Internship Report

on

Capturing data from TI radars without mmWaveStudio

Supervised by:

Professor Murat Torlak

PhD Student Josiah Smith

Submitted by:

Chase Austin Wood

chase.wood@utdallas.edu

B.S. Electrical Engineering

Summer-Fall 2020

Contents

Figures

Abstract

This report serves the purpose to record the details of work completed throughout the first portion of my internship with Professor Torlak and Josiah W. Smith. This will cover all work done throughout the completion of a modified version of Josiah's application designed to control mmWaveStudio using LUA commands. The completed application supports capturing data at the same rate as mmWaveStudio but does not require an instance of mmWaveStudio to be running. This is done using the various CLI utilities provided by TI.

Background

Data options for AWR1443

I first started out this internship by looking at basic options of data output from TI radars. I very quickly discovered the options of CAN, SPI. Further research led me to look into CLI (SPI) as it was the immediate possibility. How it is used in the demo binary, and I decided that it is a good place to start exploring options. I agree now (a few months later) that it was a good place to start learning about basic radar function as well as SPI communication for TI radars. My initial thoughts were to create a python script to read from the radar (in real time) and store the data in a binary file, similar to how mmWaveStudio stores data. Soon into doing so, just the combination of python and awr1443 in the same google search allowed for me to find a GitHub directory of code (click here to see python code) that has already been written to do exactly this, in python, for the awr1443 boost radar. Everything was running fine, so long as the correct SDK firmware is flashed to the radar using TI UniFlash.

Creating equivalent of GitHub in MATLAB

I'll first mention that the purpose of this code is to emulate the functionality that was included in the GitHub library mentioned previously, except exclusively in MATLAB. At the time, there were no intentions to incorporate this into Josiah's GUI application, but MATLAB was preferred more than Python. The process to create MATLAB scripts did not take long to have the configuration side of things recreated. The script opens user-defined COM ports and reads a configuration file to send to the radar at a baud rate of 115200. The tool parses the user's configuration file and sends each line to the user/uart com port. At this point, when discussing the best way to create the data capture side in MATLAB, it was brought up that a high data rate is critical. I found out very soon after that the number of samples/second of data is limited to 30 samples/second when using the program. I tried to mess with some values in the python code to allow a faster speed, but soon discovered that the code would not work with these new values. At the time I did not understand why, but now I know this is due to the MSS (master-sub-system) being written in a specific way for the demo-visualizer binary, which is the firmware that is used for CLI operation of the radar in both python and MATLAB.

Faster data transfer methods

The radar itself supports various types of data output depending on the radar version itself as well as SDK firmware version. For example, the xWR1243 supports CSI2 high speed output where the xWR1642 supports LVDS. When these types of outputs are mentioned, they are limited by the BSS (radar-sub-system) on the SDK versions. The BSS is a low-level binary that is fully developed by TI as opposed to the MSS which can be programmed by the end user. The process to find out all options is to look at the radar datasheet. In the AWR1443ES2 case, the BSS prevents usage of the LVDS lanes when the demo binary firmware is flashed because the demo binary utilizes the AWR1443 hardware accelerator. The demo only requires a flash of the MSS to the radar, but it actually has the BSS and MSS combined to one, for an easier out of box experience. This is problematic because end users are unable to edit the BSS firmware. The conclusion from this is there is no possible way to enable LVDS streaming from the AWR1443 while using the default demo binary.

Controlling DCA1000EVM

If it was possible to enable LVDS streaming, then we would need to be enable the DCA1000EVM to start recording data, allowing us to view it at UDP ports 4096/4098 in the format presented in the DCA1000EVM user guide document. Controlling the DCA1000EVM was not a difficult task and only required a few lines of commands to be run in Powershell or command prompt. The commands to view all available actions with the DCA1000EVM over CLI are below. Be sure to modify the script for your specific mmWaveStudio installation path.

```
cd "C:\ti\mmwave_studio_xx_xx_xx_xx\mmWaveStudio\PostProc\"
.\DCA1000EVM_CLI_Control.exe -h cf.json
```

These commands show the following, which lists all available CLI commands that the DCA1000EVM can receive.

```
Cli_Control [options]
Options:
fpga                        Configure FPGA
eeprom                      Update EEPROM
reset_fpga                  Reset FPGA
reset_ar_device             Reset AR Device
start_record                Start Record
stop_record                 Stop Record
record                      Configure Record delay
dll_version                 Read DLL version
cli_version                 Read CLI_Control tool version
fpga_version                Read FPGA version
query_status                Read status of record process
query_sys_status            DCA1000EVM System aliveness
-h                          List of commands supported
-q                          Quiet mode - No status display in the console
```

Figure 1 - DCA1000EVM CLI commands

The way to run one of these commands is to replace the -h in the command above with the appropriate command. For example, if I was already in the correct directory and wanted to start capturing data, the following command will do so:

.\DCA1000EVM_CLI_Control.exe start_record cf.json

To run the code in MATLAB, use the *system()* command or simply *$powershell "your command"* to run the script. Alternatively, I feel the better option is to only run a single Powershell command inside of MATLAB and call a powershell (.ps1) script that includes all desired operations. In the appendix, an example .ps1 script is available.

Look at mmWave SDKs

Throughout all of the mmWave SDK's, it is very difficult to find a complete example that is not for creating an SDK workspace. This led me to look for information on YouTube as well as TI's Project Explorer. Both resources have limited number of examples that explain how the SDK operates at a high-level, but that's where it stops. The best option I saw was to study the MSS ***.cpp*** and ***.h*** files located in the SDK installation folders. The file is roughly 2400 lines of code and would take a long time to grasp the high-level operation and then move forward to writing a custom firmware.

Not ready for this challenge just yet, I decided to study LVDS. I wanted to know what exactly it was, how it works in general and for these radar specific applications, as well as see how viable of an option it is to study the code for long enough to just enable this single function. This led me to begin thinking of truly how hard it could be to enable this feature for the radar while working on the mmWaveStudio firmware rather than the demo-visualizer firmware since all radars supported by the DCA1000 capture device must use LVDS to transfer data. The mmWaveStudio firmware allows LVDS operation for this board but the demo firmware does not. The issue with this is I could not revert the binary file from the mmWaveStudio installation into its C++ code, which halts all progress on that front. Even if it were possible, this would be a required step for all future versions of TI radar to be used in this format. It was at this point that we had the idea to run the backend of mmWaveStudio without the GUI.

mmWave Studio listening port 2777

The idea to run the mmWaveStudio background scripts came into my head after realizing how long it would take to write custom firmware in the SDK. My thought process was to run all of the scripts that mmWaveStudio calls for until the point of generating the listening port on port 2777. The reason for this is once the listening port is generated and functioning properly, the program should accept external LUA commands as shown in the TI mmWaveStudio user guide documents. I took a deep look into the program files and found the initialization processes of mmWaveStudio instances. These functions are located inside the path:

C:\ti\mmwave_studio_xx_xx_xx_xx\mmWaveStudio\RunTime

The files are formatted as *application extensions* (.DLL) which is a compiled C# code. To access and see these functions, you first have to decompile the .DLL files using a decompile program, I used and recommend *dotPeek*, as it worked very well and allows you to search through every .DLL file that is loaded to pick out C# functions, keywords, etc. After finding the functions to call, there were a few issues that needed to be addressed:

1. Permissions needed to call functions
2. 32-bit or 64-bit .DLL files
3. Security keys

First are the permissions needed to call functions. On my computer, the permissions are set to default which means scripts are unable to be called by a user. After looking up how to run a command in bypass mode, I was able to set the permissions of my user account to bypass mode, run the command, and then set the permissions back to default for security reasons. Next was to determine whether these were 32-bit or 64-bit .DLL files because you cannot use functions for 64-bit DLL using a 32-bit shell and vice-versa. The conclusion is that these were a combination of 32-bit and 64-bit .DLL files which means that the C# code has been compiled using both versions of a C# compiler (32/64). Last was determining if these .DLL files are signed or not. At first, I thought that all of the files were not signed but this was impossible as they wouldn't be used. After looking in my registry to find the keys for these .DLL files, I found enough that I can conclude they are all signed, which is necessary to run. I assume the signing process is something that happens during the initial mmWaveStudio installation processes. The issue with this approach is that, I was unable to find/access some of the functions that would definitely be required. I found that the functions worked whenever a mmWaveStudio application was running and concluded that the application deletes these functions whenever it is closed for security reasons (listening port left accessible).

Overview

Hardware trigger setup on AWR1443 radar

Option 1:

- **Remove R120 from the DCA1000EVM**
- **Populate R165 with 0Ω on AWR1443**

Option 2:

- **Remove R62 from the AWR1443**
- **Populate R165 with 0Ω on AWR1443**

By comparing the two options, you see that they both share populating R165 with 0Ω resistor. The decision to make is whether to modify the DCA1000EVM or the TI radar to break the pin 16 signal path.

For example, using the same DCA1000EVM to allow HW trigger for an AWR1443, AWR1642, and AWR1843, you must populate the missing resistor on all of the boards regardless of *Option 1* or *Option 2*. The benefit of *Option 1* is that you **ONLY** have to remove R120 once. *Option 2* requires you to remove R62 (for AWR1443) or equivalent on **EACH** radar.

On the radar, R62 (or equivalent) will be along the 60-pin connector. You will notice that many other resistors are in the area and depending on your soldering ability, it may be difficult to remove this resistor without causing issues to others. Alternatively, R120 (on DCA1000EVM) is in the open without additional resistors close by. Even though I am confident in my soldering and hot air rework skills, it makes no sense to take this pointless risk, especially on property that is not my own. Because of this, I chose Option 1 and removed R120 on the DCA1000EVM.

Additional Details:

- **R120 on DCA1000EVM**. This resistor is in the signal path on pin-16 (SYNC_IN) from the DCA1000EVM's FPGA which pulls the 3.3V HW trigger input down to 500mV. As a result, the device times out after 30 seconds and the error "NO_LVDS_DATA" will be present. (*R120 is located on the BACK of the DCA1000EVM as shown in* Figure 2).

- **R165 on AWR1443.** By default, radars do not have this resistor. Bridge the connection using a wire or 0Ω resistor. (*This resistor is located on the BACK of the AWR1443 radar as shown in* Figure 3).

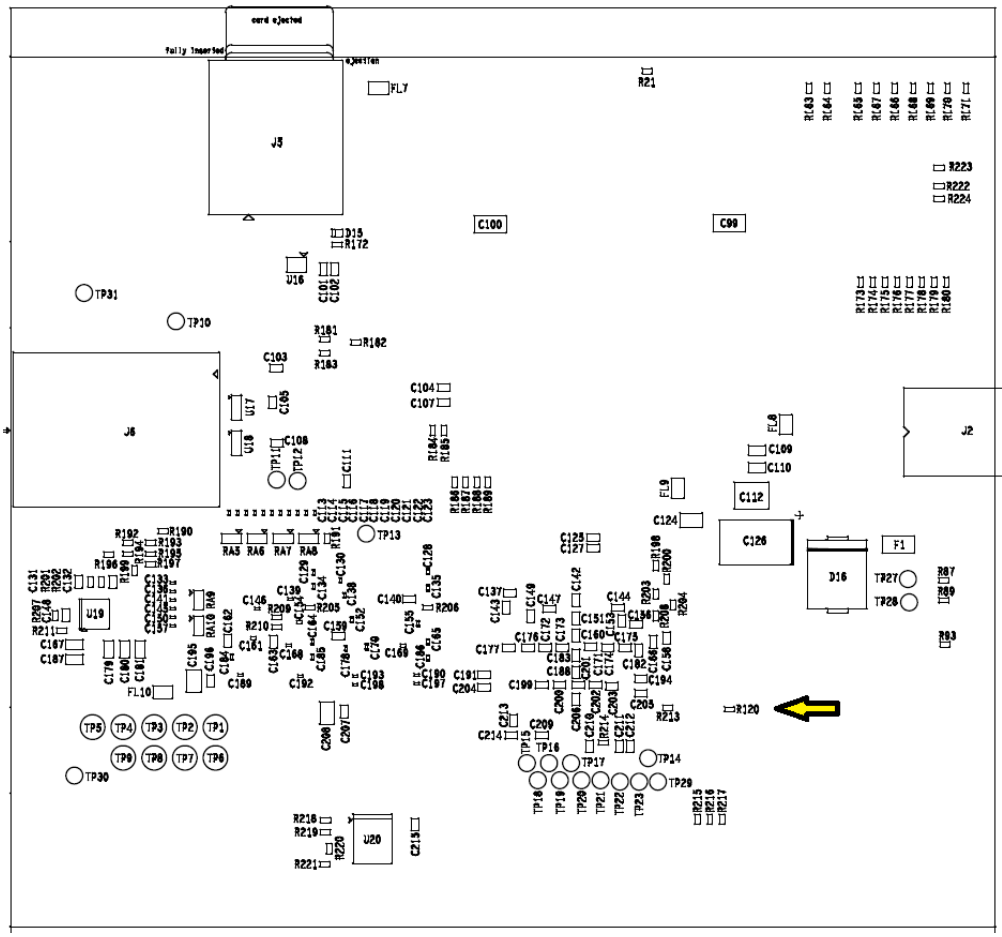
- **R62 on AWR1443**. This resistor is in the signal path on pin-16 (SYNC_IN) from the DCA1000EVM's FPGA which pulls the 3.3V HW trigger input down to 500mV. As a result, the device times out after 30 seconds and the error "NO_LVDS_DATA" will be present. (*R62 is located on the FRONT of the AWR1443 as shown in* Figure 4).

- **Locating resistors on any other radar: Use board specific schematics**

    - To find the first resistor (equivalent to R62), search on the ***HD connector*** for the resistor on pin 16. This resistor is populated with 0Ω by default (*shown in Figure 5*).

    - To find the second resistor (equivalent to R165), search on the ***BP/LP connector*** for the series resistor from the AR_SYNC_IN net. This resistor is unpopulated by default (*shown in* Figure 6).

Figure 2 - Back of DCA1000EVM (R120)



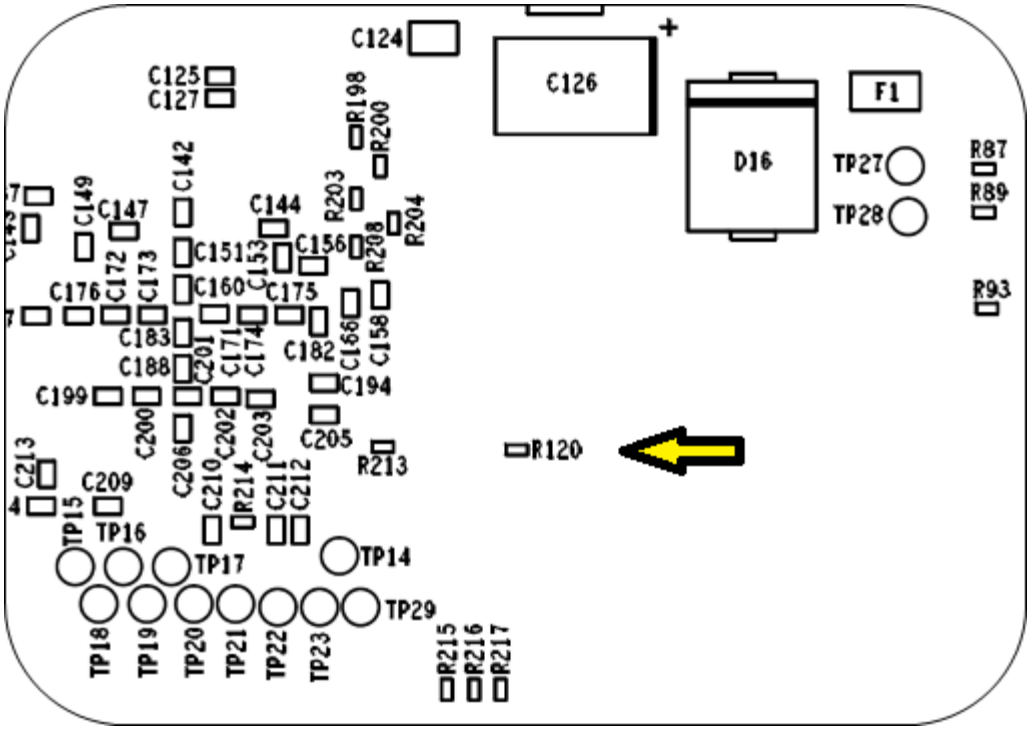


Figure 3 - Back of AWR1443 (R165)

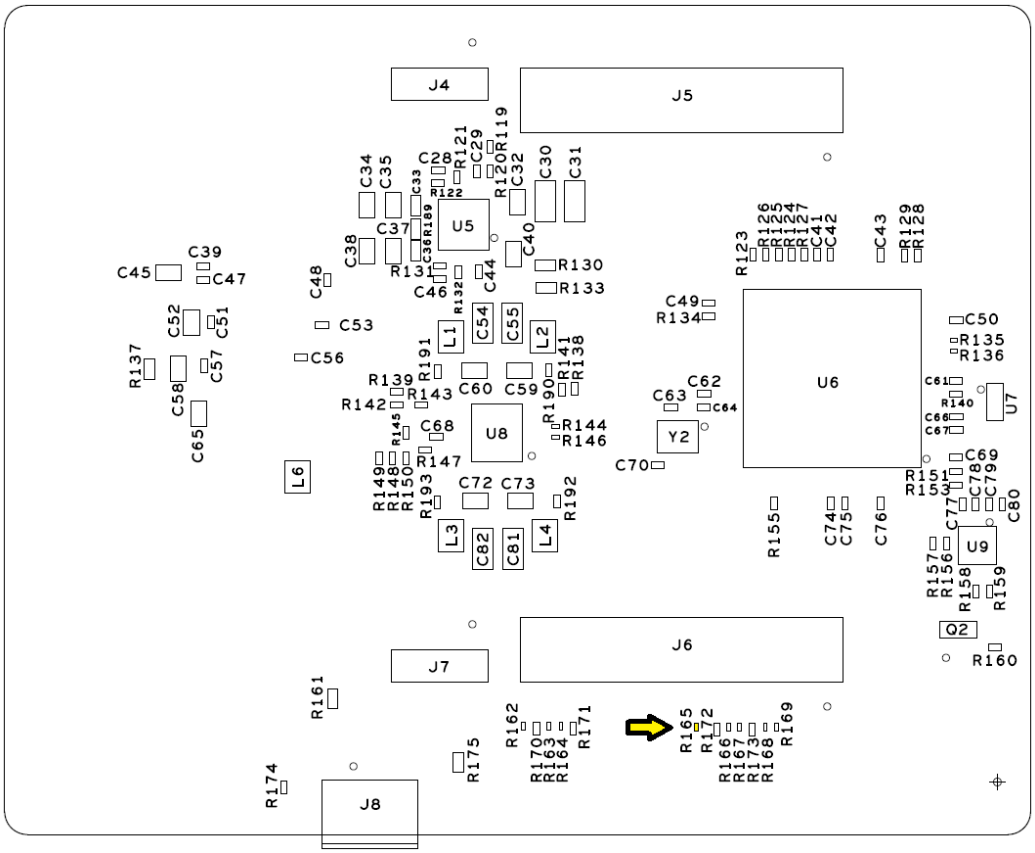


Figure 4 - Front of AWR1443 (R62)

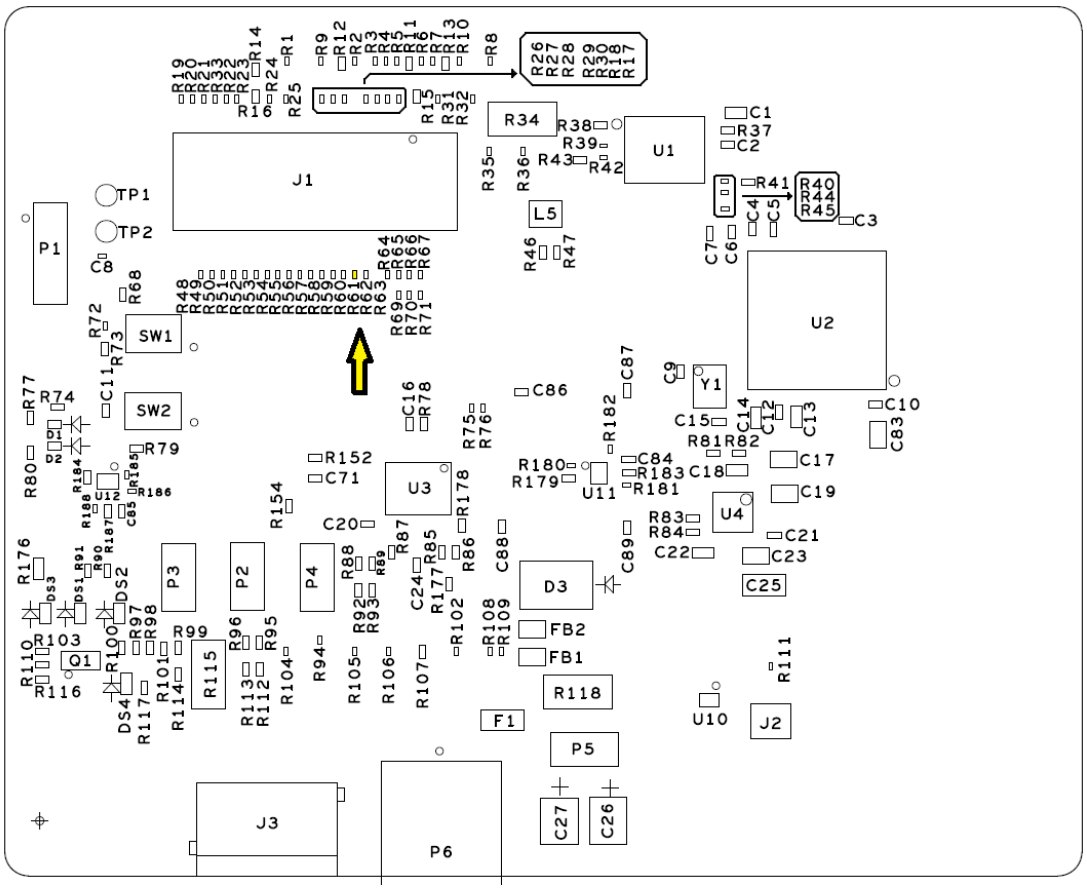


Figure 5 - 60-pin connector (from AWR1443 schematic)

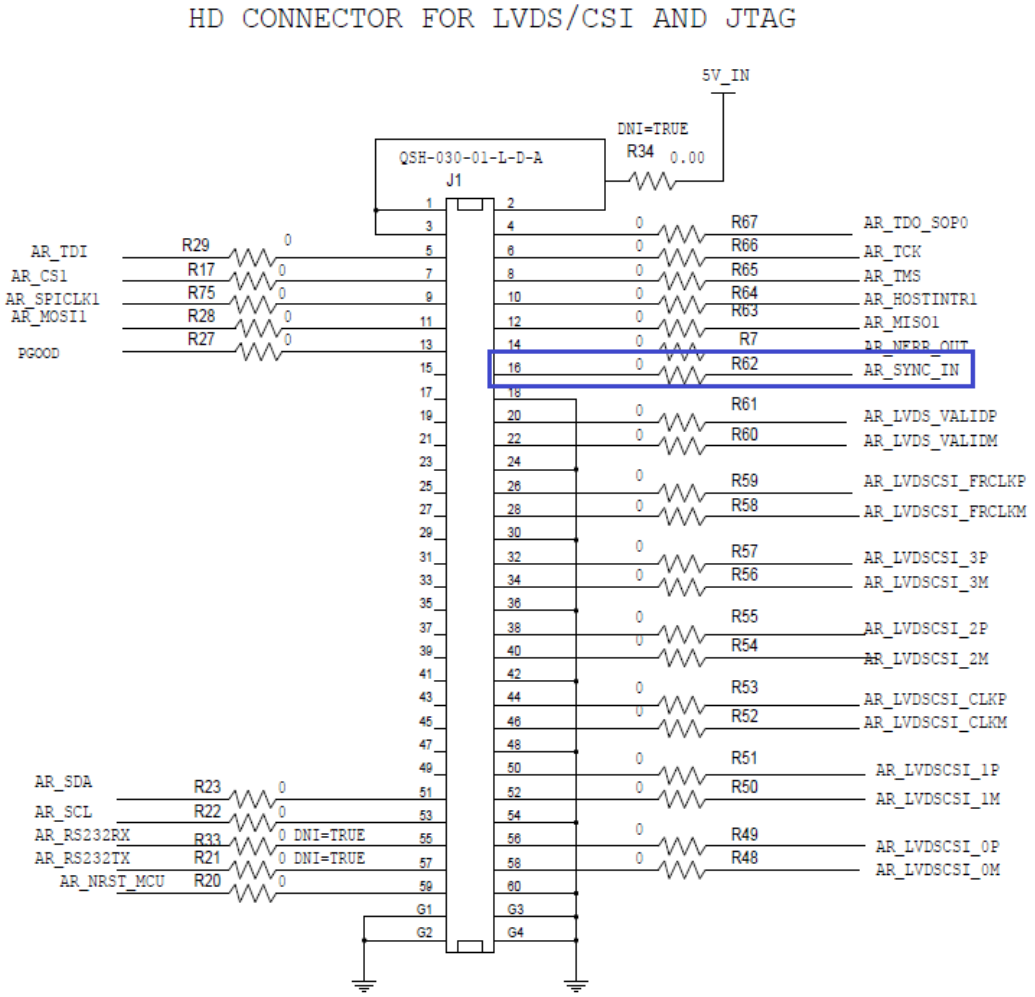
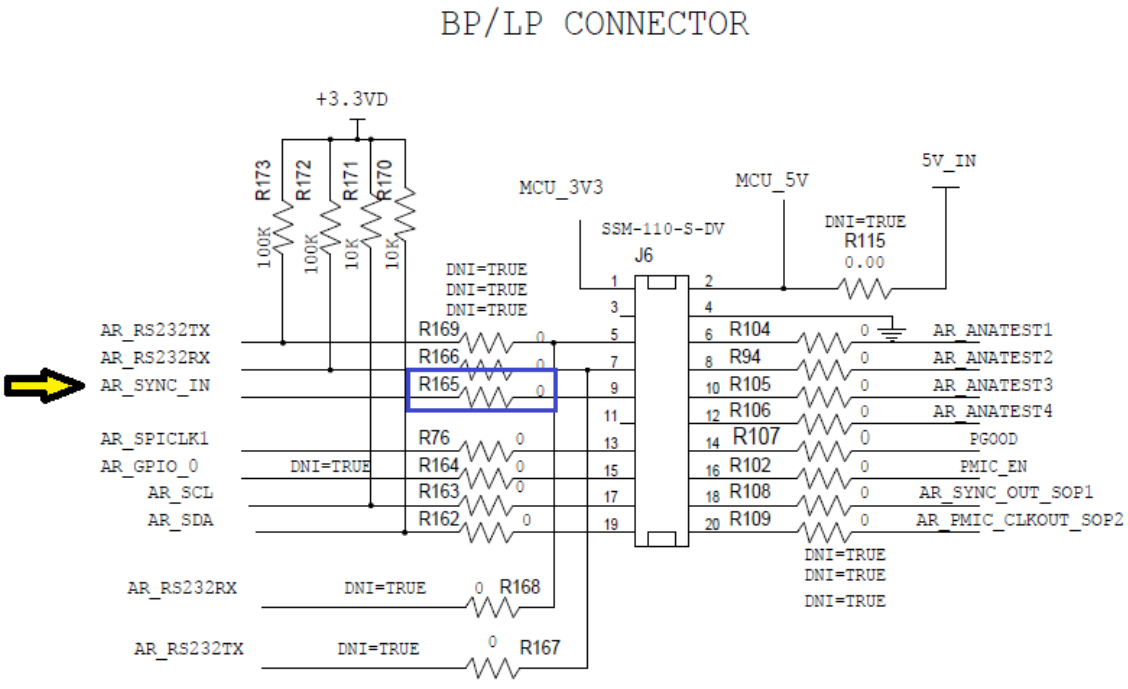


Figure 6 - Missing resistor ID (from AWR1443 schematic)



Software implementation of HW trigger pulse:

First, verify:

- Ground is be connected between the radar and MCU.
- Operating voltage of MCU is 3.3v (Use voltage divider for 5v MCUs)
- The HW trigger pulse must be applied to pin 9 of the J6 connector on TI radar.
- Period must be greater than frame time, otherwise radar will begin capture of the next frame without completing previous frame. This will cause an error and corrupt data_bin file(s).

The MCU communicates with the MATLAB application over a serial connection. Code is sent to the MCU by writing "PulseWidth,Period" to the serial port at the appropriate baud rate per MCU, where *PulseWidth* and *Period* represent length of time in milliseconds. The ESP8266 utilizes a 115200 baud rate. Since MCU's do not handle exceptions, if pulse width is larger than period, the MCU will catch this error and enter the lower value as the pulse width to avoid a 100% output signal.

As for the MATLAB application, the setup is the same as Josiah's implementation: Connect to mmWaveStudio, Initialize, Configure Radar. When entering hardware trigger mode, a LUA command is sent to mmWaveStudio to enable the hardware trigger mode of operation. Next, connect to the microcontroller using the features on the right-hand side under "MCU Configuration".

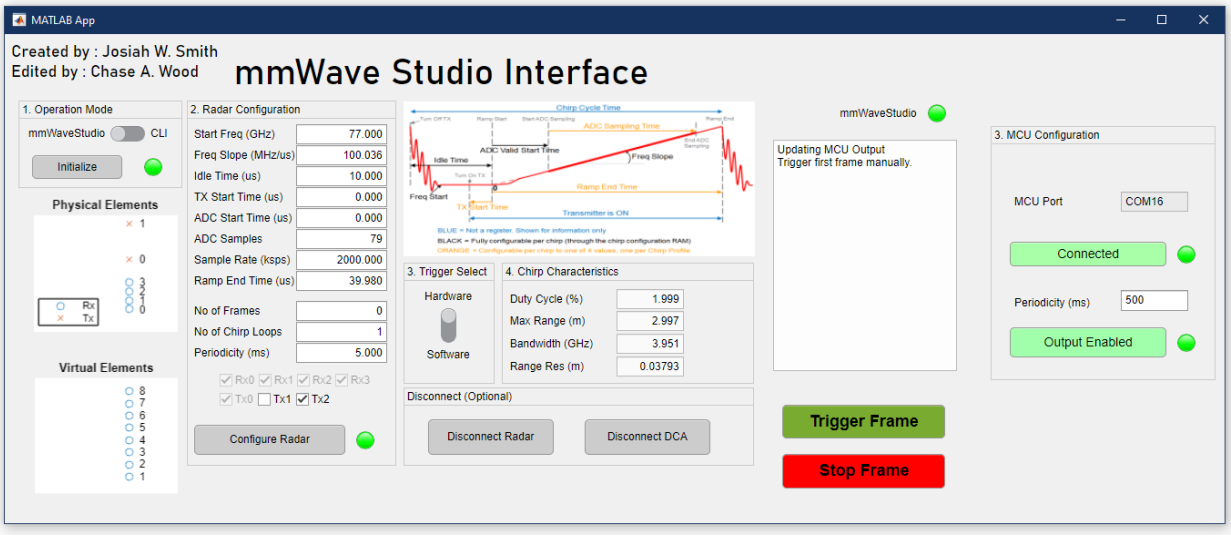


Figure 7 - Hardware Trigger Panel

Once connected, the button and lamp will change to a green color as shown above. Once periodicity is selected, pressing the "Start MCU Output" button will send the configuration to the microcontroller. The MCU button and lamp will change to a green color as shown above. If settings are adjusted after the output has been enabled, simply changing the parameters and pressing "Update MCU Signal" will send the new configuration to the microcontroller.

To capture data, pressing the “Trigger Frame” button once will begin the measurement and the microcontroller will trigger all remaining frames. The console will notify user once all frames have been captured.

Note: You may switch between software and hardware trigger modes of operation at any time so long as mmWaveStudio is connected to MATLAB and the radar is initialized.

See full ESP8266 MCU Code Here

ESP8266 example for single characteristic pulse with 20% duty cycle:

```
const int period = 1000;          //ms
const int pulseWidth = 200;       //ms

#define signalOut 5               //Define D1 (D1 is equal to pin GPIO5, or just “5”) as “signalOut”

void setup() {
  pinMode(signalOut, OUTPUT);
  delay(50);
}

void loop() {
  digitalWrite(signalOut, HIGH);
  delay(pulseWidth);
  digitalWrite(signalOut, LOW);
  delay(period-pulseWidth);
}
```

LVDS data capture using AWR1642 over CLI

This section will provide detailed instructions on how to capture data over LVDS using mmWave SDK (v3.4) and DCA1000EVM.

Know that LVDS streaming functionality over CLI is only provided by the following radars:

- **xWR16xx**
- **xWR18xx**
- **xWR68xx**

1. Positioning switches SW1 and SW2 on DCA1000EVM
   a. SW1

SW1.1	12bit_OFF
SW1.2	14bit_OFF
SW1.3	16bit_ON

   b. SW2

SW2.1	LVDS_CAPTURE	(0)
SW2.2	ETH_STREAM	(1)
SW2.3	AR1642_MODE	(1)
SW2.4	RAW_MODE	(0)
SW2.5	SW_CONFIG	(1)

2. Flashing radar using UniFlash:
   a. Use jumpers to short SOP0 and SOP2 pins. This SOP mode is only used when flashing firmware to the device with UniFlash.
   b. Power cycle the radar by holding SW2 for 1 second.
   c. Allow UniFlash to automatically detect the device. Ensure that the detected device is correct, and the icon is red. (The black icon is for chip only flashing)
   d. Select the correct COM Port (User/UART) under the “Settings & Utilities” Tab.
   e. On the “Program” Tab, for **Meta Image 1**, choose “Browse” to locate the demo binary. This binary will be located in the SDK installation path. For example:

      C:\ti\mmwave_sdk_03_04_00_03\packages\ti\demo\xwr16xx\mmw\xwr16xx_mmw_demo.bin

   f. Press the “Load Image” button. Once process is complete, remove ONLY the SOP2 jumper (leave SOP0 shorted)

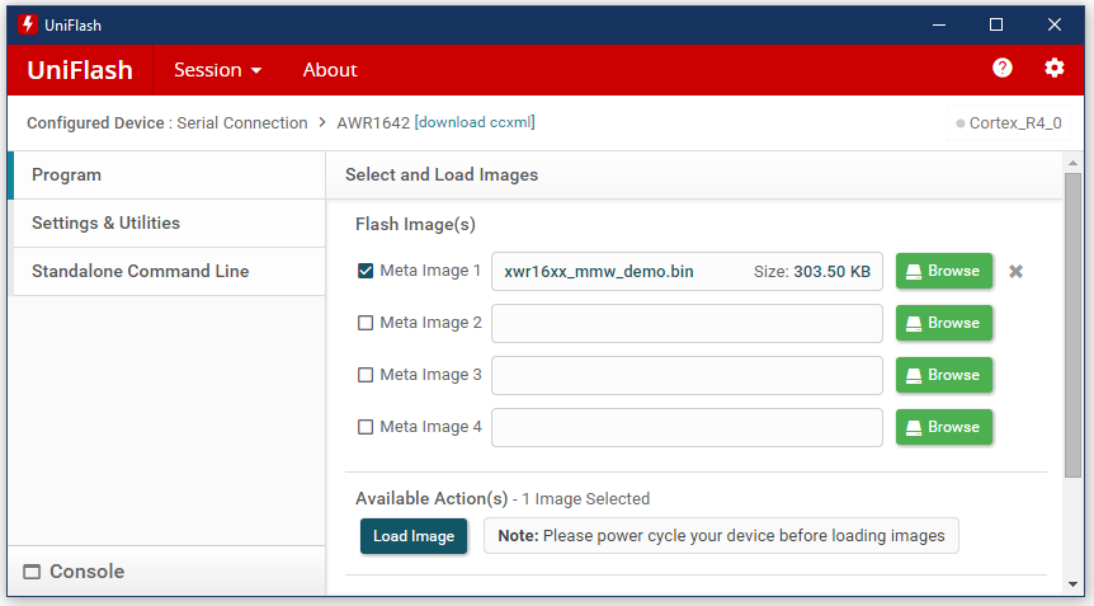


Figure 8 - UniFlash

3. Next, run the TI mmWave Demo Visualizer available online to confirm operation of radar. Once radar is confirmed working, select desired "scene selection" parameters and download the configuration file by pressing "**Save Config to PC**". I suggest renaming the configuration file to something distinct such as "awr1642_profile.cfg".

4. Modify the **lvdsStreamCfg** command in the "awr1642_profile.cfg" file according to settings shown in the Figure 9 - lvdsStreamCfg Table below. If **lvdsStreamCfg** requires modifying other commands, do so using the correct SDK user guide. (Ex: See how enabling "CP_ADC_CQ data" requires modifying command "analogMonitor")

Modify the **chirpThreshold in adcBufCfg to 1**. Otherwise, an exception will occur on line 1814 of *mss_main.c* stating that the number of chirps is not as expected. Ex: Change **adcbufCfg -1 0 1 1 0** to **adcbufCfg -1 0 1 1 1**

5. Create a copy of the DCA1000EVM's configuration file for a CLI unique version. This file is located on your mmWaveStudio installation path and by default has name **cf.json**

   C:\ti\mmwave_studio_02_01_01_00\mmWaveStudio\PostProc\cf.json

   Name the copy something distinct such as **awr1642_cf.json**

6. Modify **awr1642_cf.json**:
   a. Modify **dataLoggingMode:**
      i. If <enableHeader> is disabled (0) in lvdsStreamCfg, change "dataLoggingMode" to **"raw"**
      ii. If <enableHeader is enabled (1) in lvdsStreamCfg, change "dataLoggingMode" to **"multi"**

<enableHeader> is the **2nd** digit of **lvdsStreamCfg** (be sure to include the quotations)

b. Change **lvdsMode** to **2** (all supported devices only have 2 LVDS lanes)
c. Change **dataFormatMode** to **3** (all supported devices only allow 16-bit capture)
d. All ethernet settings should remain the same to allow for easy switching of the DCA1000EVM between radars using CLI and mmWaveStudio.
e. Modify **captureConfig:**
    i. **fileBasePath** changes where the saved data file is stored.
    ii. **filePrefix** changes the name of the saved data file.
    iii. **captureStopMode** changes the duration of the measurement. **"infinite"** will measure until there is no data present on the LVDS lanes or until the radar is told to stop. **"bytes", "frames", and "duration"** are used in combination with the next 3 lines (in the .json file) to tell the DCA1000EVM when to stop measuring data.
f. Modify **dataFormatConfig:**
    i. **MSBToggle** should be set to **0**.
    ii. **reorderEnable** should be set to **1**.
    iii. **dataPortConfig** should have all dataTypes set to **"complex"**. This is due to mmWave demo binary using all CBUFF/LVDS sessions as complex.

7. Running command to capture data with DCA1000EVM's CLI utility:
    a. Change directory to the modified (awr_1642.json) file. If in the same directory as the original cf.json file, the command to change directory will look like this:

```
cd "C:\ti\mmwave_studio_02_01_01_00\mmWaveStudio\PostProc\"
```

    b. Now use the 'DCA1000EVM_CLI_Control.exe' utility to start the capture by running the 'fpga', 'record', and 'record_start' commands, in this respective order. Doing so will look like the following command lines:

```
.\DCA1000EVM_CLI_Control.exe fpga AWR1642_cf.json
.\DCA1000EVM_CLI_Control.exe record AWR1642_cf.json
.\DCA1000EVM_CLI_Control.exe start_record AWR1642_cf.json
```

    c. To see a list of all the available commands in the DCA1000 CLI Utility, use the -h command:

```
.\DCA1000EVM_CLI_Control.exe fpga AWR1642_cf.json
```

    d. Once the start_record command is sent, the radar must begin data transmission within 30 seconds, otherwise the DCA1000EVM will run the stop_record command. Any data sent by the radar after the stop_record command will not be captured until the start_record command is run again.
    e. To properly end data capture, TI states that it is better to run the stop_record command before issuing the sensorStop command to the radar. I have found that if there is around 3 seconds of no incoming LVDS data during a capture, the DCA1000EVM will automatically issue the stop_record command. This means that anytime you issue sensorStop to the radar, then you are guaranteed that within 3 seconds the data capture will automatically end on the DCA1000EVM as well.

Figure 9 - lvdsStreamCfg Table

lvdsStreamCfg	Enables the streaming of various data streams over LVDS lanes. **When this feature is enabled, make sure chirpThreshold in adcbufCfg is set to 1.** The values in this command can be changed between sensorStop and sensorStart. This is a mandatory command.	<subFrameIdx> subframe Index	For legacy mode, this field should be set to -1 For advanced frame mode, it should be set to either the intended subframe number or -1 to apply same config to all subframes.
		<enableHeader> Enable/disable HSI header for all active data streams	0 - Disable 1 - Enable
		<dataFmt> Controls HW streaming. Specifies the HW streaming data format.	0-HW STREAMING DISABLED 1-ADC 4-CP_ADC_CQ When choosing CP_ADC_CQ, please ensure that CQRxSatMonitor and CQSigImgMonitor commands are provided with appropriate values and these monitors are enabled using analogMonitor command.
		<enableSW> Enable/disable user data (SW session)	0 - Disable 1 - Enable <enableHeader> should be set to 1 when this field is enabled.

Example: Change **lvdsStreamCfg -1 0 0 0** to **lvdsStreamCfg -1 0 1 0**

Integration into Josiah's GUI

Changes:

- Removed connect to mmWaveStudio button
    - The function the previous button called has been integrated into the "Initialize" button pressed function call.
- Added "Operation Mode" switch
    - **Changing from mmWaveStudio to CLI:**
        - Radar configuration settings are changed to a preset designed for optimal range resolution.
        - the "mmWaveStudio connection" status lamp functions as a status lamp for data capture. When red, capture inactive, when green, data capture active.
    - **Changing from CLI to mmWaveStudio:**
        - When changed to mmWaveStudio mode, the radar configuration settings are changed to a preset that contains Josiah's default settings.
        - The status lamp for mmWaveStudio connection is in the same position as Josiah's app had previously.
- Added "Trigger select" switch
    - When pressed, the application expands and shows the "MCU Configuration" panel for connecting to microcontroller for triggering frames via hardware trigger.
    - The only available parameter for change is periodicity. MATLAB calculates the necessary pulse width for selected chirp and frame configuration parameters.
    - Only works for mmWaveStudio operation at the moment, should not be difficult to implement for CLI.
- Buttons now change color to represent the next button to be pressed (similar to how mmWaveStudio directs user to the next button)

When in CLI operation, sometimes the radar does not begin capturing despite the DCA1000 beginning a data capture. This is due to the chirp configuration being out of range of the radar's available memory limit. In the instance of the demo binary, the limit is the size of the radar cube. For the awr1642, the L3 memory (radar cube memory) is 2MB. The chirp configuration should work so long as the following is true:

$$(\boldsymbol{numADCSample})\left(\boldsymbol{num_{chirploop}}\right)(\boldsymbol{numTXAnt}) \leq \mathbf{8192}$$

*Note: An integrated alert system ensures that the above condition is followed before sending a configuration to the user's radar.

Setup required for running GUI

The only necessary step to run the latest GUI is modify the parameters listed in the mmWaveStudio_GUI.mlapp. To do this, open the (.mlapp) file in MATLAB app designer and switch to the "Code View" tab. Modify the parameters jsonFileName and RSTD_DLL_Path:

1. jsonFileName is a modified version of the default **cf.json** file and is used to enable LVDS capture over CLI. By default, the new configuration file is named "CLI_LVDS.json".
    a. How to generate:
        i. Run the MATLAB GUI app.
        ii. Switch to CLI mode.
        iii. Press the button **[configure JSON]** that appeared in the top right corner of the GUI.
        iv. Once this button is pressed, you do not have to press it again.
    b. How to modify:
        i. Open the functions folder of the GUI application directory.
        ii. Open modifyJSON.m in a text editor (or MATLAB).
        iii. Comment line 14 and uncomment line 15 so it looks like this:

```
Line 14 - %jsonText = fileread(append(path,'cf.json'));
Line 15 - jsonText = fileread(append(path,FileName));
```

        iv. Uncomment and change any of the settings between line 19 – line 68.
        v. Save the modifyJSON.m file.
        vi. On the GUI app, press the **[configure JSON]** button to save the configuration.
        vii. (Optional) Comment all modified lines between line 19 – line 68 back out to avoid any possible typing mistake causing future problems.

2. RSTD_DLL_Path is located along the mmWaveStudio installation path as shown below:

    C:\ti\mmwave_studio_02_01_01_00\mmWaveStudio\Clients\RtttNetClientController\RtttNetClientAPI.dll

    a. The path is used to locate the necessary RSTD file (.dll) to communicate with an instance of mmWaveStudio for LUA commands.

    b. Additionally, this is used to point functions such as modifyJSON.m to the correct directory of a mmWaveStudio installation such as:

        C:\ti\mmwave_studio_02_01_01_00\mmWaveStudio\PostProc\

***Note: In the initial configuration file, ensure that profileCfg is on line 7, frameCfg is on line 10. Also, it is important to leave duplicate lines untouched. In my experience, removing the duplicate lines has only resulted in the radar rejecting the configuration.**

Questions

Q. Is it possible to bypass mmWaveStudio altogether?

A. The only methods to do this will be to either open the listening port for LUA externally or work with radars supporting CLI enabling of LVDS bus to extract data at high speed over DCA1000EVM's UDP ethernet ports.

Q. Can you bypass the DCA1000EVM and still have high-speed data capture?

A. This is not possible. The radar only uses the LVDS lanes through the 60-pin Samtec cable to make this data available for the DCA1000EVM. It is not possible for the radar to send data at this speed over the SPI connection, which is limited to 30 samples/second when using the demo-visualizer firmware that is provided in mmWave SDK installations.

Q. How does mmWaveStudio communicate to the DCA1000EVM?

A. Whenever mmWaveStudio performs a function related to the DCA1000EVM, certain LUA commands are executed which run Powershell scripts to perform the desired function. These commands can be replicated using a shell. See section "Controlling DCA1000EVM" for more information.

Q. How does the LVDS system work between the TI radar and DCA1000EVM?

A. The 60-pin Samtec cable is used to interface the radar to the DCA1000EVM. Looking at the DCA1000EVM user guide document, the following pins are the LVDS bus:

- Pin 20 – LVDS valid signal positive (LVDS_VALIDP)
- Pin 22 – LVDS valid signal negative (LVDS_VALIDM)
- Pin 26 – LVDS frame clock signal positive (LVDS_FRCLKP)
- Pin 28 – LVDS frame clock signal negative (LVDS_FRCLKN)
- Pin 32 – LVDS data pair 3 positive (LVDS_3P)
- Pin 34 – LVDS data pair 3 negative (LVDS_3M)
- Pin 38 – LVDS data pair 2 positive (LVDS_2P)
- Pin 40 – LVDS data pair 2 negative (LVDS_2M)
- Pin 44 – LVDS clock pair positive (LVDS_CLKP)
- Pin 46 – LVDS clock pair negative (LVDS_CLKM)
- Pin 50 – LVDS data pair 1 positive (LVDS_1P)
- Pin 52 – LVDS data pair 1 negative (LVDS_1M)
- Pin 56 – LVDS data pair 0 positive (LVDS_0P)
- Pin 58 – LVDS data pair 0 negative (LVDS_0M)

These wires are used to gather the data from the radar. The data is then processed by the DCA1000EVM according to the DIP-switches configuration, and then sent to the ethernet controller implemented on the DCA1000EVM for preparation to the PC.

Q. Does configuration when using CLI commands require mmWaveStudio to be running?

A. mmWaveStudio does not have to be running to control the radar using CLI commands.

Q. Are the CLI commands implemented in Python or MATLAB?

A. They can be implemented in any program that allows serial communication. Both Python and MATLAB allow for serial communication. To control the radar, the CLI commands listed in the appropriate SDK user guide document must be sent to the User/UART Application serial port. Whenever sending commands to the radar, the radar itself must not be capturing data, so the first commands sent should always be *"sensorStop"* followed by *"flushCfg"*. If automated, the commands can be sent too fast and the radar will not receive them properly so using a delay $(\sim 1ms)$ between commands is advised.

Q. What hardware interface is being used for CLI tool?

A. CLI must use the SPI of the radar. There is no other method for delivering CLI commands to the radar.

Q. What are the data speeds when using SPI for data transfer?

A. SPI data transfer is limited to ≤ 30 samples/second over the data port connected at 921600 baud. This means that each sample in the demo-visualizer is close to 30.7kB. (921600/30)

Q. What options are available for the data hardware interface?

A.

CAN	(1 Mbps)
SPI	(< 1 Mbps)
LVDS	(> 155.5 Mbps)

Q. What options are available for the data hardware interface from the radar to the PC directly?

A. There are only 2 options for receiving data at a PC directly from TI radars:

- SPI (Limited in data transfer speeds)
- CAN (requires USB to CAN bus adapter)

Q. What options are available for the data hardware interface from the radar to the DCA1000EVM?

A. Only LVDS.

Q. What is the fastest interface?

A. LVDS is the fastest interface at > 155.5 Mbps.

Q. How does the DCA1000EVM get data from the xWR14xx when using mmWaveStudio?

A. The only way that the DCA1000EVM gets data from the xWR14xx radar is LVDS. There is no other interface

Q. Can we use the LVDS interface to get data from the xWR14xx using CLI without mmWaveStudio?

A. The xWR14xx radar does not support a CLI command for enabling LVDS over CLI. This is due to the firmware used in the demo-visualizer utilizing the hardware accelerator included on the AWR1443 radar rather than disabling it and utilizing LVDS streaming functionality.

Q. Does the *"record_start"* command for the DCA1000EVM via CLI also enable LVDS from the radar?

A. No. It simply begins the process of transferring data from the received LVDS bus lanes to the ethernet controller. This controller prepares the data for transmission over the ethernet cable to a PC. The host computer's ethernet adapter must be manually configured for the DCA1000EVM's IP address (192.168.33.30). The data streamed to the computer appears on as UDP data on port 4098. If no data is present at the DCA1000EVM after 30 seconds of a "record_start" command, the DCA1000 will stop the process.

Q. Which radars have onboard DSP?

A. All onboard DSP enabled TI radars utilize a C674x (600 MHz). DSP enabled radars are listed below.

- xWR1642
- xWR1843
- xWR6843

Q. What are options as far as controlling the DCA1000EVM onboard DSP?

A. See the C674x DSP reference guide for details on programming.

Q. How does the DCA1000EVM get raw data samples from the radar when using mmWaveStudio?

A. Raw data is transferred via LVDS bus lanes to the DCA1000EVM. The data type is controlled by SW_1.4 on the DCA1000EVM to select RAW_DATA (off) or DATA_MODE (on). In RAW_DATA mode, the DCA1000EVM captures all LVDS data as it is. When in DATA_MODE, the user can add specific headers to different data types, and the FPGA separates out different data types based on the headers.

Q. Does the previous answer mean that theoretically the radar will support LVDS in all SDK versions?

A. In SDK versions, no. But the radar supports LVDS in all supported mmWaveStudio versions. This is because the firmware flashed to the radar to use the CLI interface is not the same as the firmware flashed to the radar for mmWaveStudio operation. From e2e: eDMA channels in the xWR14xx-mmw demo are being used for HWA data transfer, so if the user chooses to enable LVDS in the SDK, it may conflict with those channels. The CLI command is not provided to enable LVDS interfacing as it's not being tested, but the user can enable this, as certain API and interfaces support LVDS data transfer.

Q. What are the steps to capture data using CLI control for the DCA1000EVM?

A. Simply running the start_record command is all that is needed. It is the command that is called by mmWaveStudio when the ARM DCA1000EVM button is pressed to begin data collection. There is no additional setting up that is needed. This command allows the data to be visible on port 4098.

Q. Is there a way to send commands to the DCA1000EVM through MATLAB instead of using Powershell?

A. Yes. Instead of making the user enter a handful of code in a Powershell window, the CLI tool in MATLAB uses the installation path of mmWaveStudio to access the correct directory and then runs the necessary commands.

Q. Referring to the CLI tab on the MATLAB GUI, does the high-speed data streaming button utilize the DCA1000EVM?

A. This is the desired operation. I am hopeful that the LVDS_streaming functionality of the AWR1642 works properly while running the demo-visualizer firmware. This would allow the DCA1000EVM to capture the data over LVDS and present it to the user over ethernet. If the AWR1642 does not allow this, then the button will be removed.

Q. Shouldn't every radar stream through the DCA1000EVM and only some can stream high-speed data directly to the PC, bypassing the DCA1000EVM?

A. No. There is no high-speed data interface available for direct transfer between radar and PC.

Q. Will the DCA1000EVM only receive data across LVDS for certain boards?

A. The DCA1000EVM only receives data from all radars using LVDS.

Q. How does mmWaveStudio control radars as well as LVDS data transfer in general?

A. mmWaveStudio controls radars by calling .DLL functions. DLL functions are just compiled C# code.

Q. What is needed to develop custom firmware that would support functionalities that mmWaveStudio has?

A. I imagine it will primarily take having a lot of experience with SDK development. The binaries are roughly 2000 lines of code which is quite a lot to try following through at a line by line level. As Josiah suggested a few weeks ago, following along with SDK user guide examples but they are not very helpful at a first glance without some prior SDK experience. Additionally, asking questions on TI if needed.

Q. Is the firmware used for CLI control different than the one for mmWaveStudio?

A. Yes, the firmware used is different. The firmware for CLI is the same firmware used for the out-of-box demo. It is located in the SDK installation path and only has a single firmware to flash to the radar; which is a combination of the MSS and BSS firmwares. For mmWaveStudio, the firmwares are located in either the mmWaveStudio installation directory or the DFP installation directory and require a separate firmware for MSS and BSS.

Q. What version of firmware is loaded onto the DCA1000EVM's FPGA?

A. Initially, the FPGA version was v2.7 but after discovering that a new version was available, I looked into upgrading the loaded firmware and now it is v2.8.

Q. What is used on the backend to actually send commands to the radar when using mmWaveStudio?

A. mmWaveStudio controls radars by calling .DLL functions. DLL functions are just compiled C# code.

Q. Is it possible to write a binary to mimic mmWaveStudio for CLI control?

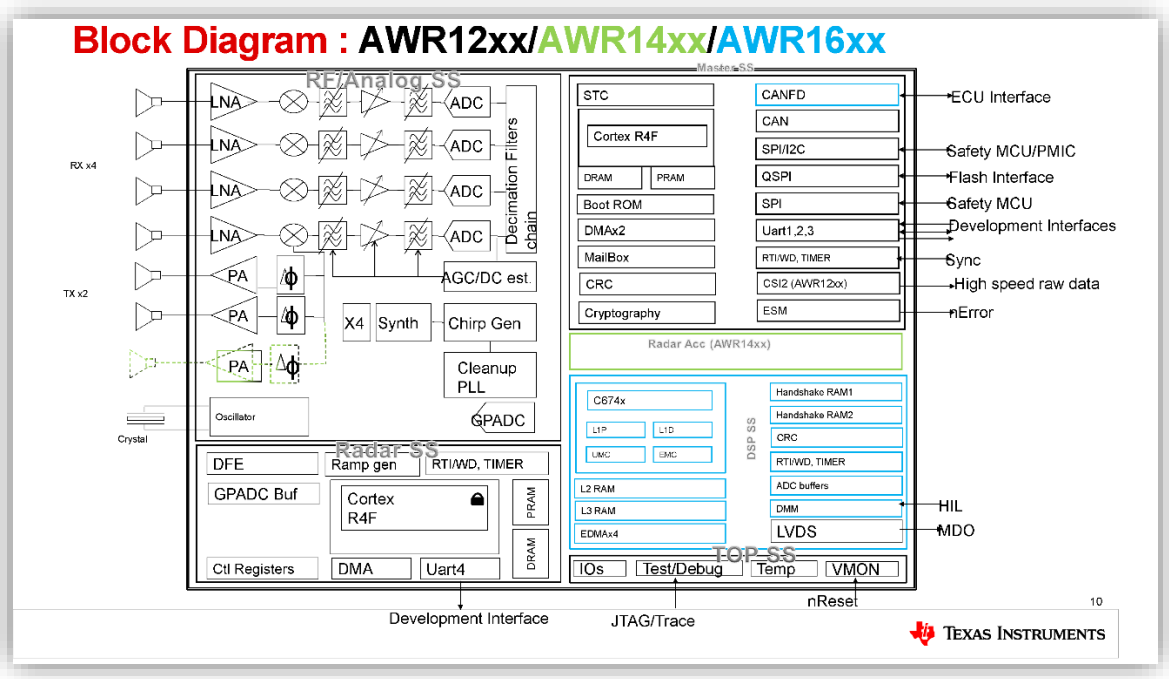
A. Since the MSS and BSS files are combined for the CLI configuration functionality of the radar, I do not believe it is possible to modify the demo-visualizer firmware to perform this task as users do not have access to reprogram the BSS file. However, it may be possible to modify a mmWaveStudio MSS file to incorporate CLI command functionality, but it is unknown if this is likely to work. The best option for this is to create a custom MSS to allow CLI functionality as well as disable any hardware through the API to the BSS. This process would need to be done for each radar to be controlled like this and without having experience doing this, it may take a long time to accomplish.

Q. What does BSS and MSS stand for?

A. The BSS (radar-sub-system) is a low-level binary that controls the hardware on the development boards and is fully developed by TI; as opposed to the MSS (master-sub-system) which performs all high-level tasks such as interfacing and accessing BSS functions through mmWave API. BSS cannot be programmed by the end user but the MSS can be.

Q. Are both the BSS and MSS in the radar?

A. Yes. See image below.



Q. Is it possible to use the existing C# functions and call them directly from MATLAB?

A. It is entirely possible to call functions inside of .DLL functions inside of MATLAB. What is not possible is using these functions to open up the mmWaveStudio listening port while mmWaveStudio is not running. This is because some of the functions are dependent on a file that is created once an instance of mmWaveStudio is running. I've come to this conclusion because the file is not available when mmWaveStudio is not running, but once it is running the file becomes available and allows the certain C# functions to run without error.

Q. What does "open a listening port inside of MATLAB" mean?

A. This was an attempt at "opening the listening port" (which is used by mmWaveStudio to receive LUA commands from MATLAB) exclusively in MATLAB rather than running an instance of mmWaveStudio. The conclusion is that this is not possible.

Q. Are there LUA commands for every mmWaveStudio command?

A. Yes. Each function in mmWaveStudio has a unique LUA command that is used for control externally.

Q. How does the existing MATLAB GUI application work?

A. LUA commands are sent via MATLAB to the RSTD listening port (on 2777) that mmWaveStudio has available whenever an instance is running. Trying to open the port to allow LUA commands to work, without having an instance of mmWaveStudio running was the goal, but I found out that this is not possible.

Q. How are the two methods different?

A. The CLI switch enables sending the data over a serial connection directly to the radar containing the configuration information, including chirp parameters, etc. and calls DCA-CLI utility to capture LVDS data completely external from mmWaveStudio. The LUA tab sends LUA commands to mmWaveStudio so it can then call .DLL files to perform the same function as if it was done in mmWaveStudio. This data is transferred over serial connection as well, in a format which I can't find information about. The DCA is controlled via a LUA command which calls the same DCA-CLI utility.

Q. What are the services that I am trying to mimic in the initialization of mmWaveStudio?

A. I was trying to follow the mmWaveStudio console up until the point that a user could make an input. The first command that is run is for RSTD initialization. The functions inside of the object RSTD_API in the file RSTDdll.dll are unable to compute due to a dependency that is unavailable.

Q. Are the problems loading .DLL files, and subsequently the C# functions, related to path or version issues?

A. To the best of my knowledge, the issues preventing me from moving forward with this idea is due to a pathing issue. In fact, the issue goes away once an instance of mmWaveStudio is opened and running. But immediately once the instance is closed and the exact same commands are run, the dependency error comes back. To the best of my knowledge, this tells me that the file these commands are dependent upon is generated once an instance of mmWaveStudio is opened and is deleted once the instance is closed.

Q. Is there a way around the above issue (DLL files missing)?

A. No. The steps needed to open an instance of mmWaveStudio requires running scripts related to the GUI of mmWaveStudio.

Q. We want to stream data to the PC using 2 DCA boards. Is this possible?

A. We have the ability to change both the IP and mac address for DCA. Also, we can change the UDP port via the json configuration file for the DCA_CLI utility. To me, it seems this is definitely possible so long as

the PC has 2 LAN adapters. If the IP address is changed, we may or may not have to change the MAC address, but we should NOT have to adjust the UDP ports since it is on a different IP.

For confirmation, I looked at the forums and found posts that also suggest this is possible. (Personally, I would be weary of using a generic USB-Lan adapter, but it may be possible for the higher end adapters known to work well. Adding another network card to a PC would be the best option.)

Q. If not, maybe we will have to stream data more slowly using serial. Is it possible to get data from multiple radars over serial?

A. Yes. I just confirmed this by running both radars at once using two instances of the TI demo visualizer (serial data transfer). If it's possible to run both at once with that tool, then we can do the same in MATLAB.

Code

(Back to table of contents)

Python

Available: https://github.com/ibaiGorordo/IWR1443-Read-Data-Python-MMWAVE-SDK-1

```
# DOUBLE CLICK (ONLY IN MS WORD) TO OPEN THIS CODE
# Additional resources: (DOUBLE CLICK (ONLY IN MS WORD) TO OPEN)
```

profile.cfg

```
import serial
import time
import numpy as np
import pyqtgraph as pg
from pyqtgraph.Qt import QtGui
import generateConfig as gC

# Change the configuration file name
configFileName = 'profile.cfg'

CLIport = {}
Dataport = {}
byteBuffer = np.zeros(2**15,dtype = 'uint8')
byteBufferLength = 0;


# ------------------------------------------------------------------

# Function to configure the serial ports and send the data from
# the configuration file to the radar
def serialConfig(configFileName):

    global CLIport
    global Dataport
    # Open the serial ports for the configuration and the data ports

    # Raspberry pi
    #CLIport = serial.Serial('/dev/ttyACM0', 115200)
    #Dataport = serial.Serial('/dev/ttyACM1', 921600)

    # Windows
    CLIport = serial.Serial('COM4', 115200)
    Dataport = serial.Serial('COM5', 921600)

    # Read the configuration file and send it to the board
    config = [line.rstrip('\r\n') for line in open(configFileName)]
    for i in config:
        CLIport.write((i+'\n').encode())
        print(i)
        time.sleep(0.01)

    return CLIport, Dataport

# ------------------------------------------------------------------

# Function to parse the data inside the configuration file
def parseConfigFile(configFileName):
```

(Back to table of contents)

MATLAB Files

```
% Additional resources: (DOUBLE CLICK ICON TO OPEN)
```

profile.cfg

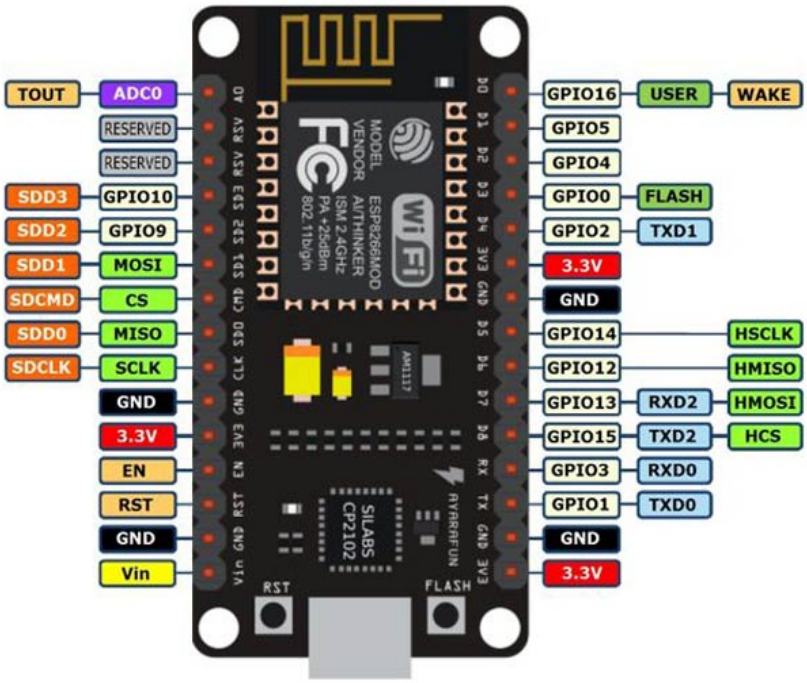
```
dca_start.ps1 (json file name is cf.json)
================================================
cd "C\ti\mmwave_studio_02_01_01_00\mmWaveStudio\PostProc\"
.\DCA1000EVM_CLI_Control.exe -h cf.json
.\DCA1000EVM_CLI_Control.exe start_record cf.json

%% ======================================================================


function connectDCA(app)
    %connect to DCA and start data capture
    path = app.RSTD_DLL_Path;

    for cnt = 1:3
        idx = regexp(path,'\');
        path = path(1:idx(end)-1);
    end

    ext = "\PostProc\";
    path = append(path,ext);

    helpCMD = ".\DCA1000EVM_CLI_Control.exe -h cf.json";
    startCMD = ".\DCA1000EVM_CLI_Control.exe start_record cf.json";

    fid = fopen('dca_start.ps1','wt');
    fprintf(fid,'%s','cd "' + path);
    fprintf(fid,'"\n');
    fprintf(fid,'%s',helpCMD);
    fprintf(fid,'"\n');
    fprintf(fid,'%s',startCMD);
    fclose(fid);
    !powershell Set-ExecutionPolicy -Scope CurrentUser Bypass
    !powershell -inputformat none -file .\dca_start.ps1
    !powershell Set-ExecutionPolicy -Scope CurrentUser Default
end
```

(Back to table of contents)

MCU

```
//ESP8266 MCU - HW Trigger - Chase A. Wood - 9/21/2020

int pulseWidth = 100;          //ms
int period = 500;              //ms    (1000) because Arduino
delay function is delay(ms)
bool result;
int idx;

//Pin Declaration
#define signalOut 5 //pin D1 == pin GPIO5

void setup() {
  Serial.begin(115200);
  pinMode(signalOut, OUTPUT);
  delay(100);
}

void loop() {
  Serial.println(pulseWidth);
  digitalWrite(signalOut, HIGH);
  delay(pulseWidth);
  digitalWrite(signalOut, LOW);
  delay(period-pulseWidth);

  if(Serial.available()) {
    digitalWrite(signalOut, LOW);

    result = newValues(Serial.readStringUntil('\n'));
    Serial.flush();
    if (result == true)
      Serial.println("Signal changed successfully");
    else
      Serial.println("Error: Invalid data sent");
  }
}

bool newValues(String str) {
  int cnt = 0;

  //Arduino does not use exceptions. Must verify data format is
correct.
  for(int i = 0; i < str.length(); i++) {
    if(str[i] == ',') {
```

```
      cnt++;
    }
  }

  if (cnt == 1) {
    idx = str.indexOf(',');
    int pulseWidthTemp = str.substring(0,idx).toInt();
    int periodTemp = str.substring(idx+1).toInt();

    if(pulseWidthTemp > periodTemp) {
      //Invalid config. to avoid MCU reset, swap variables
before programming
      pulseWidth = periodTemp;
      period = pulseWidthTemp;
      Serial.print("Warning: Parameters sent in incorrect order.
Values have been swapped to correct order.");
    }
    else {
      pulseWidth = pulseWidthTemp;
      period = periodTemp;
    }
    return true;
  }
  return false;
}
```



NodeMCU – ESP8266 Pinout