A Practical Guide to Using the ADF4351 with GNU Radio

1. Introduction

This document provides a high-level explanation and step-by-step instructions for configuring the ADF4351 frequency synthesizer chip. It addresses how to integrate the required register computations into a GNU Radio project and outlines how to pass those computations to your FPGA for final SPI control of the ADF4351.

You can imagine the ADF4351 as a device with an internal oscillator that ranges between 2.2 GHz and 4.4 GHz, but it can also output lower frequencies by means of internal dividers. This guide aims to demystify the register programming that is needed to make the chip produce the desired output frequency.

---

2. Why We Need Register Calculations

The ADF4351 must be programmed via **six (6)** internal registers: R5 down to R0. Each of these 32-bit registers contains specific fields. These fields determine:

- **Output frequency** (using integer/fractional N dividers).
- **Reference clock options** (whether to double or divide the input reference).
- **Output power** and **mute-till-lock** options.
- **Loop filter interaction** (charge pump current settings, phase detector polarity, etc.).

Manually setting these bits every time is error-prone, so we use a helper C++ class that automatically computes all the necessary bits, given only:

a) Desired output frequency (e.g., 100 MHz or 2.4 GHz).

b) Reference frequency (often 10 MHz).

c) Optional channel spacing (if fractional mode is desired).

Once the class computes these bits, we simply send the six register values to the FPGA, which then clocks them out to the ADF4351 via SPI. The synthesizer will lock to the requested frequency.

---

3. ADF4351: Key Concepts

a) **VCO Range (2.2 GHz to 4.4 GHz)**
The on-chip Voltage Controlled Oscillator (VCO) cannot go below ~2.2 GHz, nor above ~4.4 GHz.

b) **Output Divider**
When you need an output frequency lower than 2.2 GHz (e.g. 400 MHz), the ADF4351 uses internal dividers of 2, 4, 8, 16, 32, or 64 to step down the VCO frequency.

c) **N Divider**
Internally, the PLL compares the (possibly divided) VCO frequency against a reference frequency. The integer part of the divide ratio is called **INT**, and (if fractional mode is used) there is also a **FRAC** portion.

d) **PFD (Phase-Frequency Detector)**
The device's internal logic that compares the reference clock vs. the divided-down VCO. A key setting here is the charge pump current, which must be consistent with the external loop filter design.

---

4. Overview of the C++ Class

Below is a synopsis of what the included ADF4351 configuration class does

a) **Constructor**

```
ADF4351 mySynth(10e6, false, false, 1);
```

- Sets a 10 MHz reference (10e6).
- Disables reference doubler (false).
- Disables reference divide-by-2 (false).
- Sets the R counter to 1 (no extra division of the reference).

b) **setFrequency(freqHz, chanSpacingHz)**
- freqHz: The desired frequency in Hz (e.g. 2.0e9 for 2 GHz).
- chanSpacingHz: If > 0, the class uses fractional-N to achieve that spacing; if = 0, it does integer-N mode.
- This method figures out how to keep the VCO in the 2.2–4.4 GHz range by picking the correct output divider. It also sets up INT, FRAC, MOD, and prescaler bits.

c) **getRegisters()**
- Returns the final **six** 32-bit register values (R5..R0).

---

5. Example Flow: From Code to the FPGA

a) **GNU Radio Flow Graph**

- Suppose you have a block or GUI element called "LO Frequency." The user enters some frequency, say 100 MHz.

b) **Code**

```
double userFreq = 100e6;            // from GUI

ADF4351 synth(10.0e6);              // 10 MHz reference

bool ok = synth.setFrequency(userFreq, 0.0); // integer mode

auto regs = synth.getRegisters();  // get the 6 register values
```

- If ok is true, you have a valid set of registers.

c) **Send the 6 registers to the FPGA**

- In your project, the FPGA likely has memory-mapped control registers or a small bus. You write:

1. R5 → FPGA register for R5.
2. R4 → FPGA register for R4.
   …
3. R0 → FPGA register for R0.

- Then, set a "go" bit so the FPGA's **LO_Controller** picks those values up and pulses SPI lines.

d) **LO_Controller & SPI**

- The FPGA state machine writes those six registers (R5 first, R0 last) out to the ADF4351. The device calibrates the VCO and locks to the requested frequency within a few tens of microseconds (depending on your loop filter design).

6. Major Register Fields in Plain Terms

- **R0 (INT & FRAC)**
  - **INT** (integer division factor): determines the coarse multiplier from reference to VCO.
  - **FRAC** (fractional part): if you want fractional steps for more precise channel spacing.
- **R1 (MOD, Phase, Prescaler)**
  - **MOD**: sets the fraction's denominator (e.g., 50 for 200 kHz steps from a 10 MHz reference).
  - **Prescaler**: either 4/5 or 8/9 depending on how large INT is.
- **R2 (Charge Pump & Reference Settings)**
  - Charge pump current code: must match your loop filter.

- Reference doubler, reference divide-by-2, and lock detect mode.

- **R3 (ABP, Charge Cancel, etc.)**
  - Antibacklash pulse width: 3 ns for integer-N or 6 ns for fractional-N.
  - Charge cancellation, cycle slip reduction for fast locking.
- **R4 (Output Divider, Feedback Select, Output Power)**
  - **Output divider** bits to ensure VCO stays in 2.2–4.4 GHz.
  - **Feedback select**: typically from VCO.
  - **Output power**: from −4 dBm up to +5 dBm.
- **R5 (Lock Detect Pin, Defaults)**
  - Usually sets the lock detect pin to digital mode and leaves other fields at default.

7. Implementation Steps in GNU Radio

a) **Include the C++ header/class in your project**

- For instance, add a custom GNU Radio block or an out-of-tree module with the ADF4351 class.

b) **Create an instance**

```
// 10 MHz reference, no doubler, no /2:

ADF4351 myLO(10e6, false, false, 1);
```

c) **Set the frequency from user input**

```
double userFreq = /* get from GR param */;

bool success = myLO.setFrequency(userFreq);
```

d) **Retrieve final registers**

```
ADF4351::Regs regs = myLO.getRegisters();
```

e) **Push them to the FPGA** (R5 first, R4, R3, R2, R1, R0 last).
f) **Trigger the FPGA** to start the SPI write.
g) (Optional) **Wait for lock detect** or keep going, depending on your system needs.