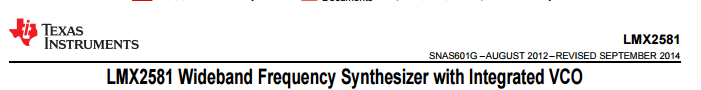
**LMX2581 – how to write the driver on embedded processor**

LMX2581 is TI RF synthesizer for frequencies how to 3457Mhz.



Datasheet reference: <http://www.ti.com/lit/ds/symlink/lmx2581.pdf>

This document will show the process of the software developer to write a driver for this device.

Unfortunately TI does not supply any source code , in C , that can be used as a reference code.

This mean that we as a developer need to work harder and more time to read the datasheet over and over , word by word, in order to access and configure that device.

Obviously, it clear that TI does not care about the customer and does not care that he is going to spend some crucial time in his project to do all the listed task in this document to get the LMX2581 device to work.

As a suggestion to TI :

Take an engineers and tell him to write a C style library for this device and many other device so the customer that is going to work with those device can do that in a few minutes.

Before we continue , I want to show you how the API I am planning is going to be:

Enum REGISTERS

{

R0

R1

R2

R3

R4

R5

}

Void Init()

bool SetFrequency(float freq)

void write\_register(REGISTERS reg, int data)

{

// do the timing diagram with the GPIO

}

int read\_register(REGISTERS reg)

{

// do the timing diagram with the GPIO

Return data;

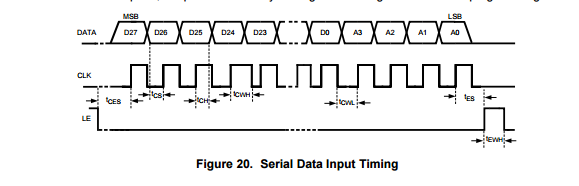
}

Of course we can have many more functions , but for now it is enough , we just want to change the output frequency.

The device does not have a known protocol like SPI I2C , or other fast bus which have known protocol.



The device as 3 wire protocol as shown in this figure:



That means that our host device should have 3 GPIO.

CLOCK, DATA and LE (letch enable)

This data input timing should help us to compose a READ and WRITE function to the internal registers of the device.

This operation can take some time to implement, and as we can see it does not tell us how the read is working, only the write.

This is a tedious work but can be accomplished with connecting a scope to see that the timing is correct.

Although it is only 3 wires it is still can do some pain while implementing.

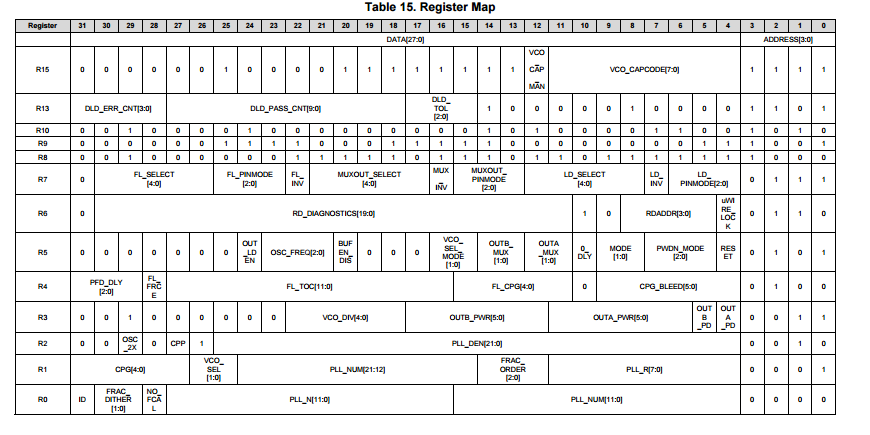
Note about the register and the writing method:

Each register is a 32 bit register but it contains 4 bits (LSB) for address and 28 bit for data.

So each register R0 will have 0000 and R1 will have 0001 at the beginning of the register value indicating it is the value of the register.

What we need to do , is just to write 32 bit , and the chip will take the first 4 bits as address.

After we understand how to write two C function, we need to learn the registers:



Fortunately In C language we have the bit field mechanism.

Using bit field, we will create 14 structures , one for each register and create the bit mapping in those structure.

It will allow us to easy access for every bit or group of bits.

The structure is looking like that:

**typedef** **union**

{

uint32\_t Value;

**struct**

{

**unsigned** address:4;

**unsigned** PLL\_NUM:12;

**unsigned** PLL\_N:12;

**unsigned** NO\_FCLA:1;

**unsigned** FRAC\_DITHER:2;

**unsigned** ID:1;

}u;

} LMX2581\_R0;

This is example of one register – we will have 13 of them like that

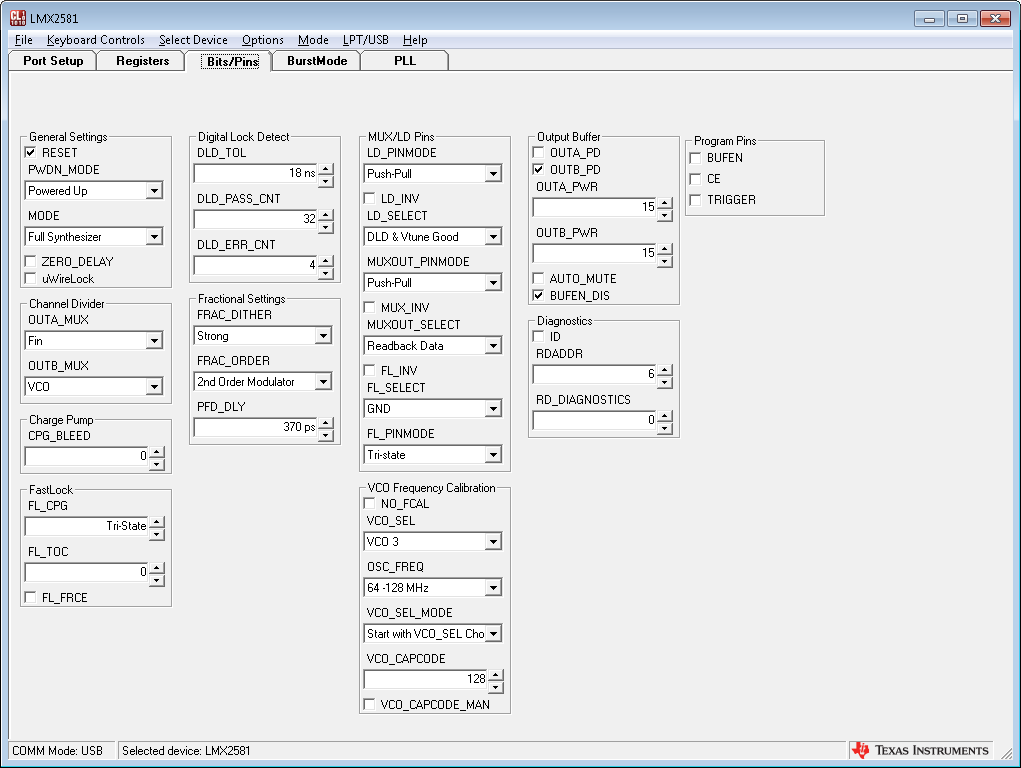
We have several options.

1. Configure each bit – will take us to learn forever and also to implement.
2. Use the Code Loader for initial configuration and the bit field structure to change bit to get only the functionality we need. – Better choice.

Until now, we did not need to understand what the registers are doing and how they actually configure that chip.

This mission the most difficult when writing a driver directly from datasheet, and we probably will never know everything about this device as the people at TI knows about it.

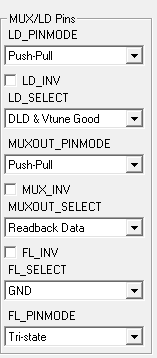
But we must continue and write our driver, so we will use TI Code Loader application



This page is the register page.

Every group, has a name, and its actually the register description and its bits.

Take for example the group name MUX/LD pins



This is actually register R7



If we combine this information, we can create some function like

SetLDPindMode()

SetMuxOut\_PinMode()

And using our API function and the bit field structure we can change bits and write to configure the device.

Some questions are open:

1. How do we actually know what to configure

* We can get help from the hardware engineer here.
* Program using the code loader to get actual signal and frequency out and see which registers where used.

1. Continue read the datasheet to understand what does every register and how it help us to complete our mission.

We have enough work to do before need to dive more and more into the document.

So let’s start (hopefully the mail we send to TI to send us the C API that we are about to write will be send to us already and we can just use it instead of writing all that by our self)

Right, it is a good thing to know how to do that kind of work for self-knowledge

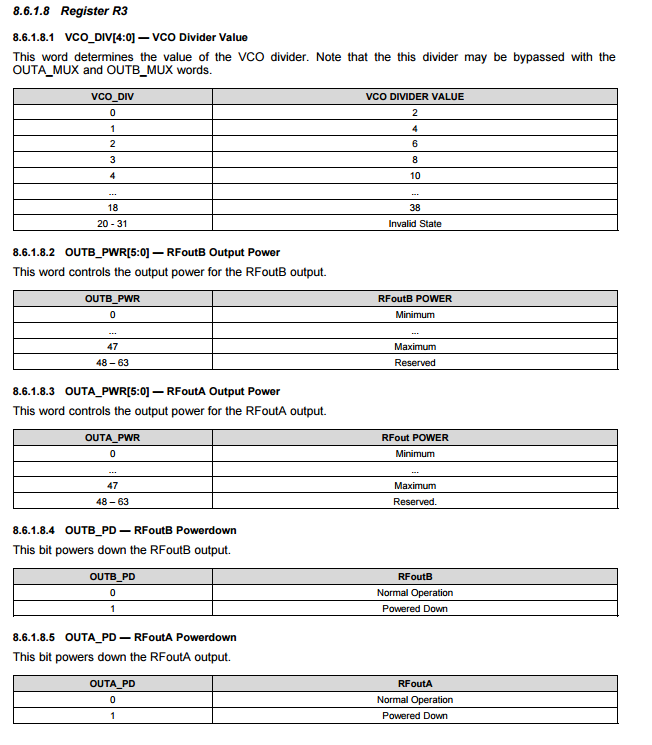
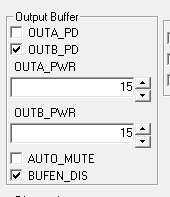
Because there are companies that does not provide any application note, and C code example or any example in any language to configure and work with their chips.

If someone from TI is reading that document, I hope he will think of creating a full package for the company chip so customer like we don’t need to struggle each time we are going to use those kind of chips.

**Registers:**

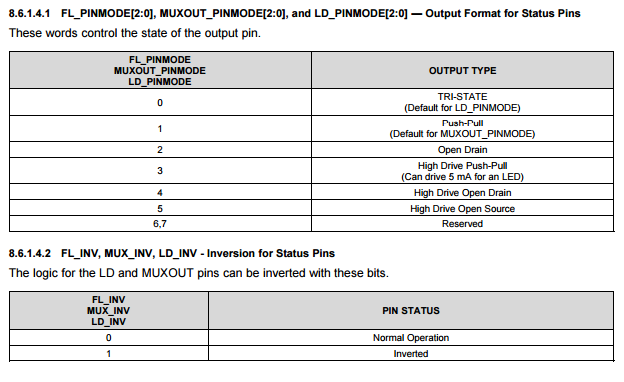
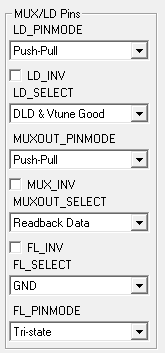
I will show here the relation between the registers and the GUI, also the valid values for each field

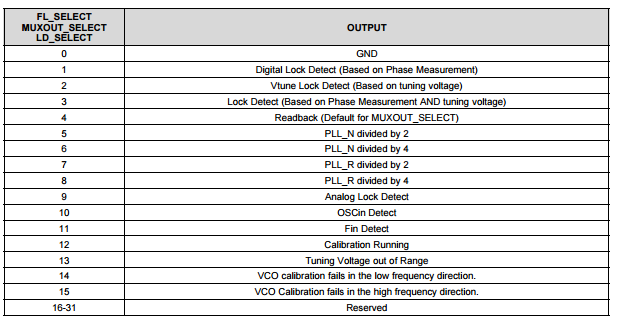
R3 –



The values for this register can be found in the datasheet, like all other registers:

R7



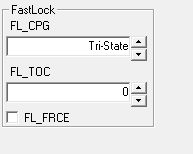


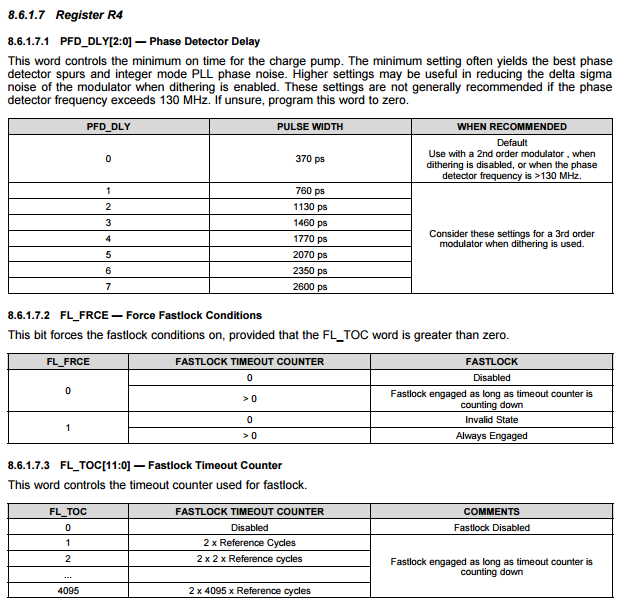
We can define each value as enum if needed to control the values.

But we will got on the approached of the changes instead of building the entire Code loader in C

Only in the place we need to change we will write code to manipulate the bits.

Register R4





Fun staff:

Registers 8 , 9 and 10 , we dong need then

