Digital Design, Design Automation, Algorithms
www.bibix.nl

Designing a GFSK Receiver in Arx

Sabih H. Gerez, Bibix
sabih.gerez@bibix.nl

Version 2 (February 7, 2013)

Abstract

This technical note documents the design of a GFSK receiver in Arx. While GFSK is used in many telecommunication standards, the version described here uses toy parameters that are not related to any standard. The code is distributed as free IP on the Bibix website.[1] The note starts by a presentation of the theory and then spends some attention on how Arx can be combined with testbenches in IT++[2] and VHDL. The intention is not so much to explain GFSK, but to show how Arx can be used in a larger design. One performs the design and verification using the C++ generated from Arx, taking advantage of the high simulation speeds attainable in C++. When the design is ready, the corresponding VHDL is directly available and can be used for synthesis.

1 About Bibix, Arx, and This Document

Bibix is a University of Twente spin-off company. Next to offering consultancy services, Bibix focuses on the development of Arx, a domain-specific language and toolset for register-transfer level design. The Arx toolset contains code generators for C++ and VHDL, such that models with identical behavior in both languages can be obtained from a single source code. The Bibix website gives free access to the description of several IP blocks, such as FIR filters, CORDICs and FFTs.

This technical note provides some background information on one of the larger examples on the website consisting of several IP blocks: a GFSK receiver. The design is a multirate system. Given the fact that the current version of Arx only supports a single clock, the system needs to be described by means of multiple Arx files. Testbenches in C++ or VHDL connect the different blocks into a single top level and provide the appropriate clocks.

This document does not explain the Arx syntax and semantics for which the reader is referred to the on-line manual available at the Bibix website.

2 Introduction to GFSK

Gaussian frequency shift keying (GFSK) is a modulation method for digital communication found in many standards such as Bluetooth, DECT and Wavenis. Digital communication amounts to

[1] http://www.bibix.nl
[2] http://sourceforge.net/apps/wordpress/itpp/

translating symbols from a discrete alphabet into a signal that the transmitting side can send into a transmission medium and from which the receiving side can recover the original symbols.

In the context of this example, the alphabet has only two symbols `0` and `1`. When the alphabet consists of just two symbols, the symbols are called *bits*. The modulation method is a variant of *frequency modulation* (FM) of some carrier frequency ω_c.[3] *Frequency shift keying* (FSK) conveys information by decreasing the carrier frequency for the duration of a `0` symbol and increasing the frequency for the duration of a `1` symbol. If one applies Gaussian filtering to the square-wave signal that would shift the carrier frequency, one gets GFSK.

The models presented here are restricted to the digital part of the entire communication system using an *intermediate frequency* ω_{IF} as carrier frequency. In a real-life system, the signals traveling between antennas have a (much) higher carrier frequency, the so-called *radio frequency* ω_{RF}. Analog circuits are normally used for upconversion to RF at the transmitter and downconversion to IF at the receiver. An *analog-to-digital converter* (ADC) at the receiver side, brings the signal back to the digital domain. The discussion below leaves out the RF part of the signal processing chain and pretends that the communication takes place at IF.

3 Transmitter

The transmitted signal $s(t)$ can be described by a cosine at ω_{IF} with a time-dependent phase:

$$s(t) = A\cos(\omega_{\mathrm{IF}}t + \phi(t)) \tag{1}$$

In this formula, A is the signal's amplitude which is constant as the modulation only affects phase. $\phi(t)$ is derived from the bits that are transmitted:

$$\phi(t) = h\pi \int_{-\infty}^{t} \sum_i a_i \gamma(\tau - iT) d\tau$$

h is the *modulation index*: the larger the value, the wider the bandwidth occupied around the carrier. A frequently encountered case is: $h = 0.5$. Note that the case $h = 0$ represents an unmodulated carrier.

a_i is a sequence of numbers: $+1$ if the ith bit is `1` and -1 if that bit is `0`.

$\gamma(t)$ is the frequency pulse. If no Gaussian filter would be applied (FSK instead of GFSK), the frequency pulse would be rectangular: $\frac{1}{T_s}$ in the interval $[0, T_s]$ and 0 outside this interval, where T_s is the duration of one symbol. So, ignoring the sign of a_i, the phase contribution of one symbol would be $h\pi$. Continuing the reasoning, transmitting a continuous series of `1`'s during 1 second would amount to a total phase shift of $\frac{h\pi}{T_s}$. Note that the total phase shift in one second is equal to the angular frequency shift. The real frequency shift is then $\frac{h}{2T_s}$. This means that the instantaneous frequency of an FSK signal is either $f_{\mathrm{IF}} - \frac{h}{2T_s}$ or $f_{\mathrm{IF}} + \frac{h}{2T_s}$ ignoring the effects of switching between the two frequencies ($f_{\mathrm{IF}} = \frac{\omega_{\mathrm{IF}}}{2\pi}$).

The Gaussian filter smoothens the shape of the frequency pulse and makes it wider than one symbol period (this causes *intersymbol interference*). The goal is to avoid the high frequencies

[3] In this document the term *frequency* is sloppily used for what is actually the *angular frequency* ω: $\omega = 2\pi f$.

caused by switching. When the sequence to be transmitted contains multiple equal bits, the effect of filtering dies out and the extreme instantaneous frequencies mentioned for FSK are reached. Otherwise, the frequency swing around ω_{IF} is smaller.

The Gaussian filter is given by:

$$g(t) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2}(\frac{t}{\sigma})^2}$$

where σ is related to the filter's 3-dB bandwidth B:

$$\sigma = \frac{\sqrt{\ln 2}}{2\pi B}$$

Note that the Gaussian filter's impulse response spans from $-\infty$ to ∞. For practical implementations, the span has to be limited.

4 Noise, SNR, and BER

The *communication channel* is the connection between transmitter and receiver. Distortion of the signal by the channel affects the quality of the received signal. The considered design models so-called *additive white Gaussian noise* (AWGN) as the only source of distortion. If $n(t)$ is the noise signal, the noisy signal $s_n(t)$ can simply be expressed as:

$$s_n(t) = s(t) + n(t)$$

Note that the model does not introduce any attenuation to the signal. The noise is parameterized by the *signal-to-noise ratio* (SNR) which is the quotient of the signal power and the noise power (often expressed in dB). In the model, the signal strength is kept constant and the noise power is chosen that corresponds to the given SNR.

When designing the GFSK receiver, the *bit-error rate* (BER), viz. the number of wrongly detected bits divided by the total number of transmitted bits, is the measure of quality. The design goal is to make the receiver as cheap as possible, for example in terms of logic gates in an ASIC realization, while satisfying the BER requirements.

5 Receiver

A common method to extract the transmitted bits from the modulated signal is to shift the signal to baseband (to reduce the carrier frequency to zero), to filter the signal and then apply a so-called *delay and multiply* transformation. These steps will be explained in short below.

First, the signal as described in Equation 1 will be rewritten to:

$$s(t) = A\cos((\omega_{\text{IF}} + \omega_d)t)$$

Here, ω_d is the instantaneous frequency offset due to modulation (remember that frequency is the time derivative of phase, so $\omega_d = \frac{d}{dt}\phi(t)$). So, if one would be able to know ω_d for a specific symbol period, one would be able to know the value of the bit from the sign of ω_d: $\omega_d > 0$ would mean that a `1` was received and $\omega_d < 0$ that a `0` was received.

In order to eliminate ω_{IF}, one can multiply $s(t)$ with an unmodulated sine and cosine:

$$i_m(t) = s(t) \cdot \cos(\omega_{\mathrm{IF}} t)$$
$$q_m(t) = s(t) \cdot - \sin(\omega_{\mathrm{IF}} t)$$

This is called *mixing*. The two signals $i_m(t)$ and $q_m(t)$ are called the *in-phase* and *quadrature* components of the new signal.

Following the product rules of trigonometry, one then gets:

$$i_m(t) = \frac{A}{2} \left(\cos((2\omega_{\mathrm{IF}} + \omega_d)t) + \cos(\omega_d t)\right)$$
$$q_m(t) = \frac{A}{2} \left(-\sin((2\omega_{\mathrm{IF}} + \omega_d)t) + \sin(\omega_d t)\right)$$

The interpretation of these formulae is that both $i_m(t)$ and $q_m(t)$ will now contain the original signal twice: once around center frequency $2\omega_{\mathrm{IF}}$ and once around center frequency 0. The signal component around $2\omega_{\mathrm{IF}}$ can be removed by low-pass filtering to obtain:

$$i_l(t) = \frac{A}{2} \cos(\omega_d t)$$
$$q_l(t) = \frac{A}{2} \sin(\omega_d t)$$

The delay-and-multiply operation is a classical technique for FM demodulation [1]. It amounts to computing:

$$d(t) = q_l(t) \cdot i_l(t - \Delta T) - i_l(t) \cdot q_l(t - \Delta T)$$

Applying once again the product rules for sines and cosines, one finds:

$$d(t) = \frac{A^2}{4} \sin(\omega_d \Delta T)$$

If $\Delta T = T_s$ (one could also choose other values) and $h = 0.5$, remembering that the maximal frequency deviation for GFSK is $\frac{h\pi}{T_s}$, the value of $d(t)$ will be at most $\frac{A^2}{4} \sin \frac{\pi}{2} = \frac{A^2}{4}$. Similarly, the minimum value of $d(t)$ will be $-\frac{A^2}{4}$. Actually, $d(t) > 0$ for $\omega_d > 0$ and $d(t) < 0$ for $\omega_d < 0$. This means that the sequence of mixing, low-pass filtering and delay and multiply has resulted in a signal from which the original sequence of bits can be extracted by sampling at the right moment.

6 Design Specification and Testbench Structure

The distributed code contains the demodulator implemented in Arx. It is a register-transfer level (RTL) description of the design that uses fixed-point data types. The design is embedded in an IT++ testbench that contains models for the modulator, the channel and BER computation. The distribution also contains a behavioral floating-point model of the demodulator in IT++. The structure of the testbench is shown in Figure 1. The numbers along the edges refer to the production and consumption rates of the blocks in the style of *synchronous data flow* [2]. So, the modulator

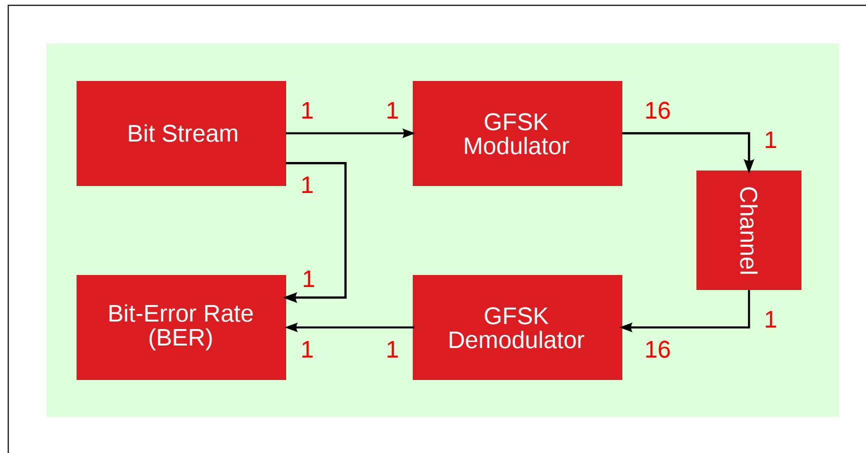

Figure 1: *The testbench structure for GFSK BER computations.*

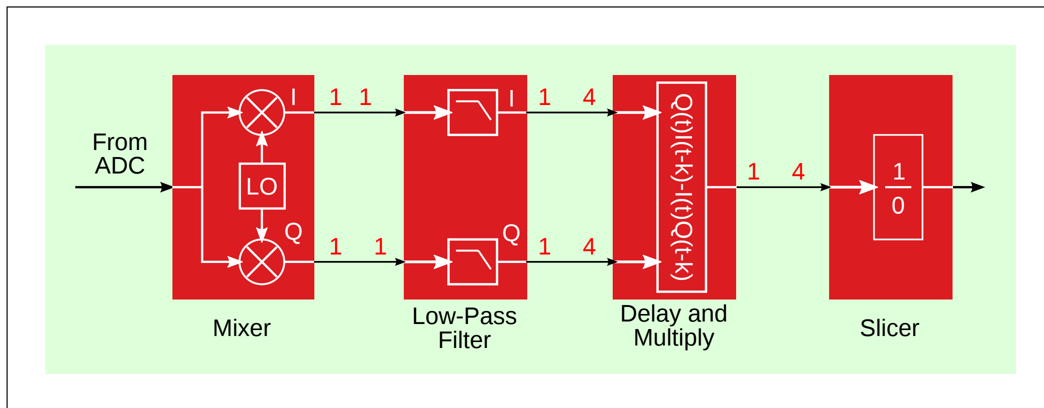

Figure 2: *The four blocks of the GFSK demodulator.*

produces 16 signal samples for each bit that it processes and the demodulator extracts 1 bit from every 16 samples that it receives. Below, the numbers will be related to the different parameters of the problem specification.

The demodulator consists of 4 blocks. They are shown in Figure 2. The blocks *mixer*, *low-pass filter* and *delay-and-multiply* are already mentioned in Section 5. The function of the last block, the *slicer*, will be explained later on in this section.

The Arx code has been developed according to the following design methodology:

- First the entire system was built in IT++ and it was verified that BERs were in the expected range.
- Then, following the signal-processing chain, each block of the demodulator was replaced by an Arx block. Fixed-point optimization was performed by reducing word lengths using the preservation of BERs as a quality criterion.

Parameter	Value
symbol rate	500 kHz
modulation index h	0.5
input sample frequency (ADC output)	8 MHz
hardware clock frequency	8 MHz
ω_{IF}	1 MHz
bandwidth low-pass filter	1 MHz

Figure 3: *Problem specification.*

Note that one could, alternatively, have directly used Arx for the initial behavioral models of the demodulator blocks. Arx support floating-point data types, although they are not synthesizable.

Some numeric specifications for the GFSK demo are given in Figure 3. The values do not belong to any real-life GFSK-based system. Modulation uses 16 samples for each symbol. For this reason, a symbol rate of 0.5 MHz corresponds to a sample rate of 8 MHz (so, using the Nyquist criterion, the maximum frequency representable in the system is 4 MHz). A hardware clock of 8 MHz implies a *one-to-one* implementation: hardware resources cannot be shared.

It was mentioned in Section 3, that the Gaussian filter has an infinite span in theory, but that the span needs to be limited in practice. In the reference design, the span has been limited to 4 symbol periods. It is therefore implemented as a 64-tap FIR filter.

The low-pass filter has a bandwidth of 1 MHz, meaning that all signals with a higher frequency are suppressed. This has consequences for the noise added in the channel. As the bandwidth of the simulated system is 4 times as large as the bandwidth considered after filtering, the SNR is corrected by a factor of 4. This can be seen as follows: the added noise is white, meaning that all frequencies are equally present; the filter removes three quarters of the noise; so, to have the correct noise energy in the frequency band of interest, a correction by a factor 4 is necessary.

The low-pass filter is an FIR filter with coefficients optimized for a multiplierless implementation as published in [3].

As the maximum frequency after filtering is 1 MHz, one can reduce the sample rate to 2 MHz, using a downsampling factor of 4. This is actually the case in the example receiver designs that are provided. In the 2 MHz domain, this means that there are 4 samples for each symbol instead of 16.

Delay and multiply consumes and produces 4 samples per symbol. The problem left is to detect the symbol boundaries in this data stream and take a decision about the bits received. Finding the symbol boundary is called *synchronization*. The optimal decision about the bits received should involve the samples of the symbol itself as well as those of its neighbors (remember that the Gaussian filter in the transmitter spreads a symbol across multiple symbol periods).

The reference design takes a pragmatic approach on synchronization and decision. It has a block called the *slicer* for this purpose. Only the four samples of the symbol itself are considered. For the sake of simplicity, no samples of neighboring symbols are taken into account. The four samples are added and then the sign of the sum is checked. Adding the four samples instead of inspecting just one, makes the decision more robust in the presence of noise. The hardware does not perform

synchronization. Synchronization is supposed to be performed by an external hardware unit that provides a parameter called `slicer_offset` which is an integer in the range from 0 to 3. The slicer continually adds the last four samples that it has received, but only updates its output when an internal modulo-4 counter has a value equal to `slicer_offset`.

As the output of the slicer only changes once in 4 samples, its output can be downsampled once again resulting in an output stream consisting of bits: one bit is delivered for each transmitted bit.

7 Files and Directories

This section presents the main files included in the distribution of the GFSK demo and the way they are organized in directories. The main directories in the distribution are:

- A root directory with the Arx code for the demodulator.
- A subdirectory `arx_c` with all models for C-based simulations.
- A subdirectory `arx_vhdl` with all models for VHDL-based simulations.

7.1 The Root Directory

The 4 blocks of the demodulator are described in the following Arx files:

- `mixer.arx`: the CORDIC-based mixer.
- `filter.arx`: the low-pass filter.
- `demodulator.arx`: the delay-and-multiply function.
- `slicer.arx`: the slicer.

Arx has been run on these source files and the generated C++ and VHDL files have been included in the distribution. The reader who is interested in modifying the Arx code, can use the on-line web interface at the Bibix website to generate new C++ and VHDL.

7.2 Subdirectory `arx_c`

The main files in subdirectory `arx_c` are:

- `gfsk.h` and `gfsk.cpp`: these files contain the IT++ code for both modulation and demodulation.
- `tb_ber.cpp`: the testbench for BER simulations. Using the C preprocessor `#ifdef` mechanism, the testbench actually codes for many variants of the design, starting from the design entirely modeled in IT++, followed by variants in which the demodulator blocks are stepwise replaced by Arx models. The `Makefile` creates separate executables for each variant.

Nested loops in the testbench reflect the multirate nature of the system. One iteration of the outer loop corresponds to the processing of one symbol. The 16 samples that the modulator generates for each symbol are processed in two nested inner loops, each of length 4. The C++ code for the mixer and low-pass filter is called in the central loop, the one for the delay-and-multiply and slicer at one level higher.

Without command-line arguments, the BER testbench calculates BER values for a sequence of SNRs and estimates the so-called *sensitivity level*. This is the SNR value for which a BER of 10^{-3} is reached. The estimation simply takes the SNRs for which the BER is just below and just above 10^{-3} and then performs a linear interpolation.

With command-line arguments, one can control the behavior of the testbench. One could e.g. instruct the testbench to dump intermediate signals for debugging purposes and then plot the signals with `gnuplot`. These are the implemented options:

- `-d`: turn on dump mode. Simulated signals are recorded in a directory the name of which corresponds to the simulated design variant.
- `-s` *value*: simulate only for the given SNR value instead of the built-in sequence of values.
- `-b` *value*: simulate the given number of bits.
- `-p` *value*: set the pipeline delay to the given value. This delay corresponds to the number of symbols that are discarded at the output before the first modulated symbols becomes available.
- `-o` *value*: set the synchronization parameter `slicer_offset` (see Section 6).

Subdirectory `plot` contains some scripts to plot a BER vs. SNR curve as well as to plot dumped signals in the time domain.

7.3 Subdirectory `arx_vhdl`

The files in this directory provide an environment in which the VHDL generated by Arx can be simulated. The simulation does not aim to repeat the system-level BER simulations. The very goal of Arx is to perform this type of computation-intensive simulations in C++. Simulation in VHDL is just for a “sanity check” to make sure that nothing went wrong in the invocation of the design tools. The VHDL testbench does not include a GFSK modulator. Instead, a stream of input samples as obtained from the C++ simulation is used to feed de demodulator. The output stream can be compared to a reference output stream.

The VHDL environment can also be used for post-synthesis simulations. Note that the demodulator is a multirate system. In VHDL, each block receives a clock with the appropriate frequency.

The following VHDL files are part of the distribution next to those that are automatically generated by Arx:

- `clock_gen.vhd`: clock generation.
- `gfsk.vhd`: top-level block that ties together the various blocks of the demodulator as well as the clock generator.

- pk_gfsk.vhd: a package containing design parameters such as word lengths.
- tb_gfsk.vhd: stimuli generation and processing and top-level testbench.

References

[1] J.H. Park. An FM detector for low S/N. *IEEE Transactions on Communication Technology*, COM-18(2):110–118, April 1970.

[2] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

[3] J.M.P. Langlois, D. Al-Khalili, and R.J. Inkol. Polyphase filter approach for high performance, FPGA-based quadrature demodulation. *Journal of VLSI Signal Processing*, 32:237–254, 2002.