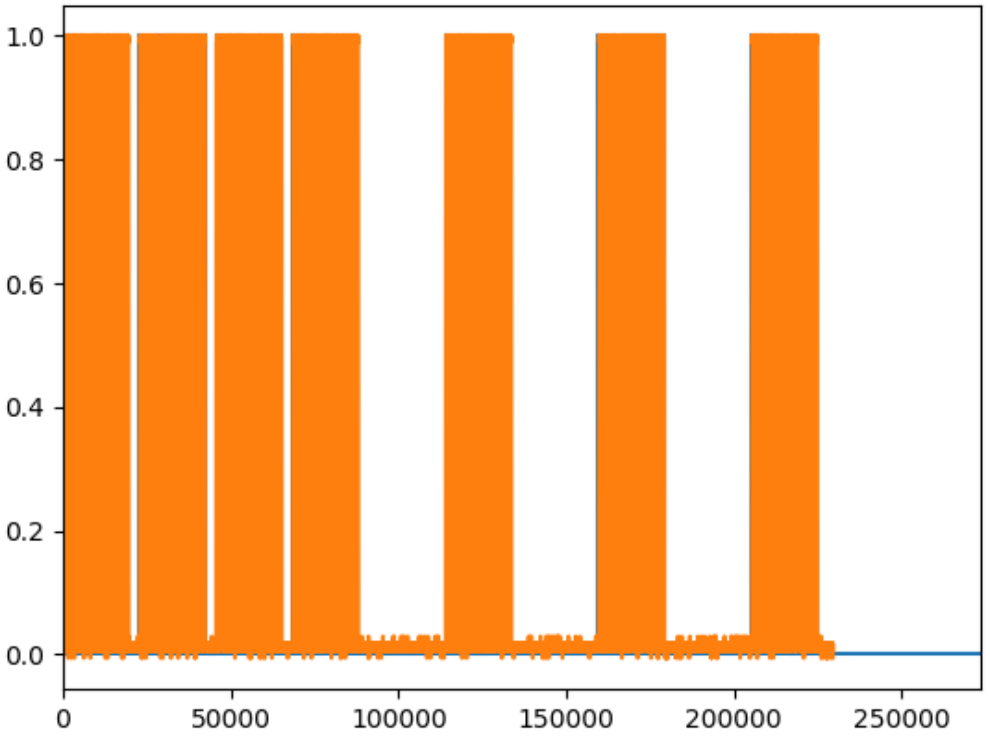
In [150]:

```
df = pd.read_csv('tune6_gpiopwm_0a00_preambleonly.csv')

%matplotlib notebook
plt.plot(df['time'], df['voltage'])
plt.show()
```



In [245]:

```
def get_tune6(samp_rate):
    df = pd.read_csv('tune6_gpiopwm_0a00_preambleonly.csv')
    df = df[ df['time'] >= 2.07638e-7] # manually-selected start of symbol

    #df['voltage'] = df['voltage'] * -1
    #df['voltage'] = df['voltage'] - df['voltage'].mean()
    df['voltage'] = df['voltage'] / df['voltage'].max()

    df['time'] = df['time'] * samp_rate
    df['time'] = df['time'] - df['time'].values[0]
    return df

def plot_tune6(samp_rate):
    df = get_tune6(samp_rate)
    plt.plot(df['time'], df['voltage'])

%matplotlib notebook
#plt.plot(generate_preamble_signal(b'\xaa', pwm_samp_rate))
plot_viz_pru(generate_preamble_sleeps(b'\xaa', pwm_samp_rate))
plot_tune6(pwm_samp_rate)
plt.xlim([0, pwm_samp_rate * 114e-6 * 12])
plt.show()
```



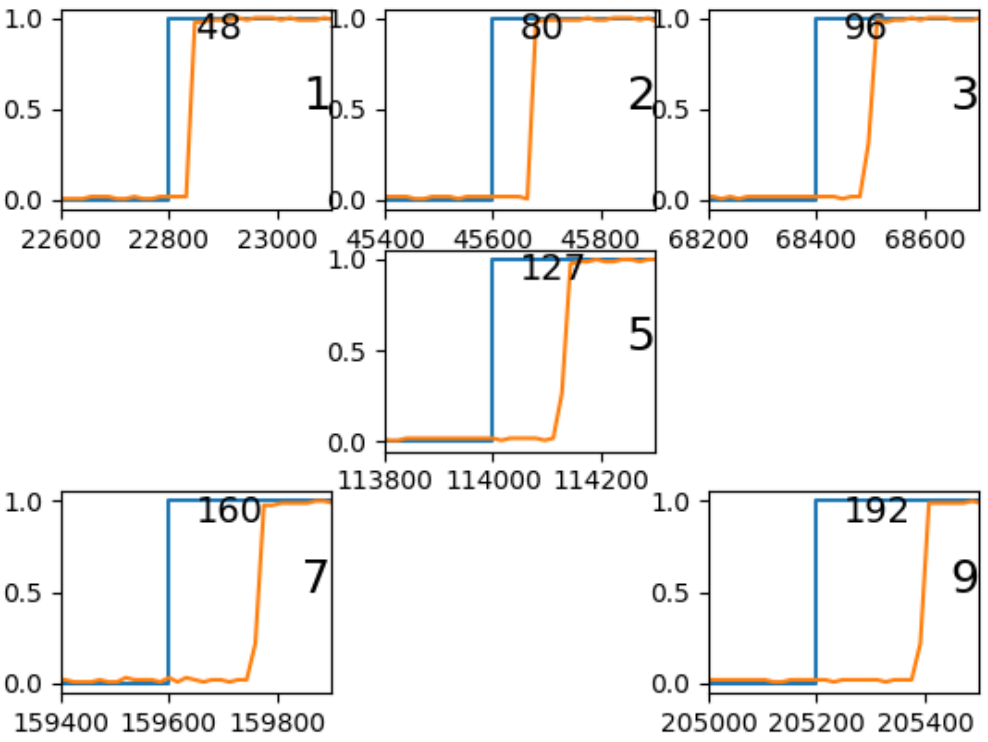
```
The alignment is as-good-as we observed after the DG tech filter, which is good. The first symbol is near-perfect. By the time we hit the last symbol something is off. This is probably due to inter-symbol timing now, not intra-symbol.
```

In [355]:

```python
def get_first_falling_edge(df, focus_start, focus_end):
    df = df[df['time'].between(focus_start, focus_end)]
    return df[df['voltage'] < 0.95].iloc[0]['time']

def get_first_rising_edge(df, focus_start, focus_end):
    df = df[df['time'].between(focus_start, focus_end)]
    return df[df['voltage'] > 0.15].iloc[0]['time']
```

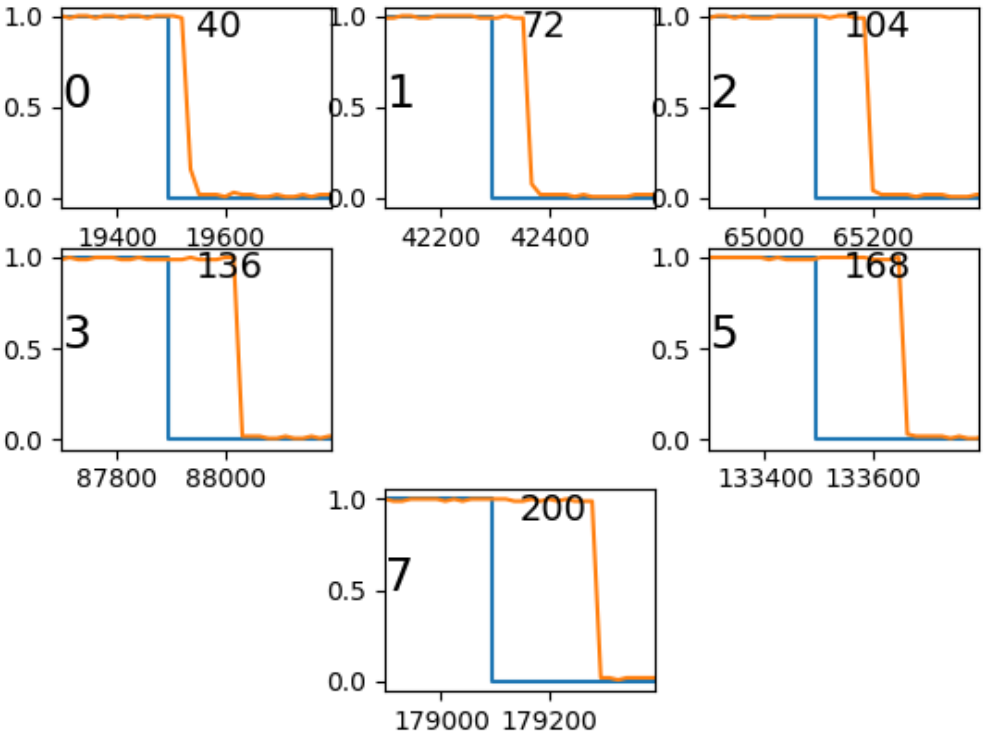
In [286]:

```
band = 200
view = 500
%matplotlib notebook

for i in [1,2,3,5,7,9]:
    ax = plt.subplot(3,3,i)
    #plt.plot(generate_preamble_signal(b'\xaa', pwm_samp_rate))
    plot_viz_pru(generate_preamble_sleeps(b'\xaa', pwm_samp_rate))
    plot_tune6(pwm_samp_rate)
    focus_start = pwm_samp_rate * 114e-6 * i - band
    focus_end   = pwm_samp_rate * 114e-6 * i - band + view
    plt.xlim([focus_start, focus_end])
    plt.text(1.0, 0.5, str(i), fontsize=18, ha='right', transform=ax.transAxe
    expected_up_cycle = pwm_samp_rate * 114e-6 * i
    measured_delay = get_first_rising_edge(get_tune6(pwm_samp_rate), focus_st
    plt.text(0.5, 0.99, str(int(measured_delay)), fontsize=14, va='top', ha='
plt.show()
```



In [271]:

```
band = 200
view = 500
%matplotlib notebook

for i in [1,2,3,4,6,8]:
    ax = plt.subplot(3,3,i)
    #plt.plot(generate_preamble_signal(b'\xaa', pwm_samp_rate))
    plot_viz_pru(generate_preamble_sleeps(b'\xaa', pwm_samp_rate))
    plot_tune6(pwm_samp_rate)
    focus_start = pwm_samp_rate * 114e-6 * i - pwm_samp_rate * 14e-6 - 504 -
    focus_end   = pwm_samp_rate * 114e-6 * i - pwm_samp_rate * 14e-6 - 504 -
    plt.xlim([focus_start, focus_end])
    plt.text(0.0, 0.5, str(i-1), fontsize=18, ha='left', transform=ax.transAx
    expected_down_cycle = pwm_samp_rate * 114e-6 * i - pwm_samp_rate * 14e-6
    measured_delay = get_first_falling_edge(get_tune6(pwm_samp_rate), focus_s
    plt.text(0.5, 0.99, str(int(measured_delay)), fontsize=14, va='top', ha='
plt.show()
```



In [292]:

```
band = 200
view = 500

df = pd.DataFrame(columns=['symbol', 'rising_delay', 'falling_delay'])
for i in range(10):
    df = df.append({'symbol': i}, ignore_index=True)

signal = get_tune6(pwm_samp_rate)

for i in [1,2,3,5,7,9]: # rising
    focus_start = pwm_samp_rate * 114e-6 * i - band
    focus_end   = pwm_samp_rate * 114e-6 * i - band + view
    expected_up_cycle = pwm_samp_rate * 114e-6 * i
    measured_delay = get_first_rising_edge(signal, focus_start, focus_end) -
    df.loc[df['symbol'] == i, 'rising_delay'] = measured_delay

for i in [0,1,2,3,5,7,9]: # falling
    focus_start = pwm_samp_rate * 114e-6 * (i+1) - pwm_samp_rate * 14e-6 - 50
    focus_end   = pwm_samp_rate * 114e-6 * (i+1) - pwm_samp_rate * 14e-6 - 50
    expected_down_cycle = pwm_samp_rate * 114e-6 * (i+1) - pwm_samp_rate * 14
    measured_delay = get_first_falling_edge(signal, focus_start, focus_end) -
    df.loc[df['symbol'] == i, 'falling_delay'] = measured_delay

display(df)
```

	symbol	rising_delay	falling_delay
0	0.0	NaN	40.0
1	1.0	48.0	72.0
2	2.0	80.0	104.0
3	3.0	96.0	136.0
4	4.0	NaN	NaN
5	5.0	128.0	168.0
6	6.0	NaN	NaN
7	7.0	160.0	200.0
8	8.0	NaN	NaN
9	9.0	192.0	232.0

It seems pretty clear from these measurements that the symbols emitted are about 40 cycles too long and the inter-symbol sleep (in the sync section at least) as about 8 cycles too long. 8 cycles too long is in the margin of error though since the GPIO outputs are sync'd on a /8 clock.

We'll try to use `round` instead of floor.

In [294]:

```
def render_asm_sleep(cycles):
    print('''\
    asm("        .newblock");
    asm("        LDI32    r1, %d ; sleep %d cycles total");
    asm("$1:     SUB      r1, r1, 1");
    asm("        QBNE     $1, r1, 0");\
''' % (round((cycles - 2)/2), cycles) )

def render_pos_toggle(index):
    if index % 2 == 0:
        print('    asm("        SET\tr30, r30, 1");')
    else:
        print('    asm("        CLR\tr30, r30, 1");')

def render_neg_toggle(index):
    render_pos_toggle(index + 1)

def render_asm_emit_symbol(sleeps, toggler):
    for i in range(len(sleeps)-1):
        toggler(i)
        render_asm_sleep(sleeps[i]
                         - 0 #custom fudge factor
                        )
    toggler(len(sleeps) - 1)
    print('    // caller is responsible for %d cycle delay' % sleeps[len(slee

print('''
// WARNING: we use r1 here because it appeared to be a safe choice when looki
// the caller. R0 is also listed as a save-on-call register in the calling
// conventions in http://www.ti.com/lit/ug/spruhv7b/spruhv7b.pdf but it turne
// out not to be a good choice. YMMV.\
''')
print('void emit_pos_symbol() {')
render_asm_emit_symbol(dumbest_pwm(generate_single_chirp(pwm_samp_rate)), ren
print('}\n\n#define EMIT_POS_SYMBOL_FINAL_CYCLES 504')

print('''
// WARNING: we use r1 here because it appeared to be a safe choice when looki
// the caller. R0 is also listed as a save-on-call register in the calling
// conventions in http://www.ti.com/lit/ug/spruhv7b/spruhv7b.pdf but it turne
// out not to be a good choice. YMMV.\
''')
print('void emit_neg_symbol() {')
render_asm_emit_symbol(dumbest_pwm(generate_single_chirp(pwm_samp_rate)), ren
print('}\n\n#define EMIT_NEG_SYMBOL_FINAL_CYCLES 504')
```

```
asm("        LDI32    r1, 276 ; sleep 555 cycles total");
asm("$1:     SUB      r1, r1, 1");
asm("        QBNE     $1, r1, 0");
asm("        SET    r30, r30, 1");
asm("        .newblock");
asm("        LDI32    r1, 272 ; sleep 546 cycles total");
asm("$1:     SUB      r1, r1, 1");
asm("        QBNE     $1, r1, 0");
asm("        CLR    r30, r30, 1");
asm("        .newblock");
asm("        LDI32    r1, 268 ; sleep 538 cycles total");
asm("$1:     SUB      r1, r1, 1");
asm("        QBNE     $1, r1, 0");
asm("        SET    r30, r30, 1");
```

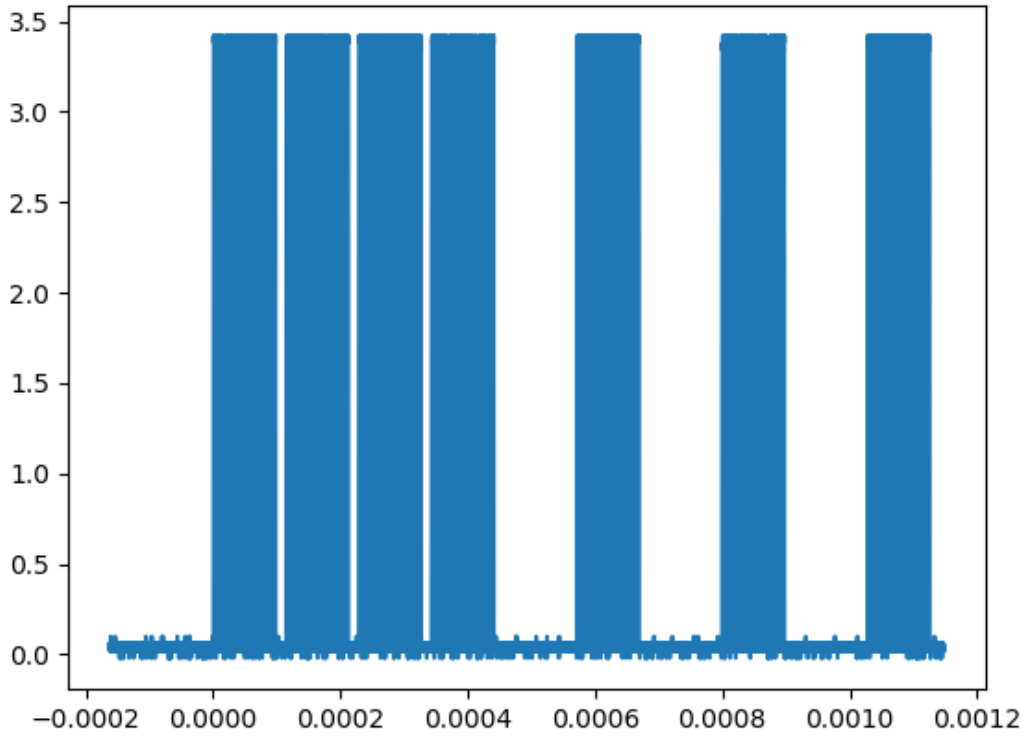
```
asm("        .newblock");
asm("        LDI32    r1, 264 ; sleep 531 cycles total");
asm("$1:     SUB      r1, r1, 1");
asm("        QBNE     $1, r1, 0");

asm("        CLR    r30, r30, 1");
```

Tune 7: Tune or Die

We trie to use the `round()` fnuction in calculating the pwm sleeps instead of math.floor to see if that would account for the extra 40 or so cycles slept in each symbol

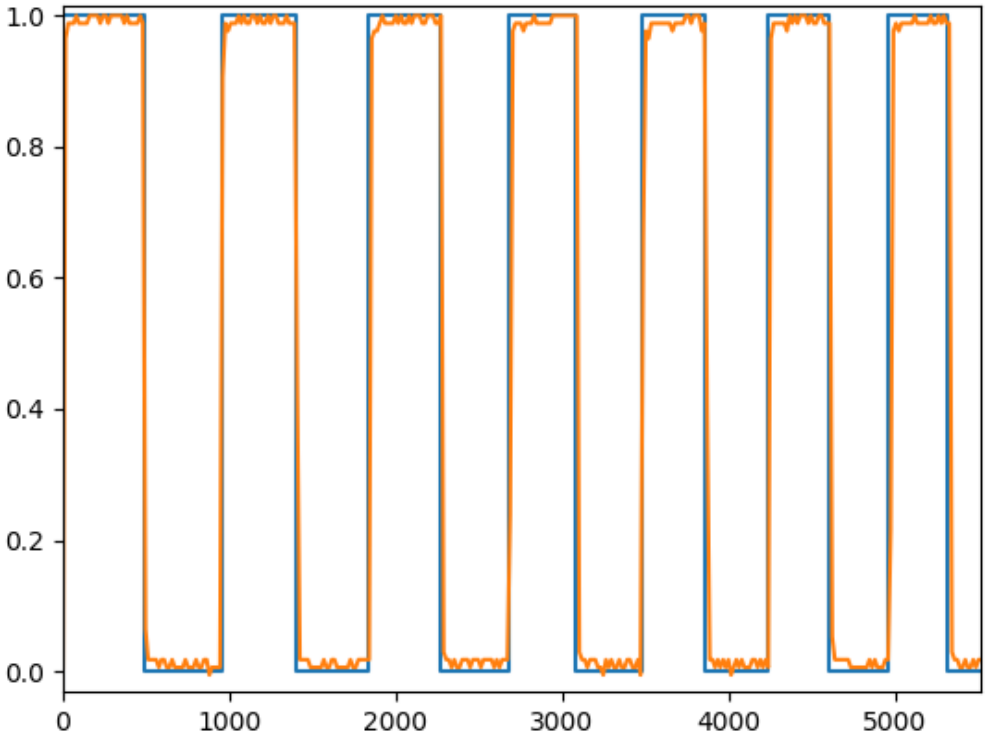
In [295]:

```
df = pd.read_csv('tune7_gpiopwm_0a00_preambleonly.csv')

%matplotlib notebook
plt.plot(df['time'], df['voltage'])
plt.show()
```



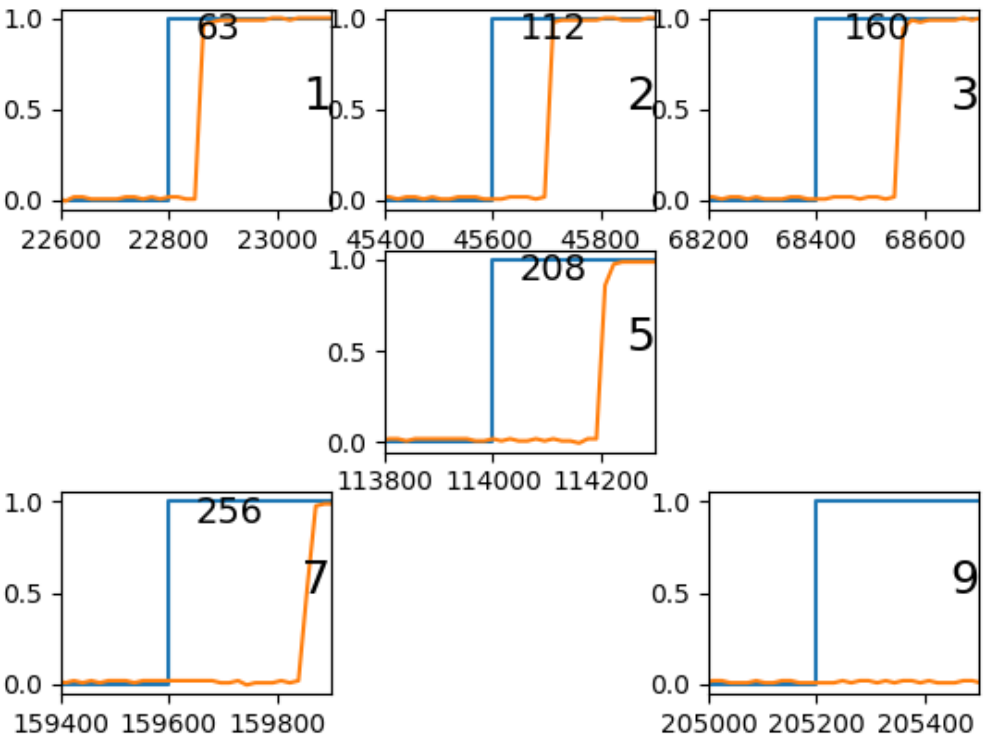
In [296]:

```
def get_tune7(samp_rate):
    df = pd.read_csv('tune7_gpiopwm_0a00_preambleonly.csv')
    df = df[ df['time'] >= 2.07638e-7] # manually-selected start of symbol

    #df['voltage'] = df['voltage'] * -1
    #df['voltage'] = df['voltage'] - df['voltage'].mean()
    df['voltage'] = df['voltage'] / df['voltage'].max()

    df['time'] = df['time'] * samp_rate
    df['time'] = df['time'] - df['time'].values[0]
    return df

def plot_tune7(samp_rate):
    df = get_tune7(samp_rate)
    plt.plot(df['time'], df['voltage'])

%matplotlib notebook
#plt.plot(generate_preamble_signal(b'\xaa', pwm_samp_rate))
plot_viz_pru(generate_preamble_sleeps(b'\xaa', pwm_samp_rate))
plot_tune7(pwm_samp_rate)
plt.xlim([0, pwm_samp_rate * 114e-6 * 12])
plt.show()
```



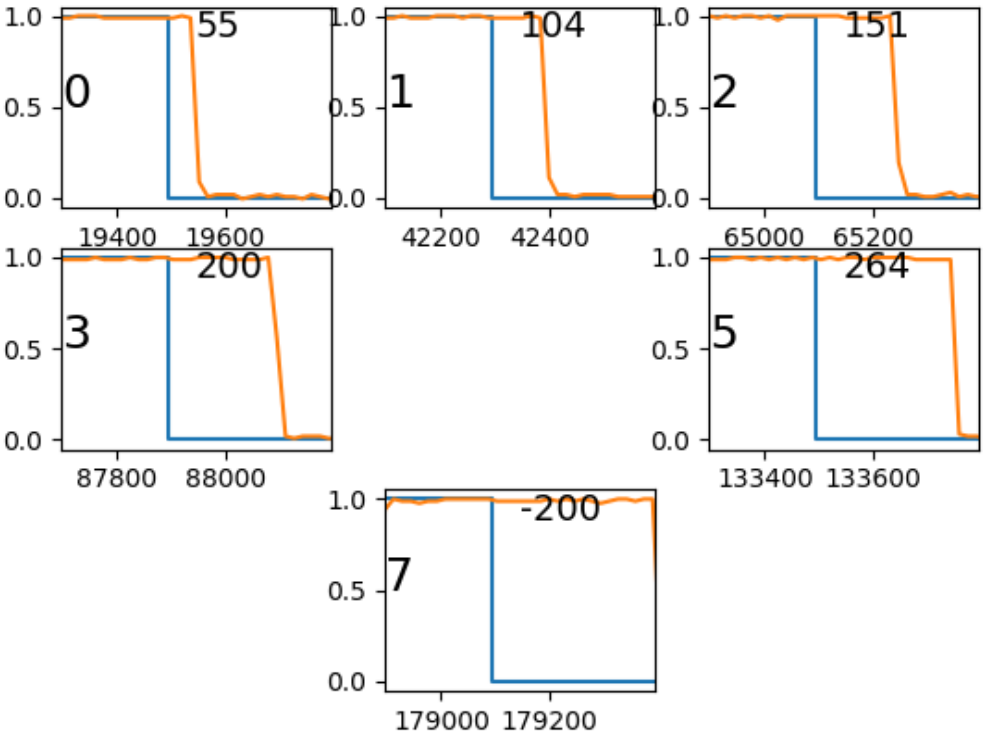
In [299]:

```
band = 200
view = 500
%matplotlib notebook

for i in [1,2,3,5,7,9]:
    ax = plt.subplot(3,3,i)
    #plt.plot(generate_preamble_signal(b'\xaa', pwm_samp_rate))
    plot_viz_pru(generate_preamble_sleeps(b'\xaa', pwm_samp_rate))
    plot_tune7(pwm_samp_rate)
    focus_start = pwm_samp_rate * 114e-6 * i - band
    focus_end   = pwm_samp_rate * 114e-6 * i - band + view
    plt.xlim([focus_start, focus_end])
    plt.text(1.0, 0.5, str(i), fontsize=18, ha='right', transform=ax.transAxe
    expected_up_cycle = pwm_samp_rate * 114e-6 * i
    try:
        measured_delay = get_first_rising_edge(get_tune7(pwm_samp_rate), focu
        plt.text(0.5, 0.99, str(int(measured_delay)), fontsize=14, va='top',
    except:
        continue
plt.show()
```



In [300]:

```
band = 200
view = 500
%matplotlib notebook

for i in [1,2,3,4,6,8]:
    ax = plt.subplot(3,3,i)
    #plt.plot(generate_preamble_signal(b'\xaa', pwm_samp_rate))
    plot_viz_pru(generate_preamble_sleeps(b'\xaa', pwm_samp_rate))
    plot_tune7(pwm_samp_rate)
    focus_start = pwm_samp_rate * 114e-6 * i - pwm_samp_rate * 14e-6 - 504 -
    focus_end   = pwm_samp_rate * 114e-6 * i - pwm_samp_rate * 14e-6 - 504 -
    plt.xlim([focus_start, focus_end])
    plt.text(0.0, 0.5, str(i-1), fontsize=18, ha='left', transform=ax.transAx
    expected_down_cycle = pwm_samp_rate * 114e-6 * i - pwm_samp_rate * 14e-6
    try:
        measured_delay = get_first_falling_edge(get_tune7(pwm_samp_rate), foc
        plt.text(0.5, 0.99, str(int(measured_delay)), fontsize=14, va='top',
    except:
        continue
plt.show()
```



It seems like using `round` made the slowness much closer to precisely 1 extra cycle per sleep (i.e. 52). This is good. We can adapt the cycles calculated per sleep.

In [360]:

```
def render_asm_sleep(cycles):
    print('''\
    asm("        .newblock");
    asm("        LDI32    r1, %d ; sleep %d cycles total");
    asm("$1:     SUB      r1, r1, 1");
    asm("        QBNE     $1, r1, 0");\
''' % (math.floor((cycles - 3)/2), cycles) )
    if cycles % 2 == 0:
        print('    asm("        XOR      r1, r1, r1 ; for even number of slee

def render_pos_toggle(index):
    if index % 2 == 0:
        print('    asm("        SET      r30, r30, 1");')
    else:
        print('    asm("        CLR      r30, r30, 1");')

def render_neg_toggle(index):
    render_pos_toggle(index + 1)

def render_asm_emit_symbol(sleeps, toggler):
    for i in range(len(sleeps)-1):
        toggler(i)
        render_asm_sleep(sleeps[i])
    toggler(len(sleeps) - 1)
    print('    // caller is responsible for %d cycle delay' % sleeps[len(slee

print('''
// WARNING: we use r1 here because it appeared to be a safe choice when looki
// the caller. R0 is also listed as a save-on-call register in the calling
// conventions in http://www.ti.com/lit/ug/spruhv7b/spruhv7b.pdf but it turne
// out not to be a good choice. YMMV.\
''')
print('void emit_pos_symbol() {')
render_asm_emit_symbol(dumbest_pwm(generate_single_chirp(pwm_samp_rate)), ren
print('}\n\n#define EMIT_POS_SYMBOL_FINAL_CYCLES 504')

print('''
// WARNING: we use r1 here because it appeared to be a safe choice when looki
// the caller. R0 is also listed as a save-on-call register in the calling
// conventions in http://www.ti.com/lit/ug/spruhv7b/spruhv7b.pdf but it turne
// out not to be a good choice. YMMV.\
''')
print('void emit_neg_symbol() {')
render_asm_emit_symbol(dumbest_pwm(generate_single_chirp(pwm_samp_rate)), ren
print('}\n\n#define EMIT_NEG_SYMBOL_FINAL_CYCLES 504')
```

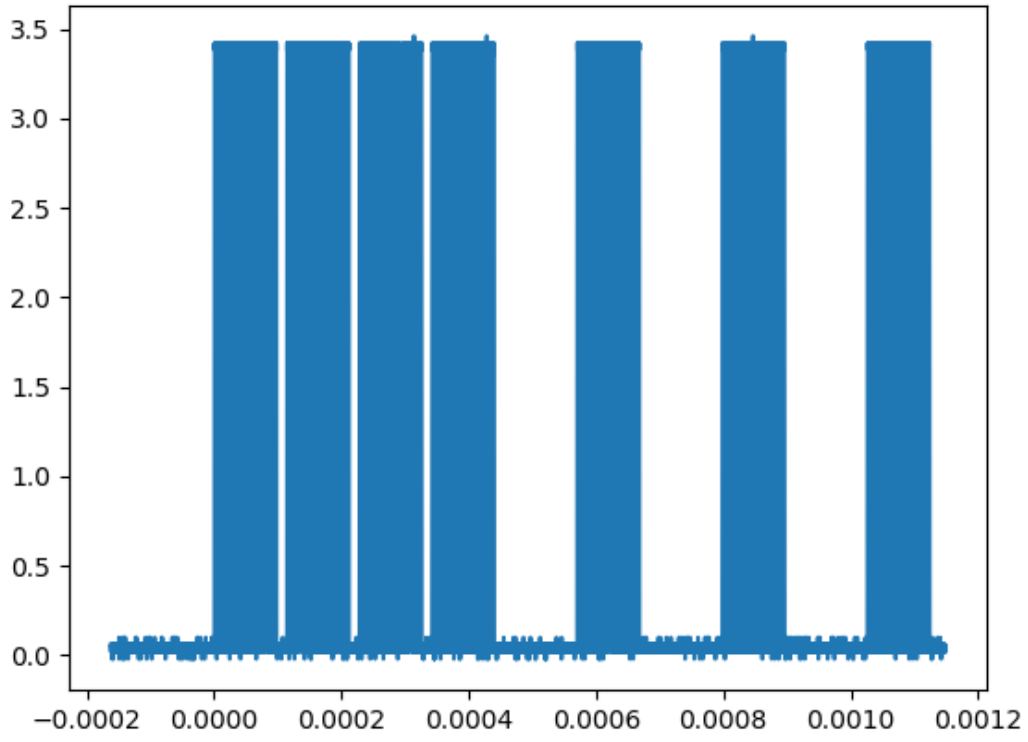
```
asm("        LDI32    r1, 267 ; sleep 538 cycles total");
asm("$1:     SUB      r1, r1, 1");
asm("        QBNE     $1, r1, 0");
asm("        XOR      r1, r1, r1 ; for even number of sleep cycles");
asm("        SET      r30, r30, 1");
asm("        .newblock");
asm("        LDI32    r1, 264 ; sleep 531 cycles total");
asm("$1:     SUB      r1, r1, 1");
asm("        QBNE     $1, r1, 0");
asm("        CLR      r30, r30, 1");
asm("        .newblock");
asm("        LDI32    r1, 260 ; sleep 523 cycles total");
asm("$1:     SUB      r1, r1, 1");
asm("        QBNE     $1, r1, 0");
```

```
asm("        SET      r30, r30, 1");
asm("        .newblock");
asm("        LDI32    r1, 257 ; sleep 517 cycles total");
asm("$1:     SUB      r1, r1, 1");
asm("        QBNE     $1, r1, 0");
```

Tune8: Why are there so many of these?

We implemented the above where we use `round` and also adjust by -1 the total cycles delayed in the loop

In [361]:
```
df = pd.read_csv('tune8_gpiopwm_0a00_preambleonly.csv')

%matplotlib notebook
plt.plot(df['time'], df['voltage'])
plt.show()
```

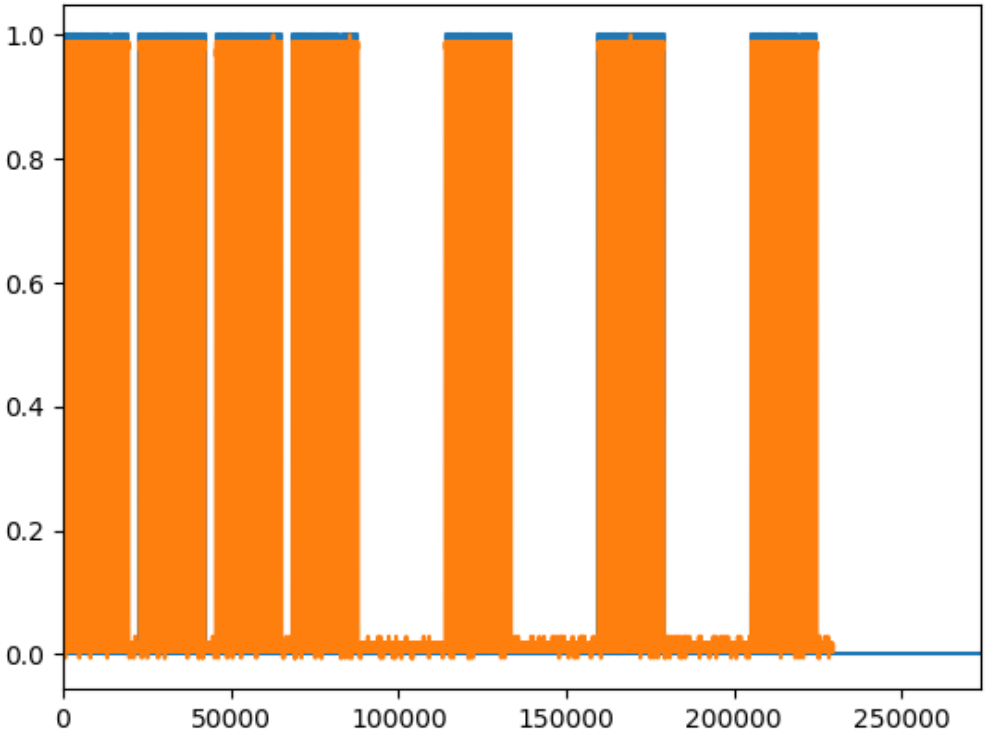


In [353]:
```
df = pd.read_csv('tune8_gpiopwm_0a00_preambleonly.csv')

focus_start = df.iloc[0]['time']
focus_end = focus_start + 0.001
focus = df[df['time'].between(focus_start, focus_end)]
focus[focus['voltage'] > 0.15].iloc[0]['time']
```

Out[353]: 3.19999999999997e-07

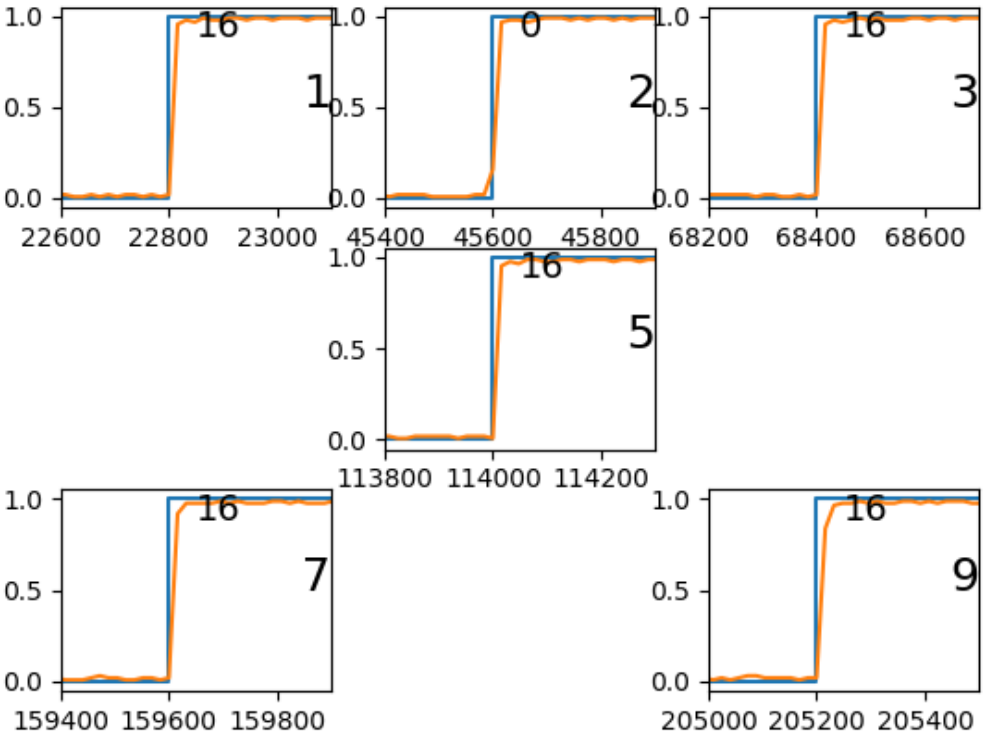
In [364]:

```
def get_tune8(samp_rate):
    df = pd.read_csv('tune8_gpiopwm_0a00_preambleonly.csv')

    focus_start = df.iloc[0]['time']
    focus_end = focus_start + 0.001
    focus = df[df['time'].between(focus_start, focus_end)]
    first_edge = focus[focus['voltage'] > 0.15].iloc[0]['time']

    df = df[ df['time'] >= first_edge ]

    #df['voltage'] = df['voltage'] * -1
    #df['voltage'] = df['voltage'] - df['voltage'].mean()
    df['voltage'] = df['voltage'] / df['voltage'].max()

    df['time'] = df['time'] * samp_rate
    df['time'] = df['time'] - df['time'].values[0]
    return df

def plot_tune8(samp_rate):
    df = get_tune8(samp_rate)
    plt.plot(df['time'], df['voltage'])

%matplotlib notebook
#plt.plot(generate_preamble_signal(b'\xaa', pwm_samp_rate))
plot_viz_pru(generate_preamble_sleeps(b'\xaa', pwm_samp_rate))
plot_tune8(pwm_samp_rate)
plt.xlim([0, pwm_samp_rate * 114e-6 * 12])
plt.show()
```



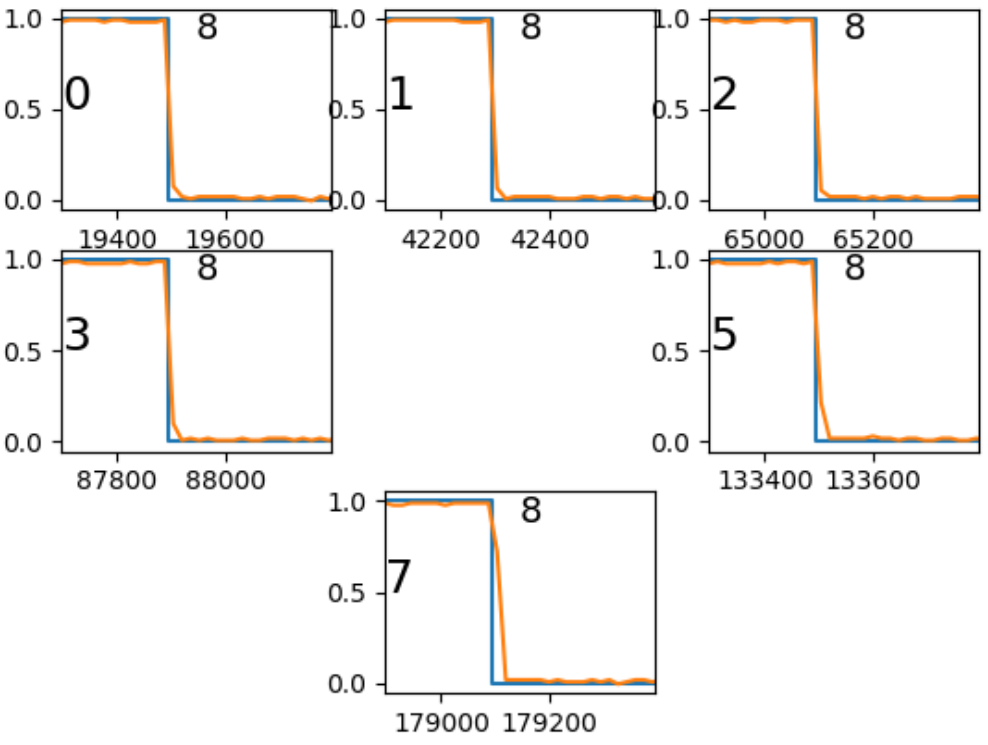
In [368]:

```
band = 200
view = 500
%matplotlib notebook

for i in [1,2,3,5,7,9]:
    ax = plt.subplot(3,3,i)
    #plt.plot(generate_preamble_signal(b'\xaa', pwm_samp_rate))
    plot_viz_pru(generate_preamble_sleeps(b'\xaa', pwm_samp_rate))
    plot_tune8(pwm_samp_rate)
    focus_start = pwm_samp_rate * 114e-6 * i - band
    focus_end   = pwm_samp_rate * 114e-6 * i - band + view
    plt.xlim([focus_start, focus_end])
    plt.text(1.0, 0.5, str(i), fontsize=18, ha='right', transform=ax.transAxe
    expected_up_cycle = pwm_samp_rate * 114e-6 * i
    try:
        measured_delay = get_first_rising_edge(get_tune8(pwm_samp_rate), focu
        plt.text(0.5, 0.99, str(int(measured_delay)), fontsize=14, va='top',
    except:
        continue
plt.show()
```



In [369]:

```
band = 200
view = 500
%matplotlib notebook

for i in [1,2,3,4,6,8]:
    ax = plt.subplot(3,3,i)
    #plt.plot(generate_preamble_signal(b'\xaa', pwm_samp_rate))
    plot_viz_pru(generate_preamble_sleeps(b'\xaa', pwm_samp_rate))
    plot_tune8(pwm_samp_rate)
    focus_start = pwm_samp_rate * 114e-6 * i - pwm_samp_rate * 14e-6 - 504 -
    focus_end   = pwm_samp_rate * 114e-6 * i - pwm_samp_rate * 14e-6 - 504 -
    plt.xlim([focus_start, focus_end])
    plt.text(0.0, 0.5, str(i-1), fontsize=18, ha='left', transform=ax.transAx
    expected_down_cycle = pwm_samp_rate * 114e-6 * i - pwm_samp_rate * 14e-6
    try:
        measured_delay = get_first_falling_edge(get_tune8(pwm_samp_rate), foc
        plt.text(0.5, 0.99, str(int(measured_delay)), fontsize=14, va='top',
    except:
        continue
plt.show()
```



In [371]:

```
band = 200
view = 500

df = pd.DataFrame(columns=['symbol', 'rising_delay', 'falling_delay'])
for i in range(10):
    df = df.append({'symbol': i}, ignore_index=True)

signal = get_tune8(pwm_samp_rate)

for i in [1,2,3,5,7,9]: # rising
    focus_start = pwm_samp_rate * 114e-6 * i - band
    focus_end   = pwm_samp_rate * 114e-6 * i - band + view
    expected_up_cycle = pwm_samp_rate * 114e-6 * i
    measured_delay = get_first_rising_edge(signal, focus_start, focus_end) -
    df.loc[df['symbol'] == i, 'rising_delay'] = measured_delay

for i in [0,1,2,3,5,7,9]: # falling
    focus_start = pwm_samp_rate * 114e-6 * (i+1) - pwm_samp_rate * 14e-6 - 50
    focus_end   = pwm_samp_rate * 114e-6 * (i+1) - pwm_samp_rate * 14e-6 - 50
    expected_down_cycle = pwm_samp_rate * 114e-6 * (i+1) - pwm_samp_rate * 14
    measured_delay = get_first_falling_edge(signal, focus_start, focus_end) -
    df.loc[df['symbol'] == i, 'falling_delay'] = measured_delay

display(df)
```

	symbol	rising_delay	falling_delay
0	0.0	NaN	8.0
1	1.0	16.0	8.0
2	2.0	0.0	8.0
3	3.0	16.0	8.0
4	4.0	NaN	NaN
5	5.0	16.0	8.0
6	6.0	NaN	NaN
7	7.0	16.0	8.0
8	8.0	NaN	NaN
9	9.0	16.0	24.0

Whew. It looks like we finally found the right tuning there. The trick is `LDI32` takes 3 cycles it seems and it also matters to emit a NOP (`XOR r1,r1,r1`) in the case where an even number of cycles needs to be delayed.

```
def render_asm_sleep(cycles):
    print('''\
    asm("        .newblock");
    asm("        LDI32    r1, %d ; sleep %d cycles total");
    asm("$1:     SUB      r1, r1, 1");
    asm("        QBNE     $1, r1, 0");\
''' % (math.floor((cycles - 3)/2), cycles) )
    if cycles % 2 == 0:
        print('    asm("        XOR      r1, r1, r1 ; for even number of slee
p cycles");')
```

Looking at Full Captures

We moved to examining the timings when we emit more than just the preamble (i.e. in the above investigation we had disabled all but the preamble)

We ran into an issue immeadiately , where the calling code was using `r1` and not saving-on-call as noted in the manual. We only need one scratch register for the emitting symbol delays. We guessed next that `r14` would be a good choice because it is the register used for returned values. But that didn't pan out either. What we ultimately had success with was saving and restoring to/from stack the `r0` register we use in the emit function.

In [387]:

```
def render_asm_sleep(cycles):
    print('''\
    asm("        .newblock");
    asm("        LDI32    r0, %d ; sleep %d cycles total");
    asm("$1:     SUB      r0, r0, 1");
    asm("        QBNE     $1, r0, 0");\
''' % (math.floor((cycles - 3)/2), cycles) )
    if cycles % 2 == 0:
        print('    asm("        XOR      r0, r0, r0 ; for even number of slee

def render_pos_toggle(index):
    if index % 2 == 0:
        print('    asm("        SET      r30, r30, 1");')
    else:
        print('    asm("        CLR      r30, r30, 1");')

def render_neg_toggle(index):
    render_pos_toggle(index + 1)

def render_asm_emit_symbol(sleeps, toggler):
    for i in range(len(sleeps)-1):
        toggler(i)
        render_asm_sleep(sleeps[i])
    toggler(len(sleeps) - 1)
    print('    // caller is responsible for %d cycle delay' % sleeps[len(slee

print('void emit_pos_symbol() {')
print('    asm("        SUB      r2, r2, 8");')
print('    asm("        SBBO     &r0, r2, 0, 4");')
render_asm_emit_symbol(dumbest_pwm(generate_single_chirp(pwm_samp_rate)), ren
print('    asm("        LBBO     &r0, r2, 0, 4");')
print('    asm("        ADD      r2, r2, 8");')
print('}\n\n#define EMIT_POS_SYMBOL_FINAL_CYCLES 504')
print('')
print('void emit_neg_symbol() {')
print('    asm("        SUB      r2, r2, 8");')
print('    asm("        SBBO     &r0, r2, 0, 4");')
render_asm_emit_symbol(dumbest_pwm(generate_single_chirp(pwm_samp_rate)), ren
print('    asm("        LBBO     &r0, r2, 0, 4");')
print('    asm("        ADD      r2, r2, 8");')
print('}\n\n#define EMIT_NEG_SYMBOL_FINAL_CYCLES 504')
```

```
asm("        LDI32    r0, 267 ; sleep 538 cycles total");
asm("$1:     SUB      r0, r0, 1");
asm("        QBNE     $1, r0, 0");
asm("        XOR      r0, r0, r0 ; for even number of sleep cycles");
asm("        SET      r30, r30, 1");
asm("        .newblock");
asm("        LDI32    r0, 264 ; sleep 531 cycles total");
asm("$1:     SUB      r0, r0, 1");
asm("        QBNE     $1, r0, 0");
asm("        CLR      r30, r30, 1");
asm("        .newblock");
asm("        LDI32    r0, 260 ; sleep 523 cycles total");
asm("$1:     SUB      r0, r0, 1");
asm("        QBNE     $1, r0, 0");
asm("        SET      r30, r30, 1");
asm("        .newblock");
asm("        LDI32    r0, 257 ; sleep 517 cycles total");
asm("$1:     SUB      r0, r0, 1");
```

```
asm("        QBNE     $1, r0, 0");
asm("        CLR      r30, r30, 1");
```

We used the above and tweaked the timings of the functions sending the preamble and the message bodies too

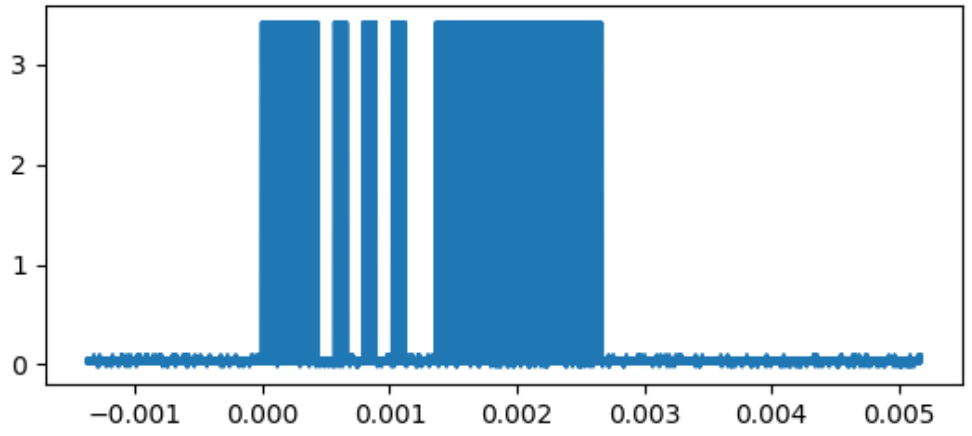
```
int hw_send_preamble(volatile tx_frame_t *msg) {
    //emit negative preamble symbol
    emit_pos_symbol();
    __delay_cycles(EMIT_POS_SYMBOL_FINAL_CYCLES
                   - 6 // overhead of emit_pos_symbol() call
                   + PREAMBLE_EXTRA_CYCLES // silence time for preamble symb
ols only
                   );
    //emit negative preamble symbol
    emit_pos_symbol();
    __delay_cycles(EMIT_POS_SYMBOL_FINAL_CYCLES
                   - 6 // overhead of emit_pos_symbol() call
                   + PREAMBLE_EXTRA_CYCLES // silence time for preamble symb
ols only
                   );
    //emit negative preamble symbol
    emit_pos_symbol();
    __delay_cycles(EMIT_POS_SYMBOL_FINAL_CYCLES
                   - 6 // overhead of emit_pos_symbol() call
                   + PREAMBLE_EXTRA_CYCLES // silence time for preamble symb
ols only
                   - 8 // overhead of loop initialization and test below
                   );

    for(int i=0; i < TX_FRAME_PREAMBLE_LEN; ++i) {
        if(msg->preamble & (1 << i)) {
            //emit positive preamble symbol
            asm("   clr r30, r30, 1");
            __delay_cycles(PREAMBLE_TOTAL_CYCLES
                           - 14 // loop and test overhead
                           );
        } else {
            //emit negative preamble symbol
            emit_pos_symbol();
            __delay_cycles(EMIT_POS_SYMBOL_FINAL_CYCLES
                           - 6 // overhead of emit_pos_symbol() call
                           + PREAMBLE_EXTRA_CYCLES // silence time for pream
ble symbols only
                           - 14 // loop and test overhead
                           );
        }
    }

    __delay_cycles(8 // loop and test overhead not executed in last pass
                   );
    //emit positive preamble symbol
    asm("   clr r30, r30, 1");
    //caller is responsible for delay of PREAMBLE_TOTAL_CYCLES
    return 0;
}
```

```
void hw_send_payload(volatile tx_frame_t *msg) {
    register uint8_t bit_length = msg->bit_length;

    for(uint8_t i=0; i < 4; ++i) {
        emit_pos_symbol();
        __delay_cycles(EMIT_POS_SYMBOL_FINAL_CYCLES
                       - 2 // loop overhead
                       - 6 // overhead of next emit_pos_symbol() call
                       );
    }
    __delay_cycles(2 /*loop overhead not executed*/);
    emit_pos_symbol();
    __delay_cycles(EMIT_POS_SYMBOL_FINAL_CYCLES
                   - 6 // overhead of emit_pos_symbol() call
                   - 17 // loop and test overhead below
                   );

    for(uint8_t i=0; i < bit_length; ++i) {
        if( msg->payload[ i / 8 ] & (1 << (i % 8)) ) { //TODO: check this is
 the correct bit direction
            emit_pos_symbol();
            __delay_cycles(EMIT_POS_SYMBOL_FINAL_CYCLES
                - 14 // loop and test overhead
                - 6 // overhead of next emit_pos_symbol() call
                );
        } else {
            emit_neg_symbol();
            __delay_cycles(EMIT_POS_SYMBOL_FINAL_CYCLES
                - 15 // loop and test overhead
                - 6 // overhead of next emit_pos_symbol() call
                );
        }
    }
    // loop exit above is only 2 cycles max, we will ignore

    emit_pos_symbol();
    __delay_cycles(EMIT_POS_SYMBOL_FINAL_CYCLES
                   - 1 // loop setup overhead
                   - 6 // overhead of next emit_pos_symbol() call
                   );
    for(uint8_t i=0; i < 5; ++i) {
        emit_pos_symbol();
        __delay_cycles(EMIT_POS_SYMBOL_FINAL_CYCLES
                       - 2 // loop overhead
                       - 6 // overhead of next emit_pos_symbol() call
                       );
    }
    emit_pos_symbol();
    //caller is responsible for delay of EMIT_POS_SYMBOL_FINAL_CYCLES
}
```

```
int __inline hw_send_frame(volatile tx_frame_t *msg) {
    tx_frame_t local_msg;

    // TODO: remove this
    msg->preamble = 0xaa;
    msg->bit_length = 1;
    msg->payload[0] = 0x55;
    msg->payload[1] = 0x55;
    msg->payload[2] = 0x55;
    msg->payload[3] = 0x55;

    // copy to a local SRAM buffer to avoid non-deterministic and long waits
 on DDR access during send
    memcpy((void *) &local_msg, (void *) msg, sizeof(tx_frame_t));

    // send preamble
    hw_send_preamble(&local_msg);
    __delay_cycles(PREAMBLE_TOTAL_CYCLES
                - 2 // return from hw_send_preamble overhead
                - 2 // hw_send_payload() call overhead below
                - 1 // first loop setup overhead in hw_send_payload()
                );

    // send payload (and checksum)
    hw_send_payload(&local_msg);
    __delay_cycles(EMIT_POS_SYMBOL_FINAL_CYCLES
                - 2 // return from hw_send_payload overhead
                );
    // sleep for bus access idle time
    asm("   clr r30, r30, 1");
    __delay_cycles(BUS_ACCESS_IDLE_CYCLES);

    return return_debug_info();
}
```

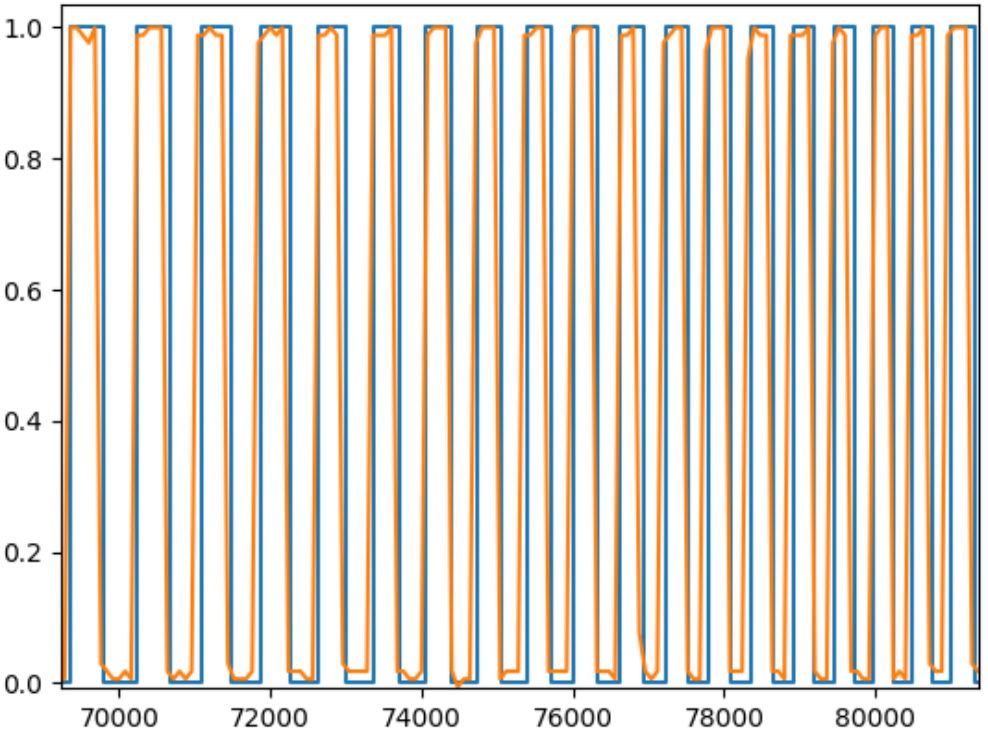
In [393]:

```
df = pd.read_csv('full1_gpiopwm_0a00_5555.csv')

%matplotlib notebook
plt.plot(df['time'], df['voltage'])
plt.show()

```



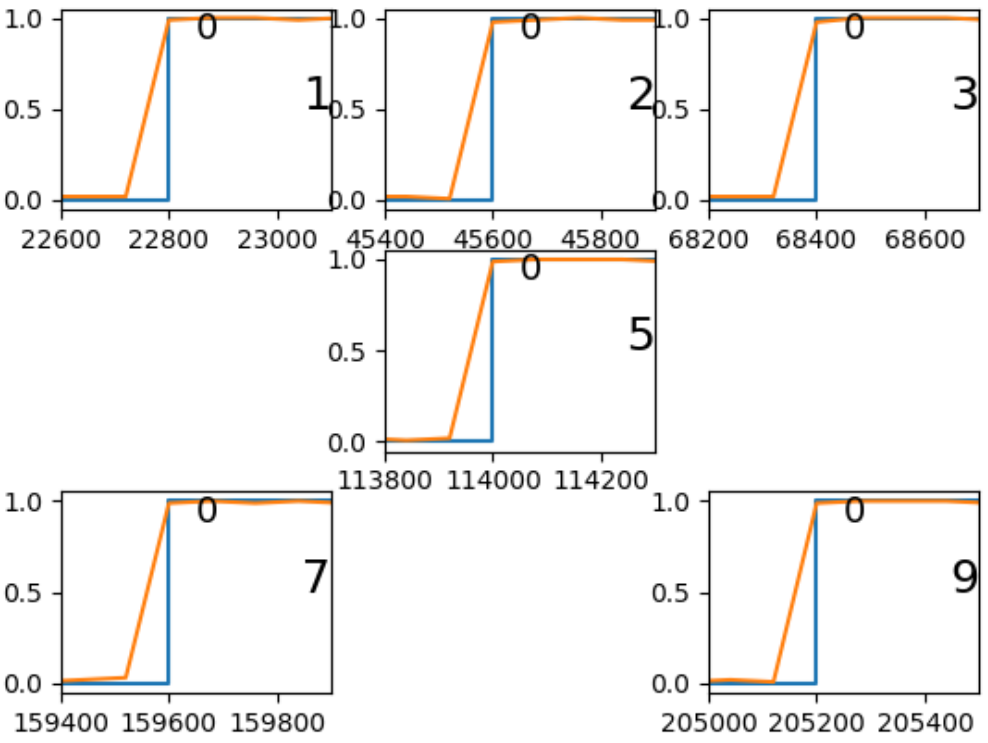
In [402]:

```
def get_full1(samp_rate):
    df = pd.read_csv('full1_gpiopwm_0a00_5555.csv')

    focus_start = df.iloc[0]['time']
    focus_end = 0.001
    focus = df[df['time'].between(focus_start, focus_end)]
    first_edge = focus[focus['voltage'] > 0.15].iloc[0]['time']

    df = df[ df['time'] >= first_edge ]

    #df['voltage'] = df['voltage'] * -1
    #df['voltage'] = df['voltage'] - df['voltage'].mean()
    df['voltage'] = df['voltage'] / df['voltage'].max()

    df['time'] = df['time'] * samp_rate
    df['time'] = df['time'] - df['time'].values[0]
    return df

def plot_full1(samp_rate):
    df = get_full1(samp_rate)
    plt.plot(df['time'], df['voltage'])

%matplotlib notebook
#plt.plot(generate_preamble_signal(b'\xaa', pwm_samp_rate))
plot_viz_pru(generate_preamble_sleeps(b'\xaa', pwm_samp_rate))
plot_full1(pwm_samp_rate)
plt.xlim([0, pwm_samp_rate * 114e-6 * 12])
plt.show()
```



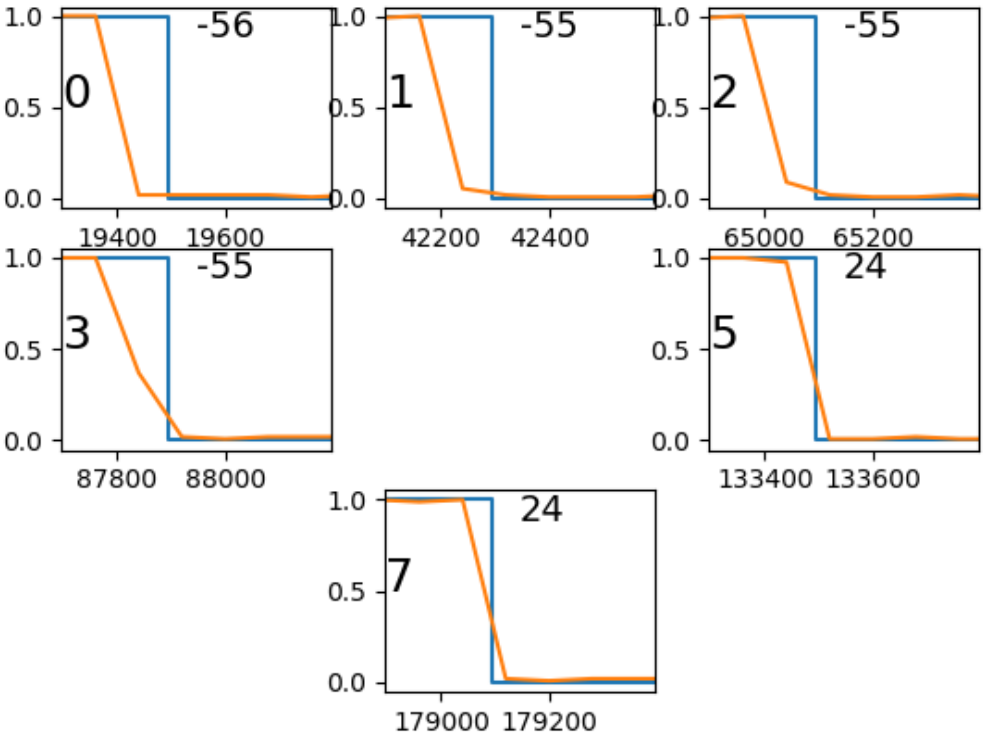
In [400]:

```
band = 200
view = 500
%matplotlib notebook

for i in [1,2,3,5,7,9]:
    ax = plt.subplot(3,3,i)
    plot_viz_pru(generate_preamble_sleeps(b'\xaa', pwm_samp_rate))
    plot_full1(pwm_samp_rate)
    focus_start = pwm_samp_rate * 114e-6 * i - band
    focus_end   = pwm_samp_rate * 114e-6 * i - band + view
    plt.xlim([focus_start, focus_end])
    plt.text(1.0, 0.5, str(i), fontsize=18, ha='right', transform=ax.transAxe
    expected_up_cycle = pwm_samp_rate * 114e-6 * i
    try:
        measured_delay = get_first_rising_edge(get_full1(pwm_samp_rate), focu
        plt.text(0.5, 0.99, str(int(measured_delay)), fontsize=14, va='top',
    except:
        continue
plt.show()
```



In [401]:

```
band = 200
view = 500
%matplotlib notebook

for i in [1,2,3,4,6,8]:
    ax = plt.subplot(3,3,i)
    plot_viz_pru(generate_preamble_sleeps(b'\xaa', pwm_samp_rate))
    plot_full1(pwm_samp_rate)
    focus_start = pwm_samp_rate * 114e-6 * i - pwm_samp_rate * 14e-6 - 504 -
    focus_end   = pwm_samp_rate * 114e-6 * i - pwm_samp_rate * 14e-6 - 504 -
    plt.xlim([focus_start, focus_end])
    plt.text(0.0, 0.5, str(i-1), fontsize=18, ha='left', transform=ax.transAx
    expected_down_cycle = pwm_samp_rate * 114e-6 * i - pwm_samp_rate * 14e-6
    try:
        measured_delay = get_first_falling_edge(get_full1(pwm_samp_rate), foc
        plt.text(0.5, 0.99, str(int(measured_delay)), fontsize=14, va='top',
    except:
        continue
plt.show()
```



In [399]:

```
band = 200
view = 500

df = pd.DataFrame(columns=['symbol', 'rising_delay', 'falling_delay'])
for i in range(10):
    df = df.append({'symbol': i}, ignore_index=True)

signal = get_full1(pwm_samp_rate)

for i in [1,2,3,5,7,9]: # rising
    focus_start = pwm_samp_rate * 114e-6 * i - band
    focus_end   = pwm_samp_rate * 114e-6 * i - band + view
    expected_up_cycle = pwm_samp_rate * 114e-6 * i
    measured_delay = get_first_rising_edge(signal, focus_start, focus_end) -
    df.loc[df['symbol'] == i, 'rising_delay'] = measured_delay

for i in [0,1,2,3,5,7,9]: # falling
    focus_start = pwm_samp_rate * 114e-6 * (i+1) - pwm_samp_rate * 14e-6 - 50
    focus_end   = pwm_samp_rate * 114e-6 * (i+1) - pwm_samp_rate * 14e-6 - 50
    expected_down_cycle = pwm_samp_rate * 114e-6 * (i+1) - pwm_samp_rate * 14
    measured_delay = get_first_falling_edge(signal, focus_start, focus_end) -
    df.loc[df['symbol'] == i, 'falling_delay'] = measured_delay

display(df)
```

	symbol	rising_delay	falling_delay
0	0.0	NaN	-56.0
1	1.0	1.818989e-11	-56.0
2	2.0	2.182787e-11	-56.0
3	3.0	2.910383e-11	-56.0
4	4.0	NaN	NaN
5	5.0	4.365575e-11	24.0
6	6.0	NaN	NaN
7	7.0	2.910383e-11	24.0
8	8.0	NaN	NaN
9	9.0	5.820766e-11	24.0

It looks like the adjustments in the symbol emitting functions to save/restore the `r0` scratch register didn't affect the timing in the preamble too much.

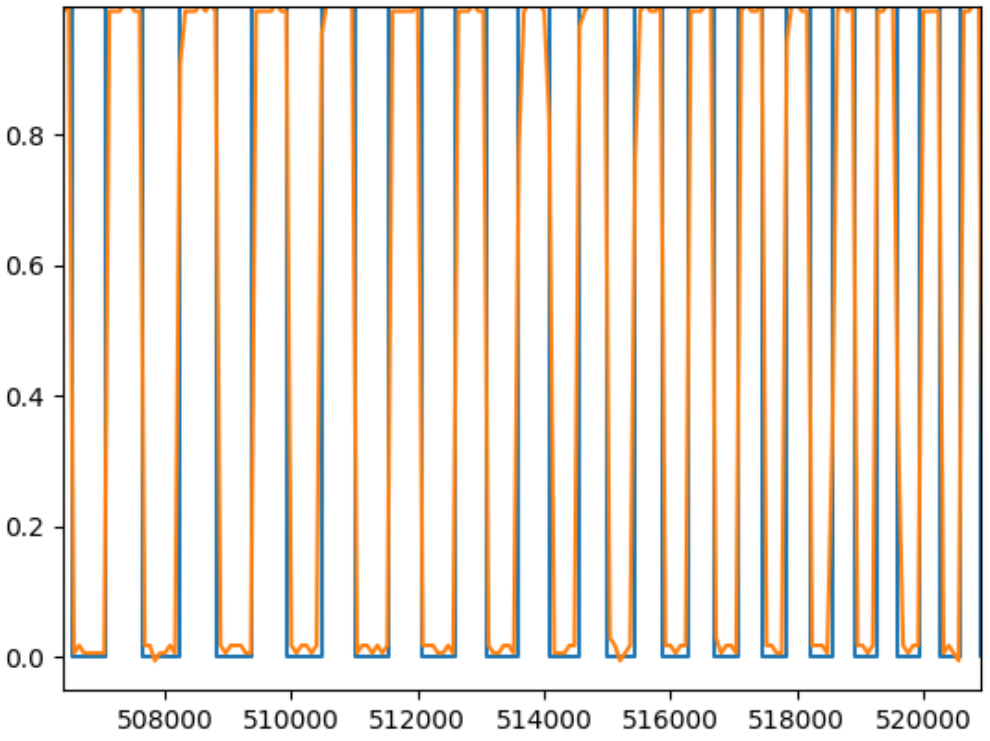
We need to look at the timing in the body of the plc signal emitted now. This will be tricky because there are no 'gaps' to use for convienient time measurement.

In [437]:

```
def get_payload_bits(payload, payload_bit_length):
    payload_bits = bitstring.BitArray()

    for b_int in bytes(payload):
        b_bytes = bytes([b_int])
        b_bits = bitstring.BitArray(bytes=b_bytes)
        b_bits.reverse()

        payload_bits.append(bitstring.ConstBitArray(bin='0')) # start bit
        payload_bits.append(b_bits) # bit-reversed byte
        payload_bits.append(bitstring.ConstBitArray(bin='1')) # stop bit

    checksum_bits = get_checksum_bits(payload)
    checksum_bits.reverse()

    payload_bits.append(bitstring.ConstBitArray(bin='0'))
    payload_bits.append(checksum_bits)
    payload_bits.append(bitstring.ConstBitArray(bin='1'))

    payload_bits = payload_bits[:payload_bit_length]

    payload_bits.prepend(bitstring.ConstBitArray(bin='11111'))
    payload_bits.append(bitstring.ConstBitArray(bin='1111111'))

    return payload_bits

def generate_payload_sleeps(j1708_message, samp_rate, payload_bit_length=None
    symbol = generate_single_chirp(pwm_samp_rate)
    pos_chirp_sleeps = dumbest_pwm(symbol)
    neg_chirp_sleeps = dumbest_pwm(-1 * symbol)

    sleeps = np.zeros(0, np.uint32)

    if payload_bit_length is None:
        payload_bit_length = len(j1708_message) * 8
    j2497_payload_bits = get_payload_bits(j1708_message, payload_bit_length)
    for n in j2497_payload_bits:
        if n:
            sleeps = numpy.append(sleeps, pos_chirp_sleeps)
        else:
            sleeps = numpy.append(sleeps, neg_chirp_sleeps)


    #append a minimum-length bus access time idle period so that returns from
    sleeps.flat[-1] = sleeps.flat[-1] + int(114e-6 * samp_rate) * 12 #  exten

    return sleeps
```

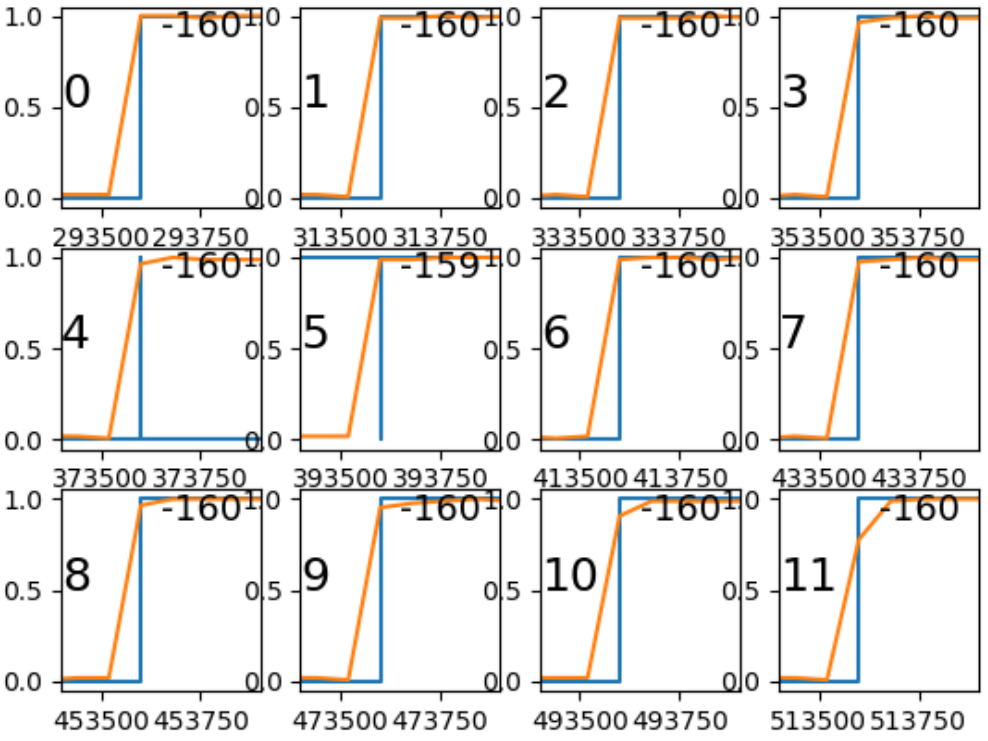
In [424]:

```
payload=b'\xaa\x55'
sleeps = np.zeros(0, np.uint32)
sleeps = numpy.append(sleeps, generate_preamble_sleeps(payload, pwm_samp_rate
sleeps =  numpy.append(sleeps, generate_payload_sleeps(payload, pwm_samp_rate

%matplotlib notebook
plot_viz_pru(sleeps)
plot_full1(pwm_samp_rate)
plt.xlim([0, pwm_samp_rate * (114e-6 * 12 + 100e-6 * (1+5+7+1) + 2e-6)])
plt.show()
```



In [425]:

```
band = 200
view = 500
%matplotlib notebook

for i in range(1, 13):
    ax = plt.subplot(3,4,i)
    plot_viz_pru(sleeps)
    plot_full1(pwm_samp_rate)
    expected_down_cycle = pwm_samp_rate * ( 114e-6 * 12 + 100e-6 * i )
    focus_start = expected_down_cycle - band
    focus_end   = focus_start + view
    plt.xlim([focus_start, focus_end])
    plt.text(0.0, 0.5, str(i-1), fontsize=18, ha='left', transform=ax.transAx

    try:
        measured_delay = get_first_falling_edge(get_full1(pwm_samp_rate), foc
        plt.text(0.5, 0.99, str(int(measured_delay)), fontsize=14, va='top',
    except:
        continue
plt.show()
```

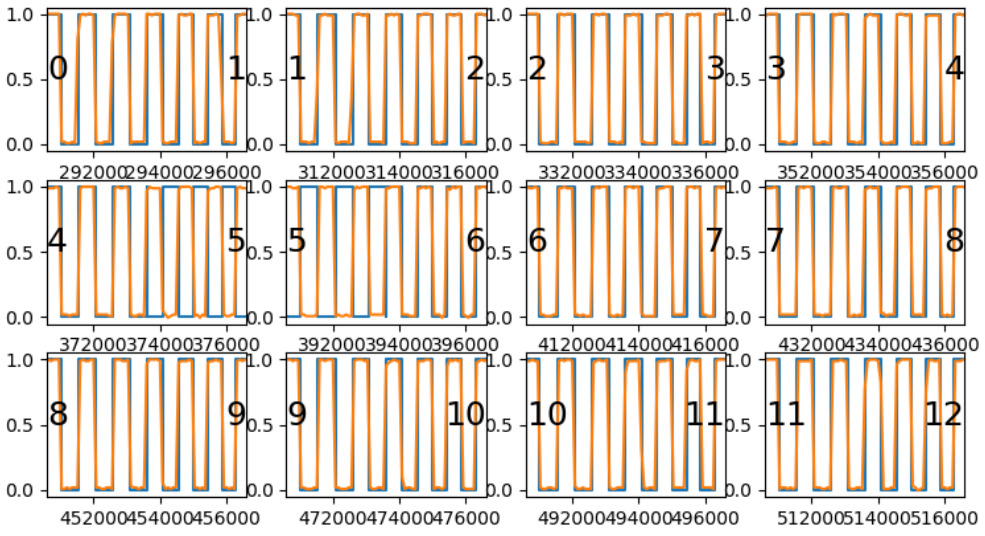


It looks like we need to have a ~80 cycle delay before moving to emitting the payload.

But otherwise the transitions line up well

In [428]:

```
band = pwm_samp_rate * 15e-6
view = 2 * band
%matplotlib notebook

for i in range(1, 13):
    ax = plt.subplot(3,4,i)
    plot_viz_pru(sleeps)
    plot_full1(pwm_samp_rate)
    expected_down_cycle = pwm_samp_rate * ( 114e-6 * 12 + 100e-6 * i )
    focus_start = expected_down_cycle - band
    focus_end   = focus_start + view
    plt.xlim([focus_start, focus_end])
    plt.text(0.0, 0.5, str(i-1), fontsize=18, ha='left', transform=ax.transAx
    plt.text(1.0, 0.5, str(i), fontsize=18, ha='right', transform=ax.transAxe

plt.show()
```



And based on how symbol 5 from the payload is out of phase I infer that the PRU code here is iterating through bits in the wrong direction.

Verifying Data sent from ARM userspace

We added a custom delay factor to accoutn for the persitent offset timing measured above and adjusted the bit iteration direction

```
--- a/src/pru/plc4trucksduck.c
+++ b/src/pru/plc4trucksduck.c
@@ -872,7 +872,7 @@ void hw_send_payload(volatile tx_frame_t *msg) {
                     );

     for(uint8_t i=0; i < bit_length; ++i) {
-        if( msg->payload[ i / 8 ] & (1 << (i % 8)) ) { //TODO: check this is
the correct
+        if( msg->payload[ i / 8 ] & (1 << (7-(i % 8))) ) { //TODO: check thi
s is the corr
             emit_pos_symbol();
             __delay_cycles(EMIT_POS_SYMBOL_FINAL_CYCLES
                 - 14 // loop and test overhead
@@ -924,6 +924,7 @@ int __inline hw_send_frame(volatile tx_frame_t *msg) {
                 - 2 // return from hw_send_preamble overhead
                 - 2 // hw_send_payload() call overhead below
                 - 1 // first loop setup overhead in hw_send_payload()
+                + 80 // custom fudge
                 );

     // send payload (and checksum)
```

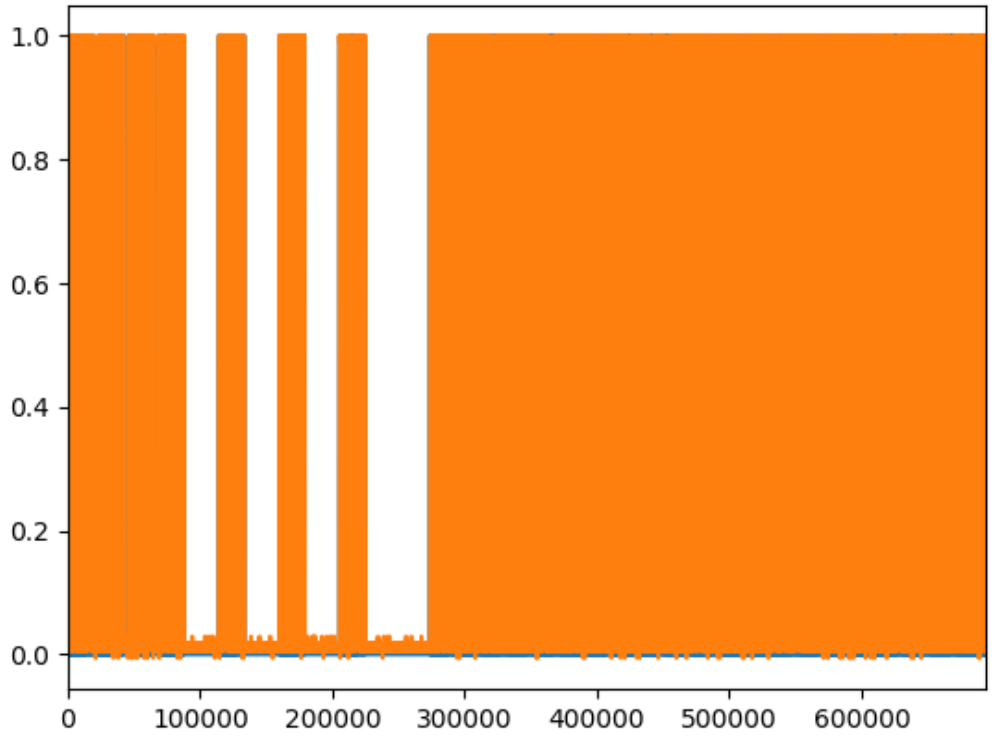
We also removed the manual setting of the payload and moved to having the userspace utility send the message 0xaaf0

In [462]:

```
def get_payload_bits(payload):
    payload_bits = bitstring.BitArray()

    payload_bits.append(bitstring.ConstBitArray(bin='11111'))
    for b_int in bytes(payload):
        b_bytes = bytes([b_int])
        b_bits = bitstring.BitArray(bytes=b_bytes)
        b_bits.reverse()

        payload_bits.append(bitstring.ConstBitArray(bin='0')) # start bit
        payload_bits.append(b_bits) # bit-reversed byte
        payload_bits.append(bitstring.ConstBitArray(bin='1')) # stop bit

    checksum_bits = get_checksum_bits(payload)
    checksum_bits.reverse()

    payload_bits.append(bitstring.ConstBitArray(bin='0'))
    payload_bits.append(checksum_bits)
    payload_bits.append(bitstring.ConstBitArray(bin='1'))

    payload_bits.append(bitstring.ConstBitArray(bin='1111111'))

    return payload_bits

def generate_payload_sleeps(j1708_message, samp_rate):
    symbol = generate_single_chirp(pwm_samp_rate)
    pos_chirp_sleeps = dumbest_pwm(symbol)
    neg_chirp_sleeps = dumbest_pwm(-1 * symbol)

    sleeps = np.zeros(0, np.uint32)

    j2497_payload_bits = get_payload_bits(j1708_message)
    for n in j2497_payload_bits:
        if n:
            sleeps = numpy.append(sleeps, pos_chirp_sleeps)
        else:
            sleeps = numpy.append(sleeps, neg_chirp_sleeps)


    #append a minimum-length bus access time idle period so that returns from
    sleeps.flat[-1] = sleeps.flat[-1] + int(114e-6 * samp_rate) * 12 #  exten

    return sleeps

def get_full_sleeps(payload, samp_rate):
    sleeps = np.zeros(0, np.uint32)
    sleeps = numpy.append(sleeps, generate_preamble_sleeps(payload, samp_rate
    sleeps = numpy.append(sleeps, generate_payload_sleeps(payload, samp_rate)
    return sleeps

```

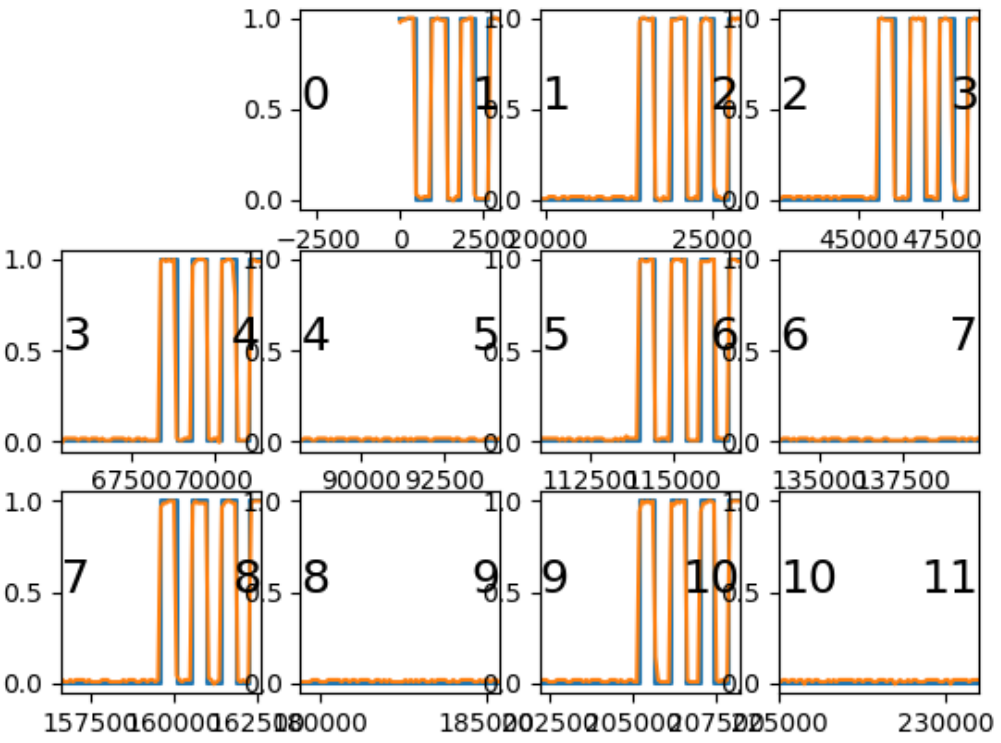
In [474]:

```
def get_full2(samp_rate):
    df = pd.read_csv('full2_gpiopwm_aaf0.csv')

    focus_start = df.iloc[0]['time']
    focus_end = 0.001
    focus = df[df['time'].between(focus_start, focus_end)]
    first_edge = focus[focus['voltage'] > 0.15].iloc[0]['time']

    df = df[ df['time'] >= first_edge ]

    #df['voltage'] = df['voltage'] * -1
    #df['voltage'] = df['voltage'] - df['voltage'].mean()
    df['voltage'] = df['voltage'] / df['voltage'].max()

    df['time'] = df['time'] * samp_rate
    df['time'] = df['time'] - df['time'].values[0]
    return df

def plot_full2(samp_rate):
    df = get_full2(samp_rate)
    plt.plot(df['time'], df['voltage'])

payload = b'\xaa'
sleeps = get_full_sleeps(payload, pwm_samp_rate)

%matplotlib notebook
plot_viz_pru(sleeps)
plot_full2(pwm_samp_rate)
plt.xlim([0, pwm_samp_rate * (114e-6 * 12 + 100e-6 * (21) + 2e-6)])
plt.show()
```



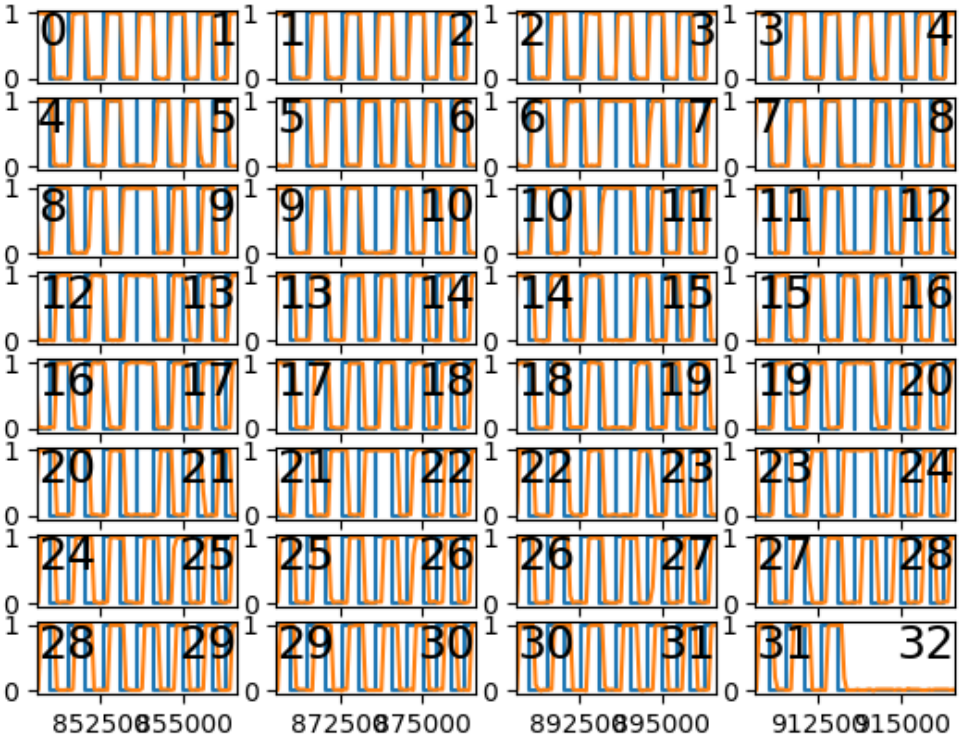
In [457]:

```
band = pwm_samp_rate * 15e-6
view = 2 * band
%matplotlib notebook

for i in range(0, 11):
    ax = plt.subplot(3,4,i+2)
    plot_viz_pru(sleeps)
    plot_full2(pwm_samp_rate)
    expected_down_cycle = pwm_samp_rate * ( 114e-6 * i )
    focus_start = expected_down_cycle - band
    focus_end   = focus_start + view
    plt.xlim([focus_start, focus_end])
    plt.text(0.0, 0.5, str(i), fontsize=18, ha='left', transform=ax.transAxes
    plt.text(1.0, 0.5, str(i+1), fontsize=18, ha='right', transform=ax.transA

plt.show()
```



In [475]:

```
band = pwm_samp_rate * 15e-6
view = 2 * band
%matplotlib notebook

for i in range(1, 33):
    ax = plt.subplot(8,4,i)
    plot_viz_pru(sleeps)
    plot_full2(pwm_samp_rate)
    expected_down_cycle = pwm_samp_rate * ( 114e-6 * 12 + 100e-6 * i )
    focus_start = expected_down_cycle - band
    focus_end   = focus_start + view
    plt.xlim([focus_start, focus_end])
    plt.text(0.0, 0.5, str(i-1), fontsize=18, ha='left', transform=ax.transAx
    plt.text(1.0, 0.5, str(i), fontsize=18, ha='right', transform=ax.transAxe

plt.show()
```



In [470]:

```
get_payload_bits(b'\xaa').bin
```

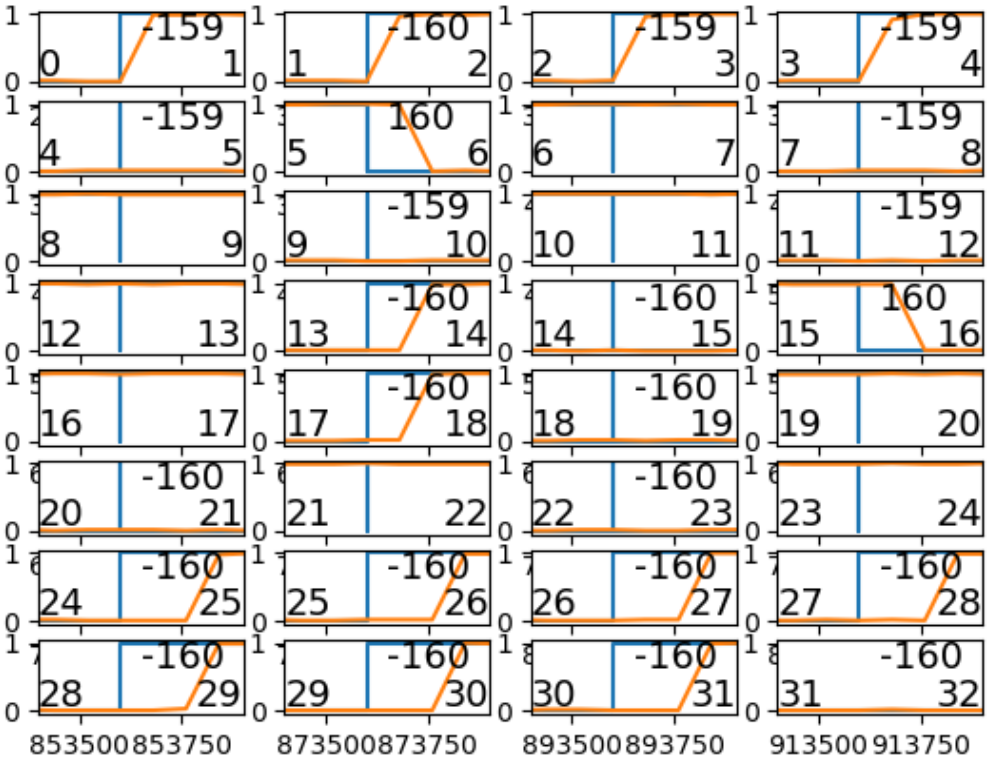
Out[470]: '11111001010101100110101011111111'

The payload bits were iterated in the correct order, the lenghts match up and the timings are 'devent' -- we could tighten up the payload body inter-symbol timings a little.

In [476]:

```
band = 200
view = 500
%matplotlib notebook

for i in range(1, 33):
    ax = plt.subplot(8,4,i)
    plot_viz_pru(sleeps)
    plot_full2(pwm_samp_rate)
    expected_down_cycle = pwm_samp_rate * ( 114e-6 * 12 + 100e-6 * i )
    focus_start = expected_down_cycle - band
    focus_end   = focus_start + view
    plt.xlim([focus_start, focus_end])
    plt.text(0.0, 0.5, str(i-1), fontsize=14, ha='left', va='top', transform=
    plt.text(1.0, 0.5, str(i), fontsize=14, ha='right', va='top', transform=a

    try:
        measured_delay = get_first_falling_edge(get_full2(pwm_samp_rate), foc
        plt.text(0.5, 0.99, str(int(measured_delay)), fontsize=14, va='top',
    except:
        continue

plt.show()
```

Figure 1



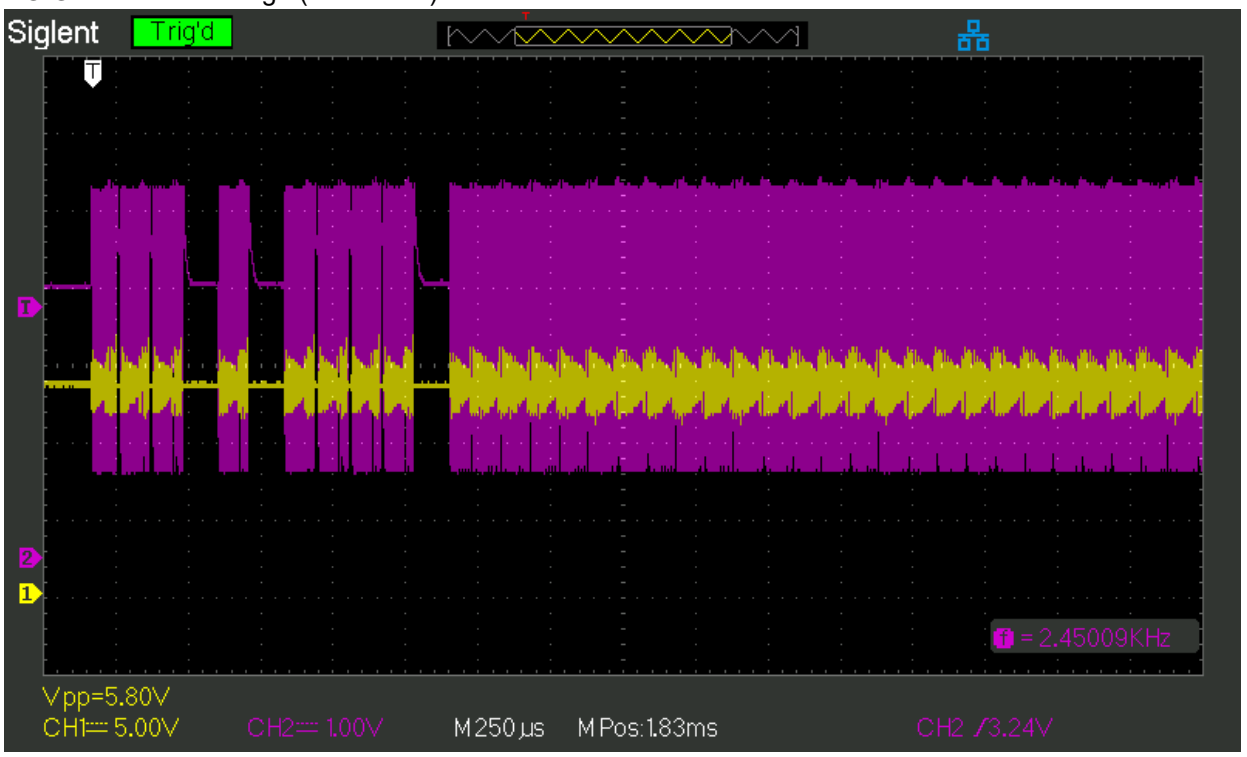
But not by much if anything because the delay is pretty constant at 160 cycles even at the end of the

body...

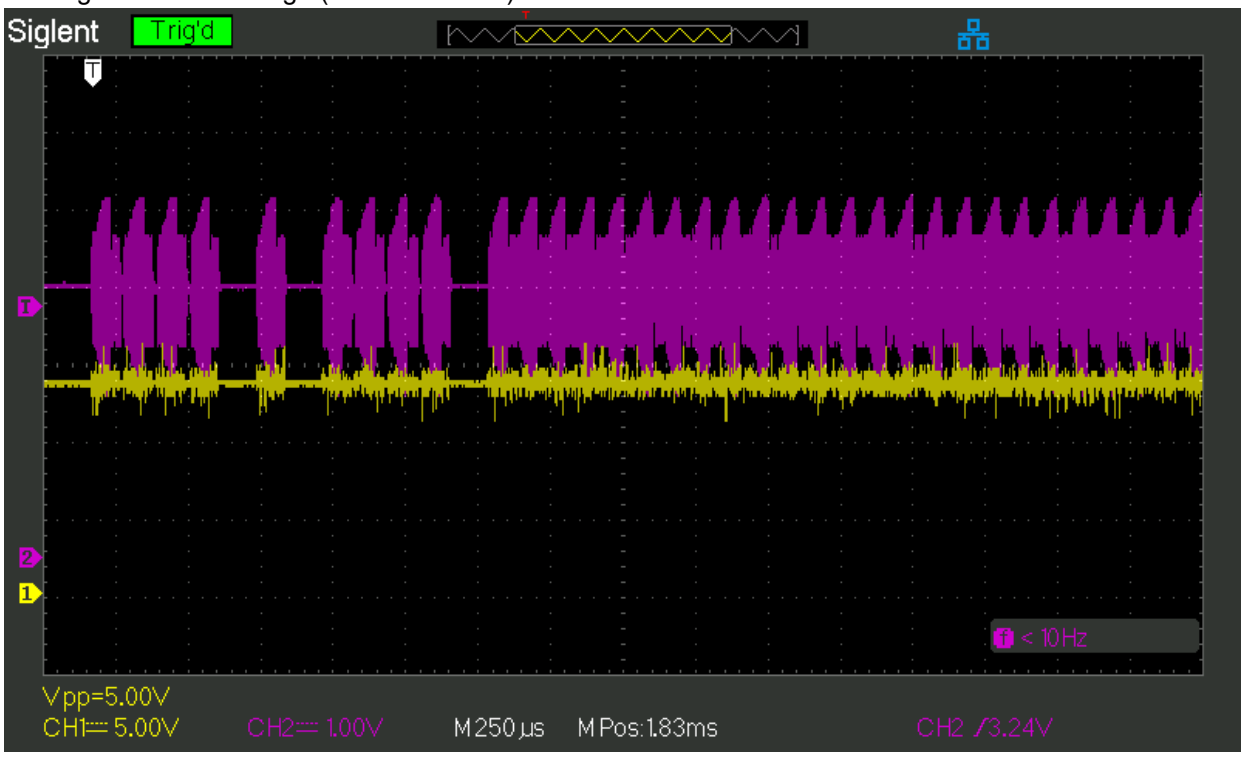
Testing it

Annnd! It works. Whew.

DG Generated message (receivable):



GPIO generated message (also receivable):



In [ ]:

```
1
```