EE 492 Biweekly Report 11

3/15/21 - 3/29/21

Group Number: SD May 21-43

Project Title: Emergency! Need backup!

Client/Advisor: Collins Aerospace / Andrew Bolstad

Team Members / Role:

James Curtis / Meeting Scribe

Caroline Easley / Meeting Facilitator

Marcelo Abrantes / Engineer (Power Systems)

Michael Kuehn / Communications Director

Benjamin Welte / Project Documentation

Abbey Wilder / Test Engineer

Stepan Zelenin / Engineer (Communication Systems)

Period Summary:

During the past work period, we developed plans for the final PCB design. This included selecting alternative parts for the mixer and local oscillator because we discovered issues with both of them during component testing. Several other team members also continued to research the best practices for making RF printed circuit boards and continued to work on the design for

the final PCB. Fortunately, the code that we developed for the local oscillator will still work on the new part because we can order a similar model from the same manufacturer. We were able to program the local oscillator over I2C to set its frequency to any value up to 200 MHz (the highest frequency we can measure with the oscilloscopes in the TLA). Furthermore, we were able to get the SNMP set command to work with integer variables.

Past Period Accomplishments:

- Reviewed parts for final PCB – James, Marcelo, Michael
- Established I2C communication between Arduino & LO – Ben, Stepan
- Programmed LO to output at any frequency – Ben
- Mixer component testing – Caroline, James, Michael
- Resolved floating point math issues on Arduino – Ben
- Got SNMP set command to work with integers – Abbey
- Started developing simulation model for final PCB – Stepan
- Preliminary S-parameter simulations - Stepan

Pending Issues:

- Resolve impedance matching issue in LO testing – everybody
- Get SNMP set command to work with floating point variables – Abbey
- Fully automate local oscillator parameter generation – Ben
- Test new mixer after it arrives – Caroline, Michael, Stepan, James,

Individual Contributions:

Name	Individual Contributions	Hours this week	Hours cumulative
James C.	• Reviewed Part Selections • Assisted oscillator testing	12	54
Caroline E.	• Part Review • Mixer Testing	13	56
Marcelo A.	• Part Review • PCB Design and Review	12	61
Michael K.	• Began final PCB design • Part research and review	12	54
Ben W.	• Created I2C test circuit • Established I2C communication with the local oscillator • Resolved floating point calculation issue on the Arduino	16	61

	• Programmed local oscillator to change frequencies • Started code to fully automate parameter selection for LO frequency changes • Investigated impedance matching problem in LO test setup		
Abbey W.	• Debugged SNMP set command • Got SNMP set command to work with integer variables • Working on getting the set command to work with floating point variables	15	57
Stepan Z.	• Started implementing simulation model for final PCB design	15	58

	• Ran preliminary S-parameter simulations • Researched impedance matching • Researched final PCB design • Researched balanced/unbalanced signal transformation		

Plans for the Upcoming Period:

For the upcoming work period, we plan to finish automating the oscillator so that we can program its output frequency using a function that only needs the desired frequency as its input. We are also planning to order a new mixer and test it to hopefully verify that it works as expected (unlike the original part we had ordered). We also need to get the SNMP set command to work with floating point values so that eventually we can integrate the SNMP code and local oscillator control scripts to set the oscillator's output frequency from a user-friendly GUI.

Advisor Meeting Summary:

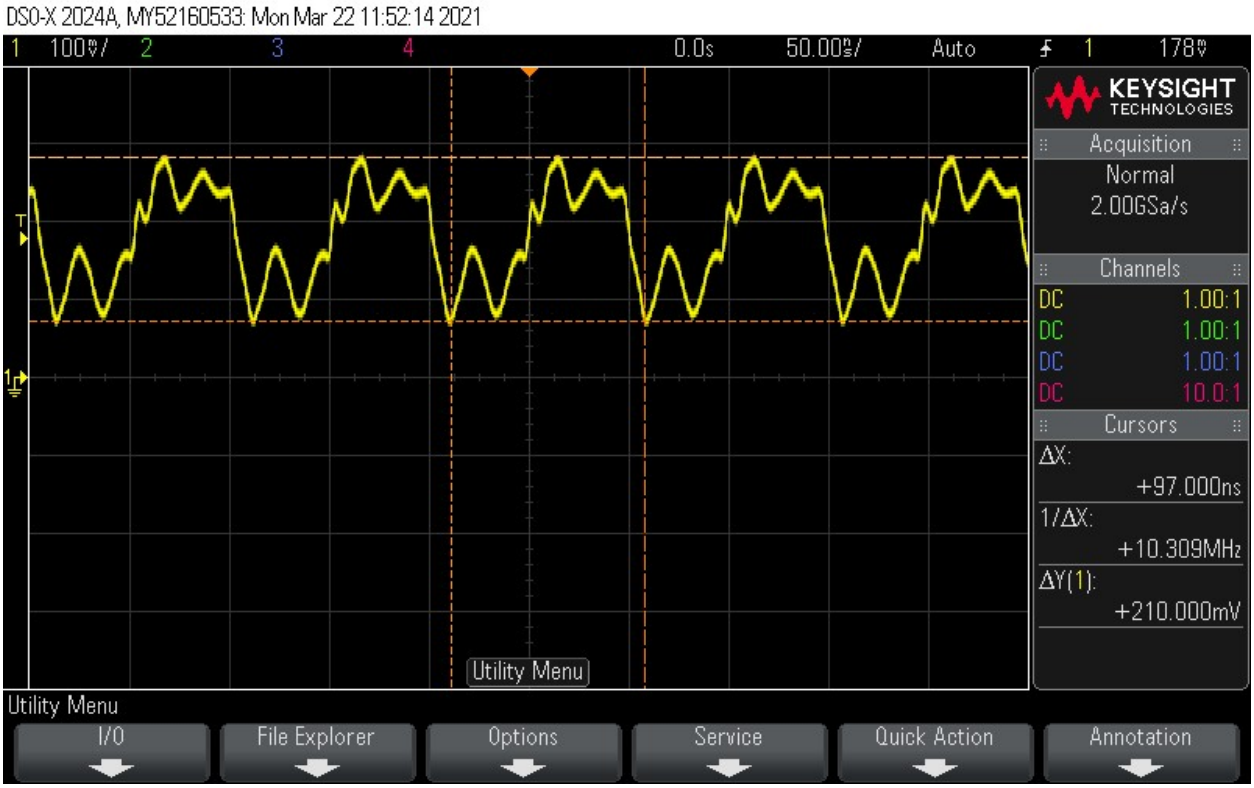
During our meetings with Dr. Bolstad, we discussed two issues we had while programming the local oscillator. Specifically, we discussed issues we were having observing the local oscillator's output on an oscilloscope. Our scope traces made it clear that we were successfully

programming the oscillator's frequency as we expected, but the shape and amplitude of the waveform both varied wildly from what we expected. After discussing with Dr. Bolstad, we concluded that there is probably a mismatch in our testing setup between the load impedance that the oscillator expects and the input impedance expected by the scope. We are planning to implement a buffer circuit in our test setup to hopefully rectify the waveform by providing the oscillator and the scope with the impedances that they expect. We also discussed issues we were having programming the oscillator that resulted from needing to multiply floating point values with very large numbers which was causing overflow on the Arduino's floating-point ALU, and Dr. Bolstad offered several possible solutions to the problem. We ultimatedly resolved the issue by shifting each decimal value in the floating point number, converting them to integers, multiplying by the required large values, then shifting each decimal place back to its original location.

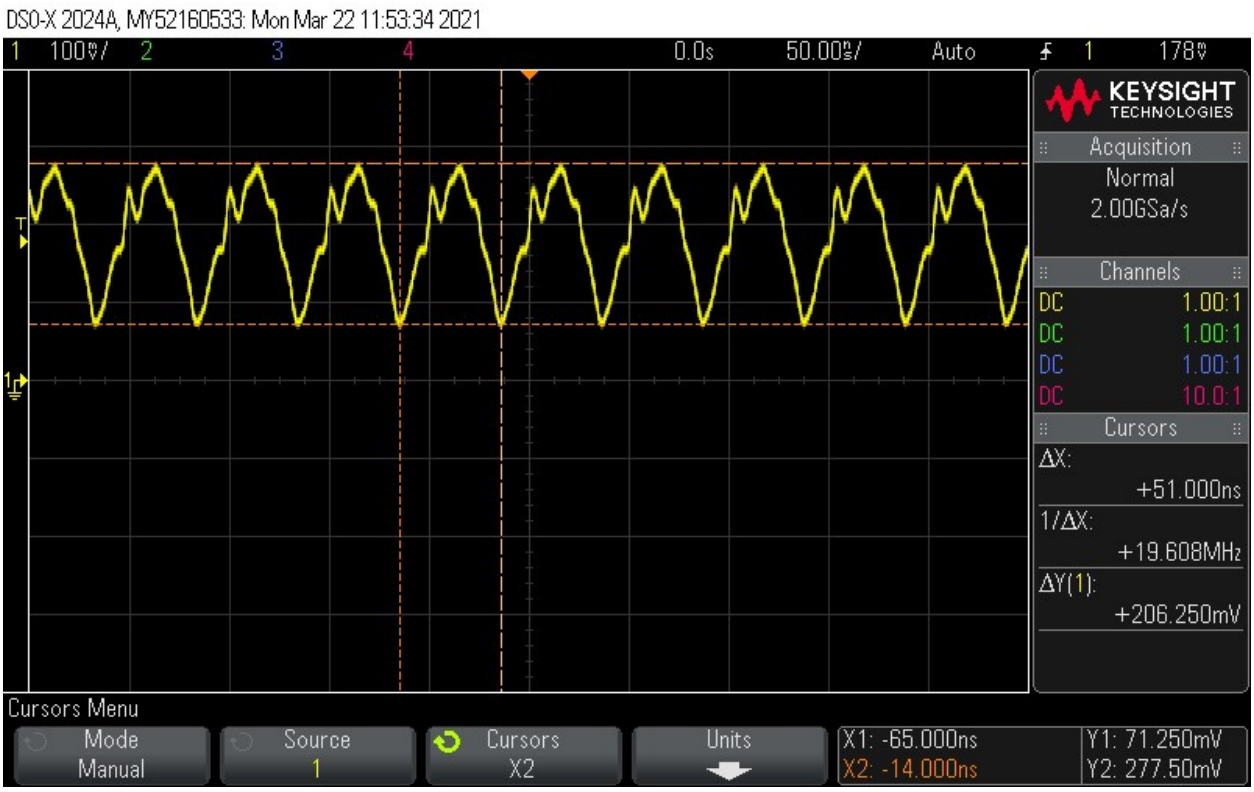
Appendix A: Images

<u>Scope traces of the local oscillator output at various frequencies:</u>

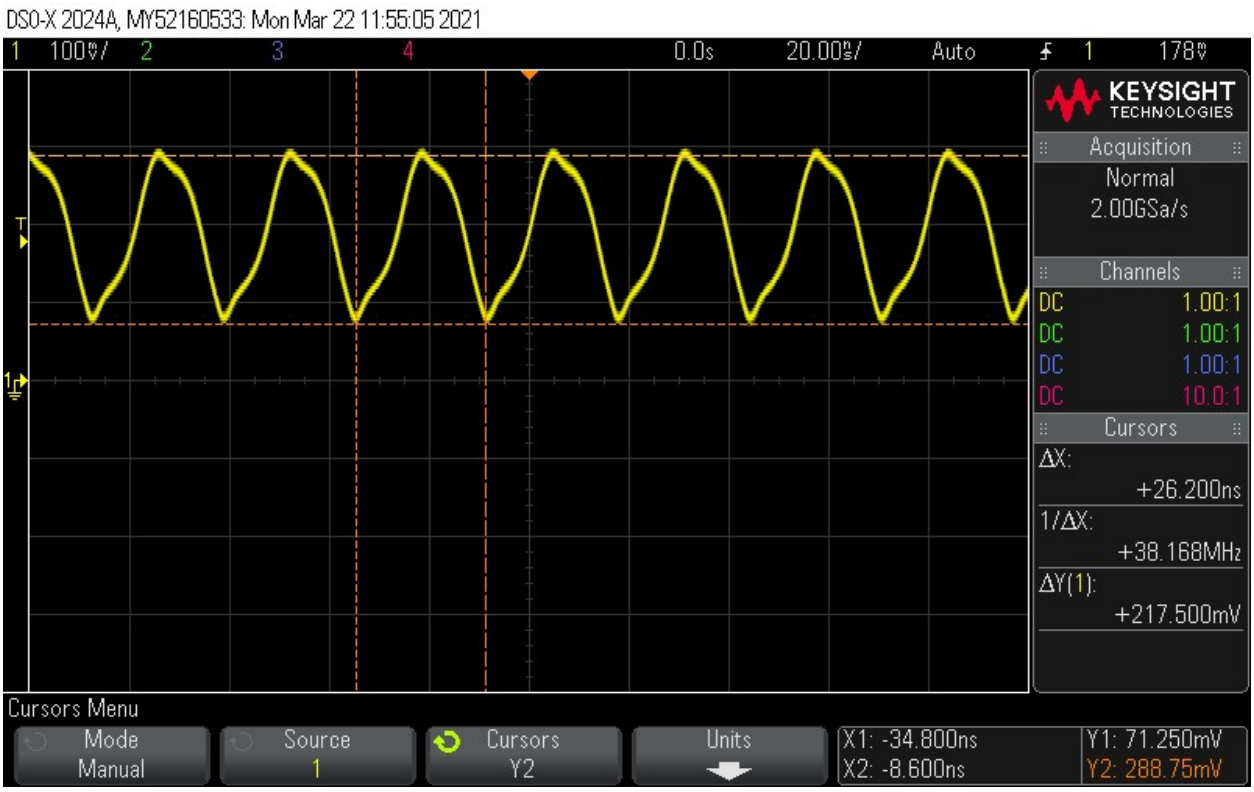
10 MHz



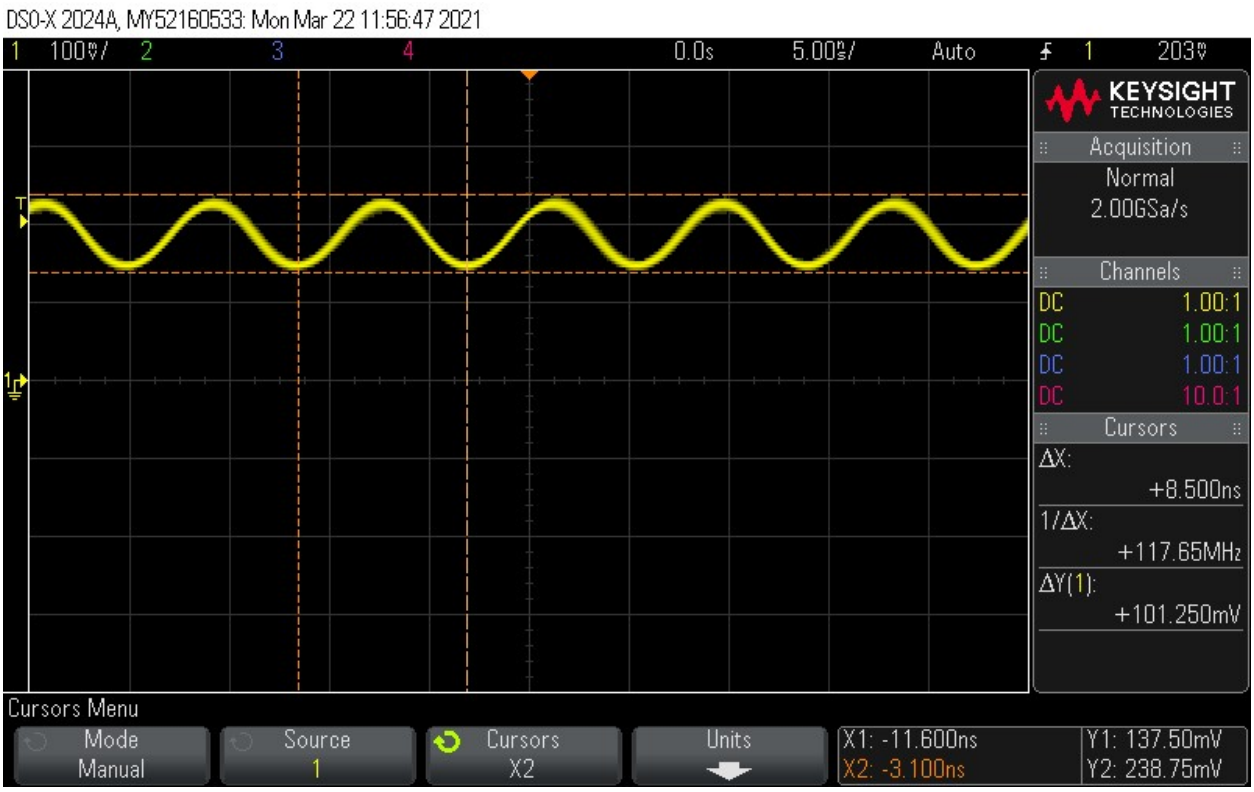
20 MHz:


DSO-X 2024A, MY52160533: Mon Mar 22 11:53:34 2021
1 100mV/ 2 3 4 0.0s 50.00s/ Auto 1 178mV
KEYSIGHT TECHNOLOGIES
Acquisition
Normal
2.00GSa/s
Channels
DC 1.00:1
DC 1.00:1
DC 1.00:1
DC 10.0:1
Cursors
ΔX:
+51.000ns
1/ΔX:
+19.608MHz
ΔY(1):
+206.250mV
Cursors Menu
Mode Manual
Source 1
Cursors X2
Units
X1: -65.000ns
X2: -14.000ns
Y1: 71.250mV
Y2: 277.50mV

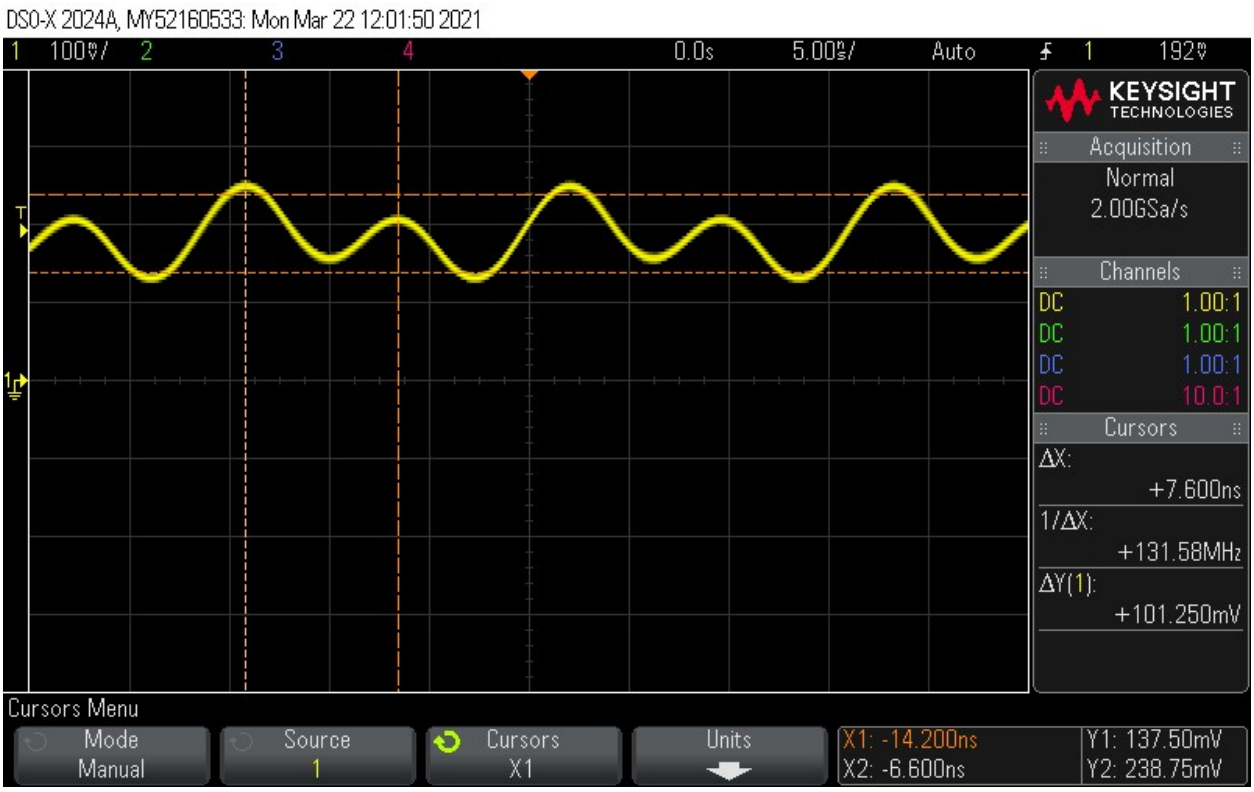
40 MHz:


DSO-X 2024A, MY52160533: Mon Mar 22 11:55:05 2021
1 100mV/ 2 3 4 0.0s 20.00ns/ Auto 1 178mV
KEYSIGHT TECHNOLOGIES
Acquisition
Normal
2.00GSa/s
Channels
DC 1.00:1
DC 1.00:1
DC 1.00:1
DC 10.0:1
Cursors
ΔX:
+26.200ns
1/ΔX:
+38.168MHz
ΔY(1):
+217.500mV
Cursors Menu
Mode Manual
Source 1
Cursors Y2
Units
X1: -34.800ns
X2: -8.600ns
Y1: 71.250mV
Y2: 288.75mV

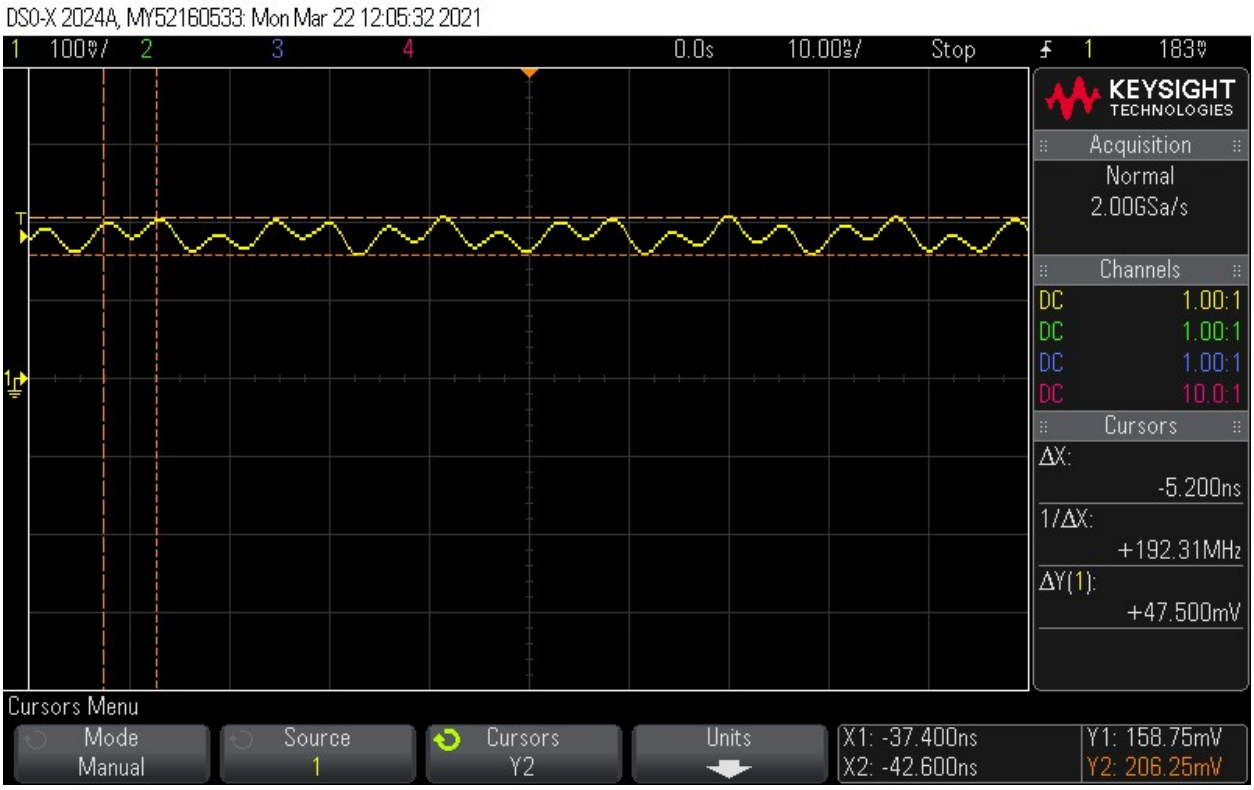
117 MHz:


DSO-X 2024A, MY52160533: Mon Mar 22 11:56:47 2021
1 100mV/ 2 3 4 0.0s 5.00ns/ Auto 1 203mV
KEYSIGHT TECHNOLOGIES
Acquisition
Normal
2.00GSa/s
Channels
DC 1.00:1
DC 1.00:1
DC 1.00:1
DC 10.0:1
Cursors
ΔX:
+8.500ns
1/ΔX:
+117.65MHz
ΔY(1):
+101.250mV
Cursors Menu
Mode Manual
Source 1
Cursors X2
Units
X1: -11.600ns
X2: -3.100ns
Y1: 137.50mV
Y2: 238.75mV

137 MHz:


DSO-X 2024A, MY52160533: Mon Mar 22 12:01:50 2021
1 100mV/ 2 3 4 0.0s 5.00ns/ Auto 1 192mV
KEYSIGHT TECHNOLOGIES
Acquisition
Normal
2.00GSa/s
Channels
DC 1.00:1
DC 1.00:1
DC 1.00:1
DC 10.0:1
Cursors
ΔX:
+7.600ns
1/ΔX:
+131.58MHz
ΔY(1):
+101.250mV
Cursors Menu
Mode Manual
Source 1
Cursors X1
Units
X1: -14.200ns
X2: -6.600ns
Y1: 137.50mV
Y2: 238.75mV

200 MHz:


DSO-X 2024A, MY52160533: Mon Mar 22 12:05:32 2021
1 100mV/ 2 3 4 0.0s 10.00ns/ Stop 1 183mV
KEYSIGHT TECHNOLOGIES
Acquisition
Normal
2.00GSa/s
Channels
DC 1.00:1
DC 1.00:1
DC 1.00:1
DC 10.0:1
Cursors
ΔX:
-5.200ns
1/ΔX:
+192.31MHz
ΔY(1):
+47.500mV
Cursors Menu
Mode Manual
Source 1
Cursors Y2
Units
X1: -37.400ns
X2: -42.600ns
Y1: 158.75mV
Y2: 206.25mV

Appendix B: Code

Preliminary code to program the local oscillator:

```
#include <Wire.h>

//solo LO dev board
/*
#define LO_I2C_ADDR 0x55
#define Fxtal 114.2  //MHz
*/

//Mixer board
#define LO_I2C_ADDR 0x55
#define Fxtal 116.8 //MHz



typedef unsigned long uint32_t;

uint32_t N1_Lookup(int N1_number);

uint32_t HS_DIV_Lookup(int HS_DIV_number);

uint32_t RFREQ_Upper_Lookup(double RFREQ_number);

uint32_t RFREQ_Lower_Lookup(double RFREQ_number);
```

```
void Write_LO_Values(int N1_reg_val, int HS_DIV_reg_val, uint32_t RFREQ_reg_upper_val,
uint32_t RFREQ_reg_lower_val);

void Reset_LO();

void Read_LO_Config();

void hardcode_test();

void setup() {

  Serial.begin(9600); // The baudrate of Serial monitor is set in 9600
  while (!Serial); // Waiting for Serial Monitor
  Wire.begin(); // Wire communication begin
}

void loop() {

  /*
   * TODO:
   *
   * Test lookup functions
   */

  //For 20 MHz:
  /*
  int N1 = 50;
  int HS_DIV = 5;
```

```
double Fnew = 20.0;
*/

//For 40 MHz:
/*
int N1 = 26;
int HS_DIV = 5;
double Fnew = 40.0;
*/

//For 60 MHz:
/*
int N1 = 18;
int HS_DIV = 5;
double Fnew = 60.0;
*/

//For 117 MHz:

int N1 = 6;
int HS_DIV = 7;
double Fnew = 117.0;


//For 137 MHz:
/*
int N1 = 8;
int HS_DIV = 5;
```

```
double Fnew = 137.0;
*/

//For 200 MHz:
/*
int N1 = 4;
int HS_DIV = 7;
double Fnew = 200.0;
*/

//eventually will need to check that 5.67 GHz > Fdco > 4.85 GHz
double Fdco = (Fnew * (double)N1 * (double)HS_DIV);

double RFREQ = Fdco / Fxtal;

uint32_t N1_reg = N1_Lookup(N1);
uint32_t HS_DIV_reg = HS_DIV_Lookup(HS_DIV);
uint32_t RFREQ_upper_reg = RFREQ_Upper_Lookup(RFREQ);
uint32_t RFREQ_lower_reg = RFREQ_Lower_Lookup(RFREQ);

//Debug:

Serial.print("\n\nDEBUG: \n");

//Expected N1_Reg: 3 (decrement 4 by 1)
Serial.print("\ncalculated N1_reg: ");
Serial.print(N1_reg, HEX);
```

```
  //Expected HS_DIV_reg: 0b101 = 9
  Serial.print("\ncalculated HS_DIV_reg: ");
  Serial.print(HS_DIV_reg, BIN);

  Serial.print("\ncalculated Fdco: ");
  Serial.print(Fdco);

  Serial.print("\ncalculated decimal value of RFREQ: ");
  Serial.print(RFREQ);

  Serial.print("\ncalculated RFREQ_upper: ");
  Serial.print(RFREQ_upper_reg, HEX);

  Serial.print("\ncalculated RFREQ_lower: ");
  Serial.print(RFREQ_lower_reg, HEX);

  Serial.print("\n\n");



  //hardcode_test();
  Write_LO_Values(N1_reg, HS_DIV_reg, RFREQ_upper_reg, RFREQ_lower_reg);
  //Reset_LO();

  Read_LO_Config();

  delay(5000);
}
```

```
/*
* hardcode_test
*  - Description:  test function to get LO behavior when reg values are hard-coded
*
*    NOTE: register values are specific to the local oscillator used for initial testing
*/
void hardcode_test(){

  //current config: 137 MHz
  uint32_t reg7_data = 0xA0;
  uint32_t reg8_data = 0xC2;
  uint32_t reg9_data = 0xB3;
  uint32_t reg10_data = 0x0A;
  uint32_t reg11_data = 0x3D;
  uint32_t reg12_data = 0x6C;

  //read reg 135
  Wire.beginTransmission(LO_I2C_ADDR);
  Wire.write(135);
  Wire.endTransmission();

  //request 1 byte from reg 137
  Wire.requestFrom(LO_I2C_ADDR, 1);

  //block while the wire isn't available
  while(!Wire.available()){}

  uint32_t reg_135_data = Wire.read();
```

```
//read reg 137
Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(137);
Wire.endTransmission();

//request 1 byte from reg 137
Wire.requestFrom(LO_I2C_ADDR, 1);

//block while the wire isn't available
while(!Wire.available()){}

uint32_t reg_137_data = Wire.read();

//write reg values:
Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(7);
Wire.write(reg7_data);
Wire.endTransmission();

Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(8);
Wire.write(reg8_data);
Wire.endTransmission();

Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(9);
Wire.write(reg9_data);
```

```
Wire.endTransmission();

Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(10);
Wire.write(reg10_data);
Wire.endTransmission();

Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(11);
Wire.write(reg11_data);
Wire.endTransmission();

Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(12);
Wire.write(reg12_data);
Wire.endTransmission();

//clear bit 4 in LO:
reg_137_data = reg_137_data & 0xEF;

Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(137);
Wire.write(reg_137_data);
Wire.endTransmission();

//set bit 6 of reg 135:
reg_135_data = reg_135_data | 0x20;
```

```
  Wire.beginTransmission(LO_I2C_ADDR);
  Wire.write(135);
  Wire.write(reg_135_data);
  Wire.endTransmission();

  //clear bit 6 of reg 135:
  /*
  reg_135_data = reg_135_data & 0b10111111;

  Wire.beginTransmission(LO_I2C_ADDR);
  Wire.write(135);
  Wire.write(reg_135_data);
  Wire.endTransmission();
  */

  return;

}

/*
* Write_LO_Values
*  - Description:  reset the local oscillator to 10 MHz
*
*    NOTE: register values are specific to the local oscillator used for initial testing
*/
void Reset_LO(){

  //solo LO
```

```
/*
uint32_t reg7_data = 0xAD;
uint32_t reg8_data = 0x42;
uint32_t reg9_data = 0xA8;
uint32_t reg10_data = 0xB4;
uint32_t reg11_data = 0x54;
uint32_t reg12_data = 0xF5;
*/

//mixer board
uint32_t reg7_data = 0xAD;
uint32_t reg8_data = 0x42;
uint32_t reg9_data = 0xA8;
uint32_t reg10_data = 0x44;
uint32_t reg11_data = 0x24;
uint32_t reg12_data = 0x16;

//read reg 135
Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(135);
Wire.endTransmission();

//request 1 byte from reg 137
Wire.requestFrom(LO_I2C_ADDR, 1);

//block while the wire isn't available
while(!Wire.available()){}
```

```
uint32_t reg_135_data = Wire.read();

//read reg 137
Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(137);
Wire.endTransmission();

//request 1 byte from reg 137
Wire.requestFrom(LO_I2C_ADDR, 1);

//block while the wire isn't available
while(!Wire.available()){}

uint32_t reg_137_data = Wire.read();

//write reg values:
Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(7);
Wire.write(reg7_data);
Wire.endTransmission();

Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(8);
Wire.write(reg8_data);
Wire.endTransmission();

Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(9);
```

```
Wire.write(reg9_data);
Wire.endTransmission();

Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(10);
Wire.write(reg10_data);
Wire.endTransmission();

Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(11);
Wire.write(reg11_data);
Wire.endTransmission();

Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(12);
Wire.write(reg12_data);
Wire.endTransmission();

//clear bit 4 in LO:
reg_137_data = reg_137_data & 0xEF;

Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(137);
Wire.write(reg_137_data);
Wire.endTransmission();

//set bit 6 of reg 135:
reg_135_data = reg_135_data | 0x20;
```

```
  Wire.beginTransmission(LO_I2C_ADDR);
  Wire.write(135);
  Wire.write(reg_135_data);
  Wire.endTransmission();

  //clear bit 6 of reg 135:
  /*
  reg_135_data = reg_135_data & 0b10111111;

  Wire.beginTransmission(LO_I2C_ADDR);
  Wire.write(135);
  Wire.write(reg_135_data);
  Wire.endTransmission();
  */

  return;

}

/*
* Read_LO_Config
*  - Description:  read LO configuration registers for debugging
*/
 void Read_LO_Config(){

  int i = 0;
```

```
  for(i = 7; i < 13; i++){

    //request reg data:
    Wire.beginTransmission(LO_I2C_ADDR);

    //specify register
    Wire.write(i);
    Wire.endTransmission();

    //ask to receive one byte from specified reg address
    Wire.requestFrom(LO_I2C_ADDR, 1);

    //block while the wire isn't available
    while(!Wire.available()){}

    int val = Wire.read();
    Serial.print ("Reg ");
    Serial.print(i);
    Serial.print(" value: ");
    Serial.print(val, HEX);
    Serial.print("\n");
    delay(1000);
  }
}

 /*
 * Write_LO_Values
```

```
 *  - Description:  Helper function to write the values of HS_DIV, N1, and RFREQ to the appropriate LO registers
 *
 *  - Inputs:      N1_reg_val - the value for N1 to be written to LO registers 7 [4:0] and 8 [7:6]
 *              HS_DIV_reg_val - the value for HS_DIV to be written to  LO register 7 [7:5]
 *              REFREQ_reg_val - the value for RFREQ to be writtent to LO registers 8 [4:0], 9, 10, 11, and 12
 *                          (this will need to be converted from a double to the actual reg value)
 */
 void Write_LO_Values(uint32_t N1_reg_val, uint32_t HS_DIV_reg_val, uint32_t RFREQ_reg_upper_val, uint32_t RFREQ_reg_lower_val){


   /*
   * Read reg 135
   */
  //specify reg 135
  Wire.beginTransmission(LO_I2C_ADDR);
  Wire.write(135);
  Wire.endTransmission();


  //request 1 byte from reg 137
  Wire.requestFrom(LO_I2C_ADDR, 1);


  //block while the wire isn't available
  while(!Wire.available()){}


  uint32_t reg_135_data = Wire.read();


  /*
```

```
 * Read reg 137
 */
//specify reg 137
Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(137);
Wire.endTransmission();

//request 1 byte from reg 137
Wire.requestFrom(LO_I2C_ADDR, 1);

//block while the wire isn't available
while(!Wire.available()){}

uint32_t reg_137_data = Wire.read();

/*
 * print reg values
 */
Serial.print("Before executing Reg_Write: \n");
Serial.print ("Reg 135 value: ");
Serial.print(reg_135_data, BIN);
Serial.print("\n");

Serial.print ("Reg 137 value: ");
Serial.print(reg_137_data, BIN);
Serial.print("\n\n");

//Try to set bit 4 of reg 137:
```

```
reg_137_data = reg_137_data | 0x10;

Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(137);
Wire.write(reg_137_data);
Wire.endTransmission();

/*
 * Write new HS_DIV, N1, and RFREQ values
 */

//shift HS_DIV to the left 5
HS_DIV_reg_val = HS_DIV_reg_val << 5;

//break N1 up into upper & lower parts
uint32_t N1_6_2 = N1_reg_val >> 2;
uint32_t N1_1_0 = N1_reg_val << 6;

//Fix RFREQ values:
RFREQ_reg_lower_val = RFREQ_reg_lower_val | ((RFREQ_reg_upper_val & 0xF) << 28);
RFREQ_reg_upper_val = RFREQ_reg_upper_val >> 4;

//make sure RFREQ_upper is only 6 bits
uint32_t RFREQ_37_32 = RFREQ_reg_upper_val & 0x3F;

//get the other parts of RFREQ:
uint32_t RFREQ_31_24 = (RFREQ_reg_lower_val >> 24) & 0xFF;
```

```
uint32_t RFREQ_23_16 = (RFREQ_reg_lower_val >> 16) & 0xFF;
uint32_t RFREQ_15_8 = (RFREQ_reg_lower_val >> 8) & 0xFF;
uint32_t RFREQ_7_0 = RFREQ_reg_lower_val & 0xFF;

//calculate reg 8 & 9 values
uint32_t reg_7_data = HS_DIV_reg_val | N1_6_2;
uint32_t reg_8_data = N1_1_0 | RFREQ_37_32;

//write reg values:
Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(7);
Wire.write(reg_7_data);
Wire.endTransmission();

Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(8);
Wire.write(reg_8_data);
Wire.endTransmission();

Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(9);
Wire.write(RFREQ_31_24);
Wire.endTransmission();

Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(10);
Wire.write(RFREQ_23_16);
Wire.endTransmission();
```

```
Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(11);
Wire.write(RFREQ_15_8);
Wire.endTransmission();

Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(12);
Wire.write(RFREQ_7_0);
Wire.endTransmission();

//clear bit 4 in LO:
reg_137_data = reg_137_data & 0xEF;

Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(137);
Wire.write(reg_137_data);
Wire.endTransmission();

//set bit 6 of reg 135:
reg_135_data = reg_135_data | 0x20;

Wire.beginTransmission(LO_I2C_ADDR);
Wire.write(135);
Wire.write(reg_135_data);
Wire.endTransmission();

//clear bit 6 of reg 135:
```

```
  /*
  reg_135_data = reg_135_data & 0b10111111;

  Wire.beginTransmission(LO_I2C_ADDR);
  Wire.write(135);
  Wire.write(reg_135_data);
  Wire.endTransmission();
  */

  return;

}

/*
 * N1_Lookup
 *  - Description:  Helper function to convert the regular number used in the frequency generation math
 *              into the value that actually needs to be written to reg 7 [4:0] and reg 8 [7:6]
 *
 *              Legal values are 1 and multiples of two.  Illegal odd values are rouned up.
 *              The value written to the register should be the desired divider minus one.  Ex: if you
 *              wanted N1 = 10, you would write 0b000_1001 (9 in decimal).
 *
 *  - Inputs:      N1_number - the regular number used in the frequency calculations
 *  - Outputs:      N1_reg_val - the value to be written to the LO registers corresponding to the value used in the math
 */
uint32_t N1_Lookup(int N1_number){
```

```
  //decrement the input value to get the appropriate reg value
  uint32_t N1_reg_val = N1_number - 1;

  //if the user is trying to write an odd value, round it up
  if(N1_number % 2 == 1)
    N1_reg_val++;

  return N1_reg_val;
}

/*
 * HS_DIV_Lookup
 *  - Description:  Helper function to convert the regular number used in the frequency generation math
 *              into the value that actually needs to be written to reg 7 [7:5]
 *
 *  - Inputs:      HS_DIV_number - the regular number used in the frequency calculations
 *  - Outputs:     HS_DIV_reg_val - the value to be written to the LO registers corresponding to the value used in the math
 */
 uint32_t HS_DIV_Lookup(int HS_DIV_number){

  uint32_t HS_DIV_reg_val;

  //return values based on the table in the datasheet
  if(HS_DIV_number == 4){
    return 0b000;
  }
  else if(HS_DIV_number == 5){
```

```
    return 0b001;
  }
  else if(HS_DIV_number == 6){
    return 0b010;
  }
  else if(HS_DIV_number == 7){
    return 0b011;
  }
  else if(HS_DIV_number == 9){
    return 0b101;
  }
  else if(HS_DIV_number == 11){
    return 0b111;
  }

  //return 0 if an invalid divider is requested
  else{
    return 0;
  }
}

/*
* RFREQ_Lower_Lookup
*  - Description:  Helper function to convert the decimal portion of the floating point RFREQ value into the
*              lower half of the value that will be written to the LO registers
*
*  - Inputs:      RFREQ_number - the floating point number used in the frequency calculations
```

```
 *  - Outputs:     RFREQ_reg_val[] - array of values to be written to LO registers for RFREQ
 *
 *  NOTES:
 *    As of 3/19/21, there is error introduced into the calculation (the lowest 3 hex characters of RFREQ are inaccurate)
 *    However, after recalculating the frequency using the new RFREQ value w/error introduced, it doesn't seem like it will
 *    have an appreciable effect on the result.  Hopefully this is good enough.
 *
 */
 uint32_t RFREQ_Lower_Lookup(double RFREQ_number){

   uint32_t dec_array[9];
   uint32_t sum = 0;
   double intermed;

   //get fractional part of RFREQ_number
   while(RFREQ_number > 1.0){
    RFREQ_number = RFREQ_number - 1;
   }

   //TODO: multiply by 2^28 without overflow
   //     2^28 = 268,435,456

   for(int i = 0; i < 9; i++){

    //shift i decimal places to the left & convert to int
    if(i == 0){
     intermed = RFREQ_number * 10;
```

```
}
else{
  intermed = intermed * 10;
}

dec_array[i] = (long)intermed;

//remove upper digits
while(dec_array[i] >= 10){
  if(dec_array[i] > 1000000){
    dec_array[i] -= (long)1000000;
  }
  else if(dec_array[i] > 10000){
    dec_array[i] -= (long)10000;
  }
  else{
    dec_array[i] -= (long)10;
  }
}

//Serial.print("\n\tisolated digit: ");
//Serial.print(dec_array[i]);

//multiply digit by 2*28
dec_array[i] = dec_array[i] * 268435456;

//Serial.print("\n\tdigit times 2^28: ");
//Serial.print(dec_array[i]);
```

```
    uint32_t divisor = 10;

    for(int j = 0; j < i; j++){
      divisor = divisor * 10;
    }

    //Serial.print("\n\tdivisor: ");
    //Serial.print(divisor);

    //unshift digit
    dec_array[i] = dec_array[i] / divisor;

    //add digit to total
    sum += dec_array[i];
  }

  return sum;
}

 /*
* RFREQ_Upper_Lookup
*  - Description:  Helper function to convert the integer portion of the floating point RFREQ value into the
*              upper half of the value that will be written to the LO registers
*
*  - Inputs:      RFREQ_number - the floating point number used in the frequency calculations
*  - Outputs:     RFREQ_reg_val[] - array of values to be written to LO registers for RFREQ
```

```
*/
uint32_t RFREQ_Upper_Lookup(double RFREQ_number){

  uint32_t digits[2];

  //remove fractional part of RFREQ w/cast:
  uint32_t RFREQ_upper_reg_val = (uint32_t)(RFREQ_number);

  //store upper digit in array
  digits[1] = RFREQ_upper_reg_val / 10;

  //store lower digit in array
  while(RFREQ_upper_reg_val >= 10){
   RFREQ_upper_reg_val -= 10;
  }
  digits[0] = RFREQ_upper_reg_val;

  //multiply each digit by 2^28, then shift right to 0th & 1st index respectively
  for(int i = 0; i < 2; i++){
   digits[i] = digits[i] * 268435456;

   digits[i] = digits[i] >> (28);

   if(i){
    digits[i] = digits[i] * 10;
   }
  }
```

```
  //recombine & return digits
  RFREQ_upper_reg_val = digits[0] + digits[1];

  return RFREQ_upper_reg_val;
}
```