CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Digital Audio Interface Jitter

Master's thesis in Electrical Engineering

FREDRIK SINKKONEN

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2024

Master's thesis 2024

Digital Audio Interface Jitter

FREDRIK SINKKONEN

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

Digital Audio Interface Jitter
FREDRIK SINKKONEN

Supervisor: Morten Fjeld, Department of Computer Science and Engineering
Examiner: Morten Fjeld, Department of Computer Science and Engineering

Master's Thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2024

Digital Audio Interface Jitter
FREDRIK SINKKONEN
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Jitter is the short-term deviation of a digital signal from its ideal position in time. Some common issues know to produce jitter in currently used digital audio interface formats were examined and multiple implementations of a Universal Serial Bus (USB) audio interface were designed with the intention of creating a device free from interface jitter. Using the three standardized clock synchronization mechanisms in the USB protocol for isochronous transmissions and a selection of suitable clock sources, USB audio class devices were created for which jitter measurements then were performed. The results were compared with jitter audibility thresholds from three studies containing listening tests. While all implementations were functionally acceptable, their jitter results did differ. For the two isochronous synchronization modes of USB that require a continuously adjustable clock source on the receiving side of the interface the jitter issue consists of two parts. Periodic adjustments of the clock signal are in itself a source of jitter and the way in which an adjustable clock source is constructed is another. The initial core idea was that a USB audio interface using isochronous transfers coupled with the asynchronous clock synchronization mode and a fixed frequency clock source would be able to provide an interface in which no additional jitter on top of the inherent jitter level of the source clock would be added by the transfer of data over the interface. The two fixed frequency clocks that were used did however not perform any better than the results of the best adjustable clock source and when they were attached to the test system their jitter levels increased even further. Analysis of the jitter measurements point in the direction of asynchronous mode being preferable for lowest possible jitter levels but the results are not completely unambiguous and jitter levels below the lowest recorded hearing thresholds were also achieved with one of the other synchronization modes for isochronous USB transfers.

Keywords: Asynchronous, Audio, Clock, DAC, Digital, Interface, Jitter, PSoC, S/PDIF, USB.

Contents

List of Figures

List of Tables

Terms and Abbreviations

This section lists terms and abbreviations that are used throughout the thesis.

ADC	Analog-to-digital converter
AES/EBU	Digital audio transfer interface standard for professional use created by the Audio Engineering Society (AES) and the European Broadcasting Union (EBU). Interchangeably sometimes called AES3.
API	Application programming interface
ASRC	Asynchronous sample rate converter
BMC	Biphase mark code
CD	Compact disc
CDF	Cumulative distribution function
CRC	Cyclic redundancy check
DAC	Digital-to-analog converter
DC	Direct current
DMA	Direct memory access
DSI	Digital system interconnect
FF	Fixed frequency
FIFO	First in, first out
GUI	Graphical user interface
I²C	Inter-integrated circuit
I²S	Inter-IC sound
IAD	Interface association descriptor
IC	Integrated circuit
IDE	Integrated design environment

IEC	International Electrotechnical Commission
IMO	Internal main oscillator
LSb	Least significant bit
MCKL	Master clock
MSb	Most significant bit
NRZI	Non return to zero invert
PDF	Probability density function
PID	Packet identifier
PLL	Phase-locked loop
ppm	Parts-per-million
PSoC	Programmable system-on-chip
Red book	The compact disc digital audio specification is printed in a book that has a red cover. Hence the term "Red book" refers to regular CD audio format.
S/PDIF	Sony/Philips digital interface. Digital audio transfer interface standard based on AES/EBU and intended for consumer audio products.
SCK	Serial clock
SD	Serial data
SOF	Start-of-frame
SWD	Serial wire debug
TIE	Time interval error
TX	Transmit
UART	Universal asynchronous receiver/transmitter
UDB	Universal digital block
USB	Universal Serial Bus
VCO	Voltage controlled oscillator
WS	Word select
XO	Oscillator
XTAL	Crystal

1

Introduction

This chapter introduces the topic of the thesis and gives an explanation as to why it was selected. It starts out with a brief description of the background and motivation for the thesis subject, followed by scope and delimitations along with the requirements for the intended hardware and software build. Lastly, an overview of the structure of the rest of the report is provided.

1.1 Background and Motivation

Digital audio data can be created by sampling and quantizing an analog sound wave by the use of an analog-to-digital converter (ADC). The number of bits used for each sample will determine the resolution, or how accurately the amplitude of the analog signal can be represented in digital form, and the sampling frequency will as described by the sampling theorem [1] effectively put an upper bound on the frequency range that can be sampled and stored digitally by the ADC. To play back the digitally stored audio, the digital audio data can be fed together with a clock signal that matches the sampling rate into a digital-to-analog converter (DAC), which then consequently converts the digital signal to an analog one that can be sent to a speaker, usually first passing through an amplifier.

A digital signal is less susceptible to interference than what an analog signal is, so there is a motive in trying to keep the audio in the digital domain for as long as possible before eventually having to convert it to analog format so that it becomes audible. This sometimes means that it will be necessary to transfer the digital signal between different audio devices, creating a need for a robust transfer protocol and a digital audio interface that will keep the audio data intact during the transfer. Any bit errors introduced into the audio data could severely degrade the sound quality, so they must be avoided. Most digital audio interfaces can be considered reliable when it comes to this characteristic [2], but it is not only the audio data itself which must be preserved in the transfer process; the clock signal, which is sent in parallel with the audio data into the DAC must also remain unaffected by the transfer from one device to another, as small timing errors in the clock signal, defined as jitter [3] can lead to subtle but still audible effects [4]. This is however something which at times has been neglected.

The AES/EBU interface [5] created for professional use by the Audio Engineer-

ing Society (AES) and the European Broadcasting Union (EBU), and its equivalent counterpart for consumer audio devices, the Sony/Philips digital interface (S/PDIF) [6], standardized in IEC 60958 [7] by the International Electrotechnical Commission (IEC), both exhibit weaknesses in how the clock signal is handled [2]. Universal Serial Bus (USB) has over time gained popularity as a dedicated digital audio interface, but its characteristics when it comes to jitter performance in the clock signal depend largely on how the interface is implemented [8]. A separate clock signal is not per se sent for USB audio, but there is still a need to keep the clocks at both sides of the interface synchronized. Historically, for audio companies recognizing the problem with jitter being introduced into the clock signal by AES/EBU and S/PDIF, the selected course of action has often been to keep using that same digital audio interface design which introduces the jitter in the first place, and then with various methods [9] try to remove or reduce the jitter from the clock signal again once the transfer of the digital audio data has been completed. The result of this approach has often been added hardware complexity and increased development and manufacturing costs, while it still remains questionable if the jitter has been removed or diminished to sufficiently low levels. A better approach it seems would be to use an audio interface design which does not introduce interface jitter into the clock signal to begin with. In theory, such an interface can be created utilizing a USB audio device class AudioStreaming interface [10] configured to run in asynchronous synchronization mode [8, 9].

1.2 Scope and Delimitations

The aim for this project is to see how digital audio interfaces can be implemented and then try to build a digital audio interface which does not add jitter to the clock signal in the process of transferring digital audio from one device to another. The interface should also be able to transfer the digital audio data reliably without bit errors or significant delay in the signal path. This is to be accomplished by implementing a USB audio device class AudioStreaming interface using a Cypress programmable system-on-chip (PSoC) development board and microcontroller. The build requires both hardware and software design in order to produce a testing platform. Jitter measurements are also to be performed and presented but no listening tests or any other kind of auditory assessment regarding jitter and its impact on audio quality will be made.

1.3 Functional Requirements

The interface should fulfill the following functional requirements:

- Support standard Red Book 2-channel Compact Disc (CD) audio data as input; 16 bits per sample at a sample rate of 44.1 kHz.
- Use inter-IC sound (I^2S) output format for the received audio data so that it can be sent to a DAC to be converted to analog and played back.

- Follow interconnect and interface standards so that the test device can be plugged into an audio source without need for special ports, cables, drivers or software.

No additional USB audio functions except for the necessary AudioControl and AudioStreaming interfaces and their associated endpoints will be implemented in the USB module.

1.4 Outline

The rest of the thesis is organized as follows. Chapter 2 contains theory about jitter audibility, statistics, and the taxonomy of jitter types. A large part of the chapter is also devoted to a walkthrough of the relevant audio interface formats and their characteristics. In Chapter 3, the common design layout for all implementation modes and the specifics for each one of them are described. The clock configuration is being examined in detail while other parts of the system setup are treated in a more general sense. Chapter 4 shows the functional results for the different implementation modes together with measurements of the jitter levels, presented both in numbers and visualized as histograms. The results and their validity are then evaluated. The report ends with Chapter 5 containing a summary of the findings and sustainability and environmental considerations. A listing of the device descriptors used for the synchronous, asynchronous and adaptive mode USB audio interfaces can be found in Appendix A, programmed register settings for the external clock generator board are located in Appendix B and in Appendix C are histograms from the period and cycle-to-cycle jitter measurements.

2

Theory

This chapter starts out by defining what jitter is and the possible causes to its existence in a system. Then follows an introduction to statistical theory and a characterization of the different jitter types. An overview of previous work related to jitter audibility testing and audibility threshold theory is presented and we will also get more acquainted with a selection of some commonly used audio interface formats. Particular attention is being paid to some of the possible sources of jitter often being associated with the AES/EBU and S/PDIF interfaces. The chapter ends with a look at clock generation with phase-locked loops (PLLs) and fractional dividers.

2.1 Jitter

The following section of this chapter defines what is meant by jitter, it introduces some statistical terminology for jitter distributions and it also characterizes the different kinds of jitter that we may encounter. Conducting listening tests of any kind is out of scope for this thesis but results and observations from auditory assessments made in other studies are presented and one purely theoretical jitter audibility threshold model is also provided.

2.1.1 Definition of Jitter

Jitter can be defined as the short-term time displacement a digital signal has relative to its ideal position in time [3, 11]. Let us start by viewing an ideal square wave clock signal with 50 percent duty cycle for which each clock cycle starts at the rising edge of the signal at time 1τ, 2τ, 3τ etc. If the clock signal is affected by jitter, the rising and falling edges can be offset from their ideal positions in time as visualized by the shaded areas in Figure 2.1. This offset of a signal compared to an ideal reference point in time is called the time interval error (TIE). Depending on the context, we may choose to compare an examined signal to an ideal reference like we do in Figure 2.1, or if no reference signal exists, we can instead look at the rising clock edges and compare their time of occurrence from one clock cycle to the next. In Figure 2.2 we see an example of the latter where the period jitter P_n for clock cycle n is the difference between two consecutive rising edges of the signal. Another commonly used measure that does not require a reference signal is the cycle-to-cycle

jitter denoted by C and it can be calculated by measuring the difference between two consecutive period jitter values as shown in Figure 2.2. Cycle-to-cycle jitter is usually expressed as an absolute value and not in terms of negative numbers.

It is worth to note that jitter only is the short-term time deviation in a signal, and that deviation over a longer period of time instead is defined as drift or wander. This could for example typically be the accumulated time deviation between a reference clock and a second free running clock that are in sync at start but where the jitter in the free running clock then causes it to become more and more out of sync with the reference as more and more clock cycles go by.

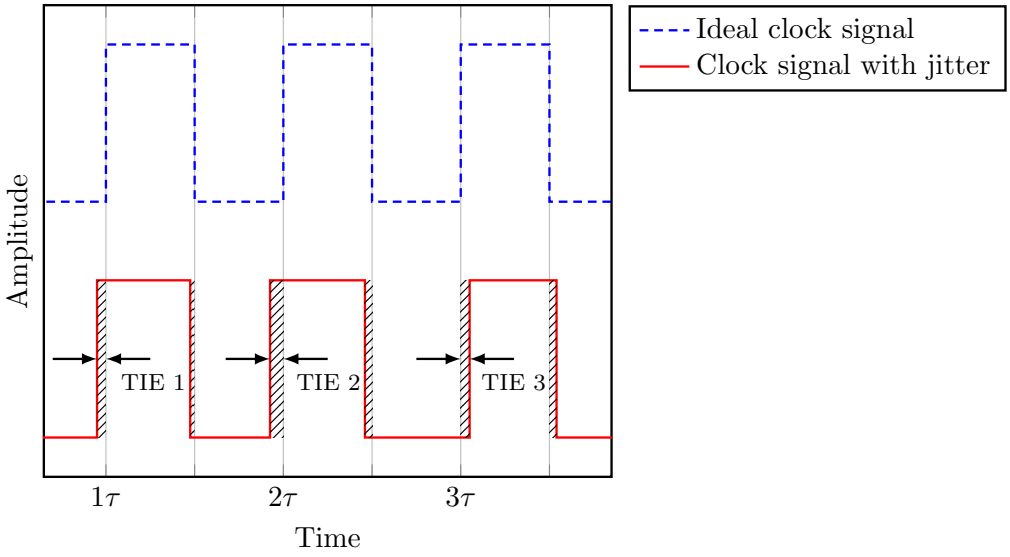


Figure 2.1: Jitter assessment for a clock signal having a reference.

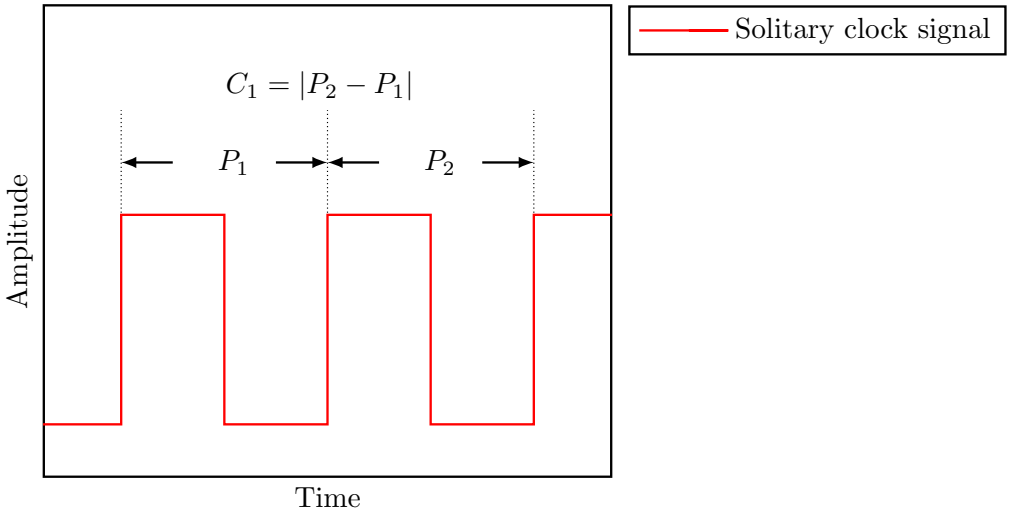


Figure 2.2: Jitter assessment for a solitary clock signal.

2.1.2 What Causes Jitter?

Although jitter is seen as a shift in the time domain, it is often caused by a disturbance in the voltage domain. In Figure 2.3 we see how noise of amplitude ΔV can create a difference in the voltage level for a rising signal edge and give rise to jitter of size Δt as it makes the signal reach the threshold level of the signal transition at a different point in time than expected.

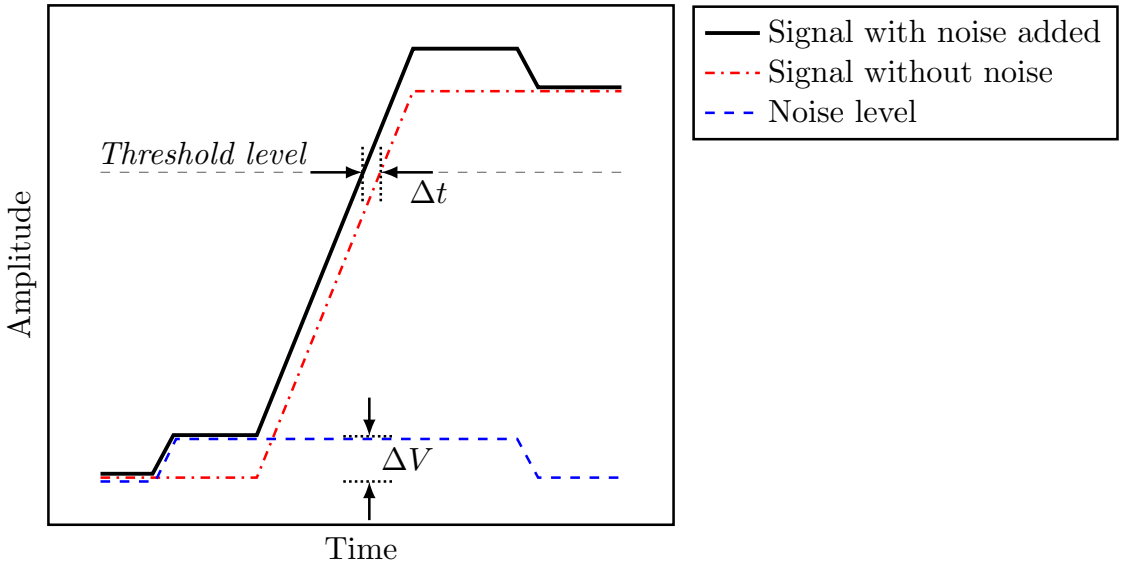


Figure 2.3: Visualization of how noise in the voltage domain can produce jitter.

There are many possible causes for voltage noise. It can originate from sources external to the signal path. Examples of this are 50 Hz to 60 Hz interference from the fundamental power line frequency, switching power supply noise and capacitive or inductive crosstalk from other cables or signal paths. Noise can also arise from sources within the signal path. Internal thermal noise caused by electrical components, shot noise appearing due to fluctuations in the flow of electrons or holes in semiconductors, burst noise and 1/f noise occurring in electrical components due to material imperfections are examples of this. Any kind of variation in voltage level can lead to jitter.

2.1.3 Probability Theory for Jitter Distributions

As we from Chapter 2.1.1 now know how to define jitter looking at one clock cycle at a time, we will introduce two terms from probability theory and statistics which will aid us when handling longer series of jitter measurements. The first one is the cumulative distribution function (CDF) [3, 12]. Looking at the transition times of the rising edges of a clock signal that is affected by jitter, we can create a function which indicates the probability of the signal having reached its high state at a certain point in time relative to the ideal transition time of the clock signal. This function is called the CDF and a theoretical example is displayed in Figure 2.4. Before τ_1, a

long time ahead of the ideal transition time for each edge, none of the rising edges have reached the high state and the probability of a state transition having happened is zero. As we move past τ_1 and closer to the ideal transition time τ_i for each edge, more and more state transitions are starting to happen. In our theoretical example in Figure 2.4, the number of state transitions happening before the ideal transition time τ_i has for simplicity been set equal to the number of state transitions happening after the ideal transition time τ_i, and more state transitions are also happening closer to the ideal transitions time τ_i than further away from it. This does not necessarily need to be true for an actual series of real world jitter measurements, but it gives us a feasible model which we can work with to understand probability theory for jitter distributions. As we move past the ideal transition time τ_i towards τ_2 in our example, fewer and fewer new state transition happen the further away from τ_i we get, while the probability of a state transition having happened, the CDF, continues to rise and it reaches its maximum value when we cross τ_2, at which point all rising edge state transitions for the theoretical measurement series have already happened. The CDF is a monotonic increasing function, meaning its value will never decrease but instead it will either always remain constant or increase as the function variable increases, and the CDF will go from $0 \rightarrow 1$ when time increases from $\tau_1 \rightarrow \tau_2$.

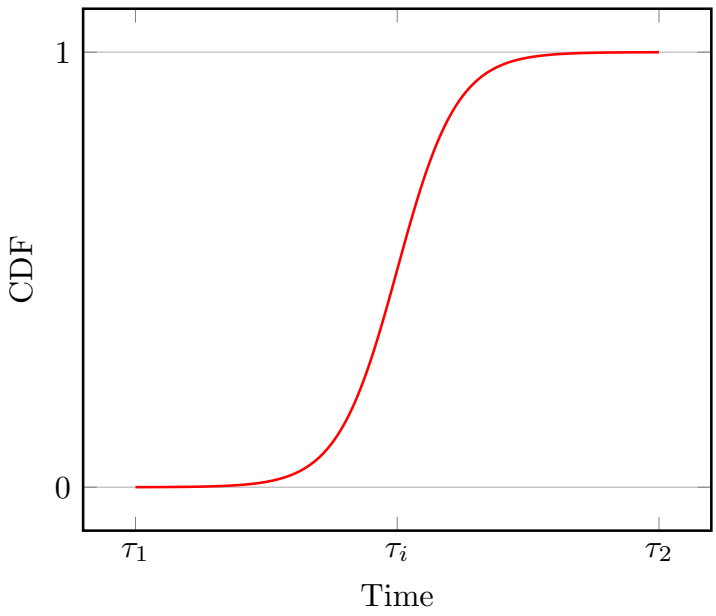


Figure 2.4: Theoretical example of a cumulative distribution function for a clock signal with ideal transition time at τ_i.

The probability density function (PDF) [3, 12] is the second term from probability theory and statistics that we will introduce in this section. Let us first consider the probability for a signal transition to happen at time τ_p. The probability of a signal transition happening exactly at an arbitrary point τ_p in time is zero as that would require the the transition to take place within an infinitely small time span, but if we instead look at a small time bracket from τ_p - γ to $\tau_p + \gamma$ as in Figure 2.7, then the probability for a transition to happen within that time range can be expressed.

The mathematical relation between the CDF and the PDF is

$$CDF(t) = \int PDF(t)dt \tag{2.1}$$

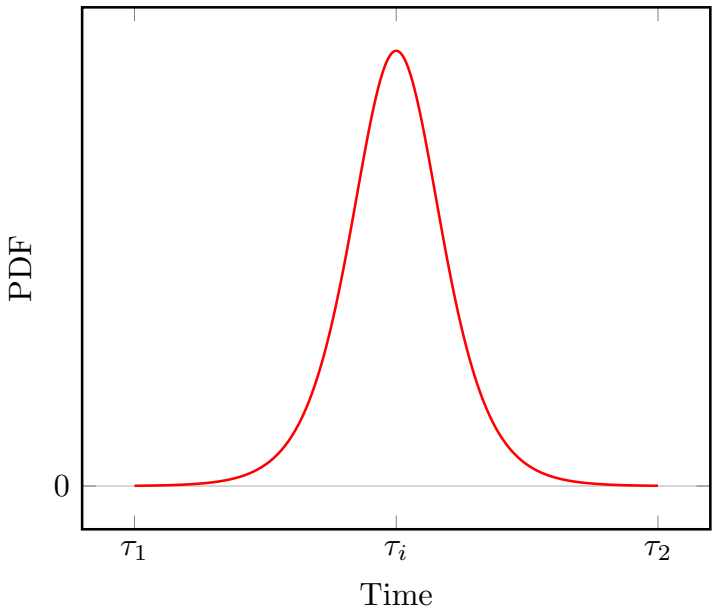


Figure 2.5: The probability density function corresponding to the cumulative distribution function in Figure 2.4.

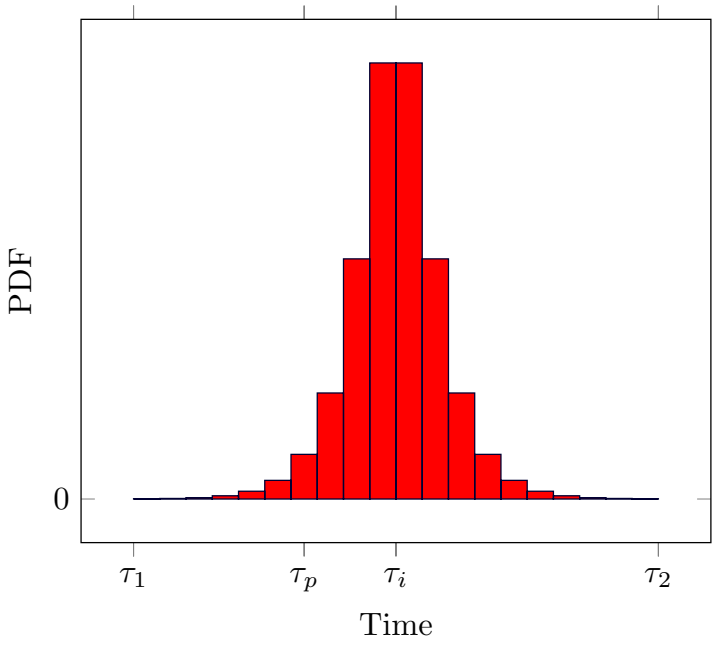


Figure 2.6: The probability density function from Figure 2.5 divided into time brackets with an arbitrary point in time τ_p selected.

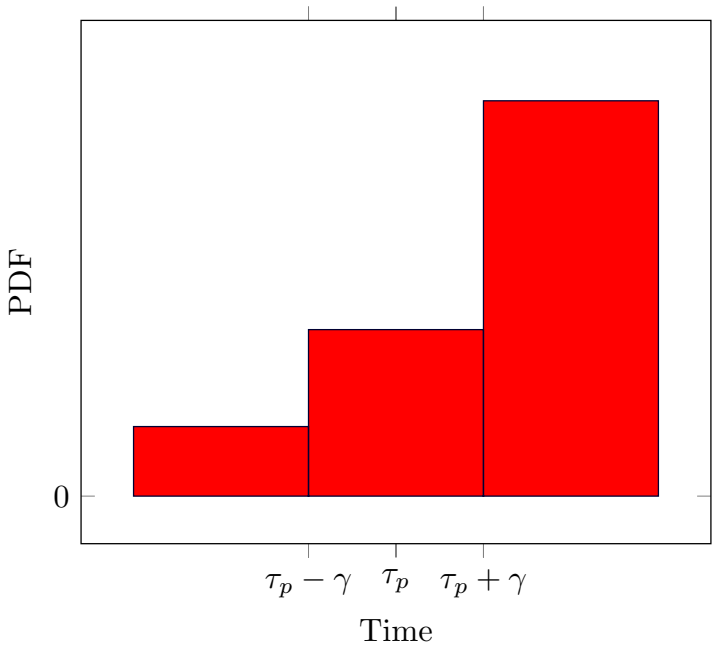


Figure 2.7: A closeup of the probability density function in Figure 2.6 around the arbitrarily selected point in time τ_p.

Figure 2.5 displays the PDF corresponding to the CDF in Figure 2.4. From Equation 2.1 we also realize that choosing to look at the PDF for a single point in time τ_p instead of a time interval τ_p - γ to τ_p + γ will give us an integration interval ranging from τ_p to τ_p, and the result of $\int_{\tau_p}^{\tau_p} PDF(t)dt$ will therefore be 0, so we need to express the probability of a signal transition happening at τ_p as the probability of it happening during a time interval τ_p - γ to τ_p + γ and not at an exact single point in time. When dealing with any real world measurement series, we will often organize our measurements to fit into predefined time brackets like we have done in Figure 2.6 for the theoretical PDF from our example.

2.1.4 Jitter Types

Jitter is often characterized as belonging to one of two categories, being either random or deterministic. The main difference between the two is that random jitter is unbounded, i.e. the jitter can in theory take on any value while deterministic jitter is bounded and therefore only has a limited range of values it can assume. Depending on the source of the deterministic jitter and its characteristics, it is often being specified further as belonging to one of a number of subcategories of its main jitter type. These subcategories of deterministic jitter are presented along with random jitter in more detail in the following sections. Looking at the plotted PDF for a measurement series may help us identify which jitter type we are dealing with so that we can try to determine its cause. The total jitter at any given moment is the sum of all jitter components that happen to be present at that point in time and jitter in any real world measurement is likely to be a composite of multiple jitter types of different origins rather than of just one single type.

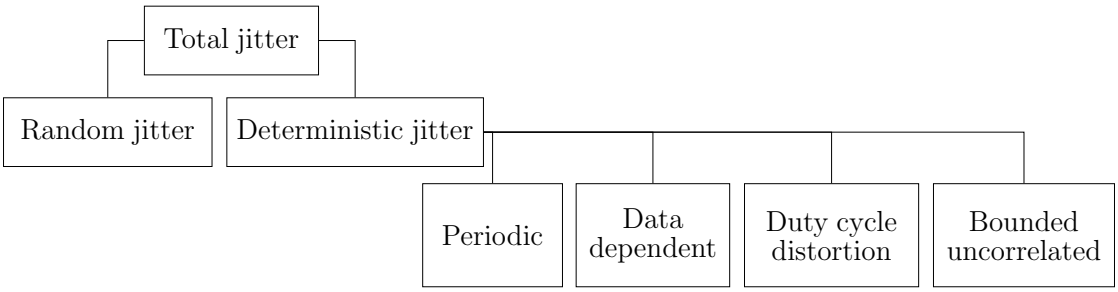


Figure 2.8: Jitter components contributing to total jitter.

2.1.4.1 Random Jitter

The most important properties for random jitter [3, 11] is that the jitter is unbounded and that the PDF for the majority of cases of can be represented by a normal distribution [12]:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{2.2}$$

By selecting the mean value μ for the normal distribution to our ideal transition time $t = 0$ for the function and setting the standard deviation σ to 1, we can simplify the general expression for the normal distribution in Equation 2.2 to

$$PDF_{random}(\Delta t) = \frac{e^{-\frac{\Delta t^2}{2}}}{\sqrt{2\pi}} \tag{2.3}$$

also replacing the variable x with Δt. A graph of the PDF for random jitter with ideal transition time 0 and standard deviation $\sigma = 1$ is displayed in Figure 2.9. All the internal types of noise listed in Chapter 2.1.2 belong to random jitter.

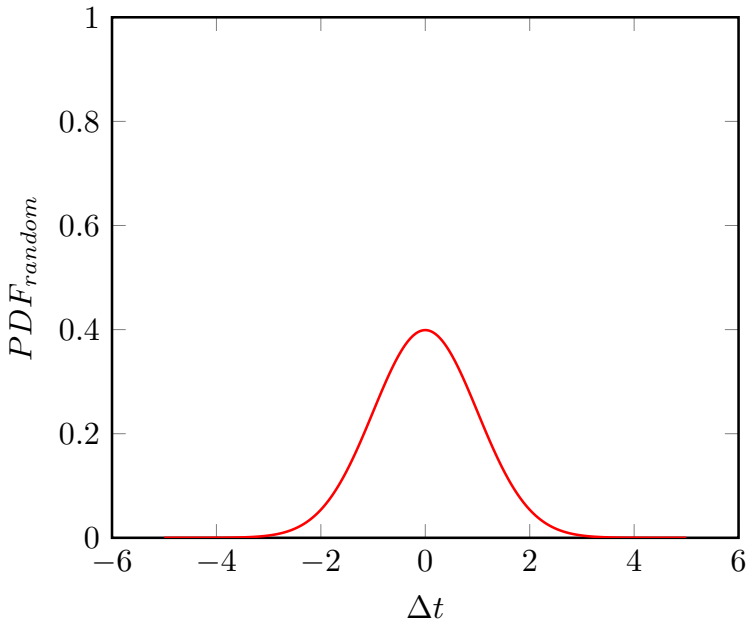


Figure 2.9: Probability density function for random jitter.

2.1.4.2 Periodic Jitter

Periodic jitter [3, 11] is jitter which repeats with a certain time interval. It is however totally uncorrelated to any clock or data signal in the system and the maximum frequency at which the jitter appears must be less than half the data rate in order for the jitter to be considered to be periodic and not data dependent. Periodic jitter can often be assumed to have a sinusoidal waveform, and for more complex cases the periodic jitter can be decomposed into a discrete Fourier series consisting of multiple sinusoidal waveforms that can be treated separately. The PDF for sinusoidal periodic jitter can be written

$$PDF_{periodic,sinusoidal}(\Delta t) = \begin{cases} \frac{1}{\pi\sqrt{a^2-\Delta t^2}} & |\Delta t| \leq a \\ 0 & |\Delta t| > a \end{cases} \tag{2.4}$$

and its graphical representation is displayed in Figure 2.10.

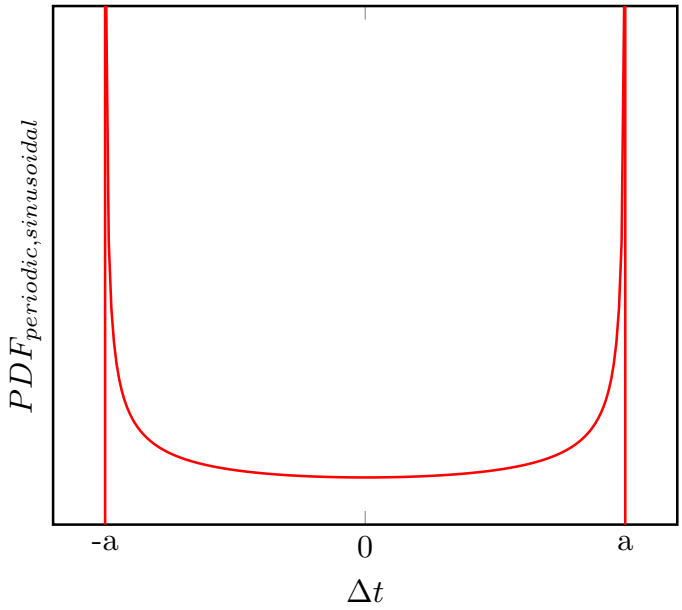


Figure 2.10: The probability density function for a sinusoidal periodic jitter distribution.

2.1.4.3 Data Dependent Jitter

Data dependent jitter [2, 3, 11] is as the name implies a type of jitter which is dependent on the data pattern that precedes the time at which the jitter manifests itself. There are multiple mechanisms that contribute to this jitter type and they are all related to the signal level being offset in relation to the threshold level which denotes the signal transition. It can be due to reflections in the signal path caused by an impedance mismatch or because the signal transition begins from a voltage level lower or higher than expected on behalf of the signal not having had time to

settle from the previous signal transition. Bandwidth limitations and asymmetrical slew rates may also affect the rise and fall times of the signal. Any reflections on the signal path will die out within a limited amount of time, resulting in just the most recent data pattern having an affect on the signal level and jitter. The PDF for data dependent jitter can be represented by

$$PDF_{dependent}(\Delta t) = \sum_{j=1}^{N}\{p_j \times \delta(\Delta t - t_j)\}, \qquad \text{where} \sum_{j=1}^{N} p_j = 1 \tag{2.5}$$

In Equation 2.5, $\delta(\Delta t - t_j)$ is the Dirac delta function[13], which has the properties

$$\delta(x) = \begin{cases} \infty, & x = 0 \\ 0, & x \neq 0 \end{cases} \qquad \text{and} \int_{-\infty}^{\infty} \delta(x)dx = 1 \tag{2.6}$$

The graphical representation of the PDF for data dependent jitter will typically have just a few discrete vertical asymptotes, which do not necessarily all have the same height as some data patterns causing the jitter could be more frequent than others.

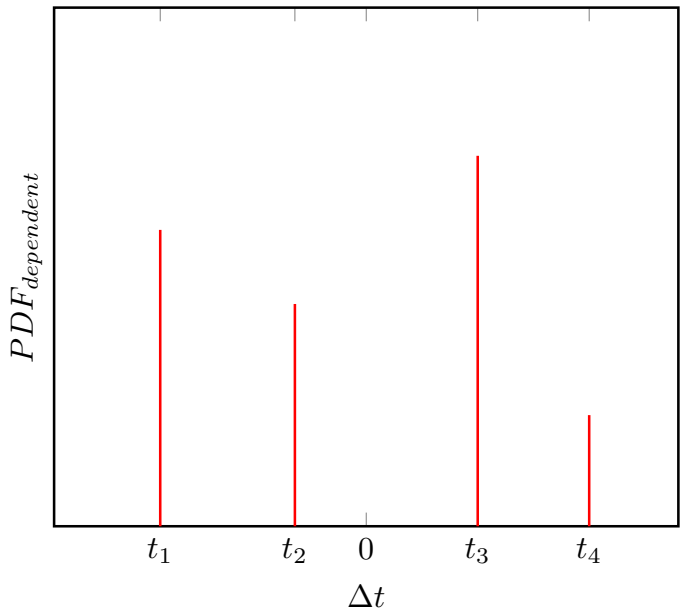


Figure 2.11: Typical probability density function for data dependent jitter.

2.1.4.4 Duty Cycle Distortion

The duty cycle defines how much time a digital signal spends in the high state versus how much time it spends in the low state. For an ideal clock signal the ratio would be 50/50 as the signal alternates back and forth between high and low, spending exactly the same amount of time in each state. Deviation from this ideal scheme, whether it is caused by an offset signal amplitude, asymmetry in rise and fall times,

or an offset threshold level for the signal transition is called duty cycle distortion [3, 11]. The PDF for duty cycle distortion will look like the two equally tall peaks in Figure 2.12, if both rise and fall transitions are included, and mathematically the PDF can be expressed as

$$PDF_{duty}(\Delta t) = \frac{\delta(\Delta t - a)}{2} + \frac{\delta(\Delta t + a)}{2} \tag{2.7}$$

where $\delta(\Delta t \pm a)$ is the Dirac delta function from Equation 2.6.

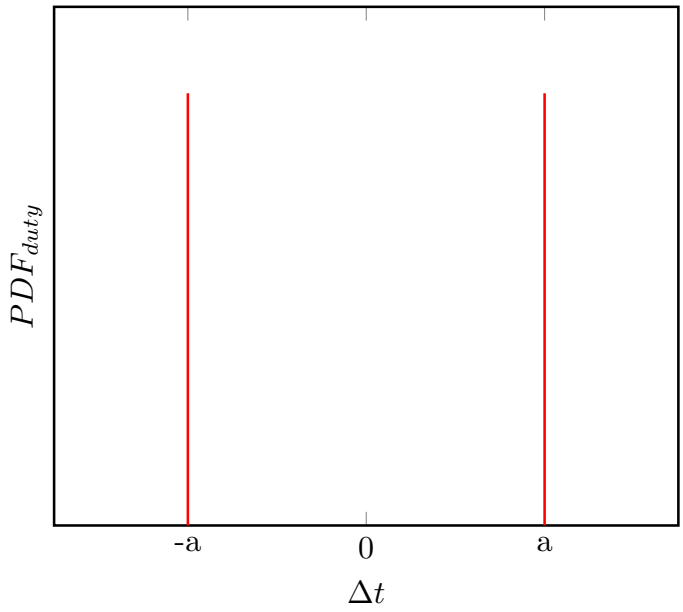


Figure 2.12: Probability density function for duty cycle distortion.

2.1.4.5 Bounded Uncorrelated Jitter

Bounded uncorrelated jitter [3, 11] covers any deterministic jitter which does not fit into any of the other three categories of deterministic jitter that have been presented in this chapter. The sources for this type of jitter can be many and the variety of causes does not make this category of jitter lend itself to making any particular generalizations about it. We will therefore just settle for using it to categorize any bounded jitter which is not periodic, data dependent or caused by duty cycle distortion.

2.1.5 Audibility of Jitter

An important question that we should ask ourselves is, "How much jitter can be tolerated before it starts to affect the sound quality?" In order to give a proper answer, we would need to ask subsequent questions such as, "What frequency range does the audio affected by jitter have?" and "What type of jitter is the audio signal being affected by?" Studies by Benjamin and Gannon [14] and Ashihara et al. [15] have shown that jitter will be more audible in source material that has more high

frequency content than one consisting of lower frequencies. This is because the effect of jitter on an audio signal not only is proportional to the amount of timing error in the signal, but also to the overall slope of the curve of the audio signal being affected. As is visualized in Figure 2.13, the same amount of timing error Δt on two sine waves of same amplitude but different frequencies will produce a bigger offset a_2 in the amplitude on the signal with a steeper slope compared to the amplitude offset a_1 on the signal with a more gentle slope, so the distortion is more likely to be audible when playing high frequency audio content.

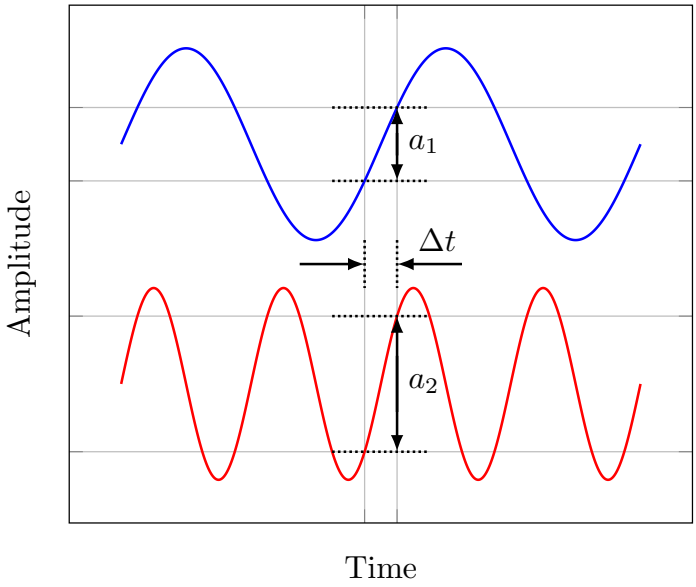


Figure 2.13: The size of an amplitude error caused by a timing error depends on the slope of the signal.

For the question regarding the type of jitter affecting the audio signal, one of the first well known studies on this subject conducted by Manson [16] back in 1974 found the hearing threshold for sinusoidal jitter to be a little bit lower than for random jitter. The mentioned studies by Manson [16], Benjamin and Gannon [14] and Ashihara et al. [15] all include listening tests by which their respective jitter audibility thresholds are determined, but none of them do in subjective terms express how the test participants experienced jitter to affect the sound or what made the test subjects pick out the tracks with jitter and distinguish them from the ones without it. We can however note that most selected test tracks contained high frequency source material as jitter audibility is greater for higher frequencies. Tracks with solitary elements and sparse sound like a single instrument were also favored in place of more complex tracks with a multitude of sound sources as that also made it easier to detect the added jitter.

Returning to the study from 1974 by Manson [16], it set the limit at which jitter could only be heard by less than 5 % of the listening audience for sinusoidal jitter with frequency above 2 kHz to 35 ns, and for random jitter the same limit was determined to be 50 ns. For sinusoidal jitter with frequency lower than 2 kHz, the

tolerance threshold proposed by Manson increases linearly as the frequency of the jitter is lowered. Test tracks consisting of piano and glockenspiel were selected to provide the most critical material out of a range of tracks auditioned by experienced listeners. All tests were conducted in one room using the same speaker system and the test participants were all described as having previous experience of assessing sound quality. During the listening tests which were carried out one person at a time, the test subject was allowed to control the listening level and was also given a control from which the jitter level in the recording could be adjusted and then another one by which the addition of jitter could be turned off completely. The test subject was then asked to find the threshold level for jitter audibility using the controls available. Jitter was added to the audio signal by passing it through two sample-and-hold units and then reclocking it in the second one by applying a control signal which perturbed the clock signal to simulate both random and sinusoidal jitter depending on the setting. Low-pass filters were also added before and after the described jitter addition circuit to comply with the sampling theorem. With the study being conducted nearly 50 years ago, tape recordings were used as source material and playback was done in monophonic audio. Surely some advances in both recording and playback technology have been made since, but whether using stereo playback instead of mono would make the jitter audibility threshold limits any lower is debatable. Small variations in timing in the microsecond range between what the left and right ear registers can be picked up by the hearing system to provide spatial information [17]. Given that the added jitter necessarily does not affect both channels equally, any disturbance caused by the jitter could possibly be picked up more easily if the audio was to be played back in stereo. On the other hand the jitter threshold levels found were way below the microsecond range and any added complexity in the source material makes it more difficult to distinguish a track with added jitter from the original, in which case adding an extra channel possibly could have made the recorded audibility threshold for jitter even higher.

In 1998 Benjamin and Gannon [14] also conducted a study where they performed listening tests in order to try to determine the jitter audibility threshold. As the audibility of jitter was found to greatly depend on the dynamic variation in the frequency spectrum of the examined audio, a lot of effort was put into finding source material where the effects of jitter would be easy to hear. Based on the criteria of having plenty of frequency content at 1 kHz or above, minimal frequency content between 400 Hz and 1 kHz, long sustain and low noise floor, this resulted in the majority of test tracks consisting of one note from a single instrument.

During the initial phase of creating the listening tests it was discovered that there was a learning effect taking place where the person being subjected to the jitter audibility testing up to a certain degree was able to increase their ability to hear the effects of jitter, thus lowering the jitter audibility threshold in successive tests. A learning phase was therefore added for all test participants prior to the listening tests used to determine the jitter audibility threshold in order to let the test subject to get familiar with the source material, controls, and test procedure to not have the threshold value decrease while the real tests were being carried out. Any intended test participants who had severe difficulties distinguishing the distortion caused by

jitter during the training phase were excluded from the final testing.

After the training phase, testing began with solitary sine wave tones of frequencies 4 kHz, 8 kHz and 20 kHz as source material to which sinusoidal jitter then was added. The jitter level was at first increased and the test subjects were asked to indicate when they were able to hear the resulting distortion. Then the jitter level began to slowly decrease until the test subject indicated that they were no longer able to hear the distortion caused by the jitter. The process was then repeated a couple of more times for all three sine wave frequencies and the top, bottom and calculated average level was recorded for each participant. Table 2.1 lists the range of calculated average threshold levels for the test participants.

Audio frequency	Jitter frequency	Jitter audibility threshold
4 kHz	2 kHz	40 ns to 150 ns
8 kHz	5 kHz	5 ns to 25 ns
20 kHz	17 kHz	7 ns to 14 ns

Table 2.1: Range of calculated jitter audibility thresholds for all test participants when playing sine wave tones with added sinusoidal jitter.

In the next part of the listening test, the earlier selected audio source material was played back to the test participant. Now given access to control the level of sinusoidal jitter added to the source material as well as having the ability to switch between the audio signal with added jitter and one with without at will, the test participant was asked to adjust the controls until the threshold level for jitter distortion audibility had been reached. Table 2.2 shows the recorded threshold ranges for the participants for each test track.

Test track	Jitter frequency	Jitter audibility threshold
1: One note, single instrument	1.70 kHz	50 ns to 270 ns
2: One note, single instrument	1.85 kHz	32.5 ns to 110 ns
3: One note, single instrument	1.70 kHz	20 ns to 310 ns
4: Synthesized music recording	1.53 kHz	112 ns to 370 ns*

**Not all test participants were able to find an audibility threshold for track 4.*

Table 2.2: Range of jitter audibility thresholds recorded for all test participants when playing program material with added sinusoidal jitter.

The audibility threshold for the jitter added to the higher frequency sine waves was slightly lower than it was for any of the other more regular program material and the results for the program material can be considered to be on par with what Manson [16] found for sinusoidal jitter added to the selected program material in his study. The same audio equipment was used for all the listening tests and a set of headphones instead of speakers were selected to reproduce the audio recordings in the study. The jitter was added to the source material by running the signal

through a jitter modulator to which a function generator was connected, through which the jitter level could be controlled. Measurements on the audio system used in the listening experiments indicated that the jitter levels inherent in the system itself were way below any of the audibility thresholds recorded during testing and should not have any influence on the test results according to the authors.

A third study including jitter audibility testing was also conducted by Ashihara et al. [15] in 2005. In it, random jitter was simulated in software by creating new sample values by interpolation after which the interpolated values were shifted to the ideal sampling points in time. An anti-aliasing filter was also added to make sure the sampling theorem was still satisfied. The test subjects, all consisting of people with backgrounds in different audio fields were asked to audition source material of their own selection using their own audio equipment. Only a computer with a digital audio interface was provided as signal source and three controls, A, B and X were given from which the playback of the source material could be controlled. Selecting X always set the original source material without any added jitter to be played back. One of the controls A and B was randomly set to also select the original non-jittered source material while the other control selected the source material with the added random jitter. The test subject was informed of this setup, asked to listen to the selected source material for a couple of minutes and then at the end decide which one of the controls A and B played back the same version as control X.

The test started with plenty of jitter being added to the source material and the test was run multiple times under the same conditions. The test subject was allowed to proceed to the next step where the jitter level was halved once 75% or more of the attempts were correct. If too many incorrect answers were given, the test was aborted and the final successfully determined jitter level was recorded. Table 2.3 shows the results from the listening test. None of the test subjects were able to audibly distinguish the next level of random jitter after 500 nanoseconds. The recorded jitter audibility threshold level being a bit higher in this study than in the ones previously presented does however not come as a big surprise as only random jitter was used and more importantly, the program material in the previous studies was tailored to maximize the audible effects of jitter while more “normal” music likely was used here as the participants were allowed to pick their own listening material.

Random jitter	Audibility among test participants
2 µs	100 %
1 µs	48 %
500 ns	26 %
250 ns	0 %

Table 2.3: Proportion of test participants that were able to hear the effects of random jitter added to self selected source material.

In all three studies mentioned so far, listening experiments were used to determine the threshold limits for jitter audibility. The lowest recorded threshold values from

any of the studies were in the single digit nanosecond range, noted when sine waves of high frequency were used as program material. Using recordings of solitary real instruments as program material increases the threshold to tens of nanoseconds and using even more varied and complex source material sends the threshold limit into the range of hundreds of nanoseconds.

2.1.5.1 A Theoretical Jitter Audibility Model

An idea commonly presented is that jitter levels below the quantization noise floor will be inaudible [14, 15, 18]. In Figure 2.14, quantization with low resolution has been used to show an example of an analog signal and its quantized digital counterpart. The horizontal grid lines indicate the digital levels available that can be assumed and the vertical grid lines mark the sampling interval points. Any real audio application is likely to use a much higher resolution to produce a smoother digitalized waveform that more closely resembles the analog signal, but even then, there will still be a least significant bit (LSb) size in the digital representation of the audio signal that together with other system parameters sets the level of the noise floor. For a DAC with a resolution of N bits, the total number of values that can be represented is 2^N and the LSb is $1/2^N$ of the total representable range.

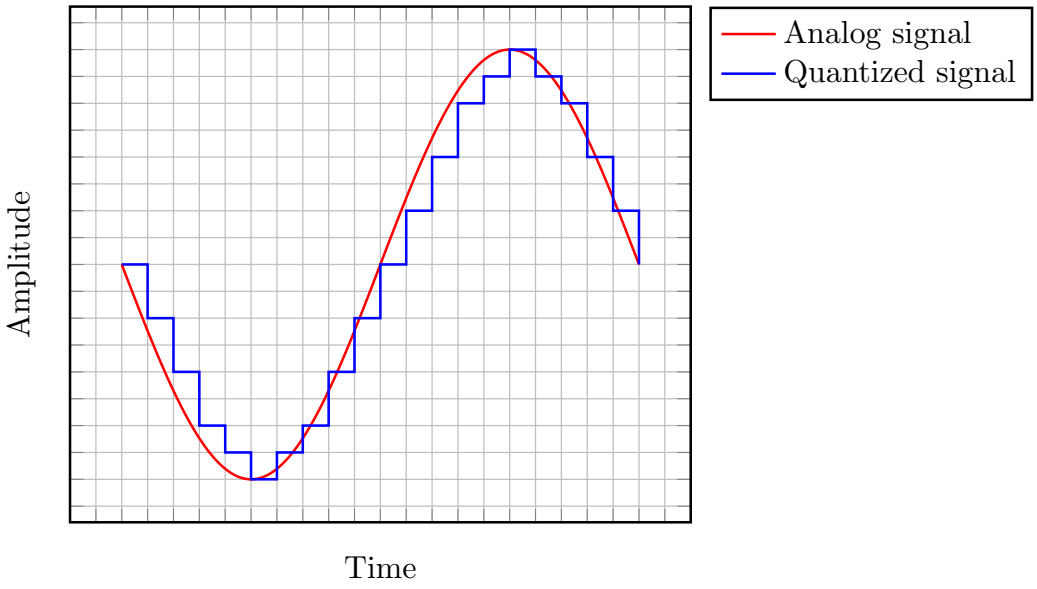


Figure 2.14: Analog sine wave and the resulting waveform after quantization in low resolution.

To find a relation between all the system parameters and the quantization noise floor, we can start by considering a sine wave:

$$y(t) = Asin(2\pi ft). \tag{2.8}$$

The rate of change for the curve is

$$\frac{dy(t)}{dt} = 2\pi fAcos(2\pi ft). \tag{2.9}$$

At $t = 0$ the rate of change will have its maximum value as the slope for the sine wave will be the steepest there. The limit $\lim_{t \to 0} cos(2\pi ft) = 1$, and we are left with

$$\left(\frac{dy}{dt}\right)_{max} = 2\pi fA. \tag{2.10}$$

If the full range of a converter is used to present a sine wave of amplitude A, then the interval from 0 up to the highest representable digital level will be of magnitude $2A$. We know that the LSb is $1/2^N$ of the total representable range, so we multiply the expression with $2A$. For any signal, the rate of change multiplied by the amount of time during which the change occurs will determine the new level of the signal. The LSb can therefore also be expressed as the the rate of change dy/dt multiplied by the amount of time t_j it takes to reach the new level when the rate of change is at its maximum. Rearranging the first expression a bit we then have

$$t_j \frac{dy}{dt} = LSb = \frac{A}{2^{N-1}}. \tag{2.11}$$

Substitution in Equation 2.10 with Equation 2.11 gives us

$$\frac{A}{t_j 2^{N-1}} = 2\pi fA. \tag{2.12}$$

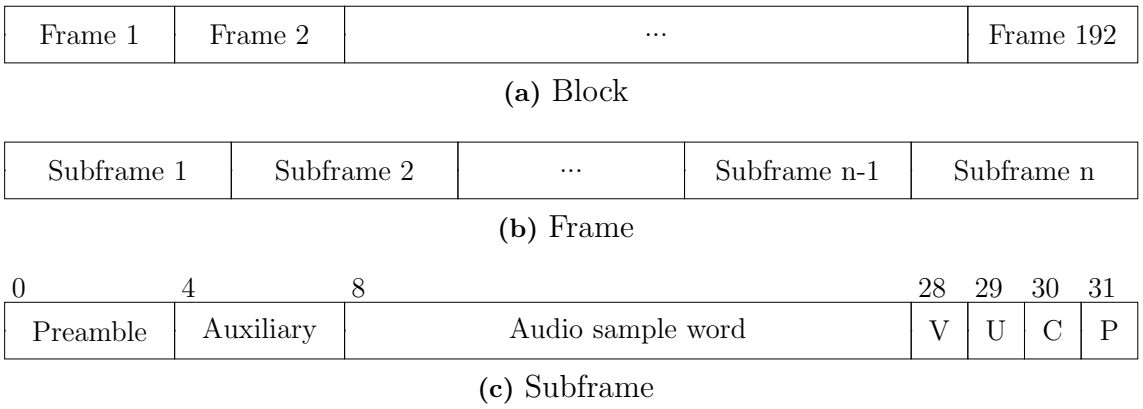
After rearranging Equation 2.12 we have an expression for t_j which corresponds to the time of one LSb:

$$t_j = \frac{1}{2\pi f 2^{N-1}}. \tag{2.13}$$

For any timing jitter to fall below the quantization level floor, it would need to be equal to less than half a LSb, so any result to Equation 2.13 needs to also be divided by a factor of 2. For a 16-bit converter with maximum sampling frequency of 20 kHz, the jitter level should therefore be lower than $t_j/2 = 121$ ps for it to fall below the quantization noise floor and be inaudible.

2.2 AES/EBU and S/PDIF

The physical appearance and the electrical characteristics of the two interfaces are different as the for professional use intended AES/EBU [5] interface uses balanced XLR connectors while the more consumer oriented S/PDIF [6] uses either a coaxial cable with RCA connectors or an optical wire with TOSLINK connectors, but beneath the dissimilar exterior they both use the same data transfer protocol. The audio data and the clock signal are transferred along the same data line, combined into one bit stream using biphase mark code (BMC). Data is transferred in blocks consisting of 192 frames, every frame is divided into multiple subframes, one for each audio channel, and each subframe contains 32 bits of data. Figure 2.15 shows the structure of the transfer scheme including the placement of the data fields inside a subframe.



(a) Block

(b) Frame

(c) Subframe

Figure 2.15: Data structure of AES/EBU and S/PDIF. The contents of a) an audio block, b) a frame and c) a subframe.

Bit no.	Subframe field	Usage
0–3	Preamble	Indicates the start of a subframe. The field specifies if it is a) the first subframe within a frame, b) the first subframe in a block, c) any other subframe.
4–7	Auxiliary bits	Used to add auxiliary information or can be assigned to carry an extra 4 bits of audio data, extending the audio sample word size from 20 to 24 bits.
8–27	Audio sample word	The bits used to carry the audio data sent with LSb first.
28	Validity bit	Indicates if the audio sample word contains valid audio data or not.
29	User data bit	Can be used to carry any user defined data.
30	Channel status bit	Used to indicate type of interface, sample rate, copy permission and other settings. The meaning of the bit is dependent on the frame number within the block in which it is transmitted. All subframes within a frame carry the same channel status bit.
31	Parity bit	Used to detect errors in the transmitted data.

Table 2.4: Description of subframe data fields.

2.2.1 Biphase Mark Code

The clock signal and the audio data together with all other parts of the subframe are for the AES/EBU and S/PDIF interfaces sent along the same data line using biphase mark encoding [19], also known as Differential Manchester encoding. For

each bit of data that is sent, at least one signal transition is guaranteed to happen. If the data bit is a “0”, the encoded signal sent to the receiver will switch polarity once at the start of the time slot, and if the data bit is a “1”, the signal will change polarity twice, once at the start and once in the middle of the time slot for that bit. The only part of the subframe data that is allowed to violate this condition is the first part containing the preamble bits. It only has three valid bit sequences used for indicating the placement of the subframes in the data stream. The reason for this is to ensure so that no other data in the subframe can contain the same bit sequence as the preamble, throwing off the synchronization of the data that is being transmitted. An example of BMC is shown if Figure 2.16.

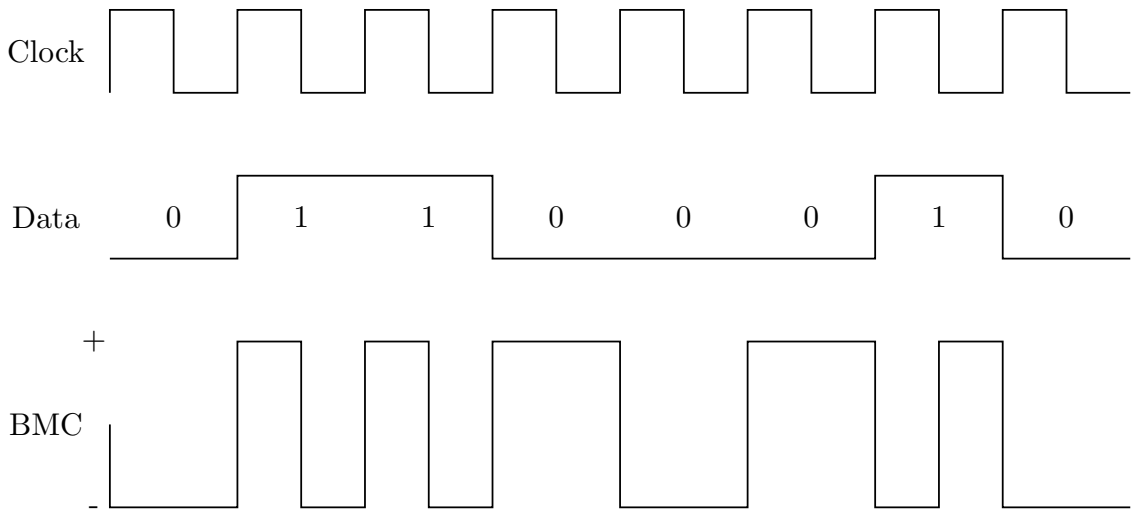


Figure 2.16: Biphase mark code timing diagram.

2.2.2 Clock Recovery

After data has been transmitted, the receiver will need to recover the clock signal and separate it from the rest of the encoded data and there are some problems that can arise in the recovery process. A study by Dunn and Hawksford [2] done relatively soon after AES/EBU and S/PDIF were started being used widely points out many of the known problems with the interfaces’ characteristics; the most significant one being that jitter can be dependent on the data pattern in the transmission. Due to bandwidth limitations, the rise and fall times of the transmitted signal will be limited. This can cause the signal to begin the transition from high to low or vice versa from a voltage level depending on the previous data pattern as the signal has not had time to settle from the previous transition due to the limited bandwidth. Dunn and Hawksford [2] created a simulation model of a bandwidth limited transmission channel using a passive low pass filter, which despite its simplicity showed a generally good agreement with measurements conducted on real systems.

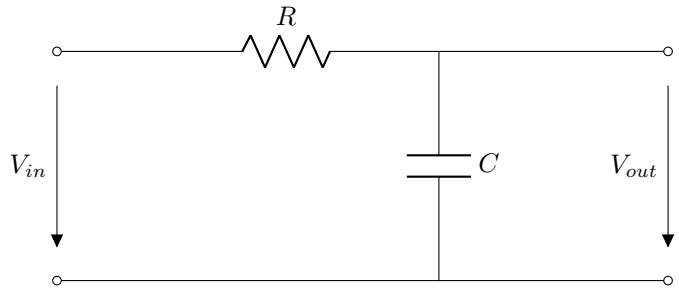


Figure 2.17: First order passive low pass filter used to simulate bandwidth limited transmission channel.

The voltage V_{out} over the capacitor in Figure 2.17 is

$$V_{out} = V_{in}\left(1 - e^{-\frac{t}{RC}}\right) + V_0 e^{-\frac{t}{RC}}, \tag{2.14}$$

where V_{in} is the amplitude of the non-filtered signal and V_0 is the voltage level from which the signal transition begins. Using the filter to simulate a bandwidth limited transmission line with time constant $\tau = RC$ of 100 ns between a S/PDIF transmitter and receiver gives the graph in Figure 2.18. Looking just at the first eight bits of the subframe in Figure 2.19, we can more clearly see that for each signal transition the starting voltage is different and that it depends on previous signal transitions. The consequence of this is that the time when the threshold level at 0 V will be reached at a signal transition will vary depending on the previous bit pattern and as we have seen before, a shift in voltage level can create timing jitter. This is shown in Figure 2.20 where bits four and five of the subframe are displayed and the difference between the signal edge and the bandwidth limited signal is inconsistent from signal transition to signal transition. The threshold level 0 V crossing time is given by the equation

$$t = RC \ln\left(1 + \left|\frac{V_0}{V_{in}}\right|\right). \tag{2.15}$$

Solutions to help lessen this issue with data dependent jitter include using data patterns in the auxiliary bits and user bits which are less prone to creating jitter [2] and to only use the first bits in the preamble to lock on to the signal and to create a local clock from only that bit sequence instead of using every transition in the whole subframe to generate it [20].

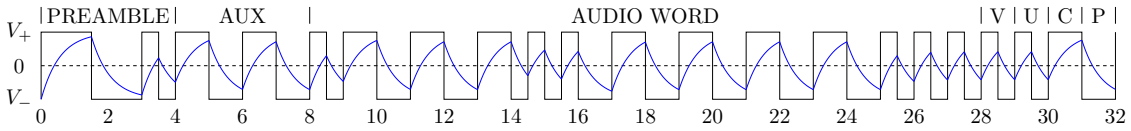


Figure 2.18: Transmission of one subframe over bandwidth limited channel.

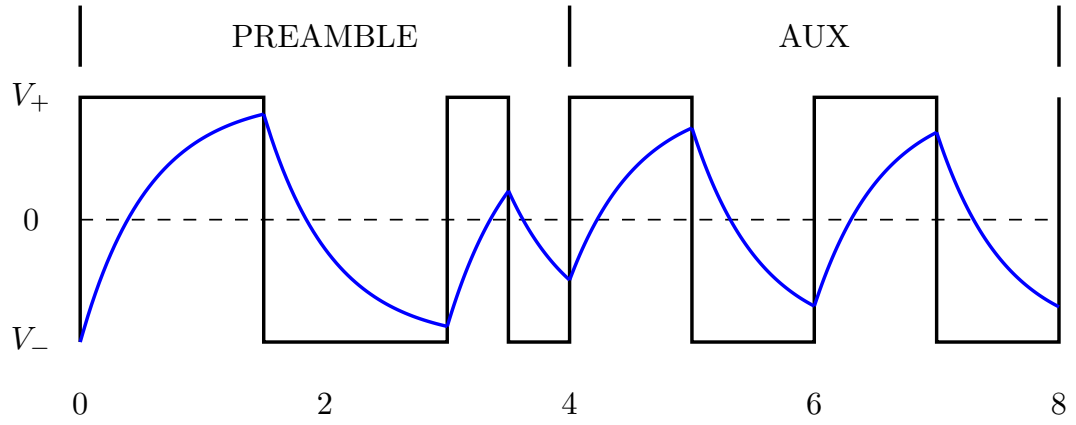


Figure 2.19: First eight bits of the subframe.

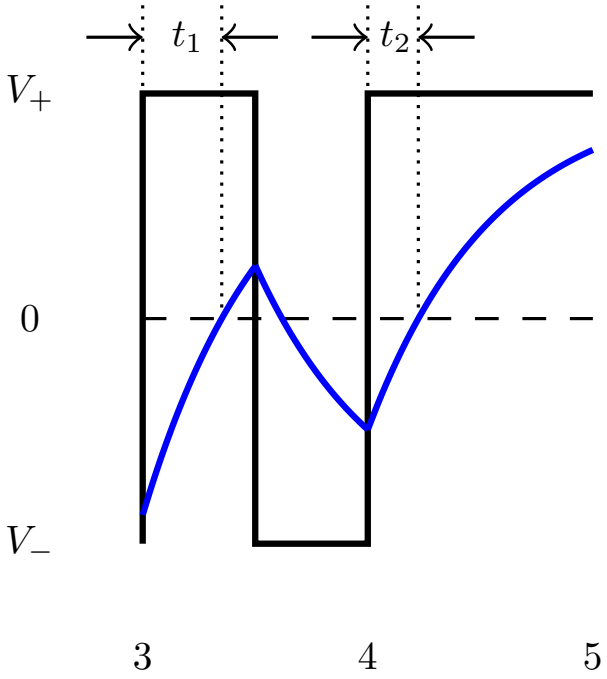


Figure 2.20: Bits four and five of the subframe.

2.2.3 Asymmetric Slew Rates

Another thing that can cause jitter in AES/EBU and S/PDIF interfaces is slew rate imbalance, giving asymmetric rise and fall times for the signal. Dunn and Hawksford [2] presented the formula shown in Equation 2.16 for the amount of jitter t_j per signal transition that is created by an asymmetry in the slew rates. V_d is the driving voltage of the transmitter, V_{SR+} is the slew rate in the positive direction and V_{SR-} is the slew rate in the negative direction.

$$t_j = \frac{|V_d|}{2} \left| \frac{1}{|V_{SR+}|} - \frac{1}{|V_{SR-}|} \right| \tag{2.16}$$

A visualization of a signal response with symmetric versus asymmetric slew rates is shown in Figure 2.21. The slew rate limited response with symmetric rise and fall times will cross the threshold level at 0 V at times τ_1 and τ_3 instead of at τ_0 and τ_2 like the ideal square wave does, but the time difference $\tau_3 - \tau_1 = \tau_2 - \tau_0$, so this does not present a problem. The falling signal edge for the slew rate limited response with asymmetric slew rates will on the other hand cross the threshold level at τ_4 instead of at τ_3 as the falling edge only has half the rate of change per time unit compared to the rise time and $\tau_4 - \tau_1 \neq \tau_2 - \tau_0$, so the slew rate asymmetry introduces jitter into the signal. One solution suggested to solve the issue is to let the receiver rely on signal transitions in one direction only, effectively removing the need for the slew rates to be even.

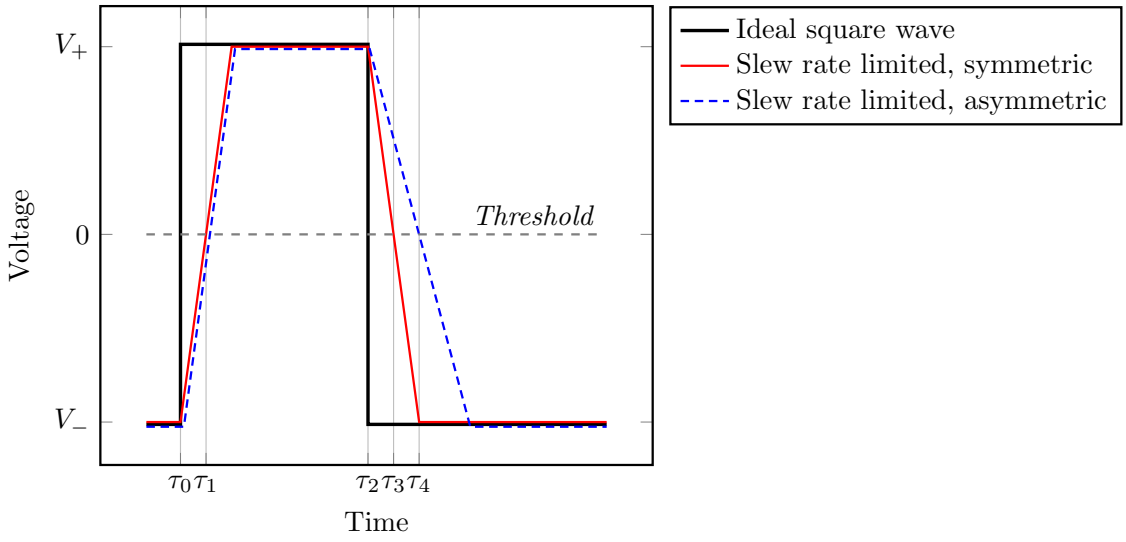


Figure 2.21: Symmetric versus asymmetric slew rate response to an ideal square wave.

2.2.4 Transmission Lines

A model for a transmission line used for high frequency signals [1, 19, 21] is shown in Figure 2.22. The parameters, resistance R, inductance L, capacitance C and conductance G are per length unit of transmission line. For a S/PDIF interface using a coaxial cable to transfer audio, the same model can be applied.

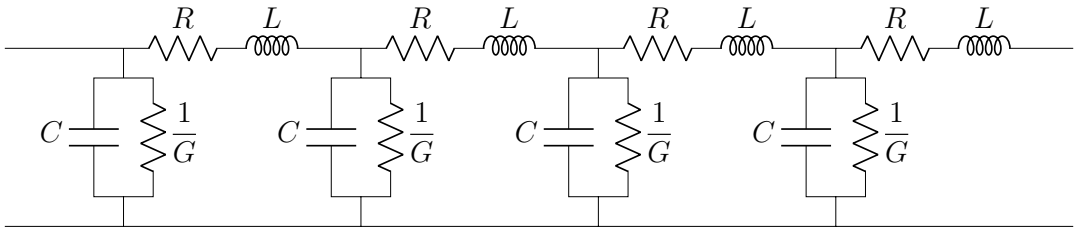


Figure 2.22: Cascaded network model for high frequency transmission line.

The transmission line has a characteristic impedance of

$$Z_0 = \sqrt{\frac{R + sL}{G + sC}} \tag{2.17}$$

with s being the frequency operator, for a sine wave often denoted by $j\omega$. Let us now attach a load with impedance Z_L to the transmission line and then apply a voltage pulse of size V_i to the other end of the line as depicted in Figure 2.23. What then happens when the voltage reaches the load depends on the impedance Z_L of the load. For a perfectly matched system where the transmission line impedance Z_0 is equal to the load impedance Z_L, the whole voltage pulse V_i will continue into the load, but if the impedances differ, a part of the voltage will be reflected back along the transmission line. Equation 2.18 gives the reflection coefficient ρ of the system.

$$\rho = \frac{V_{reflected}}{V_{incident}} = \frac{Z_L - Z_0}{Z_L + Z_0} \tag{2.18}$$

When the voltage pulse V_i in Figure 2.23 is applied, it will start to move along the transmission line towards the load with the propagation velocity

$$v = \frac{c}{\sqrt{\epsilon_r \mu_r}}, \tag{2.19}$$

where c is the speed of light, ϵ_r is the permittivity and μ_r is the permeability for the transmission line. Equation 2.18 gives the ratio between the incident voltage and the reflected voltage. If for example the load impedance Z_L and the transmission line impedance Z_0 are severely mismatched with the load impedance Z_L being twice the size of the transmission line impedance Z_0, then the reflection coefficient ρ is 0.33 and the amplitude of the reflected voltage pulse is one third of the amplitude of the applied voltage V_i. For a more tightly matched system where the transmission line impedance Z_0 and the load impedance Z_L only differ by 1 %, the amplitude of the reflected voltage pulse at the load end will still be around 5 mV per 1 V of applied voltage V_i. Figure 2.24 shows the voltage on the transmission line before it has reached the load and after a part of it has been reflected.

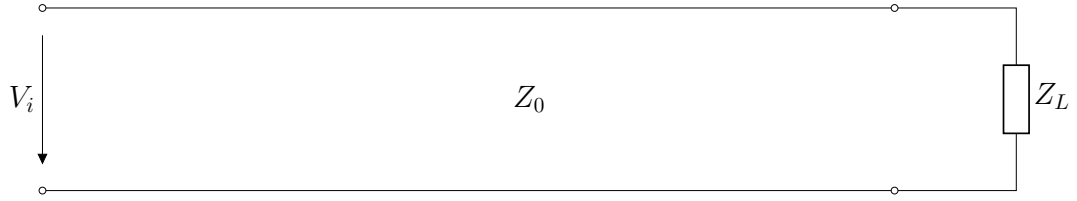
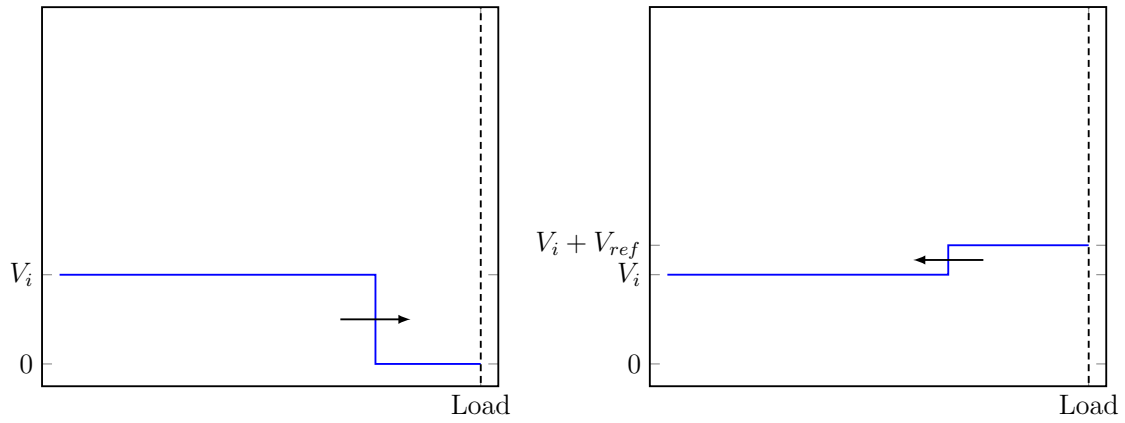


Figure 2.23: Transmission line with load attached.



(a) Approaching the load. **(b)** Reflected voltage at the load.

Figure 2.24: High frequency voltage pulse applied to a transmission line.

The transmitting side of the system where the voltage pulse V_i originates from does also have an impedance of its own and the same reasoning applies to that end of the circuit, so if there is an impedance mismatch between the transmitting side and the transmission line, then a part of the voltage pulse first reflected at the load end will be reflected once again when it reaches the transmitting side. A pulse can therefore be reflected several times between the transmitter and receiver sides if both ends have an impedance different from the transmission line impedance Z_0. In reality, any such voltage pulse bouncing back and forth is likely to diminish quickly as $|\rho| < 1$ for any case except a completely open or fully shorted circuit end. We know from Chapter 2.1.2 that voltage noise can lead to timing jitter, so even small reflections due to impedance mismatching between the transmission line and the load on high frequency transmission lines, in our case the coaxial cable connecting the transmitter and the receiver and the transmitter and receiver units themselves, can cause issues. Proper impedance matching in the audio chain between the transmitter, receiver and the cable connecting them is therefore necessary.

S/PDIF and AES/EBU signals being transmitted in coaxial cables are also affected by other attributes of the transmission channel apart from the impedance. Dielectric losses and the skin effect where a high frequency signal travels mainly along the surface of the conductor only penetrating a short distance into the core of it are some examples of things that could be expected to affect the voltage level and rise times of the signal. Both are dependent on the material parameters of the cable and on the frequency of the signal being transmitted, but as the frequency for the clock signal being sent is expected to stay the same, then all signal transmissions should be affected to an equal extent in which case no new variable jitter would be added to the signal. Optical channels also have their own share of issues that could be expected to affect a signal propagating through the optical wire such as pulse dispersion and limited bandwidth in the transmitter and received components but we will not go any further into if and how that might impact a digital audio signal being transmitted.

2.2.5 FIFO Buffers

One attempt at solving the interface jitter issues of AES/EBU and S/PDIF has been to insert a first in, first out (FIFO) buffer between the receiver chip and the DAC in the converter and to then reclock the data coming out of the FIFO buffer. It has been used by some audio manufacturing companies, but there are some drawbacks to this method. The introduction of a buffer in the audio chain will undoubtedly delay the audio signal. This might be acceptable up to certain degree if the audio only is used for music playback, but if the transmitted audio stems from a video stream, then the audio and video can become noticeably out of sync unless the buffer is small. A delay could also cause problems if the audio system would be used for communication in a telephone type of manner, as that would make the communication disruptive and less smooth.

The purpose of the added FIFO is to reclock the signal with a clock that has less jitter than the one arriving from the transmitter, so two clocks, the one supplied by the transmitter feeding the audio data into the FIFO and a second one supplied by the receiver moving data out of the FIFO will be running freely, not synchronized to each other. The clocks will essentially be running at the same rate but any difference or variation at all in the clock rates will make the clocks start drifting apart and this must be remedied by having a large enough buffer size to accommodate for the drift between the clocks so that the FIFO buffer does not underrun or overflow. If we have a system with 44.1 kHz sample rate, then one new audio sample will arrive every 22.68 µs for each audio channel. A normal oscillator (XO) like for example the one used to generate the external clock to our DAC in Figure 3.12 can have a frequency stability rating of ±100 parts-per-million (ppm), which means that the clock could in the worst case be off by one in every 10 000 samples compared to an ideal clock. If we have two clocks with the same frequency stability rating running side by side where both clocks have maximum deviation from the ideal frequency but in opposite directions, then the sample rate could be off by up to 8.82 samples per second. In an hour that amounts to 31 752 samples, so if we fill the FIFO buffer up half-way before we start extracting data from it, then it would need to be able to fit 63 504 samples for each audio channel to guarantee uninterrupted playback for one whole hour. The delay caused by the FIFO buffer would in that case be 0.72 s at the start of playback. While not ideal, this could be acceptable for audio playback, but in other applications such as video streaming, the delay between the video and the audio would just be too big unless otherwise adjusted.

Another option that has been tried together with a FIFO buffer is to use an asynchronous sample rate converter (ASRC). The average incoming data rate is first measured and then the audio data signal is resampled by the ASRC to match the rate of the clock which extracts the data from the FIFO buffer and hands it over to the DAC. In this way the buffer will not need to be so large as the ASRC will adjust the audio samples by interpolation so that the average data rate for the audio data going into the FIFO is the same as the date rate coming out of it, and the buffer will therefore not overflow nor underrun even though the clock rates at the input and output of the buffer might be slightly different. The use of an ASRC could

however give other undesirable audible effects depending on how well it has been implemented. Including not only a FIFO but also an ASRC in the design also adds to the complexity of the device.

2.3 Universal Serial Bus

The next section in this chapter will mainly focus on the parts of USB which are of relevance for the thesis like the transfer protocol structure, the device descriptors, the isochronous transfer modes and other audio and timing related subjects. With there being multiple versions of the USB specification, Universal Serial Bus specification revision 2.0 [22] is the version that the coming sections will comply with; this simply because it is the most appropriate version considering the hardware that will be used in the construction part of the project. In the specification there are several attributes declared that make USB suitable to be used as a dedicated audio interface. Among other things, guaranteed bandwidth and low latency for audio are listed as key points. The implementations in the hardware construction part of the thesis project are done using a "full-speed" device as defined by the USB 2.0 specification. Subsequent revisions of the specification [23, 24] supporting SuperSpeed devices do introduce some new concepts, but they are of no use for us here. Time units are for example handled differently. Full backward compatibility to USB 2.0 is however guaranteed for any full- or high-speed device connected to a host port using SuperSpeed. Whenever mentioning the USB specification going forward, revision 2.0 is what is being intended unless explicitly stated otherwise.

2.3.1 Network Topology

A USB system is controlled by a single host, polling the bus to which devices are connected. Devices can be grouped into different classes, such as for example the audio device class. The communication channels between the host and a device are called pipes, they can carry messages or stream data, and they are connected to endpoints at the device and at the host. At a minimum a device must implement at least one bidirectional message pipe called the default control pipe, which is connected to the control endpoints of the device. Capabilities added to a USB system for example in the form of an audio interface are called functions, so from our point of view we can use the terminology for device and function interchangeably.

The network topology of USB has a tree-like structure. At one end, there is the controlling host at which the root hub resides. Devices or other hubs can be connected to the root hub, and in extension more devices and hubs can be connected to a hub which is connected to the root hub as displayed in Figure 2.25. In order to not violate the specifications set for timing, no more than six additional levels following the root hub layer can be connected together.

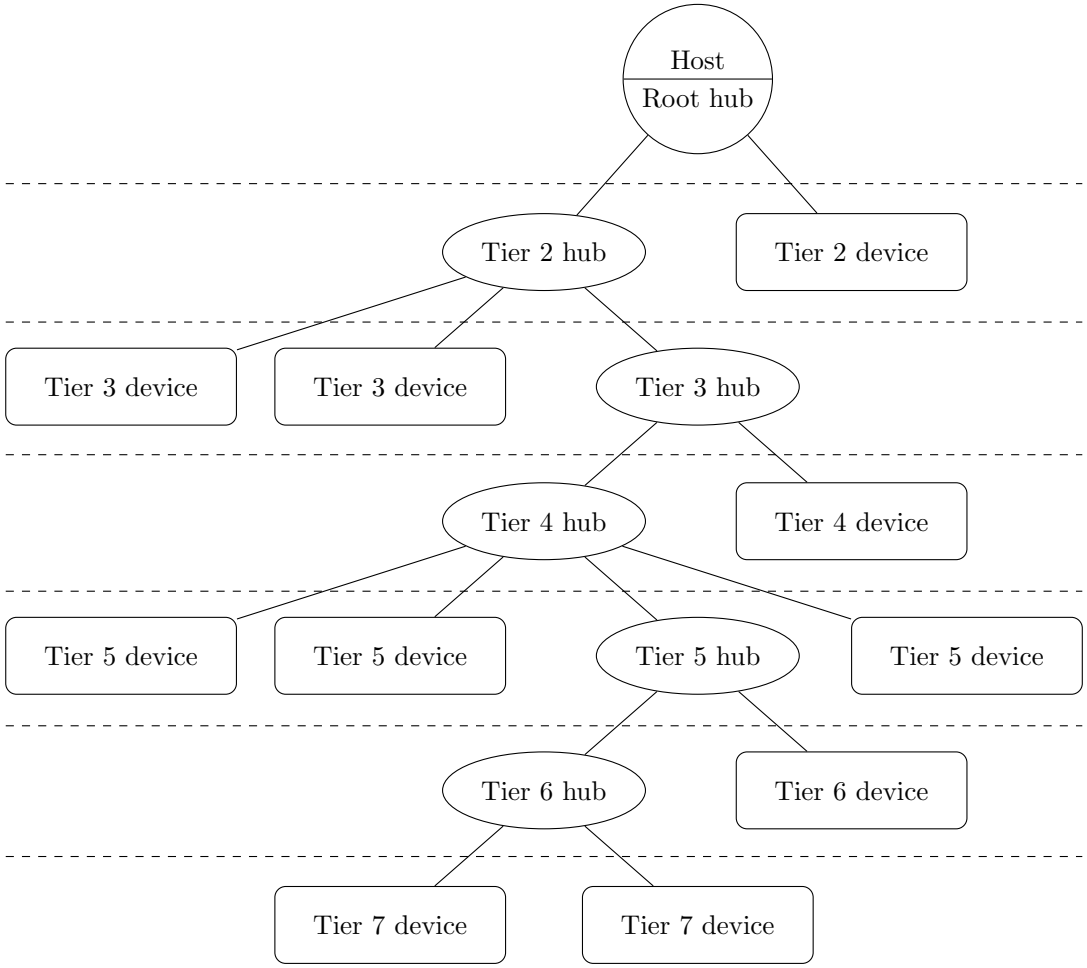


Figure 2.25: USB topology.

2.3.2 Connecting a Device to the Bus

When a device is connected to the bus, the host will need to discover and configure it before it can perform any function. The process of configuring and enabling the device is called enumeration, and it is done by performing a number of steps through which the device state is altered until configuration has completed. At first, the hub to which the device has been connected will set the device to the powered state and report to the host that its status has changed. This will cause a query to be sent from the host to the hub to find out what caused the change. Once the host knows that a device has been attached, it will send a reset command and have the port to which the device is attached to set to enabled. After the device has been reset, it will go into the default state during which the host can communicate with it using the default address. Following steps in the configuration process will assign a unique address to the device, causing it to go into the address state, and finally into the configured state once the host has read all the configuration information in the device's descriptor table and has assigned a configuration value to it. In the configured state, all endpoints described in the device's descriptor table have been enabled and the device is ready for use.

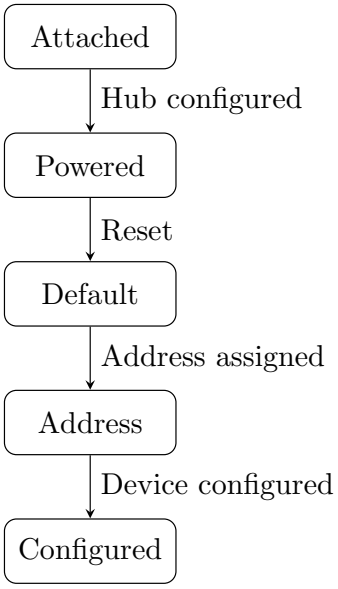


Figure 2.26: Device state changes during enumeration process.

2.3.3 Descriptors

A device reports its capabilities and settings to the host upon request through its descriptors. The host can also change some of the device settings by altering the values in the device's descriptor table by device requests. A device has exactly one main device descriptor that contains general information about the device and it also lists one or more possible configurations of the device in the underlying configuration descriptors. A configuration descriptor will in turn list one or more interface descriptors, and each interface descriptor will then list one or more endpoint descriptors. When a configuration descriptor is requested by the host, it will be returned by the device accompanied by any underlying interface and endpoint descriptors. Interface and endpoint descriptors cannot be requested on their own. Alternative settings for the interfaces may be provided by having multiple configuration descriptors. The default control endpoint is not listed among the endpoint descriptors as it must be implemented by all USB devices as a control pipe with predefined settings. Endpoint descriptors declare among other things the direction of the endpoint, if it is a control, isochronous, bulk or interrupt endpoint and what type of synchronization it uses. Endpoints are unidirectional, but two endpoints with the same endpoint number can be created with opposite data directions. A feedback endpoint associated to a single isochronous endpoint is expected to have the same endpoint number as the isochronous endpoint, and in case multiple endpoints are using the same feedback endpoint, then the endpoint number used for feedback should be the same as for the isochronous endpoint with the lowest endpoint number associated to it. A high-speed device can also have a device_qualifier descriptor and an other_speed_configuration descriptor. The device_qualifier descriptor is similar to the device descriptor, but instead of providing information about the device for the current speed setting, it will show device information for the alternative speed setting. Requests for the device_qualifier descriptor will there-

fore for a device running in high-speed return the full-speed information and for a device running in full-speed it will be the other way around. In the same way the other_speed_configuration descriptor will return a configuration descriptor of the alternative speed setting that the device is not currently using. An optional string descriptor can be included to provide information in readable Unicode text format for all devices. Figure 2.27 shows the overall structure of the descriptor table and example descriptors for the audio device implementations used in the construction build are provided in Appendix A.

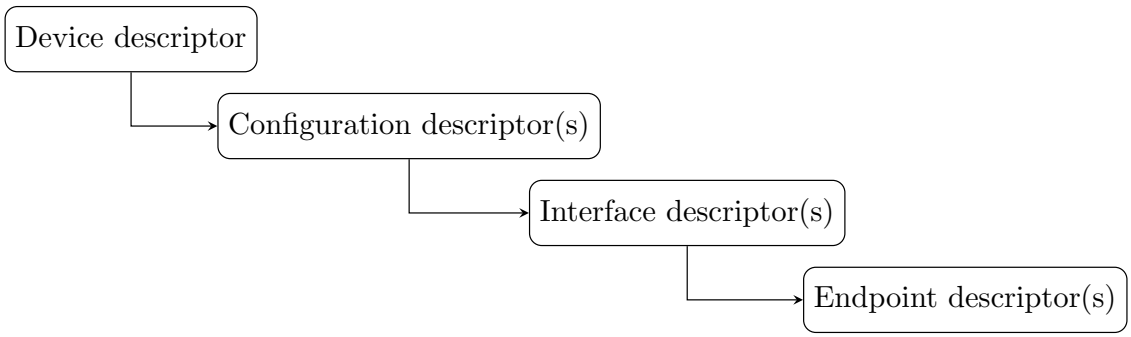


Figure 2.27: Configuration, interface and endpoint descriptor structure.

2.3.4 Device Classes and Device Requests

The descriptors on the device are made accessible to the host by replies to device requests sent from the host to the default control pipe of the device. Device requests can be of standard type, which are supported by all devices, or they can be class or vendor specific. Device requests are used to either fetch values from a device descriptor or to manipulate the values in them. The prefixes “GET” and “SET” are used in the request name to indicate if a request is meant to retrieve or change the descriptor data that is being referenced. The only standard device requests that do not use the two mentioned prefixes are the CLEAR_FEATURE request, which is used to switch off on-off toggle values and the SYNCH_FRAME request, which is used to synchronize the host and the device when the size of the transferred data varies within a frame. A device request transaction will follow the pattern for a control transfer with an initiating SETUP packet, an optional data packet depending on the request type, and a closing handshake like depicted in Figure 2.35 in Chapter 2.3.10.2. If a device receives an invalid or unsupported request, it should respond appropriately and signal the error by setting the packet identifier to STALL either in the following data packet or the next status transaction. The USB audio device class is an extension of the USB standard, so a USB audio class device will have a number of extra descriptors containing information about the device’s audio capabilities and it will also on top of the standard device requests support requests from the USB audio class specification. Software on the host communicating with an audio class device can use the standard audio class driver provided by the operating system, but it is also possible to load an external driver specific to the device and use that one instead of the generic driver.

2.3.5 Transfer Types

Most transactions on the bus consist of three interactions: 1) The host sends a "token packet" with parameters to set up a transaction with one of the connected devices. 2) An attempt to transfer the requested data is made, either in the direction from the device to the host or from the host to the device. 3) The receiver of the data sends a "handshake packet" to indicate if the data was transferred successfully or not.

There are four types of data transfers that can take place:

Data transfer type	Usage
Control transfer	Used for device configuration, commands and status requests.
Bulk transfer	Non-periodic transmission of non-time sensitive data, usually sent in larger chunks.
Interrupt transfer	Transmission of smaller amounts of time sensitive data that must be delivered reliably.
Isochronous transfer	Periodic transmission of real-time data with minimal delay.

Table 2.5: Data transfer types for USB.

We will not make any use of the bulk transfer mode or the interrupt transfer mode in any USB audio class devices described in this thesis and therefore no time will be spent on expanding the discussion around those subjects. Isochronous data transfer is the transfer mode used by USB audio devices to move audio data, so that is of most interest to us. Control transfers are also to some degree used by USB audio devices for supportive functionality.

2.3.5.1 Control Transfers

Control transfers allow the host software to configure and control device functions using the default control pipe of the device. Additional message pipes for control transfers used for other device specific purposes can be defined but are not obligatory. Requests to alter the device settings can be either standard, device class, or vendor specific. Error free message delivery is guaranteed for this transfer type and bus access is granted in a best effort manner. Time is reserved for control transfers on the bus, but that time reservation is shared between all the connected devices and it is not limited to a single device.

2.3.5.2 Isochronous Transfers

The characteristics of the isochronous transfer mode makes it the most suitable of the USB transfer types for transmission of data like audio which is consumed in real-time. USB guarantees periodic access to the bus for isochronous data transfers

with an upper bound on the maximum allowed latency. The latency of the transmitted data will depend on the amount of buffering that is done at each stage in the transmission chain. No retransmissions are made for any data lost in transmission errors, but the receiver can still discover if a transmission error has occurred by keeping track of start-of-frame (SOF) count, expected delivery interval, cyclic redundancy check (CRC) field of packets and if it is a high-speed high bandwidth device, then also the packet ID sequencing can be used. The number of transmission errors occurring is however expected to be low enough to not cause any problems. As a side note, a recommended bit error rate of less than or equal to 10^{-12} for a high-speed receiver is mentioned as a design guideline in the section for electrical characteristics in the USB specification.

2.3.6 Time Units

For full-speed devices, USB divides time into units of 1 ms called frames. High-speed devices are able to use a narrower time span of 125 µs called a microframe. Each new frame is defined by the host sending out a SOF packet every 1 ms ±0.5 µs that devices can use for synchronization. The same generation rate of SOFs for microframes is set to 125 µs±0.0625 µs by the USB specification.

2.3.7 Bus Access Period

A device using isochronous transfers must at the time of being connected to the bus inform the host software of its desired bus access period so that bandwidth can be allocated to accommodate the required data rate. This is done by setting an appropriate value in the *bInterval* field of the device's standard endpoint descriptor. Valid values for isochronous endpoints are between 1–16 and the formula

$$I = (2^{bInterval-1})F \tag{2.20}$$

expresses the desired polling interval in frames or microframes. F is the frequency of one frame or microframe depending on the speed of the connected device, so for a high-speed device F is 125 µs and for a full-speed device it is 1 ms. For a high-speed high bandwidth device up to three transactions can take place during one microframe, but the host may not always be able to fulfill the desired access interval of the device. Figure 2.6 shows the packet ID sequencing depending on the number of packets sent to the high-speed high bandwidth device during a microframe. By keeping track of the bit sequence in the packet ID field of the received packet, the device can detect if a packet is missing and if that is case then all data sent during that same microframe should be treated as incomplete.

Number of transactions per microframe	1st packet	2nd packet	3rd packet
1 transaction	DATA0		
2 transactions	MDATA	DATA1	
3 transactions	MDATA	MDATA	DATA2

Table 2.6: Packet ID sequencing for a high-speed high bandwidth device receiving isochronous data from host.

2.3.8 Endpoint Buffering

Before creating, configuring and allocating bandwidth to an isochronous stream pipe for a device, the USB host software will calculate the amount of time that the isochronous transactions are going to take to make sure that the needs of all devices sharing the bus can be accommodated. When data is sent through an isochronous stream pipe, it is first accumulated in a memory buffer and then transmitted in larger chunks in the form of packets. There must also be a buffer at the endpoint receiving the packets that can hold them until the device is ready to process them. As a rule of thumb, the recommendation is that the size of the buffers at both endpoints should be large enough to be able to fit twice the amount of data that can be sent during one frame for a full-speed device, or one microframe for a high-speed device. The larger the buffers are, the bigger the latency in the audio chain is. An appropriate buffer size B_{size} can be obtained by the formula

$$B_{size} = 2S \left\lceil \frac{F_s}{\frac{F_{SOF}}{I}} \right\rceil, \tag{2.21}$$

where F_s is the sample rate frequency of the system, F_{SOF} is the frequency of the USB clock, I is the polling interval from Equation 2.20 and S is the sample size of the device.

2.3.9 Prebuffering Delay

The way in which USB processes isochronous data through the buffers at each of the endpoints when it is transferred from source to sink will inherently add a delay. At the source, data will be accumulated and buffered during frame X for a full-speed endpoint until the SOF for frame X+1 is transmitted. The data in the buffer from frame X will then be sent during frame X+1 to the buffer at the sink endpoint. First when the SOF for frame X+2 appears can the sink start processing the data that was accumulated during frame X at the source. Figure 2.28 displays the buffering delay. The same applies for a high-speed endpoint, but instead of frames the time unit used is microframes.

Frame number	F_1	F_2	F_3	F_4	$\cdots$	F_x
Data accumulated at source	D_1	D_2	D_3	D_4	$\cdots$	D_x
Data sent on bus	$-$	D_1	D_2	D_3	$\cdots$	D_{x-1}
Data processed at sink	$-$	$-$	D_1	D_2	$\cdots$	D_{x-2}

Figure 2.28: Delay induced due to prebuffering at endpoints.

2.3.10 Transfer of Data

Data is sent on the bus using non return to zero inverted (NRZI) encoding with the LSb first. The polarity of the NRZI encoded signal changes for every data bit that is "zero" and remains the same for every data bit that is "one". Like biphase mark encoding used by S/PDIF and AES, NRZI has the same benefit of only having a small DC component. A separate signal line for a clock is likewise also not needed as the receiver can create the sample clock by itself. As a long series of data containing nothing but ones produces a NRZI encoded signal that has no transitions from high to low or vice versa until the next "one" in the data appears, extra bits are inserted into the NRZI encoded data to guarantee that a signal transition happens at least every 7^{th} bit. This is enough to ensure that the receiver can lock on to the signal. Any inserted extra data bits used for this purpose are discarded when received.

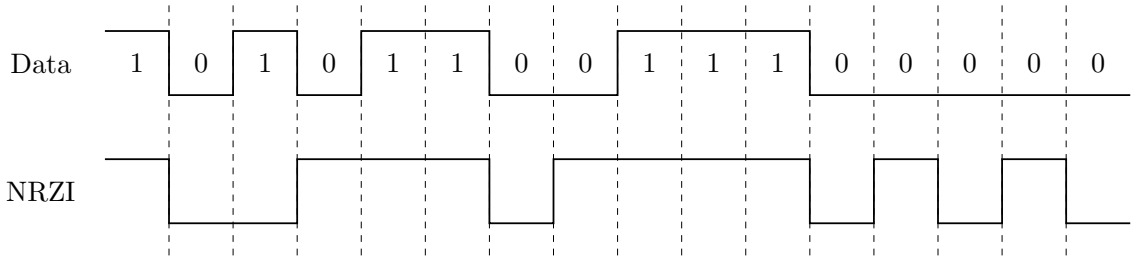


Figure 2.29: Example of NRZI encoded data.

2.3.10.1 Packet Types

Packets sent on the bus can be divided into three categories:

- Token packets
- Handshake packets
- Data packets

The following token packet types exist:

PID name	Description
IN	Request of device to host data transaction.
OUT	Request of host to device data transaction.
PING	For checking if high-speed control or bulk transfer endpoints are ready to accept more data.
PRE	Preamble to initiate low-speed transaction.
SETUP	Sent to control endpoint of device to initiate data transfer.
SOF	Indicates the start of a new frame.
SPLIT	Sent to high-speed hub to initiate or end low- or full-speed transaction.

Table 2.7: USB token packets.

The four types of handshake packets defined are:

PID name	Description
ACK	Confirmation of successfully received data.
NAK	Sent to indicate that data cannot be sent or received.
NYET	Sent to indicate that a high-speed endpoint is not ready for new data yet or that a hub has not yet completed a split transaction.
STALL	Reply to indicate that a control pipe request cannot be supported or that an endpoint has halted.

Table 2.8: USB handshake packets.

Data packets can be of type DATA0, DATA1, DATA2 or MDATA. There is not any difference between them other than their PID name which depends on the order that the packets are sent in. The DATA0 and DATA1 PIDs are alternated for every other frame sent to slow-speed endpoints, and for full-speed endpoints the usage of the different data packet PIDs are shown in Table 2.6.

2.3.10.2 Packet Fields

Every packet on the bus starts with a SYNC field of eight bits for full-speed and 32 bits for high-speed endpoints. The first bits of the field are set to create as many signal transitions as possible to enable the clock synchronization. The last two bits of the SYNC field mark the transition from the synchronization sequence to the rest of the packet. Following the SYNC field is the packet identifier field (PID). It is included for all packet types and it specifies what kind of packet it is that is being transmitted. The continuation of the packet after the PID depends on the packet type and the PID. Table 2.9 lists the packet fields defined in the USB specification.

Packet field	Size	Description
SYNC	8 or 32 bit	Synchronization of incoming data with the local clock.
PID	8 bit	Defines the packet type. The last four bits are one's complement of the first four bits to be used as error control. If the bit sequence in the first four bits does not match any of the predefined values or if any of the last four bits fail to match the values in the first part of the field, then the whole packet is disposed.
Address	6 bit	Defines the source or the destination of the packet.
Endpoint	4 bit	Defines the source or destination endpoint number to be used.
FrameNumber	11 bit	Used in SOF packets for identification. The value is incremented for each new SOF packet sent and reset once the maximum representable value has been reached.
Data	0 to 1024 byte	The field containing the requested data.
CRC	5 or 16 bit	Cyclic redundancy check for transmission errors for all fields except SYNC and PID. 5 bit length for token packets and 16 bit for data packets. Catch-all for single and double bit errors.

Table 2.9: USB packet fields.

The packets that one will typically encounter and make use of when a device with an isochronous full-speed OUT endpoint like the audio class device in the construction build is being used under normal operating conditions are the token packets with the OUT and SOF PID and the various data packet PIDs. These packet sequences are presented a bit more thoroughly in the next section. The SYNC field initiating every packet has been excluded from Figure 2.31 to 2.38.

An isochronous transaction of data from the host to a device consists of two steps; 1) First the host sends a token packet on the bus with the OUT PID, and the address and endpoint number of the device. This is received by the default control endpoint. 2) The host sends the data to the device and it is received by the isochronous OUT endpoint. Other data transfer types will often also include a third step where successful transfer of data is confirmed by returning an ACK to the sender, but that part does not exist for the isochronous transfer type due to its nature of being suited for real-time data where a failed or missing transmission rather is ignored than being sent again. An isochronous IN data transaction has the same appearance but in the first step the host sends a token packet with the IN PID instead and then the direction of the data packet in the second step is reversed.

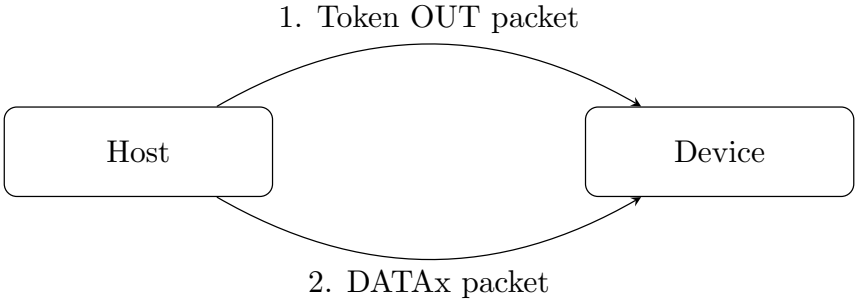


Figure 2.30: Isochronous USB OUT transaction sequence.

Figure 2.31: USB OUT token packet.

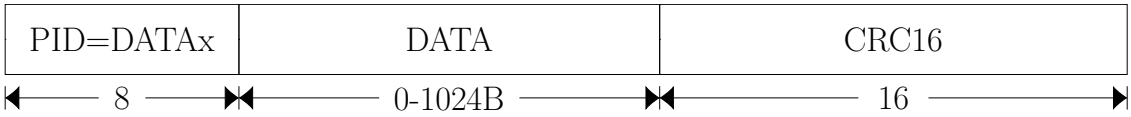


Figure 2.32: USB DATAx packet.

The start-of-frame token packets sent once every 1 ms may also be used by an isochronous audio class device for synchronization purposes and the transaction consists of a single packet sent from the host, then being received and used if needed or ignored by the full-speed devices connected to the bus. The packet is composed of the packet fields SYNC, PID, FrameNumber and CRC. The FrameNumber field is incremented for each new SOF sent every 1 ms, but the seven additional SOF microframes sent every 125 µs to high-speed devices for tighter tolerances within that same frame all use the same frame number as the first SOF packet in the frame.

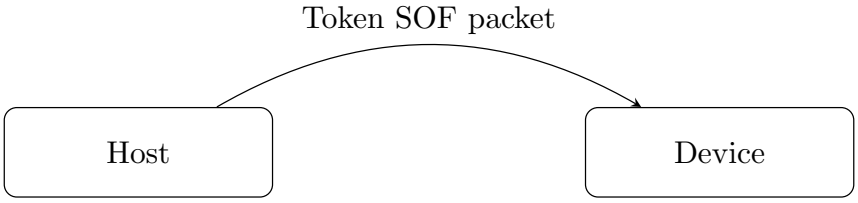


Figure 2.33: USB SOF packet transaction sequence.

Figure 2.34: USB SOF packet.

Control transfers will happen less frequently for an isochronous audio class device but they are still needed in order to initialize and configure the device when it is attached to the bus and for changing settings mid operation. An example of a control transfer transaction is when the host fetches the device descriptor in the configuration phase during the enumeration process described in Chapter 2.3.2. Any control transfer will consist of two or three transactions; 1) a token packet sent from the host to the device, 2) an optional data packet sent in either direction, and 3) a handshake packet sent to confirm success or to report failure. The "get descriptor" request has three transaction stages and they are shown in Figure 2.35. First the host sends a SETUP token packet which is received by the default control pipe of the device. The device request type and its parameters are embedded in the values of the PID field bits. The requested device descriptor is then sent back to the host using the default control pipe. Upon the descriptor having being successfully received by the host, an ACK is sent back as confirmation to the device.

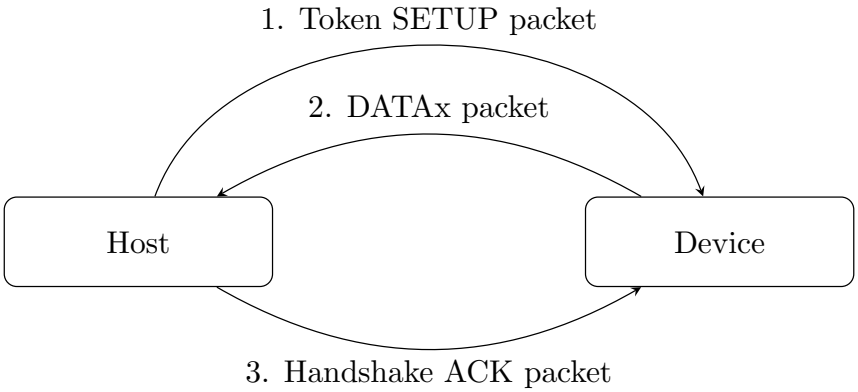


Figure 2.35: USB control transfer sequence.

Figure 2.36: USB SETUP packet used in a control transfer.

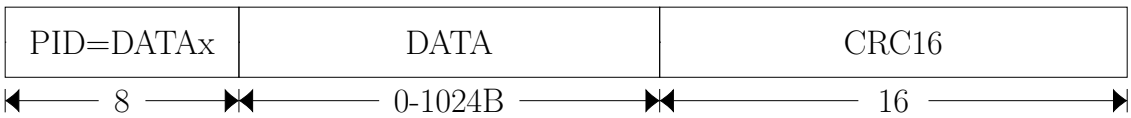


Figure 2.37: USB DATAx packet used in a control transfer.

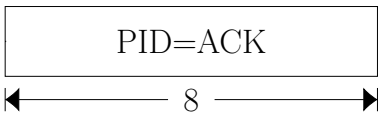


Figure 2.38: USB ACK handshake packet used in a control transfer.

2.3.11 Isochronous Synchronization Types

An isochronous endpoint will after being connected to the bus inform the host of the data rates it supports through its device descriptors. For isochronous data transfers, the USB specification presents three standardized endpoint synchronization types to choose from:

- Adaptive
- Asynchronous
- Synchronous

2.3.11.1 Synchronous

Synchronous mode devices use the SOF token packets that are generated every 1 ms for full-speed and high-speed endpoints or alternatively the microframe SOFs generated every 125 µs for high-speed endpoints as a reference to which the internal clock of the device is synchronized by the use of a phase-locked loop. Both fixed and continuously programmable data rates can be supported.

2.3.11.2 Adaptive

Adaptive mode devices adjust their internal clocks by monitoring the data rate of the sink or source endpoint to which they are connected. For an adaptive sink endpoint the data rate can be determined by averaging the number of data samples received over a period of time. An adaptive source endpoint receives feedback from the sink to determine the appropriate data rate that should be used. Data rates can be either fixed or variable.

2.3.11.3 Asynchronous

The internal clock of an asynchronous mode device is generated by a source external to the USB and it is not synchronized to any clock that is part of the USB system. Both fixed or continuously programmable data rates can be used. An asynchronous sink endpoint needs to provide feedback to the source through a separate isochronous IN endpoint in order to keep the transfer data rate at a sufficient level. For an asynchronous source endpoint the data rate can instead be determined directly from the number of samples that are transmitted during a frame and a separate endpoint for feedback is therefore not needed for an asynchronous source. The data rate is always controlled by the device operating in asynchronous mode and not by any source or sink connected to it.

2.3.12 Explicit Feedback

To avoid underflow and overflow of the buffers at either end of the communication channel, feedback of the desired data rate for an isochronous device sometimes needs to be provided. Under some circumstances this can be managed implicitly by observing the data rate of a stream when multiple isochronous pipes related to

each other send data in both directions, but for a single isochronous asynchronous endpoint consuming data, feedback needs to be provided explicitly to the source through a dedicated feedback endpoint. The same approach must be taken when a single adaptive source endpoint sends feedback to the sink to which it is connected to. To provide an accurate enough measure for the data rate calculation used in the feedback, the USB specification dictates that the measurement period T_{meas} expressed in frames or microframes for the data rate must be at least the length of one second. However, if the sampling rate F_s of the device is derived from a master clock of higher frequency than the sampling rate, then the measurement period may be reduced by that same clock multiple without losing accuracy. Equation 2.22 can be used to calculate the measurement time that is needed at a minimum to provide a data rate calculation of sufficient accuracy.

$$T_{meas} = \frac{2^K}{2^P} = 2^{K-P} \tag{2.22}$$

For a device in which the sampling rate F_s is not derived from a higher frequency master clock, $P = 0$ and Equation 2.22 can be simplified to $T_{meas} = 2^K$. The SOF packets generated for such a full-speed endpoint appear every 1 ms, so the frequency of the data rate calculation should be at least 1 kHz. Selecting $K = 10$ will give a T_{meas} value which meets that criterion. For a high-speed endpoint with a SOF rate of 125 µs, the frequency is 8 kHz, which translates to $K = 13$ being enough to meet the set demand.

In a device that does derive its sampling rate clock F_s from a master clock with higher frequency F_m,

$$F_m = 2^P F_s \tag{2.23}$$

and by rewriting the expression, the binary logarithm of the clock multiple in Equation 2.24 will give the value of P.

$$P = \log_2 \left(\frac{F_m}{F_s} \right) \tag{2.24}$$

By counting clock cycles of the master clock instead of the sampling clock that is derived from it, the accuracy of the data rate calculation can be sustained even though the measurement period is shorter and an updated data rate value is now generated every T_{meas} frame or microframe. It is of no use to update the value more often than once a frame or microframe depending on the device speed or to use a slower clock than the sampling clock itself for the calculation, so the range of P is $0 \leq P \leq K$.

The desired data rate value uses 10.10 format for full-speed endpoints and 12.13 format for high-speed endpoints. This means that both the integer part and the fractional part is represented by 10 bits each for the full-speed endpoints, and that the integer part for the high-speed endpoints is represented by 12 bits, accompanied by a 13 bit fractional part. To fit these bits into full bytes, full-speed endpoints use left-justified 10.14 bit representation, where the extra four bits can be used to improve precision or else they must be set to zero. A high-speed endpoint will

instead use 16.16 bit representation where the binary point is placed between the second and the third byte. The first four bits of the first byte for the integer part will always be zero and the last three bits of the fractional part can be used to improve accuracy or else they must also be set to zero.

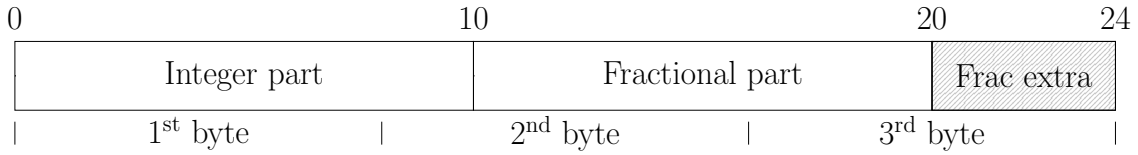


Figure 2.39: Feedback format for full-speed endpoint.

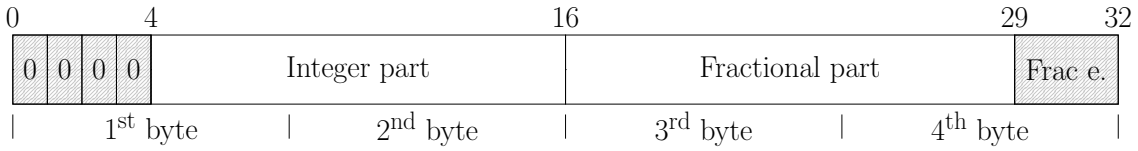


Figure 2.40: Feedback format for high-speed endpoint.

2.3.13 The Audio Device Class

This next section describes some of the main concepts of the audio class definition version 1.0 [10] which is the version of the definition that was used in the different project implementations. Some of the changes and updates in the subsequent revision 2.0 of the audio class definition [25] which was also under consideration are briefly discussed at the end of the section. A USB audio class device will typically contain one Audio Interface Collection group consisting of one AudioControl interface and one or more AudioStreaming interfaces. Settings like volume and the configuration of AudioStreaming interfaces are handled through the AudioControl interface which uses the default control pipe of the device. AudioStreaming interfaces are used to transport audio data from sender to receiver. In addition to the AudioStreaming interface type, a MIDIStreaming interface type for transmission of audio in MIDI format also exits and can be part of an Audio Interface Collection group. Interfaces follow a consecutive numbering scheme and AudioStreaming interfaces are always placed before MIDIStreaming interfaces in the collection. Audio functions of devices belong to a function category. Some examples of function categories are converters, microphones and speakers.

2.3.13.1 Clocks, Time and Synchronization

As presented in Chapter 2.3.8, the internal delay of a function receiving isochronous data will depend on the size of the internal buffer for incoming data, but for an audio device, the total delay of the device may also depend on if the audio data needs to be decompressed or processed in other ways after it has been received. This also needs to be taken into account when determining the total delay of a device.

2.3.13.2 Entities

There are two different kinds of entities called “terminals” and “units” in the audio class revision 1.0 by the use of which a USB audio class function can be described and presented to a host:

- Terminals
  - Input terminal
  - Output terminal
- Units
  - Extension unit
  - Feature unit
  - Mixer unit
  - Processing unit
  - Selector unit

Terminals represent the entry and exit points into and out of the audio function for an audio signal, and an input or output terminal is represented by a single input or output pin. Multiple audio channels are grouped together in an audio channel cluster with a predefined channel ordering and the audio data for all channels in the same cluster enters and leaves the audio function through the same input and output pins. Input and output terminals are described by class-specific terminal descriptors but some parameters closely related to the terminals are also found in the endpoint and AudioStreaming interface descriptors.

Units are the main elements that describe the internal operation of the audio function and the properties of units are reported to the host through class-specific unit descriptors. The USB audio class specification revision 1.0 defines no less than five different types of units that can be combined and together they cover most of the functionality that any ordinary audio device can be expected to have. As an example there is a selector unit that can be used to alternate between different audio sources. By sending control requests to the AudioControl interface, a host can determine which of the audio inputs the selector unit in the audio device is using at the moment and it can also have the device switch to a different audio source input. Like for all the other unit types, the selector unit only has one single output pin, but the number of input pins can be many.

2.3.13.3 Audio Class Descriptors

The combination of terminal and unit descriptors express how the parts inside the audio function and the entry and exit points are connected together. There are also other class-specific descriptors belonging to the USB audio class that exist alongside the standard USB descriptors described in Chapter 2.3.3. The device descriptor of an audio class device will in itself not indicate that the device belongs to the USB audio class; instead it will point the host towards the interface descriptor level where the device class information for the audio device can be found. Likewise, the device_qualifier descriptor will do the same. The format of the configuration and

the other_speed_configuration descriptors of an audio class device are the same as for any other USB device.

For the AudioControl interface, which must be part of an audio class device, there are two interface descriptors; the standard USB interface descriptor, and a class-specific interface descriptor. The class-specific interface descriptor defines the audio function category and what version of the audio class specification the audio function is compatible with. It also ties all the unit and terminal descriptors inside the audio function together. The unit and terminal descriptors do not have to be presented in a particular order. They follow next after the class-specific interface descriptor in the descriptor table and their total combined length is included in the descriptor length value reported by the class-specific interface descriptor. Each of the unit and terminal descriptors report their own descriptor lengths and what type of descriptor it is like any other descriptor does, but the rest of the descriptor content can vary greatly.

The AudioControl endpoint itself does not have any endpoint descriptors as it exists for all devices and uses the default control pipe for its communication. It is possible to have an additional interrupt endpoint through which the audio device can report status updates to the host, but it does not have any class-specific endpoint descriptor as the information provided by the standard interrupt endpoint descriptor is enough.

After the entity descriptors and the interrupt endpoint descriptor, if an interrupt endpoint exists, follows the AudioStreaming interface descriptors. Each AudioStreaming interface has one standard and one class-specific interface descriptor. The standard descriptor lets the host know that the interface is an AudioStreaming interface, it sets the interface number, and it also defines how many endpoints there are. The class-specific descriptor specifies the terminal to which the interface is connected, the types of formats that are supported, and the channel setup. There are also class-specific AudioStreaming format descriptors that contain details such as the sample rate and bit resolution for the data formats being used by the interface.

The endpoints of the interface have both a standard AudioStreaming isochronous endpoint descriptor and a class-specific one. The standard descriptor determines the direction of the endpoint, its synchronization type, and the maximum packet size that the endpoint can use. Information about how quickly the endpoint can lock on to a clock signal and what audio controls it supports are found in the class-specific AudioStreaming endpoint descriptor. If the interface uses a synchronization mode that includes a synchronization endpoint, then there will also be a standard endpoint descriptor for the synchronization endpoint but no class-specific one.

There is also an audio channel cluster descriptor that lists the number of channels in the audio cluster and defines the speaker positions for each of the channels using a bitmap field with predefined speaker placement positions. The audio channel cluster descriptor will never appear on its own as it always is part of either the input terminal descriptor, the mixer, processing or extension unit descriptors, or the AudioStreaming descriptor.

2.3.13.4 Audio Class Requests

The information and settings related to a unit or other entity that can be controlled through the AudioControl interface are the currently selected value and the range of available values that are allowed to be used, consisting of the minimum and maximum settings along with the resolution. The range values are always references together in an array in the AudioControl requests. For the selector unit mentioned in Chapter 2.3.13.2, the range array consisting of [MINIMUM, MAXIMUM, RESOLUTION] would be [1,n,1], where n is the number of inputs for the selector. The controls that a unit has and which can be manipulated by sending AudioControl requests are listed in the unit's descriptor. Each of the units have an individual entity number through which they can be identified and addressed so that AudioControl requests coming in through the control pipe can reach the correct unit.

In addition to AudioControl requests there are two other types of audio class specific requests, namely AudioStreaming requests and memory requests. AudioStreaming requests are used to handle the interface and endpoint descriptor settings for an AudioStreaming interface and memory requests gives access to a memory-mapped interface for any kind of unit, entity, interface or endpoint belonging to the audio function.

2.3.13.5 Audio Class Definition 2.0

Version release 2.0 [25] of the USB audio class definition includes a third entity type called the clock entity. This allows clock domains inside the audio device to be described. Due to this addition, a clock pin is also added to each of the input and output terminals. Clock entities come in three different forms; there are clock sources, clock selectors and clock multipliers. A clock source can be any internal or external clock that is used as sampling clock inside the audio function. Clock selectors are used to switch between clock sources in the same way a selector unit can be used to alternate between different audio inputs. A clock multiplier entity is used to generate new clock signals from a clock source. The output from the clock multiplier is formed by multiplying the signal fed into the unit by the fraction P/Q, where both P and Q are integers in the range $[1,2^{16}-1]$. The generated clock at the output of the multiplier unit is synchronized with the input clock source and both clocks do therefore belong to the same clock domain. An audio device defined by a set of device descriptors and revision 1.0 audio class descriptors can still include one or multiple clock entities, but there is not any way to describe the clocking structure through the descriptor table. The most significant change from version 1.0 to version 2.0 of the audio class definition is perhaps the added support for high-speed device operation, but many other smaller changes like adding a sampling rate converter unit and an effect unit, or changing some of the attributes of the already existing units are also introduced. Version 2.0 of the audio class definition is not backwards compatible with version 1.0. A version 3.0 of the USB audio class definition does also exist and version 4.0 [26] was even release no too long ago, adding even more bells and whistles. These newer releases of the specification will however not be discussed as they are not being considered as viable options for any

of the synchronization mode implementations in the thesis project.

The new clock entities and units introduced in release 2.0 of the USB audio class definition are listed below.

- Clock entities
  - Clock multiplier
  - Clock selector
  - Clock source
- Units
  - Sampling rate converter unit
  - Effect unit

2.4 Inter-IC Sound

Inter-IC sound, more commonly known as I^2S is a serial bus developed by Philips Semiconductors [27]. It is primarily intended to standardized communications between integrated circuits (ICs) used in audio applications. The communication channel consists of three separate lines:

- Serial clock (SCK)
- Word select (WS)
- Serial data (SD)

The SCK supplies the system with a controlling reference clock, WS indicates if the left or right channel is to be transmitted, and SD contains the audio data that is to be transferred. Each bus has a controlling unit governing the SCK and WS and then one or more clients nodes are connected to the controller. The controlling unit can be either the transmitter, the receiver, or an external source. Figure 2.41 shows a setup in which the transmitter controls the data flow, and this is the configuration that we will be using in the construction project of this thesis. The direction of the arrows indicate the direction of the data flow.

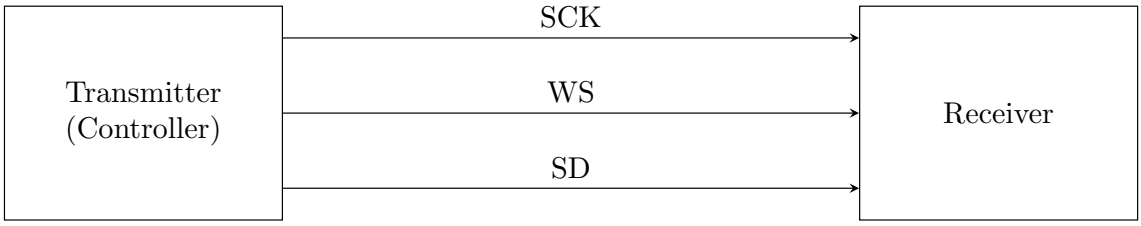


Figure 2.41: I^2S transmitter and receiver pair with the transmitter having the role of the controller.

The transmitter alternates between the audio channels in the system depending on the state of WS, sending out a sample for the right channel when WS is high, then

switching to the left channel when WS goes low and so on. This data pattern is illustrated in Figure 2.42. Data is transmitted with the most significant bit (MSb) first. The I²S specification [27] does not define any values to which SCK and WS are limited, so as long as both the transmitter and the receiver are operating within their timing constraints, the sample rate and the word length or number of bits used for data representation can be selected freely. The word length does also not have to match for a transmitter and a receiver in a system. For a setup where the transmitter is controlling the transmission like the one in Figure 2.41, the receiver will ignore the least significant bits of the transmitted data if it uses a smaller word length internally than the transmitter does, and for a receiver using a word length larger than the transmitter's, the bits missing in the transmitted data are set to zero internally in the receiver.

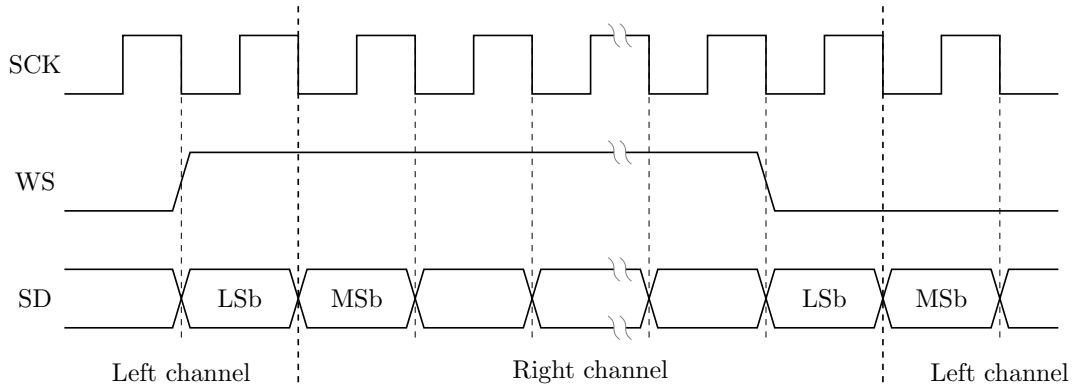


Figure 2.42: I²S interface timing diagram.

The I²S bus being constructed to connect ICs located nearby each other inside the same unit or device, often even on the same circuit board, does not have a standardized connector type to connect ICs in two separate units together with a cable. This has however not stopped some audio manufacturers in the past from creating their own solutions to allow two ICs in separate devices to be connected together using I²S. The objective for such solutions has often been to overcome the interface jitter issues attributed to S/PDIF by simply replacing it with I²S instead. Due to the lack of standardization for this type of external connection, devices from different manufacturers may however not be compatible. Three wires instead of only one for S/PDIF need to be used to connect two separate devices together with I²S, the number of audio channels that can be transmitted is limited to two while S/PDIF can handle up to eight channels, and the I²S bus will also omit any other information such as the S/PDIF control word settings as it is designed to only transfer audio data.

2.5 Frequency Synthesizers

A frequency synthesizer incorporating a phase-locked loop [19] can be used to generate a new clock signal with an operating frequency that is higher than the frequency

of the signal from which it is being derived, all the while being able to keep both signals synchronized, even when the original signal fluctuates a bit. Frequency synthesizers can be found in all kinds of audio applications, including ones featuring recovery and generation of clock signals from S/PDIF and USB audio interfaces. The main building blocks consists of a phase detector, a voltage controlled oscillator (VCO), a low pass filter, and a frequency divider. The principle of operation is that an external clock signal with frequency f_{IN} from which the new clock is to be derived from is attached to one of the inputs of the phase detector. It will first take a little bit of time for the PLL to lock on to the external signal. Once being in the locked state, the external clock signal and the feedback from the loop can be considered to be the equal. The phase detector being fed both the external signal and the feedback from the loop will compare the external clock and the feedback from the PLL. If a phase difference exists between the two, then a voltage proportional to the size of the phase difference will be output and run through the low pass filter, after which the remaining direct current (DC) component then is passed on to the VCO. The VCO has a base frequency at which it operates when no voltage is applied to its input, but once a phase difference is detected which gives rise to a voltage difference proportional to it, the applied voltage will make the VCO modulate its output frequency so that the clocks remain synchronized. The frequency divider makes it possible to generate output clock frequencies f_{OUT} that are multiples of the external clock signal f_{IN} being fed into the PLL.

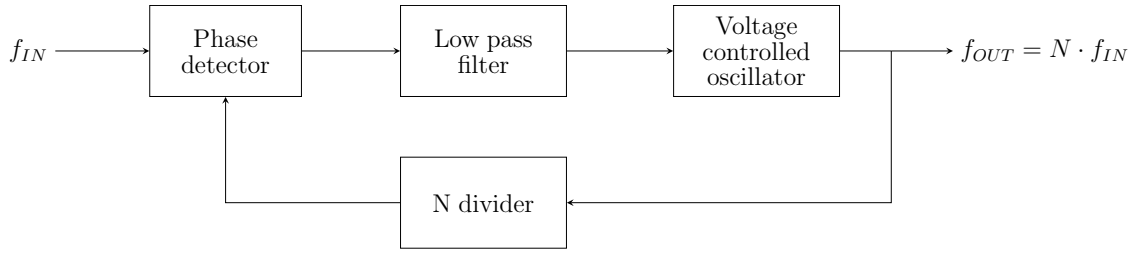


Figure 2.43: Block diagram of a phase-locked loop.

2.6 Fractional Dividers

With an integer-N divider the frequencies that can be produced from a high frequency source clock are limited to full integer multiples of the original clock frequency. To create other output frequencies from the same source, a fractional-N divider [28] can be used. As a trivial example, let us assume that we have a source clock of frequency $F_{IN} = 12\,\text{MHz}$ and that we want an output clock of frequency $F_{OUT} = 5\,\text{MHz}$. The source clock needs to be divided by 2.4 to form the wanted output frequency, so solely integer division will not suffice. We can write the divider as the fraction $\left(2 + \frac{2}{5}\right)$ and set $N = 2$, $K = 2$ and $F = 5$ in Equation 2.25 to get the correct output frequency F_{OUT} when F_{IN} is 12 MHz.

$$F_{OUT} = \frac{F_{IN}}{\left(N + \frac{K}{F}\right)} \tag{2.25}$$

There is however no rising or falling clock edge to indicate exactly when $2 + \frac{2}{5}$ of the source clock signal has been reached and therefore no simple way to just divide F_{IN} by 2.4. What fractional dividers do is to divide the source clock by N for a certain number of clock cycles followed by a division of $N + 1$ for another number of clock cycles. This way the average frequency of the output clock becomes F_{IN} divided by $\left(N + \frac{K}{F}\right)$, even though the actual frequency will alternate between $\frac{F_{IN}}{N}$ and $\frac{F_{IN}}{N+1}$. To illustrate the procedure, Figure 2.44 shows the example of 12 MHz divided by 2.4. We can see that by the time the input signal with a frequency of 12 MHz has reached 12 full clock cycles, the output signal with a frequency of 5 MHz will have completed five full cycles, and it is effectively the result of dividing the input signal by 2.4. To get the desired output, a division of the input signal by $N + 1$ can be performed K times followed by a division of N done $F - K$ times. A classic design that can be realized easily in hardware is to have an accumulator that starts at zero and then adds N to its count each time a full clock cycle of the output clock has completed. When the accumulator reaches F it overflows and this indicates that the next divider value should be set to $N + 1$. If there is no overflow, the divider value to be used is N. This accumulator pattern is what is used in Figure 2.44. The accumulated count for each completed output clock cycle is: $0, 2, 4, (\uparrow 1), 3, (\uparrow 0), 2, 4, (\uparrow 1), 3, (\uparrow 0), 2, 4$ and so forth, so there is a repeating pattern of two divisions by N, one division by $N + 1$, one division by N and then one more division by $N + 1$, after which the cycle starts over. The downside to this is that it creates spurious signals that will give rise to periodic jitter. A more novel approach that often goes by the name of delta-sigma fractional-N divider is to randomize the N and $N + 1$ dividers instead of using an accumulator while still keeping the same ratio of K cycles of $N+1$ division plus $F-K$ cycles of division by N. We will not get into the details of the mathematics involved in PLL design but it is possible to use fractional-N dividers in place of integer dividers in a feedback loop.

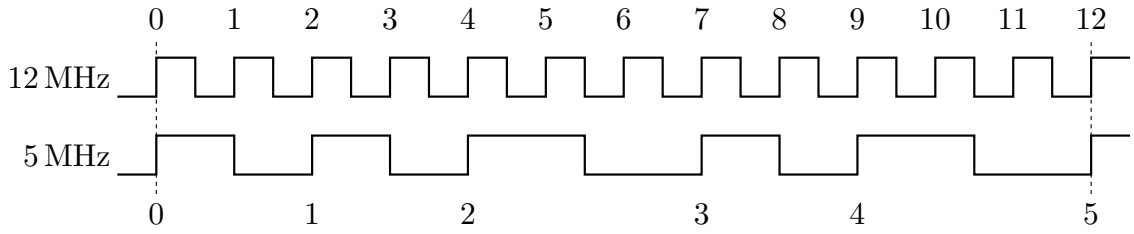


Figure 2.44: Example of fractional division by 2.4.

3

System Design and Implementation

The following chapter presents the hardware and software selected to be used in the project and the design decisions made, reconnecting with the theory concepts described in Chapter 2.

3.1 Hardware Selection

An interface of USB type was selected to be used for the hardware build as it satisfies all the criteria set, while still being manageable from a design and construction perspective. Other interface types and technologies that can carry digital audio data were considered and excluded as viable options due to different reasons including but not limited to reliability, limited bandwidth, availability and design complexity. The test interface was constructed using a PSoC 3 microcontroller unit [29] mounted on a CY8CKIT-001 PSoC development kit board [30], both produced by Cypress Semiconductor Corporation, which was acquired and is now since 2019 owned by Infineon Technologies AG. The development kit also includes the MiniProg 3 combined programmer and debugger which was used for programming the PSoC device. A programmable clock generator from Adafruit [31] equipped with the Si5351A chip [32] from Silicon Laboratories Inc. was used as an external clock source and an Adafruit I^2S stereo decoder [33] featuring the UDA1334A DAC chip [34] from NXP was used as an external converter for the digital audio signal. The PSoC device does as the name implies not only offer a microcontroller but it also includes a number configurable universal digital blocks (UDBs) that can be used to realize analog or digital peripherals so that external components do not need to be used. Both the clock generator and the DAC chip from Adafruit can also be made to work without attaching any other external electrical components than the ones already mounted to the circuit boards and support for logic levels ranging from 3.3 V to 5 V further add to the user-friendliness when including them in a design. A clock generated by an external crystal oscillator and an external fixed rate clock board was briefly used for testing at the end of the project. The external crystal required a couple of external capacitors to be connected to it but other than that, all other components needed for the project could be constructed in the PSoC Creator integrated design environment (IDE) software.

3.2 Development Environment

The designs for the PSoC 3 microcontroller were created in the PSoC Creator IDE. It includes everything needed for writing and compiling code and creating schematic designs. It also has programming and debugging support for PSoC 3 microcontroller units when attached to a device through the MiniProg 3 combined programmer and debugger. Debugging in the IDE by connecting the MiniProg 3 to the serial wire debug (SWD) port on the PSoC microcontroller board was however of limited use as real-time observation of variables are not possible when fast paced communication like the isochronous transmissions of the USB protocol needs to be upheld without interruptions. The IDE has a well documented component catalog with anything from simple logic gates to more complex peripherals for communication and signal processing that can be included in a design. Configuration of component settings can often be altered both through graphical user interface (GUI) dialog boxes or code pre-build, or through application programming interface (API) calls in the code during run-time. It is also possible to create your own custom components if the ones provided by the library are insufficient or unsuitable for your design.

3.2.1 Monitoring of Device Operation

As previously mentioned, the MiniProg 3 debugger cannot be used to monitor variables in real-time in the PSoC Creator IDE without disrupting the ongoing communication over USB between the host and the PSoC device. Fortunately there are other communication channels that can be used instead. During testing, the LCD which is part of the PSoC development kit was used for direct monitoring of variables. Alongside this, logging of values was done by connecting the PSoC device to a host terminal through the RS232 serial interface using UART. Using either of these two methods does however affect the performance of the rest of the application that is being run on the PSoC system, so some care must be taken regarding how API calls are being made to display or send data variables. For logging over UART, the data sent to the receiving terminal was minimized to single byte comma separated hex values, all sent with a time interval between the transmissions so that no long resource consuming data bursts were created. Likewise, excessive updating of the LCD may lead to problems elsewhere in the design, so the update interval and the amount of data sent to be displayed on the LCD was also limited. Both monitoring methods still provided sufficient throughput and some of the longer measurement series logged over UART even turned out to have so many data points that the whole measurement period could not be plotted in full by the PGFPLOTS package without first decimating the data.

3.2.2 The PSoC Clock System

PSoC 3 has a highly configurable clock network [35] where signals can be routed across the chip to be used as clock sources where needed. Figure 3.1 shows the significant parts of the PSoC 3 core clocking network. Part of the core PSoC clocking network configuration are also the internal low-speed oscillator and the 32 kHz ex-

ternal crystal oscillator blocks, but they have been left out of Figure 3.1 as they are not directly intertwined with the rest of clock network and as no use were made of them during the project. The MHz range external crystal (XTAL) oscillator block was also not used in any of the different synchronization implementations but it has still been included in Figure 3.1 as it is more closely interconnected with the rest of the core clocking network. The internal main oscillator (IMO) which normally is used as the main clock source for the PSoC device can only be run at the fixed clock rates listed in Table 3.1. A clock running at 48 MHz with at least ±0.25 % accuracy is needed for the USB block to function correctly. The only valid option for the IMO frequency setting when used as source for the USB block is to set it to 24 MHz which then together with the IMO doubler option produces the 48 MHz USB clock, at the same time trimming the IMO to an accuracy of ±0.25 % using the USB SOFs from the host as reference. This further limits the configuration options for the source clocks. The PLL can create clock frequencies from its input clock with a ratio of P/Q, where $4 \leq P \leq 256$ and $1 \leq Q \leq 16$. The clock outputs of the master clock and the bus clock can be divided individually by any integer. External signals can be routed in through the PSoC pins to be used as source clock in the digital system interconnect (DSI) block.

IMO clock frequency	Accuracy
3 MHz	±1 %
6 MHz	±2 %
12 MHz	±3 %
24 MHz	±4 %*
48 MHz	±5 %
62.5 MHz	±7 %

*±0.25 % *when trimmed by USB SOFs.*

Table 3.1: The fixed frequencies at which the IMO can be operated at.

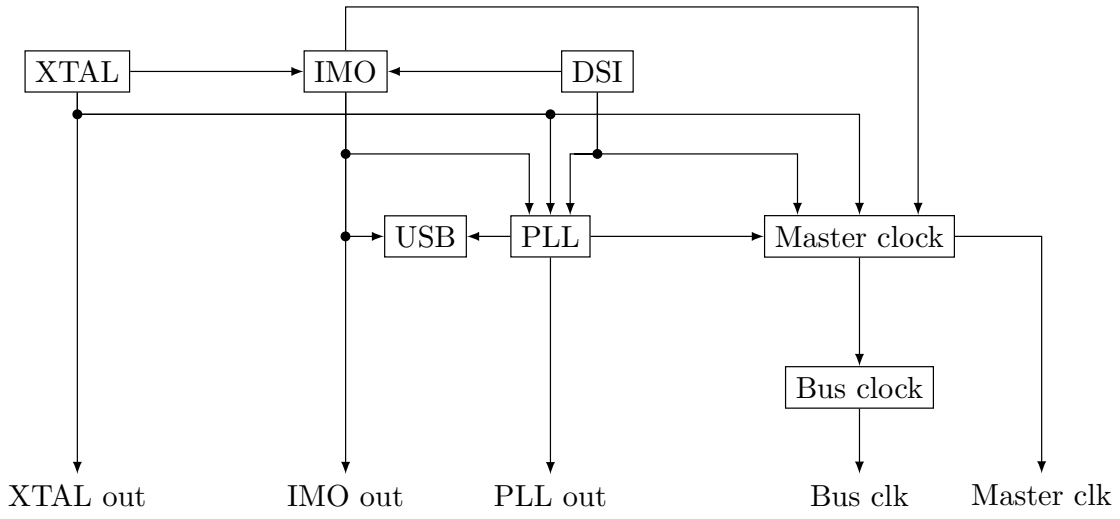


Figure 3.1: The PSoC core clocking network.

With the many different clocks in the PSoC system, it will sometimes be necessary to deal with signals crossing over from one clock domain to another. To avoid problems with metastability [36] that can arise when unsynchronized signals are used together in the same digital circuit, the system has supporting features for synchronization that can be used at different stages in the signal path. Synchronization does however come with the cost of routing delays and the reference clock which the signal is to be synchronized against needs to have a frequency that is at least double the frequency of the signal being synchronized, so it may not always be possible to perform synchronization while satisfying all other system parameters.

When a project is built, the PSoC Creator IDE will generate a static timing analysis report which will alert the designer of any possible timing violations that are found in the current setup. Timing violations do occur from time to time but there are ways to resolve them. Depending on the type of timing issue, it can sometimes be enough to just decrease the frequency of a clock source if the rest of the design allows for it or to change the system parameter for the operating temperature range if the timing violation is minor and the circumstances allow for a narrower range to be selected. In other cases more extensive changes need to be made to the design in order to meet all timing requirements.

3.3 Implementation

The synchronization modes were implemented in the same chronological order in which they are presented, starting with asynchronous mode, followed by adaptive mode and finishing with the synchronous mode implementation. They all share a common basic structure but differ in clock source selection and other design specific details.

3.3.1 Common Design Layout

Figure 3.2 shows the overall layout of the different physical building blocks used in the project build. Most DACs need to have a high frequency master clock connected directly to one of its inputs but the Adafruit I^{2}S stereo decoder DAC board can be set to work with only the three standard I^{2}S output signals, SCK, WS and SD connected to it. This is the setting that has been used, which is why there is no direct connection from the external clock to the DAC in Figure 3.2 like one might have expected. The use of an external clock source was not only a prerequisite for having access to a high quality master clock, it was also necessary in the initial phase of the construction because the clock off of which the I^{2}S component is run must have an operating frequency of $2 \cdot F_S \cdot t_{WS}$ where F_S is the sample frequency of the audio signal and t_{WS} is the word select period of the I^{2}S component. To create a source clock of suitable frequency for the I^{2}S component for 44.1 kHz audio using only the IMO and the PLL together with an integer-N divider is not possible due to the limited choice of fixed IMO frequencies and the rather narrow P/Q ratio range of the internal PLL. There is an option in the I^{2}S component settings to either have the input clock for the I^{2}S component synchronized to the system clock or to use it

without synchronization. In the end it was found best to have the I²S component's input clock sourced from the system master clock so that they both belong to the same clock domain to avoid problems with metastability.

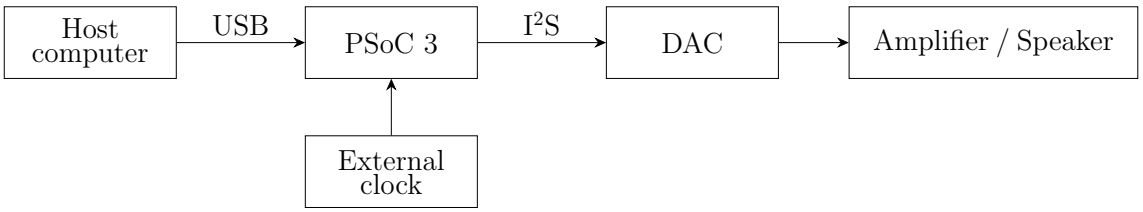


Figure 3.2: Layout of the audio path and the physical entities.

Audio data coming from a host computer or other playback device first arrives at the USB endpoint buffer of the PSoC. Data cannot be sent directly from the endpoint buffer to a peripheral so it needs to be moved into main memory before it can be transferred to the I²S component. The combined buffer space for all USB endpoints is limited to a total of 512 bytes, although USB packets of up to 1023 bytes in size can be handled when using automatic endpoint buffer management mode with direct memory access (DMA). For comparison, an ordinary USB data packet for 2 channel audio with 44.1 kHz sample rate and 2 byte bit-depth will usually be either 176 or 180 bytes in size. Transmission of data in synchronous and adaptive mode follows the transfer sequence shown in Figure 3.3 consisting of nine 44 byte packets followed by one packet of 45 bytes to produce the 44.1 kHz sample rate. Those numbers are for one single audio channel with a bit depth of eight bits, so for our two channel audio with 16 bits per sample we need to multiply everything by four. Asynchronous mode will in general use the same transfer scheme but may need to adjust it slightly to match the clock rate of the device receiving the audio stream.

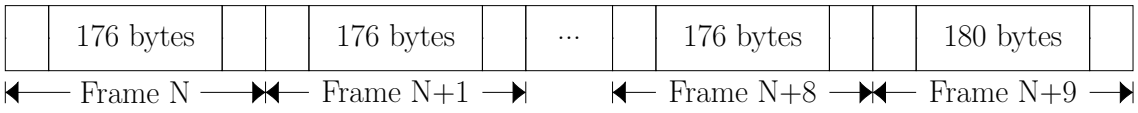


Figure 3.3: General transfer sequence for USB audio transmissions.

Data was stored in main memory in the form of a circular buffer consisting of a number of evenly sized memory chunks. Each chunk was connected to a transaction descriptor belonging to a DMA channel connecting the memory buffer to one of the I²S component's transmit (TX) registers. The transaction descriptors were chained together wrapping around from the end to the start of the circular buffer. Data received by the USB endpoint was transferred into main memory one packet at a time using a second DMA channel but the USB packet size is not constant so the data transferred into the main memory buffer is not of the same size as the memory chunks in the buffer and the data going out of it. Each time the DMA channel used for transferring data out of the circular buffer to the I²S component has completed moving one whole memory chunk, an interrupt service routine (ISR) is invoked to

update the parameters of the buffer. There are also other interrupt service routines used in the different synchronization mode implementations and most of them are related to the operation of the USB component. They are all set to have the same priority level so they can be considered to be mutually exclusive. The course of action varies a bit from one ISR to another but in order to follow sound design principles some effort was taken to make the execution time of the interrupt service routines as short as possible, sometimes only setting a flag to be checked later.

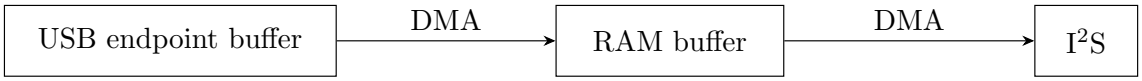


Figure 3.4: Audio data transfer path inside the PSoC device.

During audio playback the device is in one of three defined main operation modes; normal, underflow or overflow. In some cases other intermediate brief transitional operation mode states were also used when passing from one main operation mode to the next. The underflow and overflow states are there to safeguard in the event of an error happening that would make the audio data buffer in main memory either entirely run out of audio data or be completely filled so that no new audio data could be inserted into it without overwriting the audio data already present. During normal operation none of these two states are expected to be reached but there needs to exist a recovery mechanism so that audio playback can resume normally in the event that an error should happen. In the underflow state the DMA channel for transfer of data from main memory to the I^2S component and the I^2S component itself are disabled, only to be enabled again once the buffer in main memory has been filled up with data to half its total size. Depending somewhat on the audio playing, this will usually result in a very audible clicking sound. If the circular buffer is about to overflow, the operation mode will instead switch to the overflow state and the audio data in all incoming UBS packets will be discarded until half the circular buffer has been consumed. The effect this has on the audio playback is less intrusive than what happens when the device goes into the underflow state, but it is usually still noticeable. During development the occurrences of both these two error states were tracked by printing the operation mode changes on the LCD.

The device descriptors in the implementations use version 2.0 of the USB specification and the audio device class descriptors use version 1.0 of the audio device class definition. Our target device is a full-speed device and we do not need any of the additional features provided by the subsequent audio device class definitions. Using any of them would add more complexity without any real benefit for the current implementations and in addition, native support for the audio device class 2.0 or later is not always something you can count on when it comes to some older host operating systems. The configuration settings in the PSoC Creator IDE for the USB component has support for the audio device class definition version 1.0 and it also appears to have partial support for the newer audio device class 2.0 definition, but for example not all descriptor types for version 2.0 are included, so effort would have to be put into making everything fully compatible with the audio device class version 2.0 definition.

Many of the USB device and audio class descriptors used in the different synchronization mode implementations are identical. It is only the standard OUT endpoint descriptor, the class-specific AudioStreaming endpoint descriptor and the additional IN endpoint descriptor used in the asynchronous mode implementation that differ significantly between the different synchronization modes. Relevant parts of the descriptor table for the asynchronous, adaptive and synchronous mode implementations are listed in Section A.1, A.2 and A.3 of Appendix A.

3.3.2 Asynchronous Mode Implementation

In the first asynchronous mode implementation a clock signal with a frequency being a multiple of the clock needed for the I^2S component was routed through an input pin into the DSI block in the clocking network. This external clock was then fed into the PLL where it was multiplied to generate a master and a bus clock of high enough frequency to operate the design without issues. Development versions where the external clock generator was used to create the higher master clock frequency directly without using the internal PLL were also tried but in order to limit electromagnetic noise emission it was found best to use a lower frequency external clock signal and let the higher frequency master clock be generated inside the PSoC. The IMO running at 24 MHz was used as source for the USB clock.

The Adafruit Si5351 clock generator board [32] selected to be the first external clock source for the asynchronous mode implementation can be used to create up to three output frequencies simultaneously. Operating frequencies are set by programming the device registers through the I^2C interface. This can be done by using the PSoC as it has an I^2C peripheral in its component catalog. To successfully proceed with the configuration we need to know the register mapping and how the data sent to the device should be formatted. All transactions are initiated by the master device which in our case is the PSoC. Read and write operations can be done either one register at a time or by reading or writing multiple consecutive registers in a burst. Each transaction begins by setting an initiating start condition, followed by the default 7-bit address of the Si5351 chip and a bit which indicates if it is a read or write operation that is about to be performed. After the device has acknowledged this first transmission, the registry value to be accessed is sent and then follows the actual data transfer portion of the transaction. For a write operation the controlling device will send either one or multiple bytes followed by the stop condition. Each byte sent is acknowledged by the receiver before the next one is transmitted. A read operation is done in two separate steps separated by start and stop conditions. The first transmission sequence selects the register to be read and the second one reads the data from it. Figure 3.5 displays an I^2C single register write operation and Figure 3.6 shows how a burst write operation to two consecutive registers is carried out.

S	Device Address	0	A	Register	A	Data	A	P

Figure 3.5: I^2C single register write operation.

Figure 3.6: I^2C burst write operation to two consecutive registers.

The list of registers that need to be modified in order to configure the output frequencies of the Si5351 board can be found in Section B.1 of Appendix B. The registry values used to create the 2.8224 MHz external clock signal were produced by the Skyworks ClockBuilder Pro software and programming of the clock generator registers was done following the steps in Figure 3.7.

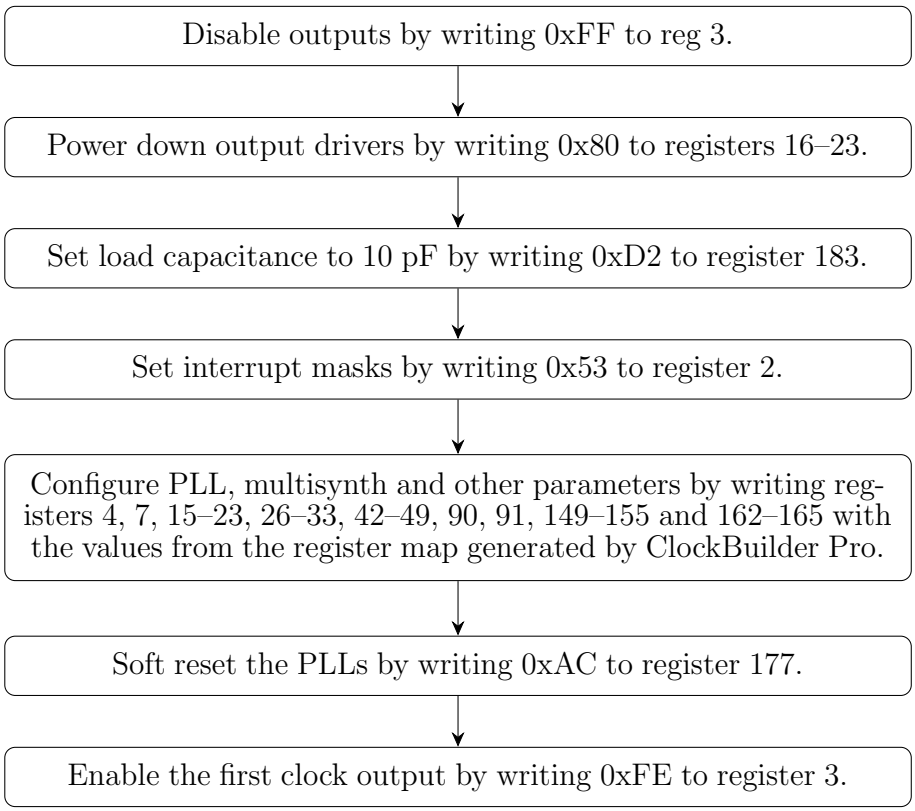


Figure 3.7: Programming procedure for the Adafruit Si5351 external clock generator board.

For collection of the input data used in the calculation of the feedback value, a design with two counter components connected to each other was created. One of the counters keeps track of the I^2S component's input clock and the other one counts the number of SOF occurrences. Each time the SOF count reaches 16 it starts over again from zero and a capture of the current source clock count for the I^2S component is triggered. This consequently also causes an interrupt service routine to be invoked, setting a flag to indicate that a new clock value has been captured and that it is ready for use. The selected length of 16 SOFs for the measurement period originates from Equation 2.22 in Section 2.3.12. For our full-speed device we have $K = 10$ and

with the help of Equation 2.24 we can determine that $P = 6$ for an audio sampling frequency of $F_s = 44.1\,\text{kHz}$ and the source clock $F_m = 2.8224\,\text{MHz}$ used in the I^{2}S component.

We could select some other clock in the system that is in sync with the I^{2}S input clock to be the source from which we calculate our feedback value, but the sensible valid interval is essentially limited to P being in the range $[0,K]$. At the lower end of the interval we have $P = 0$ which corresponds to using the audio sample frequency F_s directly and it gives a measurement period of 1 s for the counter. At the other end of the spectrum we have $P = K$ which would lower the measurement period down to 1 ms for a more frequent feedback interval, and that would require the measured clock signal to have a frequency of at least $1024 \cdot F_s$ according to Equation 2.23. The PSoC master clock is sync with the I^{2}S component's input clock and it does meet this higher range limit criterion, so there is room for adjustment of the update interval in both directions, but to keep things as simple as possible the I^{2}S clock with an update interval of 16 ms corresponding to a P value of six was selected as the starting point. The general recommendation for selection of P is to use larger values as more frequent updates will result in better control and may allow for a smaller buffer to be used without risking underflow or overflow. Setting $P = K$ should however be avoided as that could make SOF-to-SOF jitter effects more noticeable, so the design guidelines state that it is better to select a slightly lower P value so that the measurement period extends over at least two SOF cycles.

As the I^{2}S component's input clock has a frequency of 2.8224 MHz, this will result in a capture value in the counter component of approximately 45 158.4 clock cycles for each measurement period of 16 frames. The captured value will be an integer, so let us assume that 45 158 is what has been captured. The feedback value sent back to the host should indicate how many samples the device consumes during a frame. To relate the captured value from the counter to the actual data rate we need to divide it by the number of frames in the measurement period which is 16, multiplied by the multiplication factor of the I^{2}S input clock relative to the audio sample frequency F_s, which is 64. The value captured by the counter thus needs to be divided by 1024. Let us now remind ourselves of the format of the feedback array for a full-speed device shown in Figure 2.39 in Section 2.3.12. The feedback consists of an integer part and a fractional part. For the integer part we can simply perform the division and then by bitwise operations mask and shift the result into the correct locations in the first two bytes of the feedback array. Both the numerator and denominator are integers, so no floating point operations are needed for the calculation. Figure 3.8 shows the feedback value array with the integer bits populated with the result of the counter capture value of 45 158 divided by 1024.

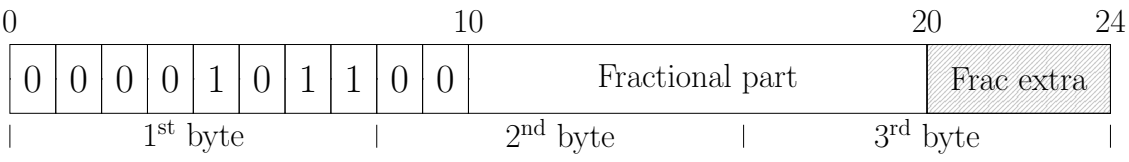


Figure 3.8: Feedback array with integer part populated.

For the the fractional part, a solution was also selected that does not require any resource consuming floating point operations to be performed. The calculation of the fractional part is done in very much the same way as a manual calculation of the fractional bit values would have been carried out. After performing the integer division by 1024 of the captured counter value of 45 158 in the example, the remainder left is 102. If we were to calculate the first bit of the fractional part in the feedback array by hand, we would first compare the value of the remainder $102/1024 = 0.099609375$ with the value for the first fractional bit which is $2^{-1} = 0.5$.

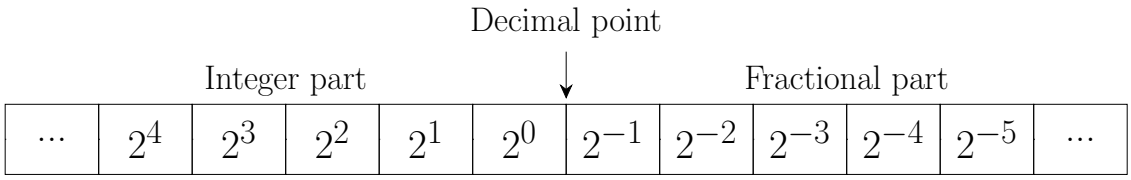


Figure 3.9: Bit values around the decimal point in the feedback array.

Since the remainder is not equal to or larger than the bit value, the first bit in the fractional part would be set to zero. We then move on to compare the remainder with the next bit, which has the value $2^{-2} = 0.25$, and since 0.099609375 is not equal to or larger than the bit value for the second bit, also the second bit is set to zero. The third fractional bit has the bit value $2^{-3} = 0.125$ and it is therefore also set to zero. It is first when we reach the fourth bit of the fraction that the remainder is bigger than the bit value of $2^{-4} = 0.0625$. We now set the fourth bit to 1 and subtract the bit value from the remainder leaving $0.099609375 - 0.0625 = 0.37109375$. Then we continue in the same way only now comparing 0.37109375 instead of the original remainder, and we do this until we have gone through the rest of the fractional bits setting them to either one or zero.

If we return to the beginning again and look at the expression for the first fractional bit comparison, we notice that instead of comparing $102/1024$ with 0.5 it is possible to multiply both sides of the expression and instead get $102/512 \geq 1$. We can therefore instead do an integer comparison between 102 and 512 and conclude that if $102 \geq 512$, then the first fractional bit should be set to one and the value of the denominator should then also be subtracted from the numerator, but since this expression does not hold true, the first bit is set to zero and we move on to the second fractional bit for which we need to divide the denominator with two once more before we do the comparison. This same procedure then continues for the rest of the fractional bits. The resulting bit values are displayed in the rightmost column of Table 3.2 and in Figure 3.10 the feedback array has been populated with the same fractional bit values.

Fractional bit	Numerator	Denominator	Bit value
1	102	512	0
2	102	256	0
3	102	128	0
4	102	64	1
5	38	32	1
6	6	16	0
7	6	8	0
8	6	4	1
9	2	2	1
10	0	1	0

Table 3.2: Calculation of the fractional bit values.

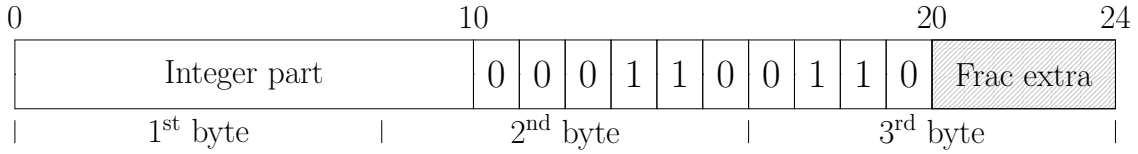


Figure 3.10: Feedback array with the fractional part populated.

Putting it all together, we have the complete feedback array in Figure 3.11. The value is precisely 44.099609375 so there is no rounding error and we arrive at the result without using floating point operations. No use is made of the four extra fractional bits as no more precision can be had, so they are set to zero.

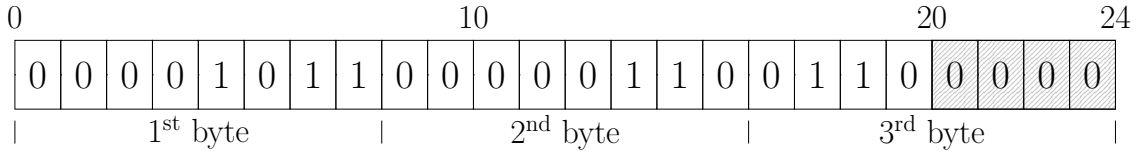


Figure 3.11: The complete USB feedback array.

The feedback value is then loaded into the endpoint buffer for the USB IN endpoint to be sent to the host each time a new value has been has been calculated every 16$^{\text{th}}$ SOF. An IN endpoint providing feedback for a single isochronous OUT endpoint should have the same endpoint number as the OUT endpoint but different direction to comply with the USB specification. The PSoC device does however only allow for an endpoint number to be used in one single direction, so it is necessary to select a different endpoint number for the feedback IN endpoint than what the AudioStreaming OUT endpoint is using. The USB specification does also declare that in the case of multiple data endpoints sharing the same feedback endpoint, the feedback endpoint should always have an endpoint number equal to or lower than any of the the data endpoints for which it is providing feedback. The most

logical choice was therefore to set the feedback IN endpoint number to one and the AudioStreaming OUT endpoint number to two, even though only feedback for one single endpoint was being provided. One other thing to mention about the feedback endpoint is that the USB protocol uses little endian ordering, so the byte order of the feedback array needs to be reversed before it is sent to the host.

The asynchronous mode implementation was then later tried with other clock sources. The Si5351 clock board using the ClockBuilder Pro generated register map was first replaced by an external oscillator for which the circuit diagram is shown in Figure 3.12. A second test was then conducted using a fixed frequency (FF) clock board driven by a 11.2896 MHz crystal oscillator mounted on it. Later on, after the two other USB synchronization mode test designs had been completed, the asynchronous mode implementation was then also tried with the Si5351 clock board using the manually generated multisynth integer mode register map, and finally also a test with the design using the PSoC IMO coupled with a custom fractional component was conducted. The results of the different tests are presented in Chapter 4.

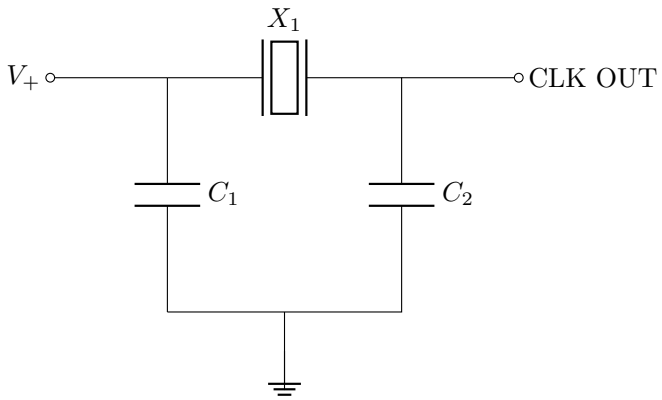


Figure 3.12: Crystal oscillator circuit used for generating external clock.

3.3.3 Adaptive Mode Implementation

An adaptive mode sink should adjust its internal clock to match the data rate of the source that it is connected to. We can use much of the same basic design from the asynchronous mode device for the adaptive mode implementation, but we now need to be able to adjust the master clock of the device on the fly instead of running it at the same rate like in the asynchronous mode implementation. To determine if a feasible solution can be built using the Adafruit Si5351 clock generator, we must first find out a little bit more about its inner workings. In the asynchronous mode implementation the register map with the frequency settings for the external clock generator was generated by the ClockBuilder Pro software and programmed at startup into the device to remain constant during its operation. Now we would need to generate multiple configurations to match the variations in the data rate which to some extent can be done, but preferable would be if the values that set the frequencies of the clock outputs could be calculated during operation to have a better degree of accuracy than what a set of fixed configurations can provide.

Another problem is that when programming a new configuration into the clock generator by following the process described in Figure 3.7, the resetting of PLLs and disabling and enabling of the clock outputs will cause interruptions in the sound. As it however turns out, it is possible to update the output frequencies during operation without having to go through the whole programming procedure. By only changing configuration settings that apply to the later synthesis and output stages, there is no need to restart the PLLs or to disable the clock outputs. The Si5351 is in fact designed in a way so that the switch from one frequency to another can be done seamlessly by starting the first clock cycle of the new frequency setting right at the end of a clock cycle of the previous frequency setting which will make the transition completely glitch-free [37]. Configurations generated by the ClockBuilder Pro software do have a tendency to use different register settings in all of the clock configuration stages including the ones for the PLLs even for frequencies which are closely related and could be switched between by only updating the parameters in the later stages of the synthesis. Using ClockBuilder Pro to produce the configurations that we need now is therefor not the best option.

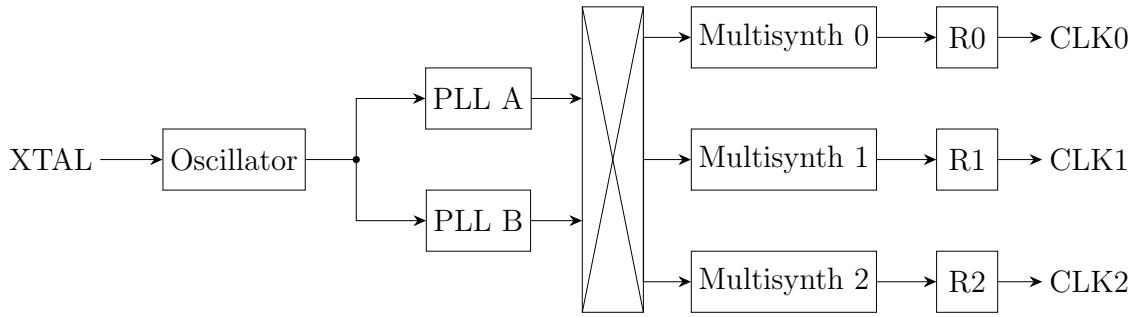


Figure 3.13: Si5351 block diagram.

A block diagram for the clock path inside the Si5351 is shown in Figure 3.13. The clock source is an external 25 MHz crystal driving the oscillator block from which the clock signal then is routed into the two PLLs. Both can be configured individually to act either as integer or fractional multipliers of the XTAL frequency. The frequency of a PLL is set by selecting parameters m, n and d in the equation

$$F_{PLL} = F_{XTAL}\left(m + \frac{n}{d}\right) \tag{3.1}$$

where the value for $\left(m + \frac{n}{d}\right)$ must be within the range 15–90. The documentation [38] is not crystal clear on all the fine details but it appears as if d in the fraction $\frac{n}{d}$ has a maximum value of 1 048 575, so its range should then be $1 \leq d \leq 1\,048\,575$ and presumably the range of n would then be $0 \leq n \leq d$. There exists an integer only mode setting for each of the PLL blocks which can be enabled to improve jitter performance when the sum of $\left(m + \frac{n}{d}\right)$ is an integer and not a fraction. The output clocks from the PLLs can be routed to any of the multisynth blocks where the clock frequency is divided and the output frequency F_{mult} from a multisynth

block is determined by

$$F_{mult} = \frac{F_{PLL}}{a + \frac{b}{c}} \tag{3.2}$$

where $a + \frac{b}{c}$ is in the range of 8–2048 with discrete values 4 and 6 also accepted outside the given range. Valid values for the multisynth for b and c are presumably $0 \leq b \leq c$ and $1 \leq c \leq 1\,048\,575$. On top of this there is also an additional R divider value by which the PLL clock frequency F_{PLL} can be divided further to generate an even more fine grained output frequency. Valid values for the R divider are 1, 2, 4, 8, 16, 32, 64 and 128. Putting all this together we have

$$F_{OUT} = \frac{F_{XTAL}\left(m + \frac{n}{d}\right)}{\left(a + \frac{b}{c}\right)R} \tag{3.3}$$

where F_{OUT} is the output frequency from the clock generator. It is now time to find a set of variables that can produce our wanted output frequency F_{OUT} of 2.8224 MHz. Lets us start by writing the output frequency as a fraction:

$$2.8224 \cdot 10^6 = \left(2 + \frac{514}{625}\right) \cdot 10^6. \tag{3.4}$$

If we put the numerical values for the the output frequency F_{OUT} and the oscillator frequency F_{XTAL} into Equation 3.3 we then get the expression

$$\left(2 + \frac{514}{625}\right) \cdot 10^6 = \frac{25 \cdot 10^6\left(m + \frac{n}{d}\right)}{\left(a + \frac{b}{c}\right)R}. \tag{3.5}$$

Dividing both sides with F_{XTAL} and then simplifying the left hand side by removing the factors of ten to the power of six leaves us with

$$\frac{\left(2 + \frac{514}{625}\right)}{25} = \frac{\left(m + \frac{n}{d}\right)}{\left(a + \frac{b}{c}\right)R}. \tag{3.6}$$

The numerators on the left and right hand sides both have the same format. We could start out by setting $m = 2$, $n = 514$ and $d = 625$ would it not be for the fact that this would put $\left(m + \frac{n}{d}\right)$ outside the permitted range of 15–90, so a little bit of adjustment is needed before we can proceed. Multiplying the numerator with for example eight gives us

$$8 \cdot \left(2 + \frac{514}{625}\right) = \left(22 + \frac{362}{625}\right) \tag{3.7}$$

which is inside the permitted range. Multiplying the denominator with the same number we then have

$$8 \cdot 25 = \left(a + \frac{b}{c}\right) R. \tag{3.8}$$

A simple solution to Equation 3.8 is to set R, b and c equal to one, after which a then can be determined to be 199. The choice of $b = 1$ and $c = 1$ will also let us use the integer mode setting for the multisynth which produces less jitter compared to the fractional mode setting. Fractional mode is however still used in the generation of the PLL clock. We now have a complete set of variables for Equation 3.3 that will produce $F_{OUT} = 2.8224\,\text{MHz}$ and we can proceed with the creation of the registry values that need to be programmed into the clock generator to set the desired output starting frequency.

PLL A was selected to be used as the source for multisynth 0, which through the R0 divider is used for the creation of the clock signal at the CLK0 output. The settings for PLL A reside at registers 26–33. Table 3.3 shows the registry mapping for the PLL A configuration settings.

Register	Register contents
26	MSNA_P3[15:8]
27	MSNA_P3[7:0]
28	Bits [7:2] reserved & MSNA_P1[17:16]
29	MSNA_P1[15:8]
30	MSNA_P1[7:0]
31	MSNA_P3[19:16] & MSNA P2[19:16]
32	MSNA_P2[15:8]
33	MSNA_P2[7:0]

Table 3.3: PLL A settings in the register mapping.

Register values for one of the PLLs can be calculated by the use of:

$$MSNA\_P1[17:0] = 128 \cdot m + \left\lfloor 128 \cdot \frac{n}{d} \right\rfloor - 512 \tag{3.9}$$

$$MSNA\_P2[19:0] = 128 \cdot n - d \left\lfloor 128 \cdot \frac{n}{d} \right\rfloor \tag{3.10}$$

$$MSNA\_P3[19:0] = d \tag{3.11}$$

The settings for multisynth 0 are found at registers 44–51 and Table 3.4 shows the register mapping. R0_DIV is the setting for the extra R divider through which the clock signal passes on its way to the CLK0 output and MS0_DIVBY4 is used to indicate if a divider value of four is used in the multisynth.

Register	Register contents
42	MS0_P3[15:8]
43	MS0_P3[7:0]
44	Bit [7] N/A & R0_DIV[2:0] & MS0_DIVBY4[1:0] & MS0_P1[17:16]
45	MS0_P1[15:8]
46	MS0_P1[7:0]
47	MS0_P3[19:16] & MS0_P2[19:16]
48	MS0_P2[15:8]
49	MS0_P2[7:0]

Table 3.4: Multisynth 0 settings in the register mapping.

The register values for the multisynth 0 registers can be determined by

$$MS0\_P1[17:0] = 128 \cdot a + \left\lfloor 128 \cdot \frac{b}{c} \right\rfloor - 512 \tag{3.12}$$

$$MS0\_P2[19:0] = 128 \cdot b - c \left\lfloor 128 \cdot \frac{b}{c} \right\rfloor \tag{3.13}$$

$$MS0\_P3[19:0] = c \tag{3.14}$$

When using integer mode for the multisynth, Equations 3.12, 3.13 and 3.14 can be simplified to

$$MS0\_P1[17:0] = 128 \cdot a - 384 \tag{3.15}$$

$$MS0\_P2[19:0] = 0 \tag{3.16}$$

$$MS0\_P3[19:0] = 1 \tag{3.17}$$

The Si5351 chip on the clock generator board will be programmed at startup using the procedure described in Figure 3.7. Values for PLL A and multisynth 0 in registers 26–33 and 44–51 are generated using the methods described in this chapter and for the other remaining registers the values listed in Section B.1 of Appendix B that were used for the asynchronous mode implementation are used here again. During operation when the clock frequency needs to be adjusted, we are only going to update the register values for multisynth 0 which can be done without having to disable the CLK0 output and the PLL does not have to be reset either, so switching clock frequencies should not result in glitches in the music playback.

To follow the data rate so that the clock frequency of the I^2S component can be updated to match it, we can monitor the buffer fill level of the audio buffer in the PSoC RAM. Updates can be done either at fixed time intervals using for example a count of the SOF pulses as trigger, or we can choose to make adjustments only when

the buffer fill level has drifted some predefined distance away from the center of the buffer. We should avoid making unnecessary frequency switches, but presumably it is better to make multiple smaller frequency adjustments instead of making one big frequency jump as the former will allow us to follow the data rate more closely and it should be less likely to be noticed.

3.3.4 Synchronous Mode Implementation

In the synchronous mode the connected USB audio device must synchronize its sample clock to the USB start-of-frame frequency. It is normally done by the use of a programmable PLL. In our hardware we have the internal PLL in the PSoC and the two PLLs inside the Si5351 on the clock generator board. The Si5351 mounted on the Adafruit external clock generator board is the A version of the chip which lacks a clock input, so there is no way to feed an external clock signal into one of its internal PLL blocks. There is a version C of the Si5351 which does have an external clock input, but it only accepts signals that are within the range 10 MHz to 100 MHz, so the 1 kHz SOF signal would not be possible to route directly into that version of the chip either. The same goes for the PSoC PLL which only accept input signals in the range of 1 MHz to 48 MHz. There is thus no way to feed the SOF clock directly into any of the PLLs in our hardware. Another clock of higher frequency which follows the SOF generation rate must therefore instead be selected to be routed into one of the PLLs. If we count the number of cycles for the selected high frequency clock during each SOF period, then we can determine a variable that describes the relation between the SOF and the high frequency clock that we choose to feed into the PLL. Using the 24 MHz IMO as an example, if the host clock and the local clock both are running at the exact same pace, then the clock count during one SOF will read 24 000, but if the local clock is a little bit faster, then the counter might read for example 24 001 instead. To match the local clock frequency to the current SOF rate we need to multiply the local clock with the ratio of the local clock frequency divided by the counter value for the SOF period multiplied by 10^3. What the host side perceives to be one full SOF would then be matched by a local clock rate of $\frac{24 \cdot 10^6}{24\,001 \cdot 10^3} = 23\,999\,000.04\,\text{Hz}$.

The design decision to use the internal PLL of the PSoC in favor of the PLL blocks in the Si5351 chip was then made. This allowed for the external clock generator board to be removed completely from the design. A fractional divider was however still needed to be able to create the clock frequencies necessary, so a custom fractional divider component was designed in Verilog and implemented using the PSoC digital building blocks. With the USB block of the PSoC needing a 24 MHz reference clock, the clock source to generate the 2.8224 MHz clock for the I^2S component was selected to be the IMO set to 24 MHz. This is also the only IMO setting that can produce a clock of high enough accuracy to run the USB component, so there was no other option available. To get the correct starting frequency for the I^2S component, the IMO clock needs to be divided by the fraction

$$\frac{24000 \cdot 10^3}{2.8224 \cdot 10^6} = 8 + \frac{7104}{14112} = 8 + \frac{74}{147}. \tag{3.18}$$

This is our starting value and the value that will be used when the counter capture is exactly 24 000. The custom fractional divider component uses the accumulative method described in Chapter 2.6 to create a new clock which then is fed into the PSoC PLL. With $N = 8$, $K = 74$ and $F = 147$ put into Equation 2.25 it means that the input clock would need to be divided 74 times by nine and 73 times by eight to produce the correct output frequency. Should the captured value instead be 24 001 or 24 002, then the fractions to use for the calculations will become

$$\frac{24001 \cdot 10^3}{2.8224 \cdot 10^6} = 8 + \frac{7109}{14112} \tag{3.19}$$

and

$$\frac{24002 \cdot 10^3}{2.8224 \cdot 10^6} = 8 + \frac{7114}{14112} = 8 + \frac{3557}{7056}. \tag{3.20}$$

The smallest common denominator for the fractions produced by any capture value near the starting frequency is 14 112, so that will be the actual F value that we will be using. Calculating a couple more plausible clock capture values by hand, it becomes apparent that for each ± 1 that is registered by the counter, the value of K will increase or decrease by five. In reality this means that five more division cycles will now use a divider value of nine instead of eight or vice versa. This will also make the total period of all the divide by eight plus all the divide by nine cycles five input clock pulses longer or shorter. The attribute of the full division cycle being of a length that varies along with the captured clock values is rather non-wanted. It was therefore decided to update the K value using a fixed time interval instead of running it one full cycle at a time. The output clock from the fractional divider component will therefore not be fully precise to the SOF rate but it should provide a good enough approximation. There is also a second reason to why this approach was taken and that is because using larger numbers in the Verilog code will consume many UDB blocks up until the point where there are none left, so by being able to skip the variable that keeps track of the total cycle count, the design can be made to fit on the PSoC 3 without consuming all the digital building block resources.

A problem with the design that became apparent was the fact that to divide a clock by an uneven number, the switch from high to low in the output signal must then be made on a falling edge of the source clock. The recommendation in the documentation [39] from Infineon is to not use the negative edges as a trigger in Verilog as it can cause timing and synchronization problems. To work around this issue, the positive edges of a source clock with double the frequency can instead be used but our only possible IMO setting was to run it at 24 MHz. Instead the output frequency can be divided in half as it still will be a multiple of the I^{2}S component clock and the frequency will still be over the 1 MHz minimum PLL input clock limit. What we are going to do here is to divide the input clock $K = 7104$ times by $N + 2 = 18$ and $F - K = 7008$ times by $N = 16$. The actual fraction of $24\,000 \cdot 10^3$

divided by $1.4112 \cdot 10^6$ is $\left(17 + \frac{96}{14112}\right)$ and it would have resulted in division of the input clock $F = 96$ times by $N + 1 = 18$ and $F - K = 14\,016$ times by $N = 17$. Comparing the total number of input clock cycles needed to complete a full set of both of these two division cases, it is clear that the amounts are equal. Using the dividers $N = 16$ and $N + 2 = 18$ instead of the more common parameter selection of $N = 17$ and $N + 1 = 18$ should therefore be possible as long as the number of times we divide by each is correct.

The setup through which the clocks in the system are generated is shown in Figure 3.14. It is not possible to start directly with the pictured configuration as the DSI block has not been fully configured when the system starts. The IMO was therefore set to be the initial input signal for both the PLL and MCKL blocks. Once the DSI block getting its clock from the fractional divider component is up and running, the PLL can then be stopped and reconfigured to use the DSI signal as its input. After also changing the $\frac{P}{Q}$ ratio of the PLL, it can be started again and used as the input signal for the MCKL block. The output frequency from the fractional divider is initially set to 1.4112 MHz and the PLL block then multiplies this by 40 to produce the master clock running at 56.448 MHz. Part of the fractional divider component is also a control register that allows us to adjust the output frequency to match the SOF rate.

Figure 3.14: Clock configuration for the synchronous mode implementation.

4

Results

The results from the different synchronization mode implementations are presented in this chapter. This includes plots from data logged over UART, jitter histograms based on oscilloscope measurements, and other observations gathered during testing and development.

4.1 Functional Results

Functional results for the implementations are presented in the following subsections. The plots are generated from data logged from the PSoC device over UART.

4.1.1 Functional Results for Asynchronous Mode

When first taking a look at the communication between the host and the PSoC device during normal operation, it was noticed that most of the feedback values were slightly below the expected average of 44.1 kHz. Calculation of the feedback value for exactly 44.1 kHz by hand using the method listed in Section 3.3.2 gives the result {0x0B, 0x06, 0x66}, but the packets captured by the Wireshark network protocol analyzer software were mostly containing feedback values of {0x0B, 0x06, 0x50}, and {0x0B, 0x06, 0x40}. Later by the use of an oscilloscope the external clock generator was confirmed to run a bit slower than what the configuration values used would suggest.

In one of the earlier versions during development, the counter component that was used to capture the clock count of the I^2S clock was configured to be reset and restarted after each captured value. This caused it to miss some of the clock pulses on its count input so that the captured value was not accurate all the time and the buffer fill started drifting away from the center of the buffer, eventually leading to the buffer running out of data. Letting the counter run continuously looping over the terminal count and then comparing the most recently captured count value to the previous one instead of resetting the counter for each capture turned out to work much better with the buffer fill then staying consistently between 45 % and 55 %. Figure 4.1 shows a logged playback session of 4 hours in length where the buffer fill stays perfectly at half the buffer size. The total buffer size for the audio buffer in the PSoC RAM was set to 5120 byte during this test.

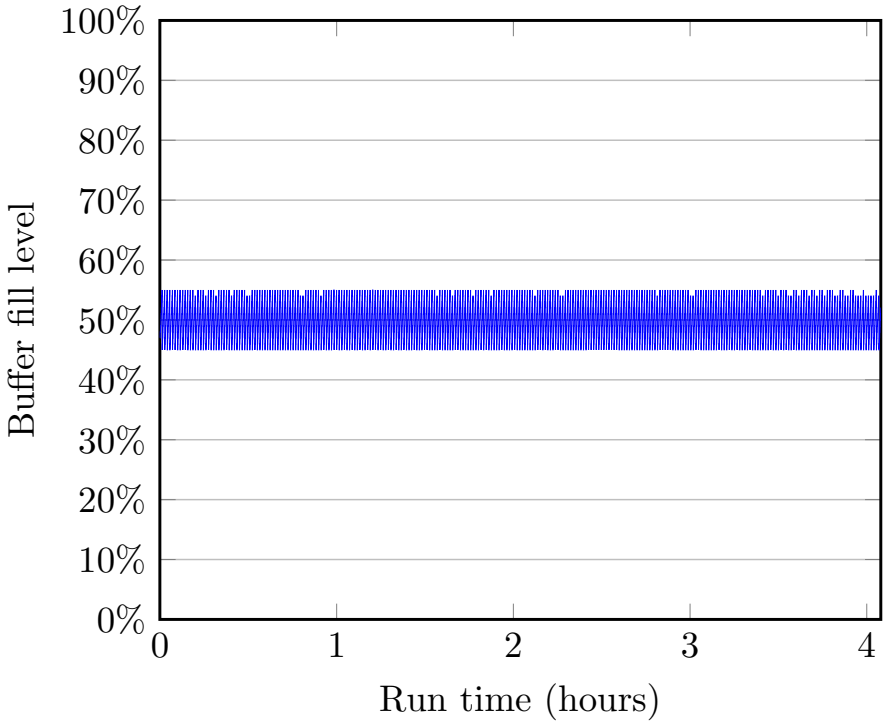


Figure 4.1: Buffer fill level during four hour long playback session using asynchronous mode.

Judging by the logged data, the total size of the buffer in the PSoC RAM would only need to be around 10 % of its current size. Making the buffer smaller would decrease the buffering delay, but keeping the bigger buffer adds a bit of safety margin against buffer underrun and overrun conditions. For the bigger buffer size to be most useful, the design should then also include a mechanism that steers the buffer fill level back towards the middle of the buffer if it gets offset from it due to an error, and that can be done by adjusting the feedback value sent back to the host. It was found easiest to add or subtract a constant from the counter capture value of approximately 45 158, gradually increasing or decreasing the adjustment value the further away from the center of the buffer that the buffer fill level gets. Table 4.1 shows the selected adjustment profile that was used.

Buffer fill level	Capture value adjustment
>90 %	-3
80–90 %	-2
60–80 %	-1
40–60 %	0
20–40 %	+1
10–20 %	+2
<10 %	+3

Table 4.1: Adjustment of the capture value depending on buffer fill level.

In Figure 4.2 we see the slowly accumulating error created by resetting the I^{2}S clock counter instead of letting it run continuously, which then eventually leads to the buffer reaching the underflow state. In Figure 4.3 is the same test run again, now with the capture value being adjusted depending on the buffer fill level. As can be seen, the buffer fill level first starts to drift away from the middle of the buffer as the error accumulates, but once it reaches the first adjustment level of 40 % the drifting stops and the lowest levels of the buffer fill value do not go much below the 40 % mark. With some adjustment and a bit of tighter control, the buffer fill level could be pushed back closer to the center level of the buffer space, but we can now conclude that protection against errors can be had using a simple method that does not require much in terms of computing performance. The test also revealed that the buffer space is not being used to its full potential. The device is set to go into the operational underflow mode when the I^{2}S component has completed transmitting a chunk of audio data and the next full chunk is not yet available in the RAM buffer. With the buffer in RAM divided into only ten data chunks, underflow will essentially occur when there still is around 10 % of the RAM buffer left unused. The plot shows that underflow happened slightly before the 10 % buffer fill mark was reached, but this is not unexpected as the values were logged at intervals and were thus not precisely following the buffer fill level. We do not want to start transmission of the next buffer chunk before all of the audio data is in the RAM buffer even if most of the time the missing data would get populated before it is due for transmission. What we can do is to increase the number of chunks in the RAM buffer. Doubling them would put us closer to only 5 % of the buffer not being utilized and by quadrupling we would be down to 2.5 %. Up to 128 transaction descriptors are available in the PSoC so running out of them should not be an issue. Trying both of these two suggested configurations while keeping the same total buffer size by making the buffer chunks smaller by the same amount that their numbers increased with showed that dividing the buffer into 40 memory chunks with accompanying transaction descriptors was too much for the PSoC to handle performance wise, but using 20 worked out fine. Figure 4.4 shows a test run with the accumulating error again with our increased number of buffer chunks but still the same total buffer size. We now get below the 10 % buffer fill mark before underflow occurs and the short range top to bottom variation of the buffer fill level has also gone down a bit. Running a test session without the accumulative error again shows in Figure 4.5 that we now only use around 5 % of the total buffer size of 5120 bytes when increasing the number of buffer chunks from ten to twenty. In Figure 4.4 it can be seen that the starting value of the buffer fill level has been shifted slightly below the 50 % mark only as a result of changing the data chunk size and the number of buffer elements. A small adjustment to the compare value of the buffer fill level at which the audio playback starts needed to be made to get the buffer fill level to center closer to half the buffer size. The buffer fill level in Figure 4.5 does still appear to be a little off center even after the adjustment as it is difficult to divide the 5 % top to bottom range evenly on both sides of the 50 % mark with the accuracy of the logging being ± 1 %. It is up to the designer to decide if a smaller buffering delay or added safety against buffer underflow and overflow should be prioritized or some combination thereof when selecting the size of the RAM buffer.

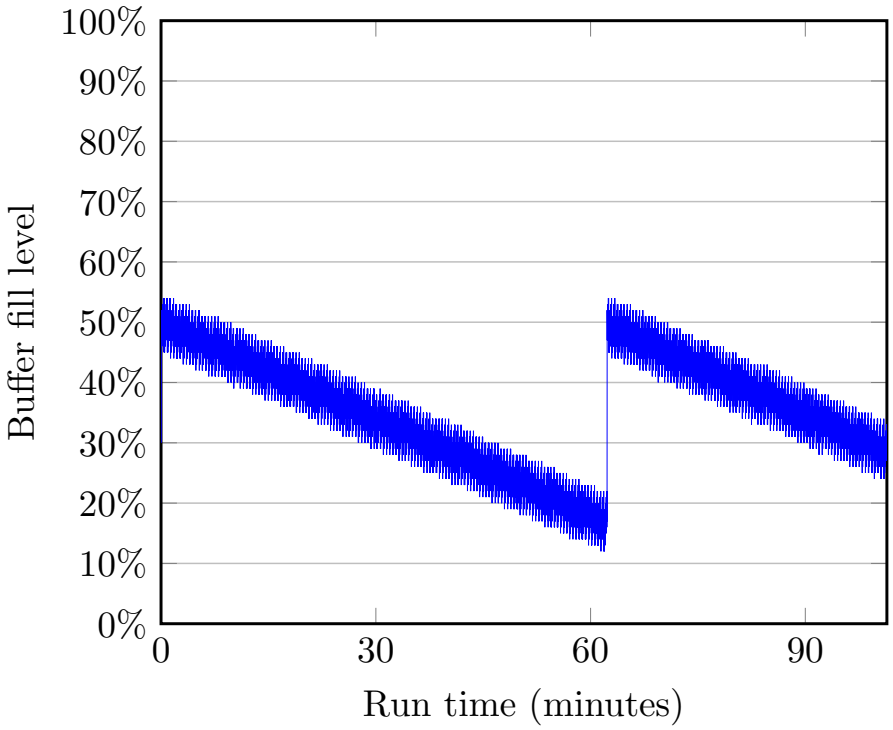


Figure 4.2: Accumulative error without correction of buffer fill.

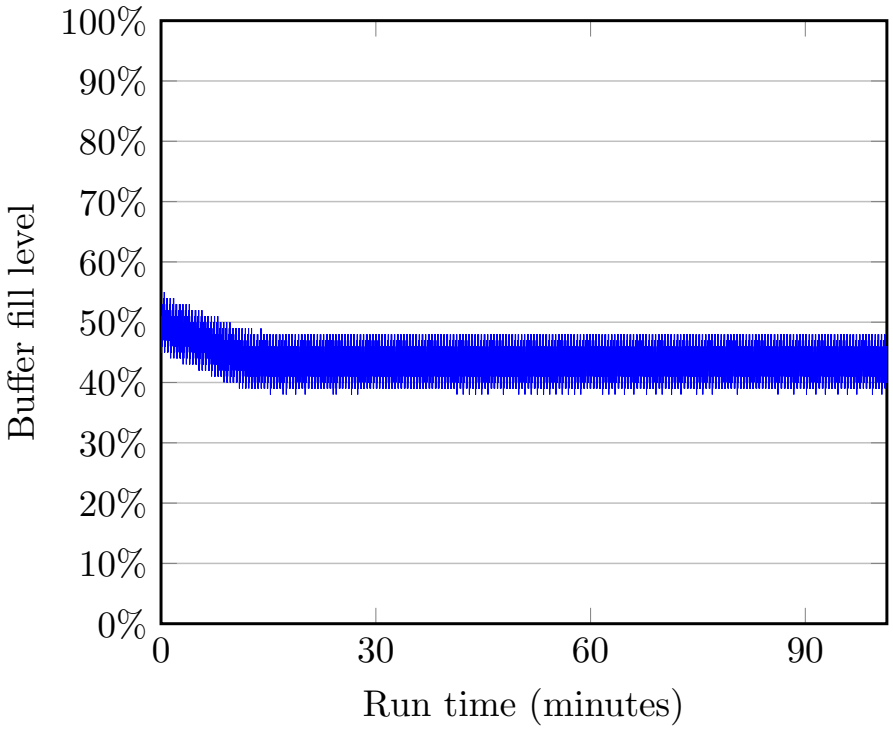


Figure 4.3: Accumulative error with correction of buffer fill.

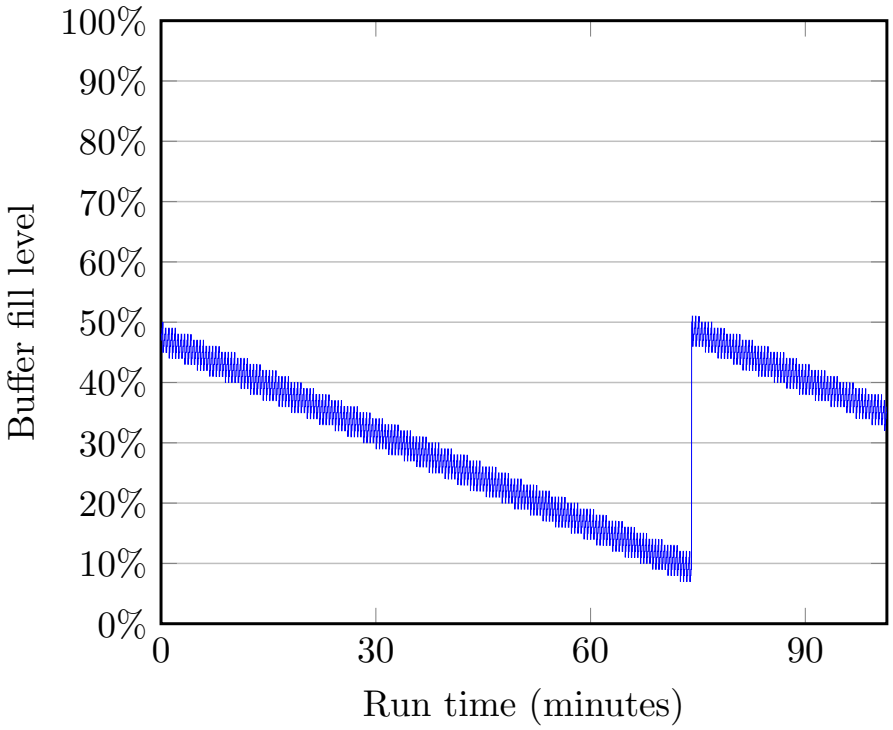


Figure 4.4: Accumulative error without correction and with increased number of buffer chunks.

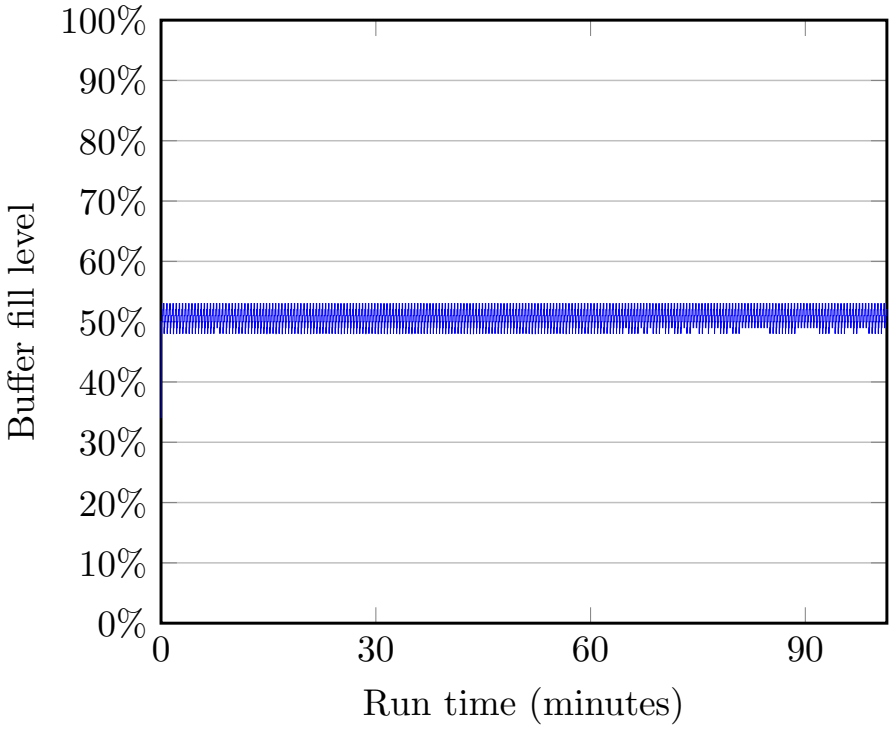


Figure 4.5: Increasing the number of buffer chunks to twenty for asynchronous mode.

4.1.2 Functional Results for Adaptive Mode

When the buffer level has drifted away from its equilibrium state close to the center of the buffer, the clock frequency of the Si5351 needs to be updated to better match the data consumption rate. The same update interval of once every 16^{th} SOF that was used for the feedback value in the asynchronous mode case was also used here as the monitoring interval for the buffer fill level and the total buffer size was also the same, split into ten memory chunks of 512 bytes each. Choosing to adjust the clock by only changing the variable a in the multisynth settings and keeping b, c and R constant using the values from Section 3.3.3 results in a minimum adjustment step size of around 0.5 % of the total frequency, which is an unnecessary big jump in frequency. Multiplying the numerator and denominator of Equation 3.6 with some other value than eight would make it possible to place the variable a closer to the middle of the valid range for $\left(a + \frac{b}{c}\right)$ of 8–2048, giving a minimum frequency step size of around 0.1 % for adjustments while still keeping the multisynth in integer mode. A normal feedback value update for the frequency adjustment in the asynchronous mode implementation was however only around 0.002 % of the sample frequency, so to keep things comparative and similar, the decision to opt for a switch from integer to fractional mode for the multisynth was made even though this meant there would be more jitter in the output signal from the clock generator board. To get to the same adjustment level size here in our adaptive mode implementation we can keep the variables $a = 199$ and $R = 1$ unchanged in Equation 3.6 and set b and c to 250. Adjusting b by $\pm$ 1 will then give us a frequency jump size in the same ballpark as for the asynchronous mode case.

Updating the clock generator settings for the multisynth using dynamically generated values during runtime turned out to be difficult as the PSoC 3 had a hard time keeping up with the flow of audio data and any other operations it had to perform when at the same time now also having to calculate new values and update the register map. Even using only the simplified formulas in Equations 3.15, 3.16 and 3.17 for integer mode and only updating the multisynth registers where the value changed still resulted in audible distortion and consequently failure because the PSoC 3 could not handle the added workload. There is not much else that can be optimized without making more far-reaching alterations to the design and doing things like switching off logging of data would make it difficult to verify any results. Therefore it became necessary to use a lookup table with pregenerated register values for the clock adjustment of the multisynth block.

In the tests for asynchronous mode it was noted that the external clock generator frequency was a bit slower than the host clock, and it was the same here for both the integer and fractional mode multisynth settings generated by manual calculations for the adaptive synchronization mode. Figure 4.6 shows how the buffer fill level with the multisynth set to integer mode drifts off from the center of the buffer until it overflows due to the frequency difference between the host clock and the clock produced by the external clock generator board, if no frequency adjustments are made. In Figure 4.7 the buffer fill level starts to drift, but then it gets pushed back towards the middle of the buffer by the frequency adjustments. The same

test runs with the multisynth set to fractional mode are displayed in Figure 4.8 and Figure 4.9. In Table 4.2 the clock adjustment levels for both the integer and fractional multisynth mode test runs are shown and the calculated register values are listed in Section B.2 of Appendix B.

Buffer fill level	Adjustment in integer mode	Adjustment in fractional mode
>90 %	-2.0 %	-0.008 %
80–90 %	-1.5 %	-0.006 %
70–80 %	-1.0 %	-0.004 %
60–70 %	-0.5 %	-0.002 %
40–60 %	No adjustment	No adjustment
40–30 %	+0.5 %	+0.002 %
30–20 %	+1.0 %	+0.004 %
20–10 %	+1.5 %	+0.006 %
<10 %	+2.0 %	+0.008 %

Table 4.2: Clock frequency adjustment based on buffer fill level.

We can now establish that the external clock frequency updates are working as expected, keeping the buffer fill level within the buffer limits. The frequency switches also appear to be glitch-free as no audible distortion in connection with the register writes could be detected when listening to the audio playback. A simple method with fixed frequency levels was selected to minimize the computational work needed to update the clock generator registers, but ideally a method that would guide back the buffer fill level even closer to the center of the buffer would be preferable. Here we simply stop the progression of the drift and for testing purposes this is enough as it keeps us within the buffer limits while also letting us view how the frequency adjustment size affects the buffer fill level. With the main goal to make the frequency jumps as small as possible and also remembering that the feedback values from which the frequency adjustment levels in the fractional mode setting were calculated from are only feedback values and not the actual frequency adjustments that the host makes to the clock, one could presume that the frequency adjustment levels could be made smaller than the 0.002 % that was used here with the multisynth set to fractional mode. How low you can go depends on other parameters such as how stable the host clock is and on the selected update interval for the register writes, so no in-depth tests were done to find a lower limit frequency adjustment range. Minimizing the time between frequency adjustments was of secondary priority and some testing using an update interval other than once every 16^{th} SOF was tried with the conclusion that there is room to increase or decrease the update interval if necessary, at least when continuing to use the same frequency adjustment levels as before. Focus was now instead directed at seeing how the frequency adjustments of the external clock generator board affected the jitter levels.

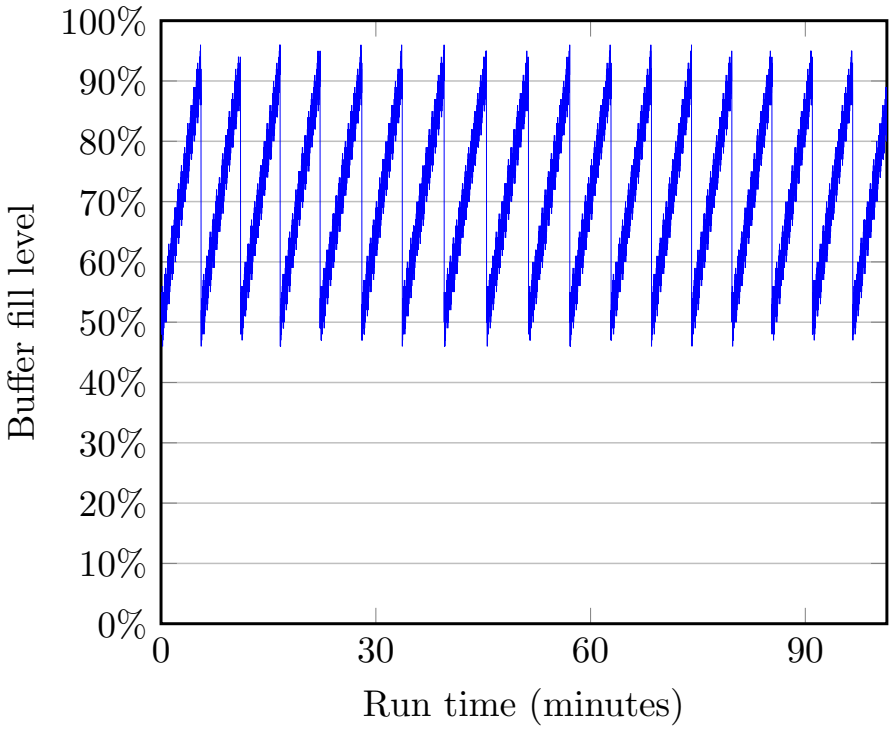


Figure 4.6: Buffer fill level for USB in adaptive mode with the multisynth in integer mode without clock adjustment.

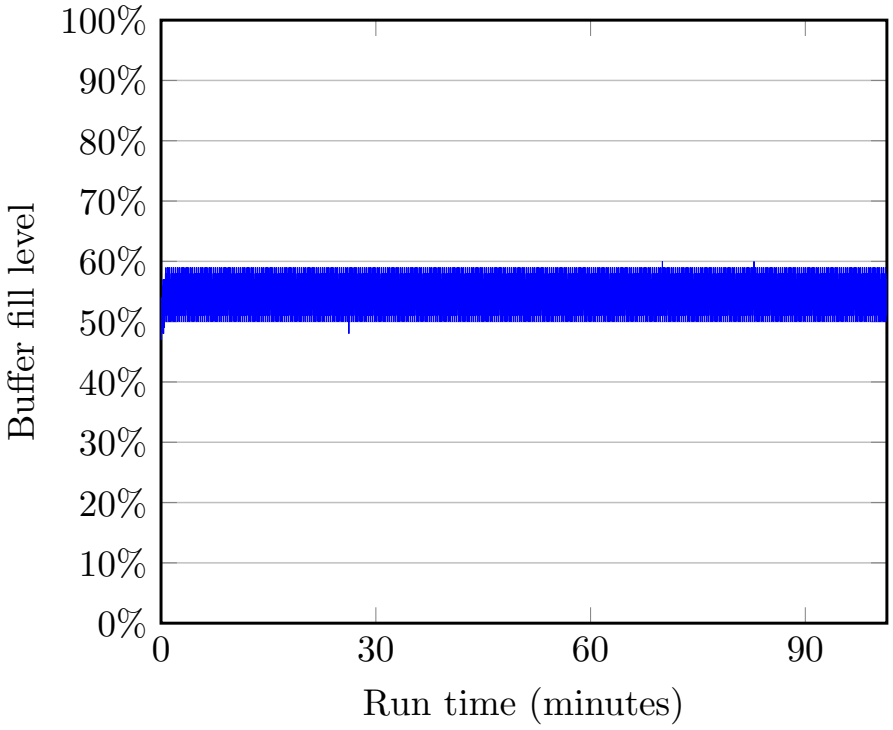


Figure 4.7: Buffer fill level for USB in adaptive mode with the multisynth in integer mode and clock adjustment activated.

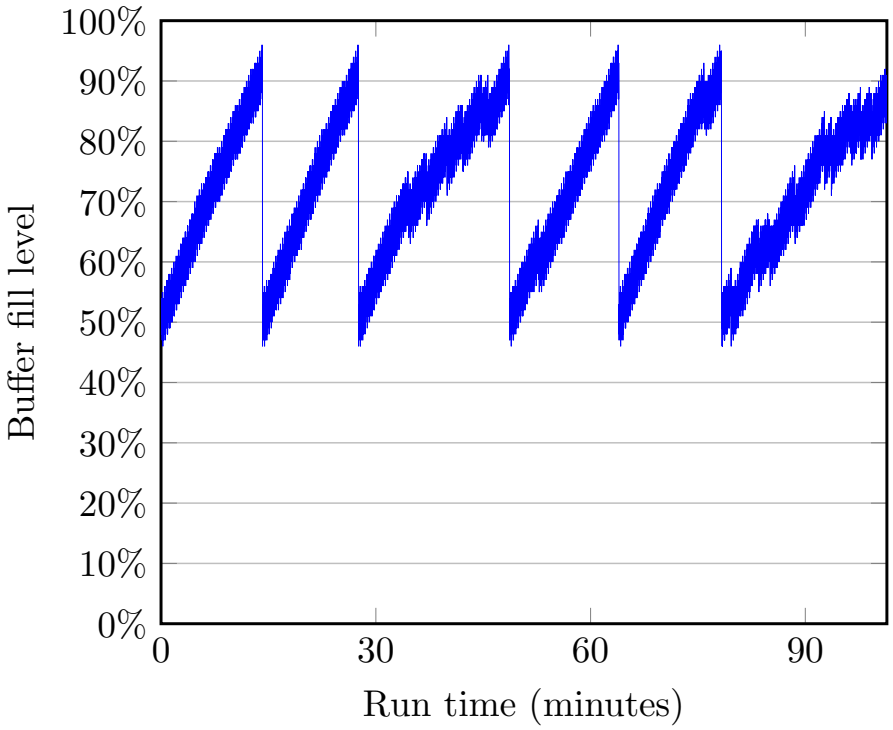


Figure 4.8: Buffer fill level for USB in adaptive mode with the multisynth in fractional mode without clock adjustment.

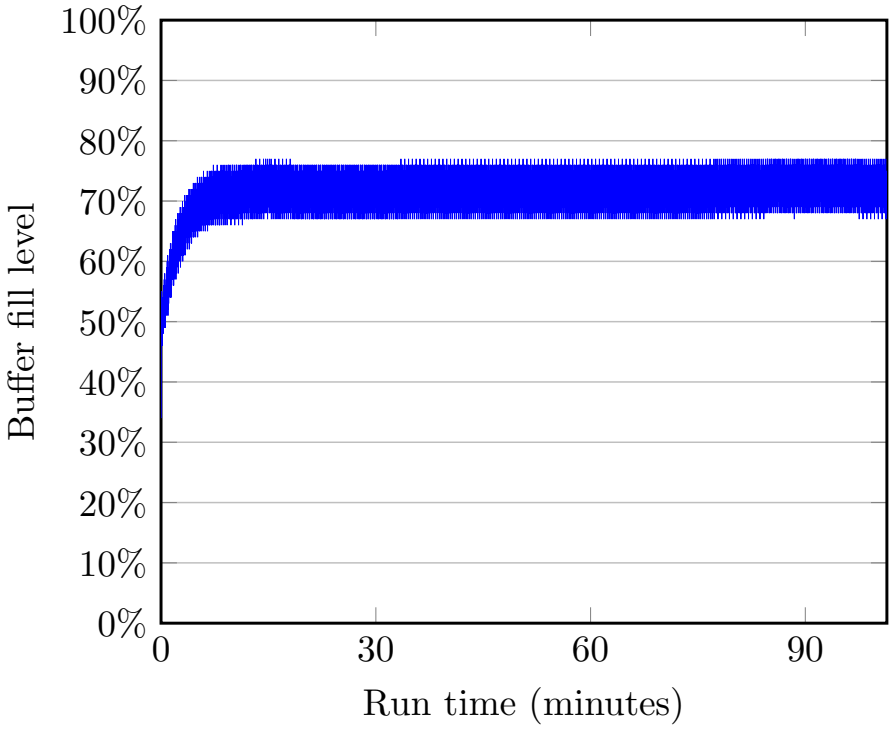


Figure 4.9: Buffer fill level for USB in adaptive mode with the multisynth in fractional mode and clock adjustment activated.

4.1.3 Functional Results for Synchronous Mode

An initial test of the custom fractional divider component with the 24 MHz IMO used as input resulted in an output frequency of around 1.4 MHz as expected. The frequency reading on the oscilloscope fluctuated a bit so giving a more exact frequency number was difficult but when replacing the external clock source with the clock generated by the custom fractional divider setup described in Figure 3.14 in the asynchronous mode implementation, feedback values between {0x0B, 0x06, 0x30} and {0x0B, 0x06, 0x70} were observed. This is in line with the nominal feedback value of {0x0B, 0x06, 0x66} for the audio sample frequency 44.1 kHz. Following the clock generation chain backwards, a more exact reading of the output frequency from the fractional divider component referenced to the host clock would then be between 1.411 093 75 MHz–1.411 218 75 MHz which can be compared with the wanted nominal value of 1.4112 MHz. With the initial frequency set up, it was then time test how the component would respond to frequency adjustments made by writing updated values to the control register in the fractional divider. The control register is made available for CPU writes by API calls and the size of one control register is 8 bits. When monitoring the variation of IMO clock cycles counted during a SOF, the amount stayed within the range 24 000±16. One single control register is therefore enough to communicate the change in frequency if only the deviation from the nominal frequency is reported. Having confirmed the fractional divider component functioned correctly, the synchronous mode design was then implemented incorporating the custom fractional divider component.

Figure 4.10 shows a test run of the synchronous mode implementation and for reference in Figure 4.11 is a test run of the same design with the frequency updates turned off. The update interval was set to once per SOF and the same buffer size as before was configured using 10 buffer chunks of 512 bytes each. The update interval was set lower than in the other two synchronization mode implementations simply because the 16 bit counter used in all the designs would not have been large enough for the IMO clock at 24 MHz if the capture interval would have stayed the same at once every 16^{th} SOF. Looking at the plot in Figure 4.10, the buffer fill level is fairly consistent but it does decrease even if just by a little as time goes by. The rate of change appears to be around 1 % per hour and if that number is correct and consistent, then the buffer should run out of data in approximately two days time if music is being played continuously and no other compensation is applied. The cause of this discrepancy has not been determined but possible reasons could be that clock cycles of the IMO are missed by the counter due to synchronization issues, or that using a fixed length update period instead of running one full set of N and $N + 2$ clock cycles at a time produces a small error that accumulates over time. Using a fixed length update period was necessary to limit the number of UDB blocks used for the fractional divider component. Otherwise it would not have been possible to make it fit in the implementation without exceeding the number of available UDB blocks as some of the other peripherals also need access to them.

In Figure 4.11 where the frequency adjustments have been turned off, the first part of the plot looks a bit inconsistent. From around 30 minutes up until the end of the

measurement series, the host clock is consistently faster than the clock generated by the PSoC, resulting in many buffer overflow conditions. Both clocks are however much closer in frequency in the first part of the plot, giving rise to the slightly strange looking behavior. Any particular reason as to why the frequency deviation varies and which of the clocks if not both it is that is changing in frequency has not been determined. Dependency to temperature and supply voltage in clock generation circuits and other factors such as host system load level may possibly affect the clock frequencies and the generation rate of SOFs.

The range given for SOF generation in the USB specification is 1.0 ms±500 ns, so a bit of variation can be expected. Figure 4.12 shows the number of IMO clock cycles registered by the counter component in the PSOC per SOF during a typical test run of the synchronous mode implementation. If we assume that the clock in the PSoC is totally accurate, then all the registered SOFs would have arrived within the interval 1.0 ms±700 ns, but most likely the PSoC clock will have fluctuated a bit so the SOFs may very well have been received within the interval presented in the USB specification. Needless to say, there is going to be variation in the arrival times of the SOFs, so using them to synchronize the device clock to the host clock as is done in synchronous mode USB is expected to result in added jitter.

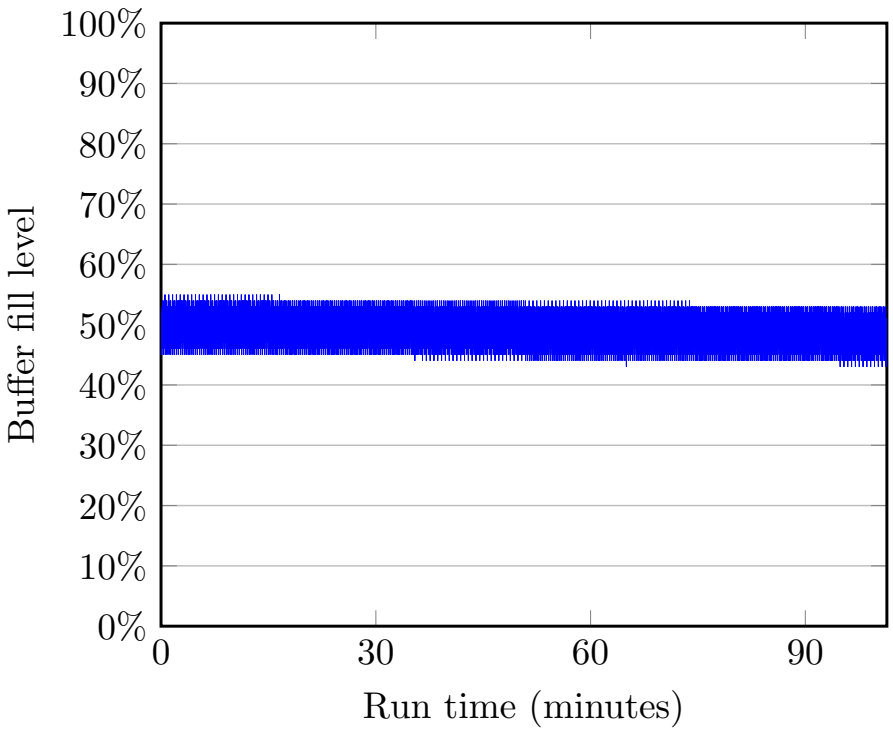


Figure 4.10: Buffer fill level for USB in synchronous mode with frequency updating turned on.

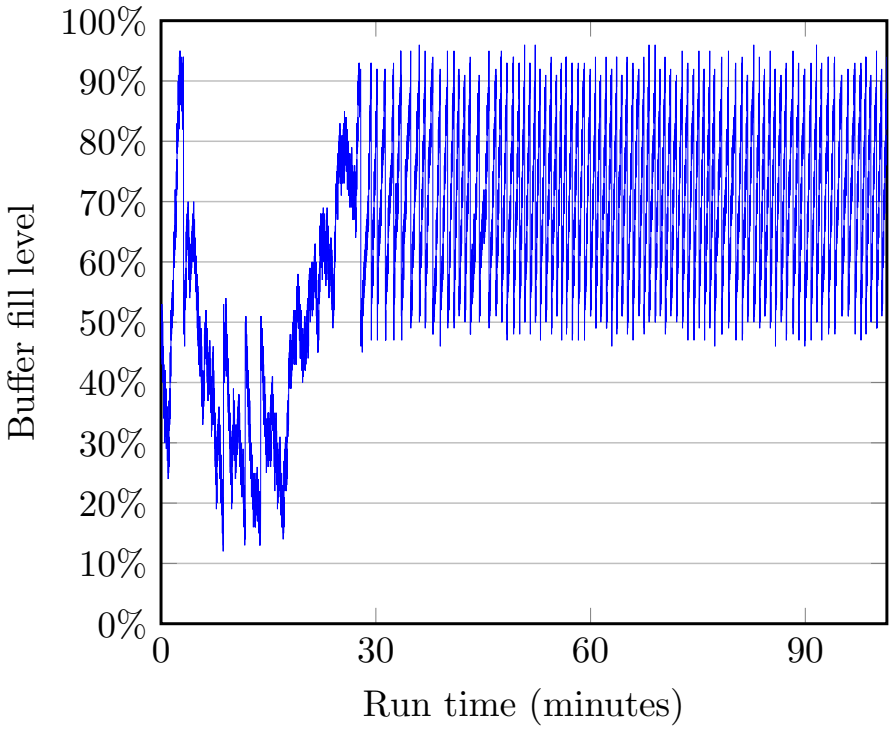


Figure 4.11: Buffer fill level for USB in synchronous mode with frequency updates turned off.

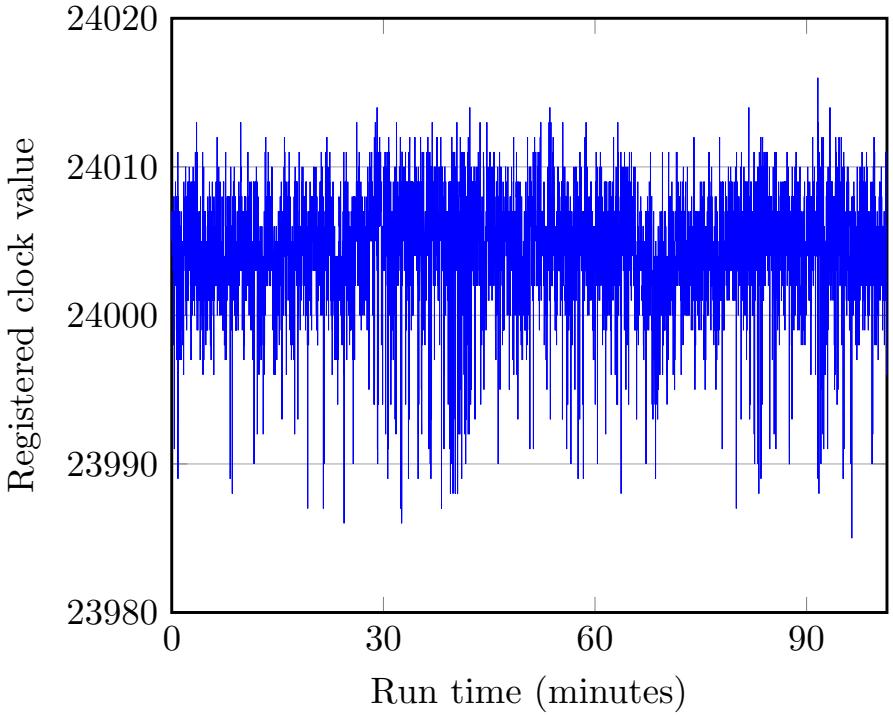


Figure 4.12: PSoC IMO clock pulses registered per SOF.

4.2 Jitter Measurements

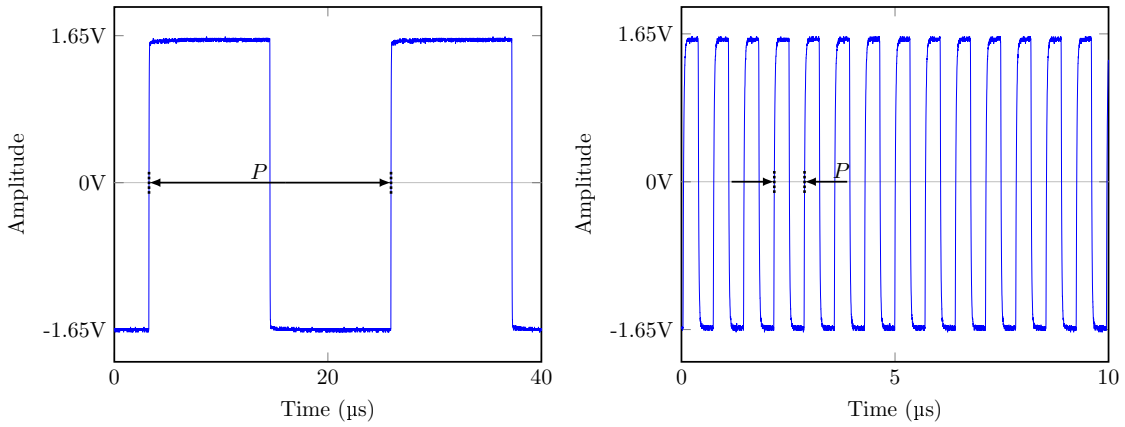
Jitter measurements for all three USB synchronization modes are presented in this section along with comparisons and discussion of the results. The collected data comes from digital real-time oscilloscope measurements, out of which period and cycle-to-cycle jitter histograms then were created. The oscilloscope used for measurements had a bandwidth of 100 MHz, 1 GSa/s sample rate and 10k sample depth, so it was possible to use it to measure any of the signals in the different implementations but for some of the signals with higher frequencies, the measurements became somewhat coarse as the smallest sample interval that can be used for data export is 1 ns. The clock signals going into and coming out of the I^{2}S component inside the PSoC being on the slower side regarding frequency were selected as suitable for doing comparisons as they are present in all implementations and have the same nominal frequencies in all of them. It is also presumable that jitter present in any higher frequency signal from which the I^{2}S clocks are being sourced will propagate down the line to the slower clock signals [14].

To get the best resolution, measurements were done using the lowest possible sample interval. For the WS signal which is the lowest frequency audio clock in the system with a frequency of 44.1 kHz, this meant setting the sample interval of the oscilloscope to 4 ns to be able to capture at least one full clock cycle. In retrospective it may seem a bit excessive to produce measurement series where one clock cycle is composed of more than 5600 sample points, but tests where longer sample intervals were used produced jitter values for the WS signal that were much higher than with the 4 ns sample rate. The downside to just capturing one clock cycle at a time as in Figure 4.13a apart from having to perform more measurements is that only the period jitter can be extracted from the measurement as cycle-to-cycle jitter calculations require at least two full consecutive clock cycles. With cycle-to-cycle jitter values being presented for the SCK output and I^{2}S input clock signals and all three clocks being derived from the same source clock, not producing any cycle-to-cycle jitter measurements for the WS signal was decided to be an acceptable trade-off for a more reliable period jitter measurement.

The frequencies of the SCK and the I^{2}S input clocks are 1.4112 MHz and 2.8224 MHz respectively, so even with the oscilloscope set to the smallest possible sample interval of 1 ns, the minimum number of captured clock cycles per measurement will be 14 for the SCK and 28 for the I^{2}S input clock. This results in around 700 sample points being captured per full clock cycle for the SCK signal and 350 for the I^{2}S input clock, which should be plenty to give a reliable reading. Figure 4.13 shows an example each of a measurement of the SCK and WS output clock signals and Figure 4.14 contains an example of a measurement series of the I^{2}S component input clock.

Measurements were then repeated for each signal until a sufficient amount of clock cycles had been collected to be able to get a sense of the jitter distributions. The period time P was calculated from the exported data using the rising edge 0 V crossings as start and end points for the period and intersection of the signal with V=0 was found by linear interpolation of the closest sample points. The period values for

each of the signals were then collected in bins and plotted as histograms with the period P being rounded off to the nearest bin value. Cycle-to-cycle jitter was also calculated from the period values in accordance with the example in Figure 2.2 and plotted in histogram form for the I²S component input clock and the SCK output clock signals. Histograms for each synchronization mode and clock source used are presented in Appendix C.



(a) I²S WS output signal. **(b)** I²S SCK output signal.

Figure 4.13: Capture of one measurement each for the I²S output signals.

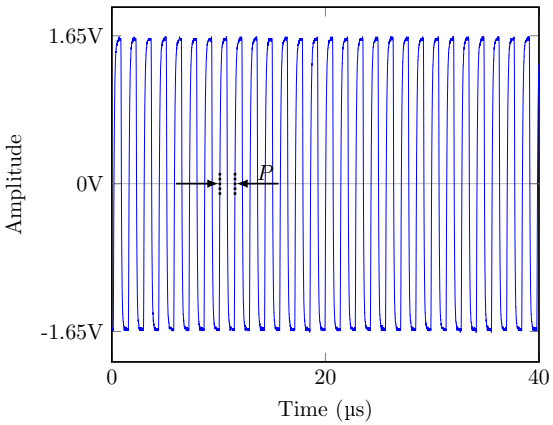


Figure 4.14: Capture of one measurement for the I²S input signal.

4.2.1 Discussion of Measurement Results

A summary of the period peak-to-peak and peak cycle-to-cycle values can be found in Table 4.3 with corresponding graphical representations in Figure 4.15 and Figure 4.16. We can start by comparing the measured values to the jitter hearing threshold levels presented in Chapter 2.1.5. The lowest reported threshold level in any of the three examined studies was the 5 ns average value for the 8 kHz sine tone with added sinusoidal jitter in the study from Benjamin and Gannon [14]. All

measurement values where the Adafruit clock generator board with the Si5351 chip was used as clock source are clearly below 5 ns and the jitter levels created by the clock generator board for those designs would therefore be expected to be inaudible regardless of the type of audio content played.

There is however one additional thing to consider for the USB adaptive mode measurements and that is that what has been captured by the oscilloscope possibly only is the jitter reading for one single clock adjustment level. If the clock at the host side is reasonably stable, then the PSoC will likely after an initial adjustment phase alternate back and forth between the two adjustment levels which are closest to the clock frequency of the host clock. For a period of time, the device clock will stay at one of the adjustment levels until the buffer level has drifted far enough for the clock to have to be readjusted. In the USB adaptive mode implementation with the external clock generator board in multisynth integer mode the difference between two adjustment levels is 0.5 %, which translates to a difference of 1.77 ns for the 2.884 MHz I^2S input clock. The same number for the adaptive mode USB implementation with the multisynth in fractional mode which has adjustment levels of 0.002 % is around 70 ps. The implementation with the smaller adjustment intervals using the fractional multisynth mode will likely be switching between the levels more often than the implementation with longer distances in between, but 70 ps is not much in relation to the total measured jitter of the clock generator board, so it is difficult to tell if values from more than one adjustment level have been captured during the limited time measurement values were collected.

For the design using the integer multisynth mode, the difference between two adjacent adjustment levels is comparable to the size of the total inherent jitter for the clock generator board in the USB asynchronous mode case, so it appears as if only values from one single adjustment level were captured during the adaptive mode measurements. Figure 4.17 shows one single measurement series each from two adjustment levels located next to each other, and when viewing it, it becomes clear that when an adjustment is made, then the total jitter will increase approximately to the size of the inherent jitter of the clock generator board plus the size of the adjustment, making the total jitter become a function of the size of the adjustment level for the adaptive mode USB implementations. For the implementation with the multisynth in integer mode, the total jitter during normal operation would then be expected to increase more than twofold compared to the measurement results. The adjustment level size can be decreased but only down to around 0.05 % due to the parameter limits of the Si5351 in multisynth integer mode, which would equal a clock adjustment of 0.177 ns for the I^2S input clock. This could put the results for the SCK and WS clocks in adaptive and asynchronous USB mode very closely together. The 70 ps difference between two adjustment levels for the USB adaptive mode implementation with the multisynth in fractional mode should however be small enough to not make any significant difference to the results in Table 4.3.

An unexpected thing was that the external XO and the external fixed frequency clock board performed worse than the Si5351 clock generator board which includes a fractional divider. The unsatisfying results with the external XO was first thought

to be caused by difficulties in finding correct values for the capacitors in the circuit in Figure 3.12 as also parasitic capacitances from the rest of the circuitry must be included in the total [40], but when later testing the same design with the external XO replaced by the fixed frequency clock generator board, the results were on par for both clock sources. The external fixed frequency clock board is marketed as a clock upgrade for consumer audio products and it has a jitter measurement rating of less than 2 ps, which is nowhere near any of the measurement results produced with it connected to the PSoC. We should remember that the smallest possible sampling interval of the oscilloscope used was 1 ns and that we with the use of interpolation can get a little below that limit, but the measurements listed in Table 4.3 for the external XO and the fixed frequency clock generator board are around 20 ns.

One thought was that the voltage regulator on the CY8CKIT-001 PSoC development board or interference from other components could have something to do with the discrepancy between the specification and the test results and when using a battery pack as power source with no other components connected to the clock board, peak-to-peak period jitter levels of less than 1 ns were observed using the oscilloscope. Looking at the shape of the waveforms of the jitter plots for both the external XO and the fixed frequency clock board, it does seem plausible that there could be some duty cycle distortion taking place. Figure C.8 and C.13 displaying the period jitter for the SCK output signals are examples of this. Possible causes for duty cycle distortion could be that there is a slew rate imbalance, or that the threshold level that marks the signal transition has been offset [11].

A second thing that stood out was the measurement results for the synchronous mode USB using the PSoC IMO as clock source through the custom made fractional divider component. With all the other implementations, the I^2S input and SCK and WS output clock measurements were much closer in magnitude, but for the WS output clock in synchronous mode, the jitter was more than 25 times larger than for any of the other two measured signals in the same design. Tests were therefore then later run again in which roughly the same numbers were confirmed once more, so there was no obvious temporary error in the system during the test that could explain why the WS period jitter differed so greatly from the other two measured clocks. There was suspicion of that the captured data for the I^2S input clock and the SCK output clock were not giving the complete picture for this measurement. All three clocks are after all derived from the same clock source, so one would expect them to produce jitter readings of similar magnitude, even though the measurements of the clocks were not performed simultaneously. The clock generated by the IMO through the fractional divider component was then also tried in the asynchronous mode design where it showed similar results, but the magnitude of the SCK clock jitter had now grown the be somewhere in the middle between the I^2S and WS clock readings due to there being outliers of larger magnitude, such as for example the most deviant measurement value in Figure C.19 which more than doubled the cycle-to-cycle peak jitter for the SCK signal. The most extreme values tend to increase in size with the number of sample points in the measurement series [11] when there is Gaussian unbounded jitter, so to make the results comparable, one hundred sample points each were collected in all of the measurements listed in Table 4.3.

USB mode	Clock source	Signal	Jitter type	Value
Asynchronous	Si5351 (fractional)	I²S in	Period pk-pk	1.86 ns
Asynchronous	Si5351 (fractional)	I²S in	Cycle-to-cycle pk	1.86 ns
Asynchronous	Si5351 (fractional)	SCK	Period pk-pk	1.15 ns
Asynchronous	Si5351 (fractional)	SCK	Cycle-to-cycle pk	0.95 ns
Asynchronous	Si5351 (fractional)	WS	Period pk-pk	2.23 ns
Asynchronous	External XO	I²S in	Period pk-pk	21.6 ns
Asynchronous	External XO	I²S in	Cycle-to-cycle pk	21.4 ns
Asynchronous	External XO	SCK	Period pk-pk	20.6 ns
Asynchronous	External XO	SCK	Cycle-to-cycle pk	20.1 ns
Asynchronous	External XO	WS	Period pk-pk	20.9 ns
Asynchronous	External FF clock	I²S in	Period pk-pk	21.2 ns
Asynchronous	External FF clock	I²S in	Cycle-to-cycle pk	21.0 ns
Asynchronous	External FF clock	SCK	Period pk-pk	20.3 ns
Asynchronous	External FF clock	SCK	Cycle-to-cycle pk	19.8 ns
Asynchronous	External FF clock	WS	Period pk-pk	22.4 ns
Asynchronous	PSoC IMO	I²S in	Period pk-pk	3.75 ns
Asynchronous	PSoC IMO	I²S in	Cycle-to-cycle pk	3.75 ns
Asynchronous	PSoC IMO	SCK	Period pk-pk	22.7 ns
Asynchronous	PSoC IMO	SCK	Cycle-to-cycle pk	21 ns
Asynchronous	PSoC IMO	WS	Period pk-pk	75.6 ns
Asynchronous	Si5351 (integer)	I²S in	Period pk-pk	1.10 ns
Asynchronous	Si5351 (integer)	I²S in	Cycle-to-cycle pk	1.00 ns
Asynchronous	Si5351 (integer)	SCK	Period pk-pk	1.50 ns
Asynchronous	Si5351 (integer)	SCK	Cycle-to-cycle pk	1.50 ns
Asynchronous	Si5351 (integer)	WS	Period pk-pk	2.48 ns
Adaptive	Si5351 (integer)	I²S in	Period pk-pk	1.14 ns
Adaptive	Si5351 (integer)	I²S in	Cycle-to-cycle pk	0.92 ns
Adaptive	Si5351 (integer)	SCK	Period pk-pk	1.40 ns
Adaptive	Si5351 (integer)	SCK	Cycle-to-cycle pk	1.05 ns
Adaptive	Si5351 (integer)	WS	Period pk-pk	1.70 ns
Adaptive	Si5351 (fractional)	I²S in	Period pk-pk	1.92 ns
Adaptive	Si5351 (fractional)	I²S in	Cycle-to-cycle pk	1.73 ns
Adaptive	Si5351 (fractional)	SCK	Period pk-pk	1.70 ns
Adaptive	Si5351 (fractional)	SCK	Cycle-to-cycle pk	1.30 ns
Adaptive	Si5351 (fractional)	WS	Period pk-pk	3.91 ns
Synchronous	PSoC IMO	I²S in	Period pk-pk	1.55 ns
Synchronous	PSoC IMO	I²S in	Cycle-to-cycle pk	1.55 ns
Synchronous	PSoC IMO	SCK	Period pk-pk	2.25 ns
Synchronous	PSoC IMO	SCK	Cycle-to-cycle pk	1.50 ns
Synchronous	PSoC IMO	WS	Period pk-pk	56.8 ns

Table 4.3: Period peak-to-peak and cycle-to-cycle maximum values from the jitter measurements.

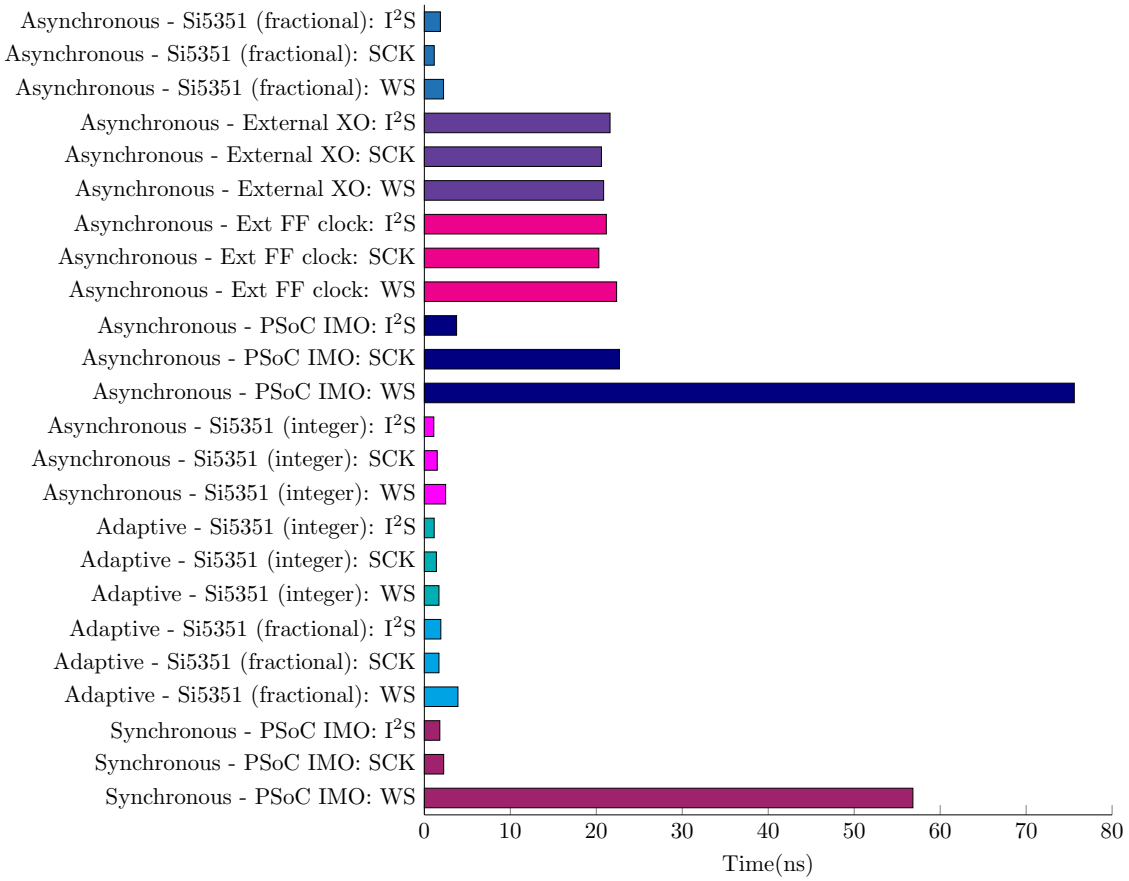


Figure 4.15: Period peak-to-peak jitter for the I²S component input and SCK and WS output clocks for each of the implementations.

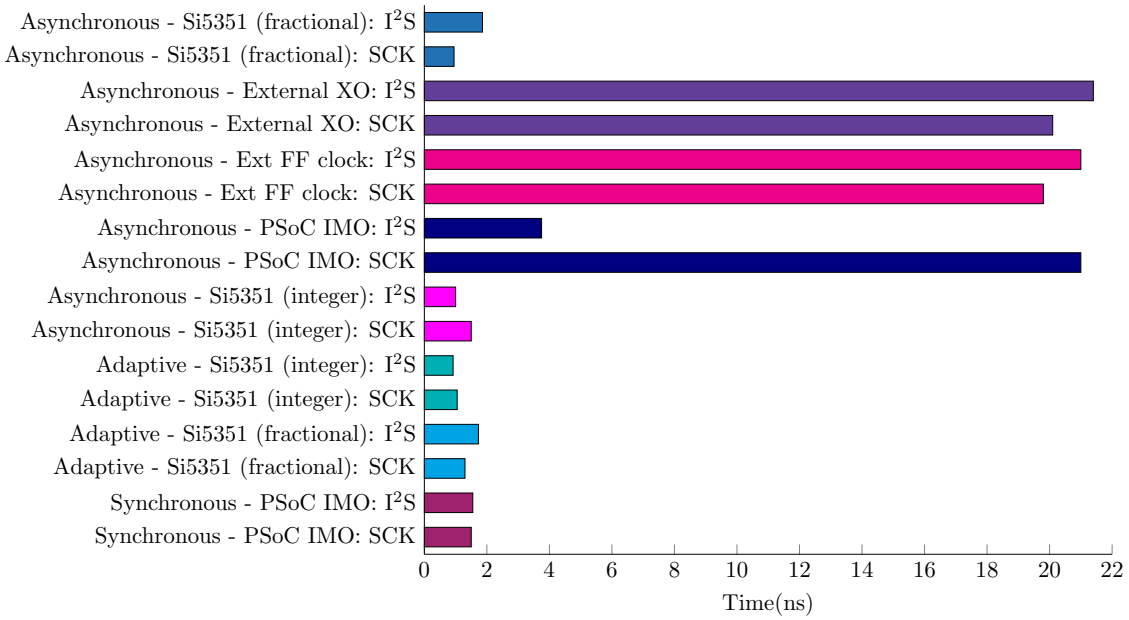


Figure 4.16: Cycle-to-cycle peak jitter for the I²S component input and SCK output clocks for each of the implementations.

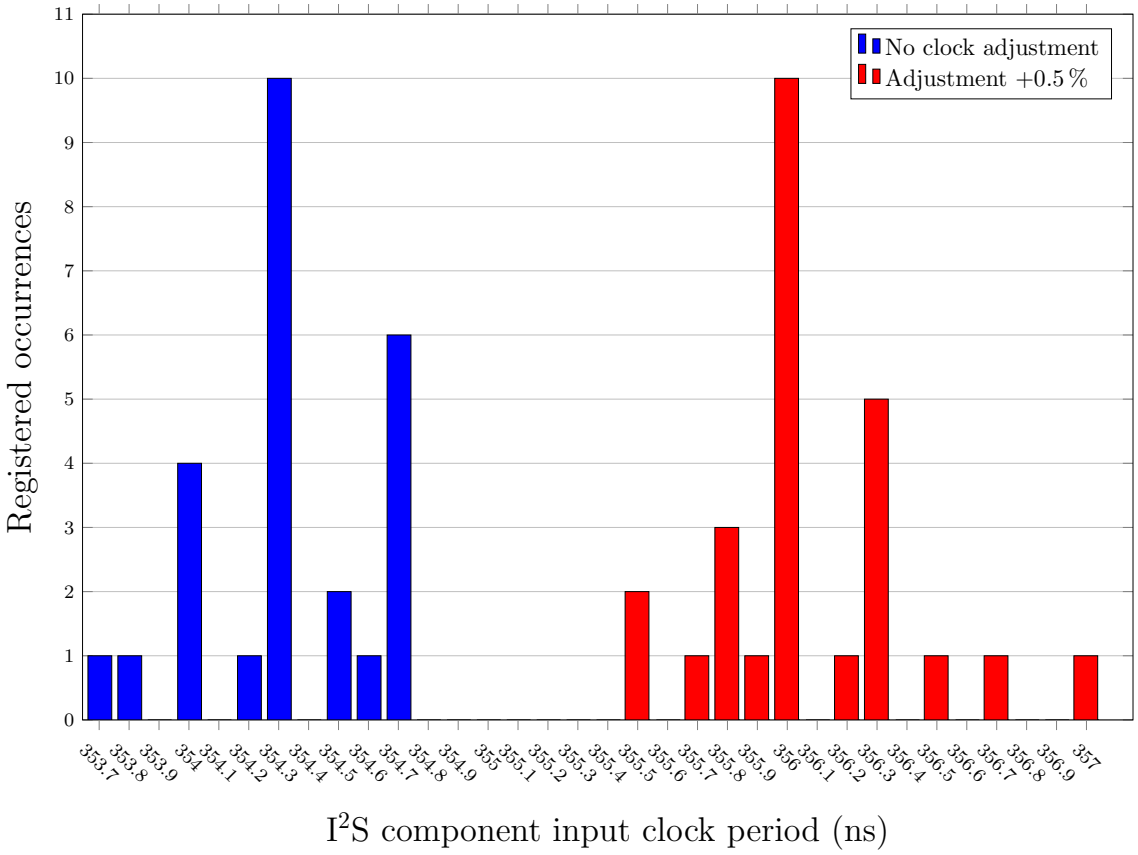


Figure 4.17: Period histogram for two adjacent adjustment levels for the I²S component input clocks in USB adaptive mode with the multisynth set to integer mode.

The Si5351 clock signal in the asynchronous synchronization mode design used the ClockBuilder Pro generated register map for the fractional multisynth mode implementation while the USB adaptive synchronization mode design used the manually generated register values for its multisynth fractional mode test, so a direct comparison between the two might not be totally fair as one would expect the ClockBuilder Pro generated registry settings to be optimized for low jitter performance while the manually created registry mapping only is concerned with getting the correct frequency and adjustability for the clock. In multisynth integer mode the Si5351 clock used the exact same registry mapping for both the asynchronous and adaptive mode implementation and it did actually perform a bit better for the slower WS clock and in the cycle-to-cycle jitter measurement for the SCK clock in adaptive mode. This could be caused by natural variation and measurement uncertainty but we should not rule out the possibility that there could be something in the design of the asynchronous mode implementation adding one more USB endpoint with its own DMA channel and interrupt service routine that could have an impact on clock generation timing inside the PSoC, which could be why it performed worse than the adaptive mode implementation before the clock adjustments were also taken into account. All measurements listed in Table 4.3 were done individually, so for example the I²S input clock and the WS output clock for one implementation are

from different measurement series and they are therefore not directly related to each other even if they were conducted around the same time using the same design.

With the synchronous mode implementation using an update interval of once per SOF and the asynchronous mode implementation coupled with the IMO clock as source being updated every 16th SOF, some additional testing was done to see if the frequency of the clock updates could have any impact on the jitter measurement results. Increasing the update interval in steps did not appear to give rise to any obvious pattern where the measured jitter would increase or decrease along with the update frequency but the measured values did vary quite a bit between the different measurement series and in some, both the period and the cycle-to-cycle jitter for the SCK output clock in synchronous mode did now also go above 20 ns as in the asynchronous mode measurement. A few outliers that extended way beyond the rest of the measurement values is what once again caused the increase in the period peak-to-peak and cycle-to-cycle peak jitter levels for the SCK output clock signal. As previously mentioned, an equal amount of samples should be used to keep the comparison against the other measurements fair, but it is debatable if a larger sample size than 100 samples for each of the signals should have been used. Tests with larger sample sizes for the I^{2}S input clock signal for the fixed frequency clock board and the Si5351 clock board did not make any bigger difference in the peak jitter values but the designs including the fractional divider component seem to be prone to giving rise to more and larger outliers which did not get registered with a shorter sample length.

In an attempt to look at the effect on the I^{2}S clocks of the adjustments made to the fractional divider settings, the asynchronous mode design was run with the fractional divider using no adjustment at all and also with the two outmost adjustment levels at $\pm$16 needed to match the most extreme values of the SOF variation. Figure 4.18 shows this in which 1000 samples for each setting were collected. The main volume of data points is located between 352 ns and 356 ns just like all the measurement values in the shorter series for the same signal displayed in Figure C.16. There are however now clusters of values outside that range which did not get registered in the shorter measurement series for the I^{2}S input clock.

Unlike the plot in Figure 4.17 for the adjustment levels in adaptive mode using the Si5351 clock generator, Figure 4.18 does not show a simple linear dependence between the registered jitter levels and the settings of the fractional divider component. Both the setting with no level adjustment of the fractional divider at all and the setting where +16 was added have similar measurement ranges from around 252 ns up to 373 ns while the setting of -16 ranges from 343 ns to 356 ns. The nature of the design using the fractional divider component does not allow us to make the same straightforward kind of statement of how clock adjustments affect the I^{2}S clocks like for the adaptive mode implementations using the Si5351 board as clock source. The clock chain that leads to the I^{2}S input clock consists of the IMO, the fractional divider component, the PLL, the master clock, and a final divider to get the correct clock multiple. As for the color scheme of Figure 4.18, the vertical bars are semitransparent, so red on top of blue does for example result in purple.

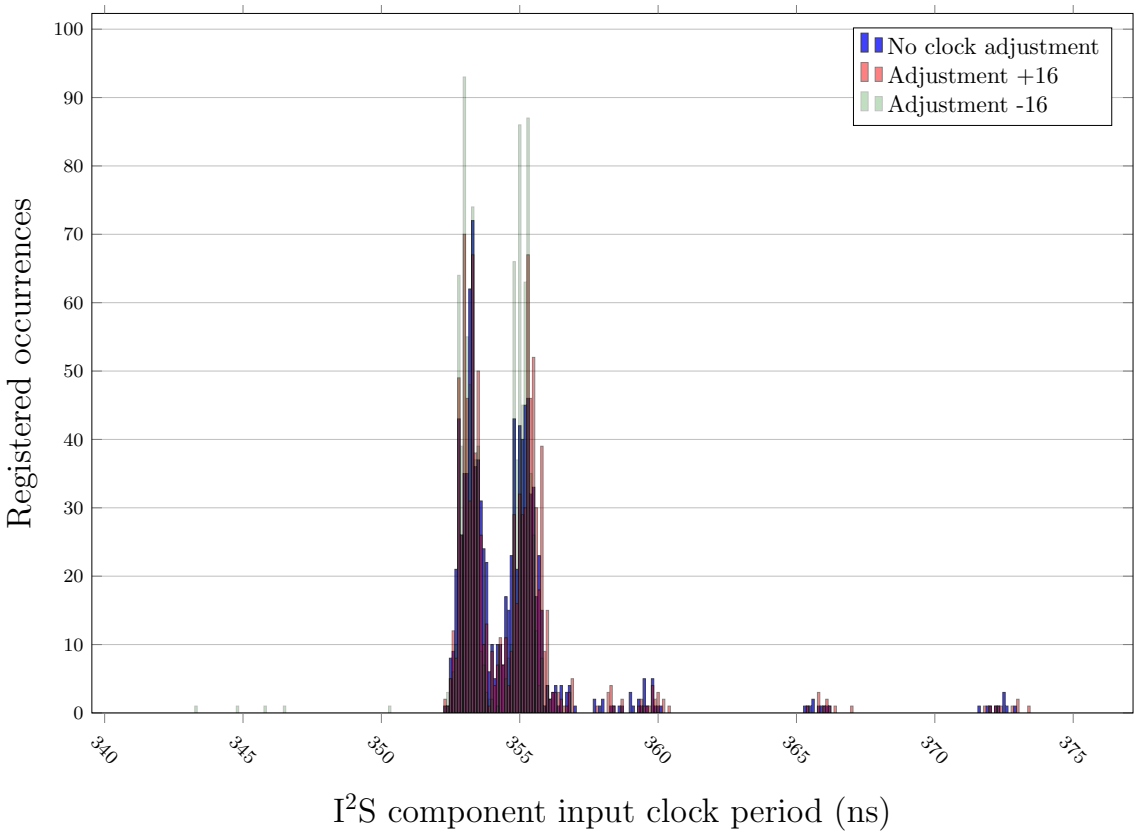


Figure 4.18: Period histograms with and without clock adjustments for the I²S component input clock in USB asynchronous mode using the IMO together with the fractional divider as clock source.

Returning to the results from the studies in Chapter 2.1.5 in which listening tests were conducted, for single note instruments threshold levels down to a couple of tens of nanoseconds were recorded and for normal music the lowest threshold levels increased to be in the hundreds of nanoseconds. For the external XO, the external fixed frequency clock board, and the clock generated from the PSoC IMO through the custom fractional divider we are crossing the line into the territory of that first threshold level, so for audio with sparse sounds, jitter could possibly be within the audible range for the mentioned implementations. For normal and more complex music there should be too much of a masking effect [14] for jitter of the measured magnitude to be heard in any of the test implementations.

The semilog plot in Figure 4.19 displays the range of period jitter peak-to-peak values from Table 4.3 measured for the three I²S clocks in each implementation, comparing the results to the lowest jitter audibility thresholds for each program material audio type determined by the listening tests listed in Chapter 2.1.5 and also with the theoretical jitter audibility model from Chapter 2.1.5.1. Jitter type, methodology and other parameters differ between the three audibility tests, so any comparison between the threshold limits in Figure 4.19 themselves should be done with this in mind.

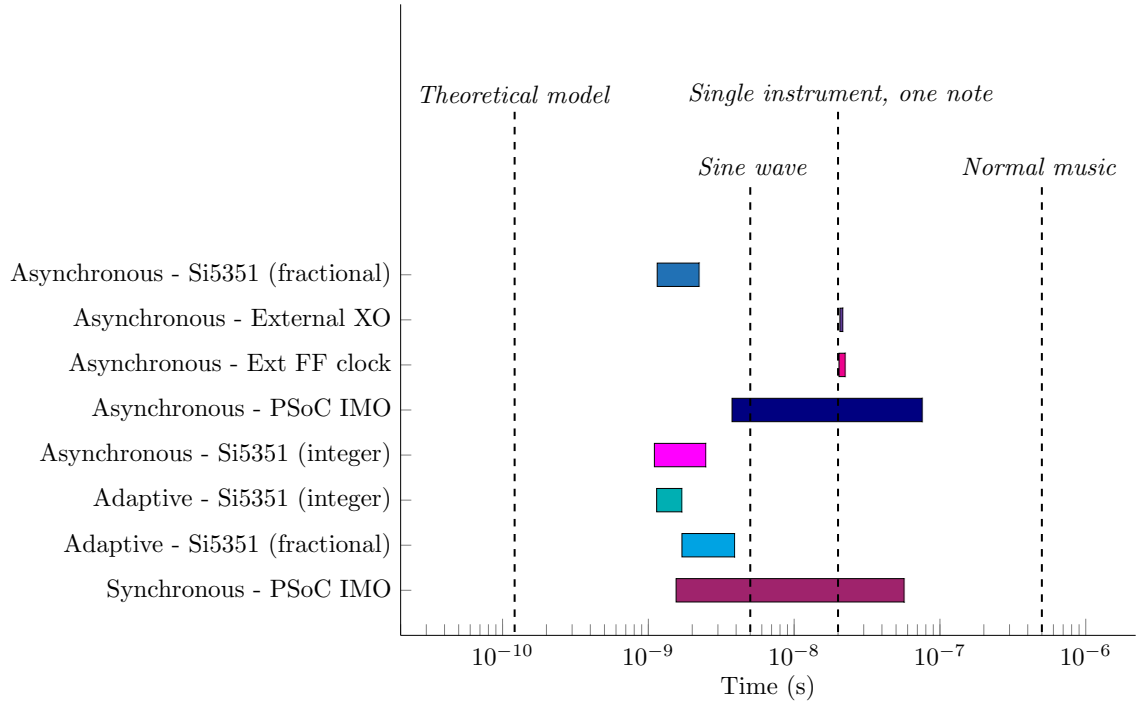


Figure 4.19: The range of period jitter peak-to-peak values for all three I^2S component clocks compared to the theoretical jitter audibility model and the jitter audibility thresholds determined by listening tests.

4.2.2 Start-of-Frame Jitter

The oscilloscope at hand cannot be used for proper jitter measurements of the SOF. A measurement series for the SOF signal is shown in Figure 4.20. The SOF pulses arriving at $\tau, 2\tau, 3\tau$ etc. are precisely evenly spaced apart by 2500 samples and there is not enough resolution to get a good reading of the jitter levels this way. SOF jitter is expected to be within approximately ±500 ns according to the USB specification. The sampling interval in Figure 4.20 is 0.4 µs and the sample depth is 10 000 samples, giving a total measurement time of 4 ms. SOF packets are sent once per 1 ms and at least one full period is needed the to compare the arrival times of two consecutive SOF packets. Even if the sampling time interval is made shorter, there is no way to get anywhere near the expected range of SOF jitter presented in the USB specification while still fitting one full SOF period in the 10 000 samples as the SOF pulses are approximately 1 ms apart. What we can do instead is to do a TIE measurement using the PSoC IMO clock as reference. As already concluded in Chapter 4.1.3, the measured variation in SOF arrival time was 1.0 ms±700 ns. Here is now also a histogram presented for the SOF variation in Figure 4.21. It has been derived directly from the data for the registered IMO clock pulses per SOF shown in Figure 4.12. As can be seen, the SOF jitter seems to predominantly be of random type, but there are also two small peaks at minus six and minus ten IMO samples per SOF, possibly indicating that there also are other types of jitter involved.

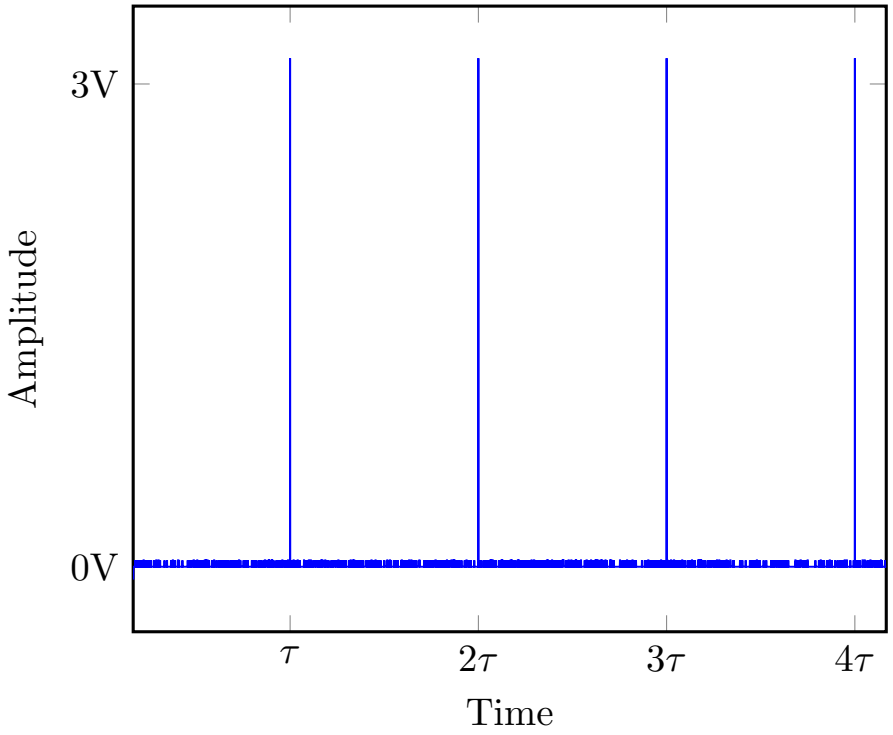


Figure 4.20: Start-of-frame packets received by the PSoC USBFS component.

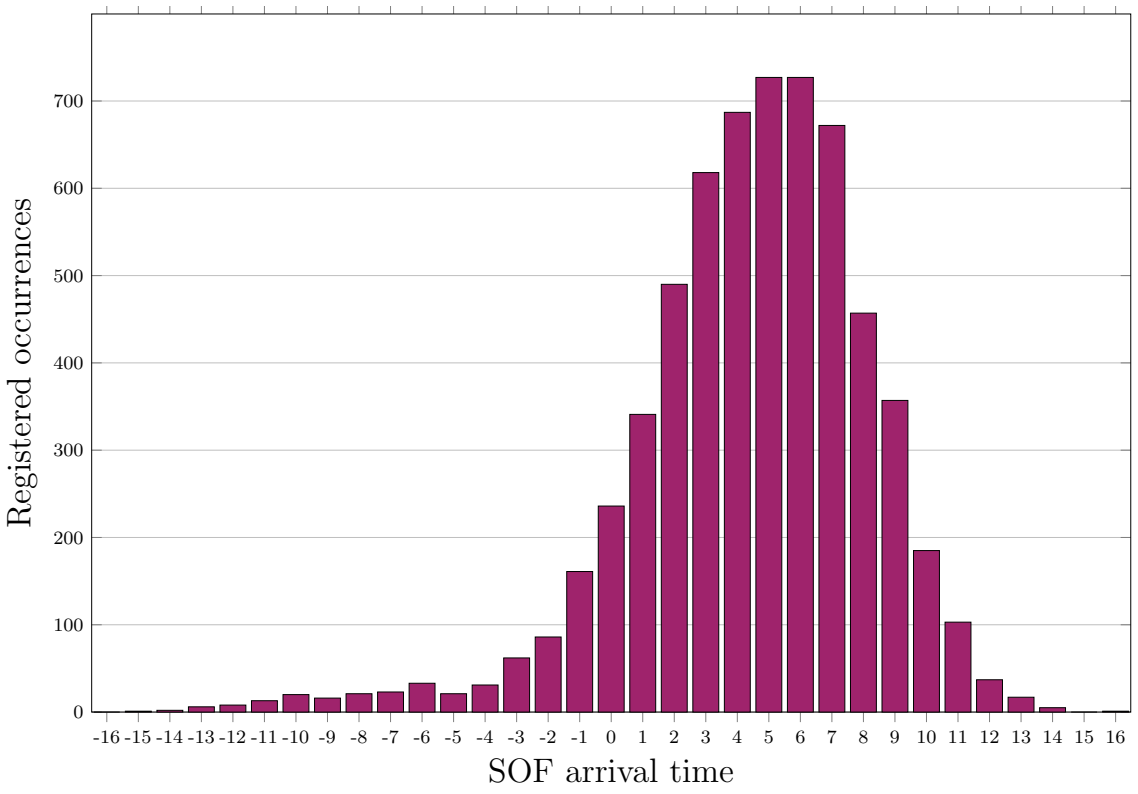


Figure 4.21: Histogram of SOF arrival time variation referenced to IMO clock data from Figure 4.12.

5

Conclusion

This chapter contains a summary of the findings from the measurement results and tests. A paragraph considering sustainability issues is included at the end.

5.1 Summary of Results

One of the goals with the thesis was to try to construct a digital audio interface that would not produce any interface jitter in the process of transferring the audio signal between two devices. With the USB audio device working in asynchronous synchronization mode, the expectation was for the jitter to be as low as the inherent jitter of the clock used as master clock on the device side of the interface. Tests did however show that when attaching a clock with a jitter specification way below the lowest recorded audibility test thresholds for jitter in different program source materials, the measured jitter did in some cases increase to levels above the lowest jitter audibility thresholds when the clock was connected to the test system and the audio device was operating in asynchronous mode.

Designs for all three standard USB audio synchronization modes were implemented and tested, which in itself was a great learning experience. All three synchronization mode implementations are working from a functional point of view, but the expectation of USB asynchronous mode outperforming adaptive and synchronous mode in terms of jitter due to not having to adjust the internal clock of the device is not directly obvious by looking at the period peak-to-peak and cycle-to-cycle peak jitter measurements alone. Many of the clock signal measurements with for example the Si5351 clock generator board with the multisynth in integer mode as source display worse jitter readings in asynchronous mode than for the same clock source counterpart in adaptive mode. That is however before the clock adjustments of the adaptive mode are included as they do not always get registered during the limited measurement time.

If we compare the adaptive and asynchronous mode implementations using the Si5351 clock generator board, we can see that for the designs with the multisynth in fractional mode the I^2S input clock measurements are around the same levels, but for both the SCK and WS clocks, the jitter levels of the asynchronous mode implementation are noticeably lower. The two different USB synchronization mode implementations do however not use the same register mapping for the Si5351 clock

source with the multisynth set to fractional mode, which possibly could have some effect on the results. It is not totally clear whether the periodic frequency adjustments of the clock in adaptive mode with the Si5351 multisynth set to fractional mode have been captured in the measurement results, but if they have not, then a fair presumption is that the jitter of the adaptive implementation would further increase just by a tiny fraction of the current jitter value.

The same two synchronization mode implementations with the Si5351 as clock source and the multisynth set to integer mode share the same fundamental registry mapping for the source clock, and by further tests it was determined that the clock level adjustments of the adaptive mode implementation did not get registered during the limited time measurements were conducted. When viewing the numbers in Table 4.3, the results for the I^2S input clock and the SCK output clock period jitter are not far apart, but for the SCK output clock cycle-to-cycle and the WS period jitter the adaptive mode implementation does show lower jitter measurement levels than the asynchronous mode implementation. When the frequency level adjustments then also are included in the results for adaptive mode USB, the jitter will increase for each new clock adjustment level that is reached. The increase is expected to be approximately the same as the registered measurement value for each clock signal adjustment, for example then doubling the jitter level if the clock alternates between two clock adjustment levels as expected. This still makes asynchronous mode the better choice of the two if low jitter is the topmost priority, even though only looking at the jitter measurements alone would make it seem as if the opposite is true.

In the comparison of the synchronous and the asynchronous mode USB using the IMO coupled with the fractional divider component as clock source, the registered measurement results once again showed higher jitter readings for the asynchronous mode implementation. It was however also determined that a much longer series of measurements made the peak values grow a lot higher than in the shorter series used for comparison, so the results in Table 4.3 should in this case be treated as less reliable. The effects on jitter by the clock adjustments made to the fractional divider component in synchronous mode are not as clear as for example in the adaptive mode case with the Si5351 clock board operating with the multisynth in integer mode, so it is difficult to say exactly how much of a difference they make. As the designs including the fractional component in its current form show WS clock results way above any of the other implementations and the I^2S input and SCK output clock peak values are expected to also increase with longer measurement time, there is less use in trying to precisely determine which one of the two implementations using the fractional divider is better or worse than the other.

A property of the USB asynchronous mode design is that not having to make any clock adjustments on the fly allows for a fixed frequency clock to be used as clock source, something which is not possible when the device operates in adaptive or synchronous mode. A fixed frequency clock should in theory be able to perform at the same level or better than any similar clock design also including a fractional divider if given the correct conditions. It was a little bit disappointing to see that neither of the two fixed frequency clock sources that were tried performed anywhere

near the best results of the Si5351 clock board in the tests conducted. More effort would need to be put into ensuring that the working conditions of the fixed frequency clocks are being optimal.

There are varying opinions on jitter and its effect or lack thereof on audio quality. Theoretical jitter audibility threshold models like the one in Chapter 2.1.5.1 will usually place the threshold level below a couple of hundreds of picoseconds or even lower while the lowest recorded levels in the listening tests in Chapter 2.1.5 for different jitter types and source material never went below a couple of nanoseconds. In the different USB synchronization mode implementations using different clock sources there are some designs which land above and others where all clocks have jitter levels below the auditory assessed threshold values. The sample rate interval limit of the digital oscilloscope at 1 ns did not allow for more accurate measurements than the ones in Table 4.3, including interpolation of the sample points closest to the signal transition level, so no meaningful comparison with the theoretical jitter audibility threshold could be made.

5.1.1 Sustainability and Environmental Considerations

Using the PSoC IMO as clock source coupled with the custom fractional divider component removes the need for an external clock source. Fewer hardware components is desirable from a sustainability perspective, especially when it also reduces the total cost of a built device. In this case using the internal clock as source in the current configuration produced enough jitter to place the device performance above the lowest measured audibility threshold, making it unsuitable when audio quality is the top priority. For devices where other parameters are of more concern, using the internal clock of the PSoC could be an acceptable trade off. Replacing the external fractional clock divider with the internal custom fractional divider component is essentially replacing hardware with software, and that is what the PSoC system is all about. The components used in the designs, the counters, clock dividers, the USB block etc. would all have needed to be constructed using many separate external components if it would not have been for the programmable UDB blocks of the PSoC, making it a flexible and resource saving solution.

Bibliography

[1] J. Dunlop and D. G. Smith, *Telecommunications engineering*, 3rd ed. Chapman & Hall, 1994, pp. 77–80.

[2] C. Dunn and M. Hawksford, "Is the AESEBU / SPDIF digital audio interface flawed ?" 1992, presented at the 93rd convention of the Audio Engineering Society, San Francisco 1992, preprint 3360.

[3] W. Maichen, *Digital timing measurements: From scopes and probes to timing and jitter*. Springer, 2006.

[4] J. Dunn, "Jitter: Specification and assessment in digital audio equipment," *Journal of the Audio Engineering Society*, October 1992.

[5] European Broadcasting Union, "Specification of the digital audio interface (The AES/EBU interface)," 2004. [Online]. Available: https://tech.ebu.ch/docs/tech/tech3250.pdf

[6] T. Engdahl, "S/PDIF interface." [Online]. Available: https://www.epanorama.net/documents/audio/spdif.html

[7] ISO/IEC 60958:2022, "Digital audio interface," International Electrotechnical Commission, Geneva, CH, Standard, 2022.

[8] Naim Audio Ltd., "Asynchronous USB," 2013. [Online]. Available: https://www.naimaudio.com/sites/default/files/products/downloads/files/dac-v1_asynchronous-usb_mwp_jan13_0.pdf

[9] Ayre Acoustics, "A new era in computer-based audio," 2004. [Online]. Available: https://www.ayre.com/wp-content/uploads/2020/08/Ayre_USB_DAC_White_Paper.pdf

[10] USB Implementers Forum, Inc., "Universal Serial Bus device class definition for audio devices release 1.0." [Online]. Available: https://usb.org/document-library/audio-device-document-10

[11] Tektronix, "Understanding and characterizing timing jitter," 2019. [Online]. Available: https://download.tek.com/document/Understanding-and-Characterizing-Timing-Jitter_55W_16146_6.pdf

[12] U. Dahlbom, *Matematisk statistik för teknologer*. Matematiklitteratur i Göteborg, 2001, pp. 100–144.

[13] L. Bergström and B. Snaar, *Elektriska kretsar och linjära system: Laplacetransformer och z-transformer*. Natura läromedel, 1997, pp. 33–34.

[14] E. Benjamin and B. Gannon, "Theoretical and audible effects of jitter on digital audio quality," 1998, presented at the 105th convention of the Audio Engineering Society, San Francisco, September 1998.

[15] K. Ashihara, S. Kiryu, N. Koizumi, A. Nishimura, J. Ohga, M. Sawaguchi, and S. Yoshikawa, "Detection threshold for distortions due to jitter on digital audio," *Acoustical Science and Technology*, vol. 26, pp. 50–54, 01 2005.

[16] W. I. Manson, "Digital sound signals: Subjective effect of timing jitter," *BBC Reserch Department Report*, November 1974.

[17] M. Kleiner, "Audioteknik och akustik," Institutionen för teknisk akustik, Chalmers Tekniska Högskola, Göteborg, 2000.

[18] Z. Kulka, "Sampling jitter in audio A/D converters," Warsaw University of Technology, Institute of Radioelectronics, Electroacoustics Division, 2011.

[19] P. H. Young, *Electronic communication techniques*, 5th ed. Pearson Education International, 2006.

[20] E. Meitner, "Very low jitter clock recovery from serial audio data," U.S. Patent US5 404 362A, Apr. 4, 1995. [Online]. Available: https://patents.google.com/patent/US5404362A

[21] D. K. Cheng, *Fundamentals of Engineering Electromagnetics*. Prentice-Hall Inc., 1993.

[22] USB Implementers Forum, Inc., "Universal Serial Bus specification revision 2.0." [Online]. Available: https://www.usb.org/document-library/usb-20-specification

[23] USB Implementers Forum, Inc., "Universal Serial Bus 3.2 specification." [Online]. Available: https://www.usb.org/document-library/usb-32-revision-11-june-2022

[24] USB Implementers Forum, Inc., "Universal Serial Bus 4 (USB4®) specification." [Online]. Available: https://www.usb.org/document-library/usb4r-specification-v20

[25] USB Implementers Forum, Inc., "Universal Serial Bus device class definition for audio devices release 2.0." [Online]. Available: https://www.usb.org/document-library/audio-devices-rev-20-and-adopters-agreement

[26] USB Implementers Forum, Inc., "Universal Serial Bus device class definition for audio devices release 4.0." [Online]. Available: https://www.usb.org/document-library/usb-audio-devices-release-40-and-adopters-agreement

[27] Philips Semiconductors, "I²S bus specification," 1986.

[28] Texas Instruments Incorporated, "Technical brief SWRA029 - Fractional/integer-N PLL basics." [Online]. Available: https://www.ti.com/lit/an/swra029/swra029.pdf

[29] Infineon Technologies AG, "PSoC™ 3 - Infineon Technologies." [Online]. Available: https://www.infineon.com/cms/en/product/microcontroller/legacy-microcontroller/legacy-8-bit-16-bit-microcontroller/psoc-3

[30] Infineon Technologies AG, "Release notes CY8CKIT-001 PSoC® development kit." [Online]. Available: https://www.infineon.com/dgdl/Infineon-CY8CKIT-001_Release_Notes-UserManual-v01_00-EN.pdf?fileId=8ac78c8c7d0d8da4017d0eec44d17773

[31] Adafruit Industries, "Adafruit Si5351 clock generator breakout." [Online]. Available: https://learn.adafruit.com/adafruit-si5351-clock-generator-breakout

[32] Silicon Labs, "Si5351A/B/C - I²C-programmable any-frequency CMOS clock generator + VCXO." [Online]. Available: https://cdn-shop.adafruit.com/datasheets/Si5351.pdf

[33] Adafruit Industries, "Adafruit I²S stereo decoder - UDA1334A." [Online]. Available: https://learn.adafruit.com/adafruit-i2s-stereo-decoder-uda1334a

[34] NPX Semiconductors, "UDA1334ATS - Low power audio DAC with PLL." [Online]. Available: https://www.nxp.com/docs/en/data-sheet/UDA1334ATS.pdf

[35] Infineon Technologies AG, "AN60631 - PSoC® 3 and PSoC 5LP clocking resources." [Online]. Available: https://www.infineon.com/dgdl/Infineon-AN60631_PSoC_3_and_PSoC_5LP_Clocking_Resources-ApplicationNotes-v09_00-EN.pdf?fileId=8ac78c8c7cdc391c017d072f02b65370

[36] Infineon Technologies AG, "AN81623 - PSoC® 3, PSoC 4, and PSoC 5LP digital design best practices." [Online]. Available: https://www.infineon.com/dgdl/Infineon-AN81623_PSoC_3_PSoC_4_and_PSoC_5LP_Digital_Design_Best_Practices-ApplicationNotes-v07_00-EN.pdf?fileId=8ac78c8c7cdc391c017d0726b5b94b78

[37] Skyworks Solutions, Inc., "Glitch-free frequency shifting simplifies timing design in consumer applications." [Online]. Available: https://www.skyworksinc.com/-/media/Skyworks/SL/documents/public/white-papers/Si5350-51-Frequency-Shifting-WP.pdf

[38] Skyworks Solutions, Inc., "AN619 - Manually generating an Si5351 register map for 10-MSOP and 20-QFN devices." [Online]. Available: https://www.skyworksinc.com/-/media/Skyworks/SL/documents/public/application-notes/AN619.pdf

[39] Infineon Technologies AG, "PSoC® Creator™ - Component author guide." [Online]. Available: https://www.infineon.com/dgdl/Infineon-PSoC_Creator_Component_Author_Guide-Software%20Module%20Datasheets-v21_00-EN?fileId=8ac78c8c7d0d8da4017d0eb0c2672a9f

[40] IQD Ltd, "IQXO-350." [Online]. Available: https://www.farnell.com/datasheets/47655.pdf

A

USB Descriptors

A.1 Asynchronous Mode USB Descriptor Table

The USB descriptor table for the asynchronous mode implementation is provided in this section for reference. String descriptors and system generated settings and dispatch tables and have been omitted.

```
/*********************************************************************
* Device Descriptor
*********************************************************************/
/* Descriptor Length                    */ 0x12u,
/* DescriptorType: DEVICE               */ 0x01u,
/* bcdUSB (ver 2.0)                     */ 0x00u, 0x02u,
/* bDeviceClass                         */ 0x00u,
/* bDeviceSubClass                      */ 0x00u,
/* bDeviceProtocol                      */ 0x00u,
/* bMaxPacketSize0                      */ 0x08u,
/* idVendor                             */ 0xB4u, 0x04u,
/* idProduct                            */ 0x51u, 0x20u,
/* bcdDevice                            */ 0x00u, 0x00u,
/* iManufacturer                        */ 0x05u,
/* iProduct                             */ 0x0Au,
/* iSerialNumber                        */ 0x00u,
/* bNumConfigurations                   */ 0x01u
/*********************************************************************
* Configuration Descriptor
*********************************************************************/
/*  Config Descriptor Length            */ 0x09u,
/*  DescriptorType: CONFIG              */ 0x02u,
/*  wTotalLength                        */ 0x70u, 0x00u,
/*  bNumInterfaces                      */ 0x02u,
/*  bConfigurationValue                 */ 0x01u,
/*  iConfiguration                      */ 0x00u,
/*  bmAttributes                        */ 0x80u,
/*  bMaxPower                           */ 0x32u,
```

```
/*********************************************************************
* Standard AudioControl Interface Descriptor
*********************************************************************/
/*  Interface Descriptor Length             */ 0x09u,
/*  DescriptorType: INTERFACE               */ 0x04u,
/*  bInterfaceNumber                        */ 0x00u,
/*  bAlternateSetting                       */ 0x00u,
/*  bNumEndpoints                           */ 0x00u,
/*  bInterfaceClass                         */ 0x01u,
/*  bInterfaceSubClass                      */ 0x01u,
/*  bInterfaceProtocol                      */ 0x00u,
/*  iInterface                              */ 0x07u,
/*********************************************************************
* Class-Specific AudioControl Interface Descriptor
*********************************************************************/
/*  AC Header Descriptor Length             */ 0x09u,
/*  DescriptorType: AUDIO                   */ 0x24u,
/*  bDescriptorSubtype                      */ 0x01u,
/*  bcdADC                                  */ 0x00u, 0x01u,
/*  wTotalLength                            */ 0x1Eu, 0x00u,
/*  bInCollection                           */ 0x01u,
/*  baInterfaceNr                           */ 0x01u,
/*********************************************************************
* Input Terminal Descriptor
*********************************************************************/
/*  AC Input Terminal Descriptor Length     */ 0x0Cu,
/*  DescriptorType: AUDIO                   */ 0x24u,
/*  bDescriptorSubtype                      */ 0x02u,
/*  bTerminalID                             */ 0x01u,
/*  wTerminalType                           */ 0x01u, 0x01u,
/*  bAssocTerminal                          */ 0x00u,
/*  bNrChannels                             */ 0x02u,
/*  wChannelConfig                          */ 0x03u, 0x00u,
/*  iChannelNames                           */ 0x00u,
/*  iTerminal                               */ 0x00u,
/*********************************************************************
* Output Terminal Descriptor
*********************************************************************/
/*  AC Output Terminal Descriptor Length    */ 0x09u,
/*  DescriptorType: AUDIO                   */ 0x24u,
/*  bDescriptorSubtype                      */ 0x03u,
/*  bTerminalID                             */ 0x02u,
/*  wTerminalType                           */ 0x02u, 0x06u,
/*  bAssocTerminal                          */ 0x00u,
/*  bSourceID                               */ 0x01u,
/*  iTerminal                               */ 0x00u,
```

```
/************************************************************
* Standard AudioStreaming Interface Descriptor, Alternate Setting 0
************************************************************/
/*  Interface Descriptor Length             */ 0x09u,
/*  DescriptorType: INTERFACE               */ 0x04u,
/*  bInterfaceNumber                        */ 0x01u,
/*  bAlternateSetting                       */ 0x00u,
/*  bNumEndpoints                           */ 0x00u,
/*  bInterfaceClass                         */ 0x01u,
/*  bInterfaceSubClass                      */ 0x02u,
/*  bInterfaceProtocol                      */ 0x00u,
/*  iInterface                              */ 0x08u,
/************************************************************
* Standard AudioStreaming Interface Descriptor, Alternate Setting 1
************************************************************/
/*  Interface Descriptor Length             */ 0x09u,
/*  DescriptorType: INTERFACE               */ 0x04u,
/*  bInterfaceNumber                        */ 0x01u,
/*  bAlternateSetting                       */ 0x01u,
/*  bNumEndpoints                           */ 0x02u,
/*  bInterfaceClass                         */ 0x01u,
/*  bInterfaceSubClass                      */ 0x02u,
/*  bInterfaceProtocol                      */ 0x00u,
/*  iInterface                              */ 0x09u,
/************************************************************
* Class-Specific AudioStreaming General Interface Descriptor
************************************************************/
/*  AS General Descriptor Length            */ 0x07u,
/*  DescriptorType: AUDIO                   */ 0x24u,
/*  bDescriptorSubtype                      */ 0x01u,
/*  bTerminalLink                           */ 0x01u,
/*  bDelay                                  */ 0x0Eu,
/*  wFormatTag                              */ 0x01u, 0x00u,
/************************************************************
* AudioStreaming Format Type I Descriptor
************************************************************/
/*  AS Format Type I Descriptor Length      */ 0x0Eu,
/*  DescriptorType: AUDIO                   */ 0x24u,
/*  bDescriptorSubtype                      */ 0x02u,
/*  bFormatType                             */ 0x01u,
/*  bNrChannels                             */ 0x02u,
/*  bSubframeSize                           */ 0x02u,
/*  bBitResolution                          */ 0x10u,
/*  bSamFreqType                            */ 0x00u,
/*  tLowerSamFreq                           */ 0x44u, 0xACu, 0x00u,
/*  tUpperSamFreq                           */ 0x44u, 0xACu, 0x00u,
```

```
/************************************************************
* Standard OUT Endpoint Descriptor
************************************************************/
/*  Endpoint Descriptor Length              */ 0x09u,
/*  DescriptorType: ENDPOINT                */ 0x05u,
/*  bEndpointAddress                        */ 0x02u,
/*  bmAttributes                            */ 0x05u,
/*  wMaxPacketSize                          */ 0xC0u, 0x00u,
/*  bInterval                               */ 0x01u,
/*  bRefresh                                */ 0x00u,
/*  bSynchAddress                           */ 0x81u,
/************************************************************
* Class-Specific AudioStreaming Endpoint Descriptor
************************************************************/
/*  Endpoint Descriptor Length              */ 0x07u,
/*  DescriptorType: CS_ENDPOINT             */ 0x25u,
/*  bDescriptorSubtype                      */ 0x01u,
/*  bmAttributes                            */ 0x01u,
/*  bLockDelayUnits                         */ 0x00u,
/*  wLockDelay                              */ 0x00u, 0x00u,
/************************************************************
* Standard IN Endpoint Descriptor
************************************************************/
/*  Endpoint Descriptor Length              */ 0x09u,
/*  DescriptorType: ENDPOINT                */ 0x05u,
/*  bEndpointAddress                        */ 0x81u,
/*  bmAttributes                            */ 0x11u,
/*  wMaxPacketSize                          */ 0x03u, 0x00u,
/*  bInterval                               */ 0x01u,
/*  bRefresh                                */ 0x04u,
/*  bSynchAddress                           */ 0x00u,
/************************************************************/
```

A.2 Adaptive Mode USB Descriptors

Most descriptors in the adaptive mode implementation are identical to the descriptors used for the asynchronous mode presented in Section A.1. Due to there not being any IN endpoint providing feedback, there is also no standard IN endpoint descriptor in the descriptor table for the adaptive mode implementation. Side effects of this are that the number of endpoints reported by the bNumEndpoints parameter in the standard AudioStreaming interface descriptor changes from two to one and that the wTotalLength field of the configuration descriptor also must be updated as removal of the standard IN endpoint descriptor shortens the total length for all the descriptors by nine bytes. The bmAttributes field in the standard AudioStreaming endpoint descriptor is the parameter that decides the synchronization mode. Only the descriptors that differ from the ones used in the asynchronous mode implemen-

tation have been included in this section and all fields that have a different value have been highlighted with yellow. The rest of the descriptors are identical to the ones used in the asynchronous mode implementation.

```
/**********************************************************************
* Configuration Descriptor
**********************************************************************/
/*  Config Descriptor Length                */ 0x09u,
/*  DescriptorType: CONFIG                  */ 0x02u,
/*  wTotalLength                            */ 0x67u, 0x00u,
/*  bNumInterfaces                          */ 0x02u,
/*  bConfigurationValue                     */ 0x01u,
/*  iConfiguration                          */ 0x00u,
/*  bmAttributes                            */ 0x80u,
/*  bMaxPower                               */ 0x32u,
/**********************************************************************
* Standard AudioStreaming Interface Descriptor, Alternate Setting 1
**********************************************************************/
/*  Interface Descriptor Length             */ 0x09u,
/*  DescriptorType: INTERFACE               */ 0x04u,
/*  bInterfaceNumber                        */ 0x01u,
/*  bAlternateSetting                       */ 0x01u,
/*  bNumEndpoints                           */ 0x01u,
/*  bInterfaceClass                         */ 0x01u,
/*  bInterfaceSubClass                      */ 0x02u,
/*  bInterfaceProtocol                      */ 0x00u,
/*  iInterface                              */ 0x09u,
/**********************************************************************
* Standard OUT Endpoint Descriptor
**********************************************************************/
/*  Endpoint Descriptor Length              */ 0x09u,
/*  DescriptorType: ENDPOINT                */ 0x05u,
/*  bEndpointAddress                        */ 0x02u,
/*  bmAttributes                            */ 0x09u,
/*  wMaxPacketSize                          */ 0xC0u, 0x00u,
/*  bInterval                               */ 0x01u,
/*  bRefresh                                */ 0x00u,
/*  bSynchAddress                           */ 0x00u,
/**********************************************************************/
```

A.3 Synchronous Mode USB Descriptors

The situation is almost exactly the same for the synchronous mode descriptor table as it is for adaptive mode, therefore only the descriptors that differ from the listing of the asynchronous mode descriptor table presented in Section A.1 are shown here. The only difference between the adaptive mode and the synchronous mode

descriptors is the bmAttributes field in the standard OUT endpoint descriptor which determines the synchronization mode of the endpoint. Everything else is the same and as there also is no IN endpoint for feedback in synchronous mode, therefore there is no standard IN endpoint descriptor and any change in the descriptor table related to it not existing applies in the same way for the synchronous mode descriptors as it does for adaptive mode. Field values differing from the asynchronous descriptors have been highlighted with yellow.

```
/**********************************************************************
* Configuration Descriptor
**********************************************************************/
/*  Config Descriptor Length                */ 0x09u,
/*  DescriptorType: CONFIG                  */ 0x02u,
/*  wTotalLength                            */ 0x67u, 0x00u,
/*  bNumInterfaces                          */ 0x02u,
/*  bConfigurationValue                     */ 0x01u,
/*  iConfiguration                          */ 0x00u,
/*  bmAttributes                            */ 0x80u,
/*  bMaxPower                               */ 0x32u,
/**********************************************************************
* Standard AudioStreaming Interface Descriptor, Alternate Setting 1
**********************************************************************/
/*  Interface Descriptor Length             */ 0x09u,
/*  DescriptorType: INTERFACE               */ 0x04u,
/*  bInterfaceNumber                        */ 0x01u,
/*  bAlternateSetting                       */ 0x01u,
/*  bNumEndpoints                           */ 0x01u,
/*  bInterfaceClass                         */ 0x01u,
/*  bInterfaceSubClass                      */ 0x02u,
/*  bInterfaceProtocol                      */ 0x00u,
/*  iInterface                              */ 0x09u,
/**********************************************************************
* Standard OUT Endpoint Descriptor
**********************************************************************/
/*  Endpoint Descriptor Length              */ 0x09u,
/*  DescriptorType: ENDPOINT                */ 0x05u,
/*  bEndpointAddress                        */ 0x02u,
/*  bmAttributes                            */ 0x0Du,
/*  wMaxPacketSize                          */ 0xC0u, 0x00u,
/*  bInterval                               */ 0x01u,
/*  bRefresh                                */ 0x00u,
/*  bSynchAddress                           */ 0x81u,
/**********************************************************************/
```

B

Register Maps for Si5351

B.1 Register Map Generated by ClockBuilder Pro

Presented in Table B.1 are the register values generated by the Skyworks ClockBuilder Pro software that were used to program the Adafruit Si5351 clock generator board in the asynchronous mode implementation, setting the clock frequency of the first output clock port to 2.8224 MHz.

Register	Register Value	Register	Register Value
2	0x53	43	0x04
3	0x00	44	0x00
4	0x20	45	0x9D
7	0x00	46	0x60
15	0x00	47	0x00
16	0x0F	48	0x00
17	0x8C	49	0x00
18	0x8C	90	0x00
19	0x8C	91	0x00
20	0x8C	149	0x00
21	0x8C	150	0x00
22	0x8C	151	0x00
23	0x8C	152	0x00
26	0x02	153	0x00
27	0x71	154	0x00
28	0x00	155	0x00
29	0x0F	162	0x00
30	0xFE	163	0x00
31	0x00	164	0x00
32	0x00	165	0x00
33	0x62	183	0xD2
42	0x00		

Table B.1: Register values generated by ClockBuilder Pro for the Adafruit Si5351 clock generator.

B.2 Manually Generated Register Values

Listed in this section are the manually generated register values that were used in the adaptive mode implementation. The PLL A settings are located at registers 26–33 and the multisynth 0 settings reside at registers 42–49. To activate the integer mode setting to improve jitter performance for the multisynth 0 block, the 7^{th} bit of register 16 must be set to 1. The value in register 16 used in the adaptive mode implementation was therefore changed from 0x0F to 0x4F when integer mode for multisynth 0 was activated. Other than that, all other register values used in any of the adaptive mode implementations remained the same as in the ClockBuilder Pro generated asynchronous mode implementation register map presented in Table B.1.

Register	Register Value
26	0x02
27	0x71
28	0x00
29	0x09
30	0x4A
31	0x00
32	0x00
33	0x56

Table B.2: Manually calculated register values used in adaptive mode for the Adafruit Si5351 clock generator PLL A block.

	Registers							
	42	43	44	45	46	47	48	49
Clock adjustment	Register value							
+2.0 %	0x00	0x01	0x00	0x64	0x00	0x00	0x00	0x00
+1.5 %	0x00	0x01	0x00	0x63	0x80	0x00	0x00	0x00
+1.0 %	0x00	0x01	0x00	0x63	0x00	0x00	0x00	0x00
+0.5 %	0x00	0x01	0x00	0x62	0x80	0x00	0x00	0x00
No adjustment	0x00	0x01	0x00	0x62	0x00	0x00	0x00	0x00
-0.5 %	0x00	0x01	0x00	0x61	0x80	0x00	0x00	0x00
-1.0 %	0x00	0x01	0x00	0x61	0x00	0x00	0x00	0x00
-1.5 %	0x00	0x01	0x00	0x60	0x80	0x00	0x00	0x00
-2.0 %	0x00	0x01	0x00	0x60	0x00	0x00	0x00	0x00

Table B.3: Manually calculated register values used in adaptive mode for the Adafruit Si5351 clock generator multisynth 0 block with the integer bit set.

	Registers							
	42	43	44	45	46	47	48	49
Clock adjustment	Register value							
+0.008 %	0x00	0xFA	0x00	0x62	0x02	0x00	0x00	0x0C
+0.006 %	0x00	0xFA	0x00	0x62	0x01	0x00	0x00	0x86
+0.004 %	0x00	0xFA	0x00	0x62	0x01	0x00	0x00	0x06
+0.002 %	0x00	0xFA	0x00	0x62	0x00	0x00	0x00	0x80
No adjustment	0x00	0xFA	0x00	0x62	0x00	0x00	0x00	0x00
-0.002 %	0x00	0xFA	0x00	0x61	0xFF	0x00	0x00	0x7A
-0.004 %	0x00	0xFA	0x00	0x61	0xFE	0x00	0x00	0xF4
-0.006 %	0x00	0xFA	0x00	0x61	0xFE	0x00	0x00	0x74
-0.008 %	0x00	0xFA	0x00	0x61	0xFD	0x00	0x00	0xEE

Table B.4: Manually calculated register values used in adaptive mode for the Adafruit Si5351 clock generator multisynth 0 block with the fractional bit set.

C

Jitter Histograms

This section lists the period and cycle-to-cycle jitter histograms created from the digital oscilloscope measurements for all the different design implementations. Images are organized firstly by USB synchronization type and secondly by clock source, with each set of plots belonging to a particular design having its own color scheme.

C.1 Asynchronous Mode Jitter Histograms

C.1.1 Asynchronous Mode with Si5351 Integer Multisynth

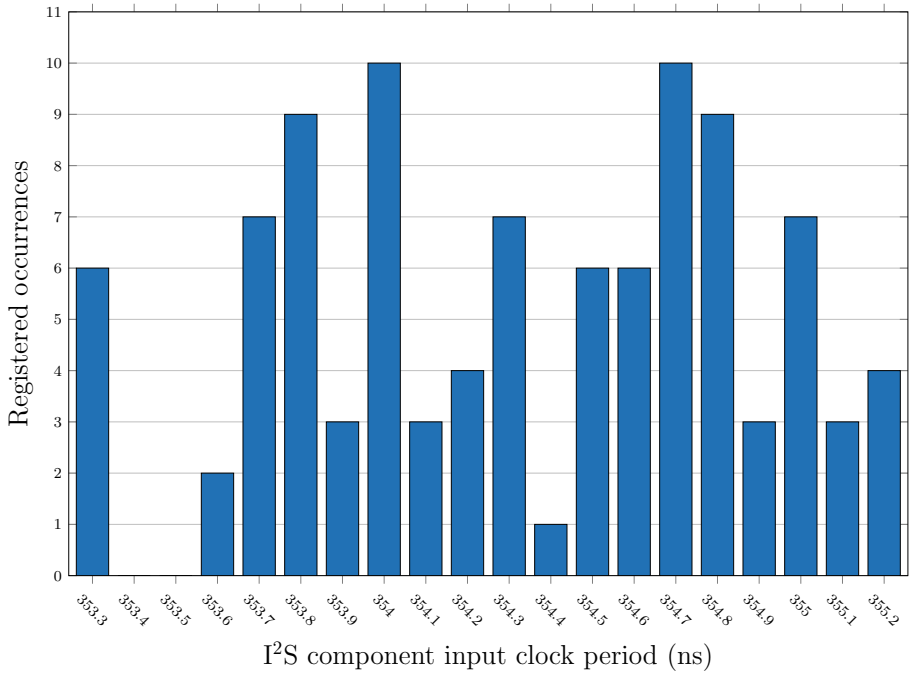


Figure C.1: I^{2}S component input clock period jitter histogram for asynchronous mode USB with the Si5351 multisynth set to fractional mode.

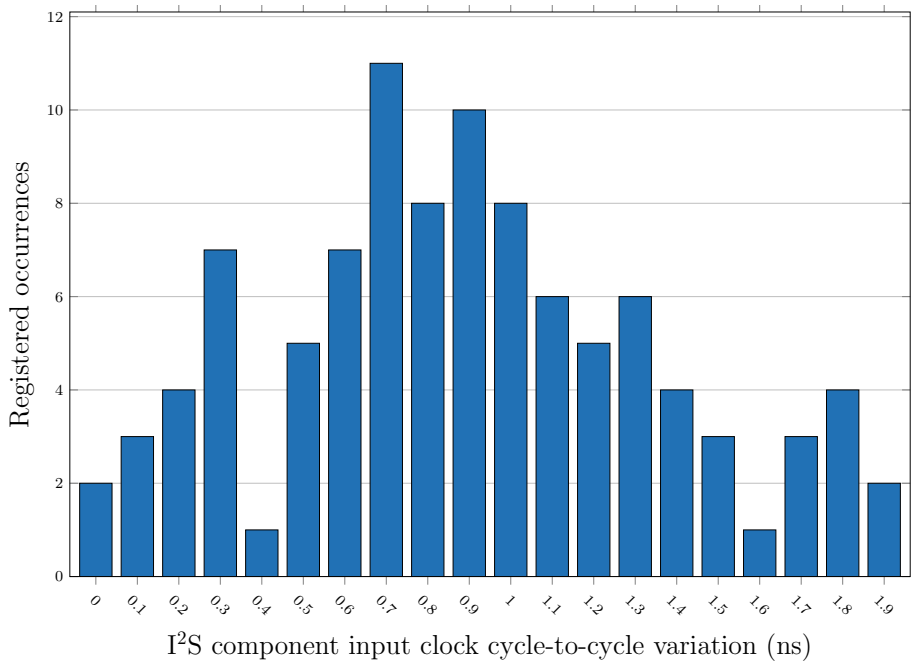


Figure C.2: I²S component input clock cycle-to-cycle jitter histogram for asynchronous mode USB with the Si5351 multisynth set to fractional mode.

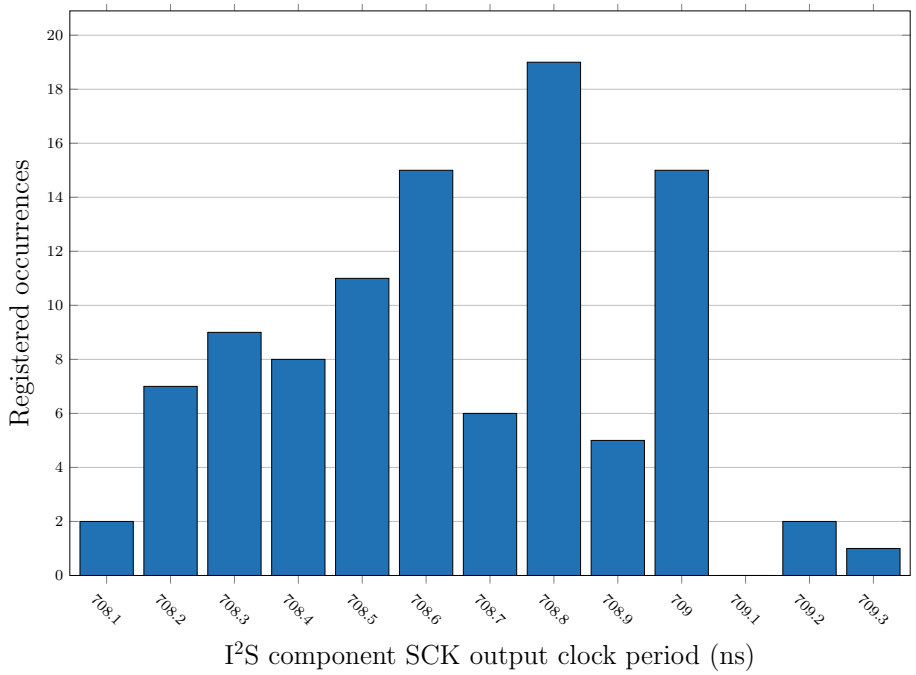


Figure C.3: I²S SCK signal period jitter histogram for asynchronous mode USB with the Si5351 multisynth set to fractional mode.

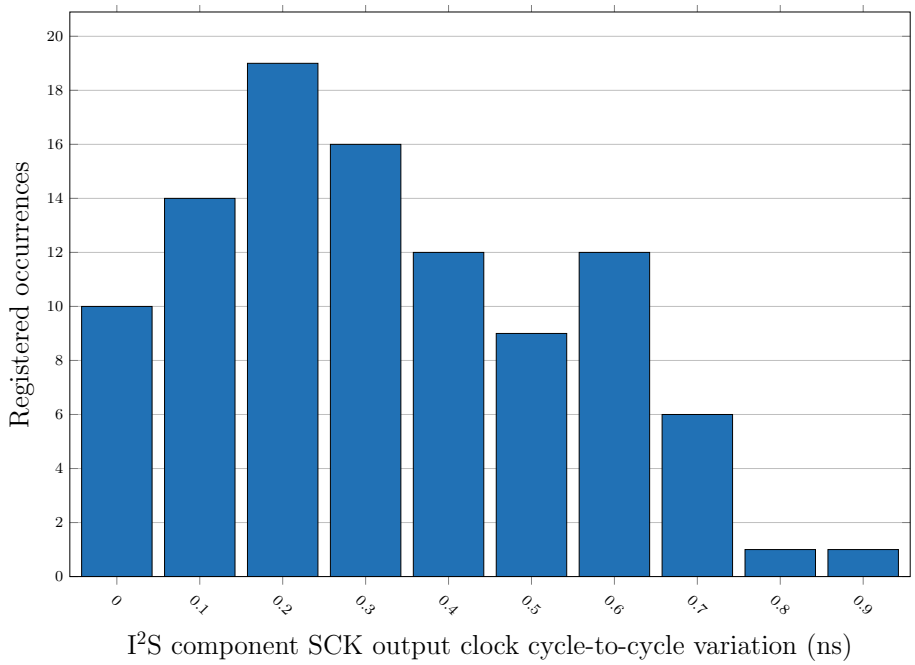


Figure C.4: I²S SCK signal cycle-to-cycle jitter histogram for asynchronous mode USB with the Si5351 multisynth set to fractional mode.

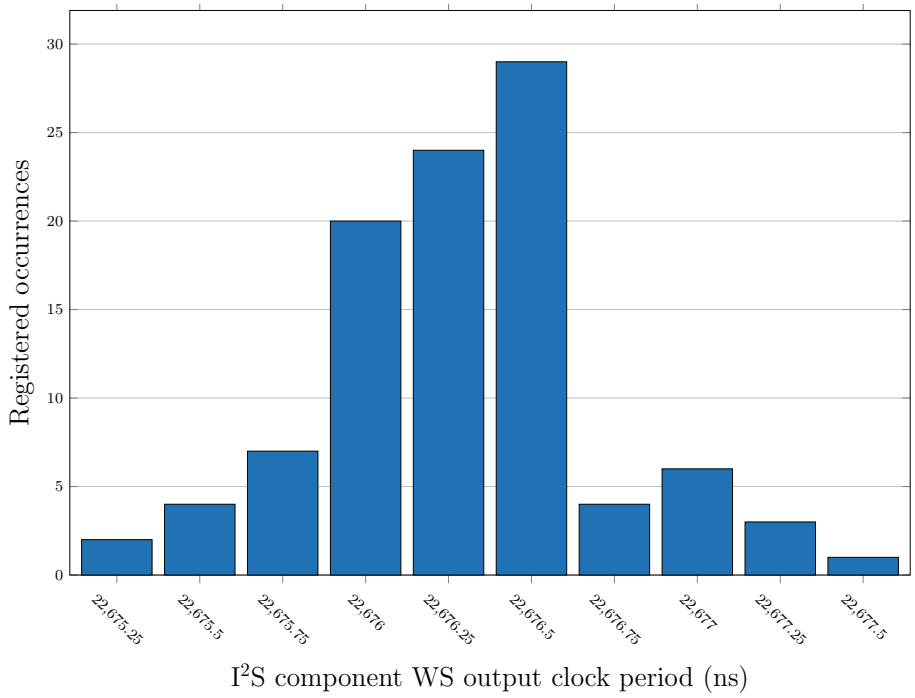


Figure C.5: I²S WS signal period jitter histogram for asynchronous mode USB with the Si5351 multisynth set to fractional mode.

C.1.2 Asynchronous Mode with External Crystal Oscillator

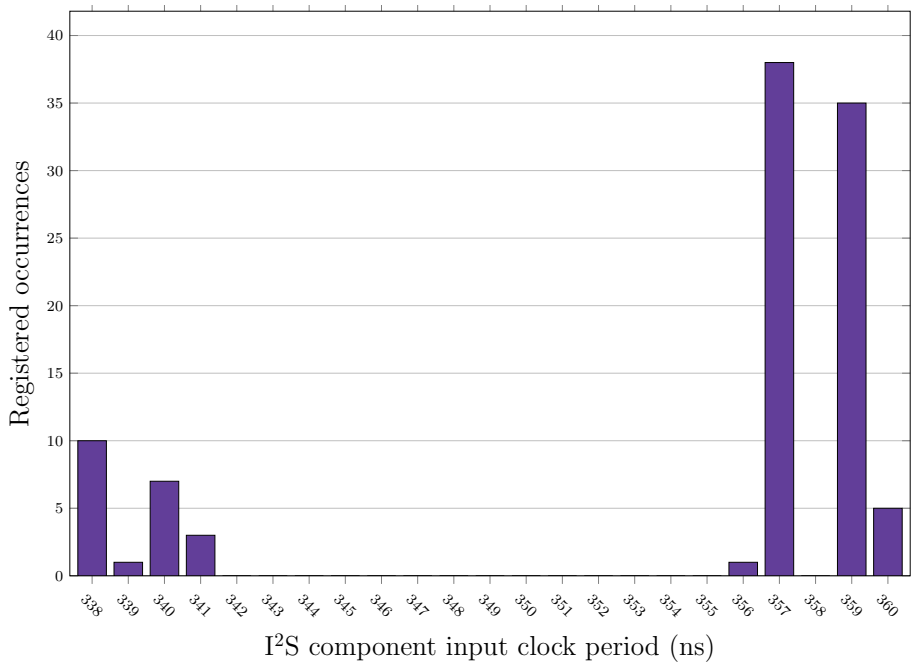


Figure C.6: I²S component input clock period jitter histogram for asynchronous mode USB using the external XO as source clock.

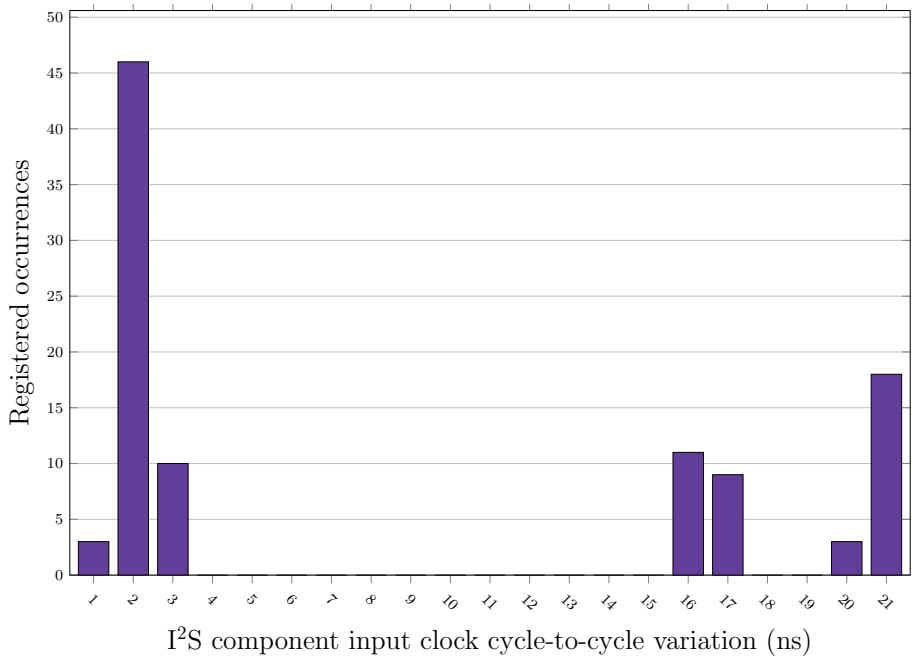


Figure C.7: I²S component input clock cycle-to-cycle jitter histogram for asynchronous mode USB using the external XO as source clock.

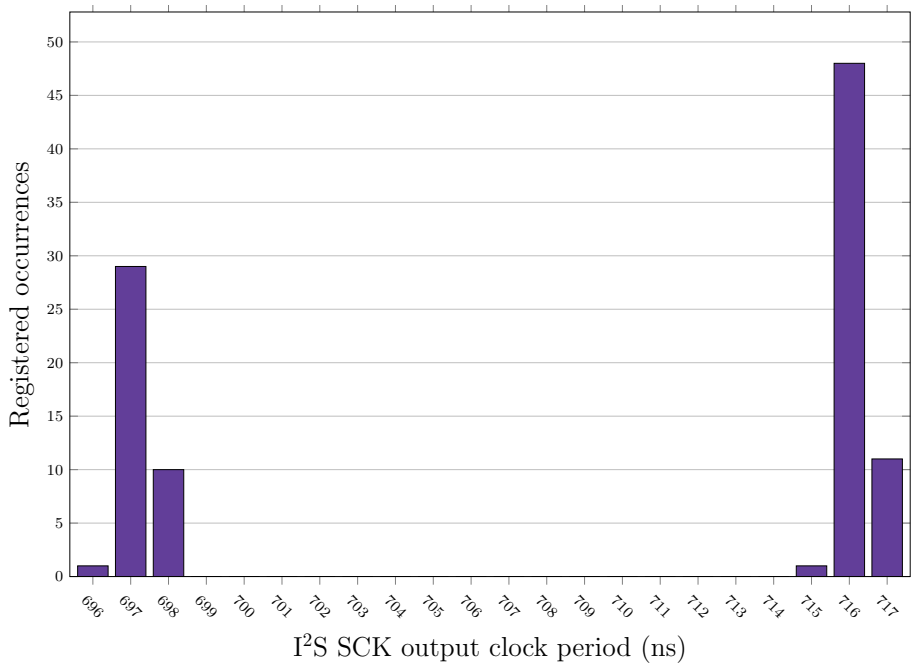


Figure C.8: I²S SCK signal period jitter histogram for asynchronous mode USB using the external XO as source clock.

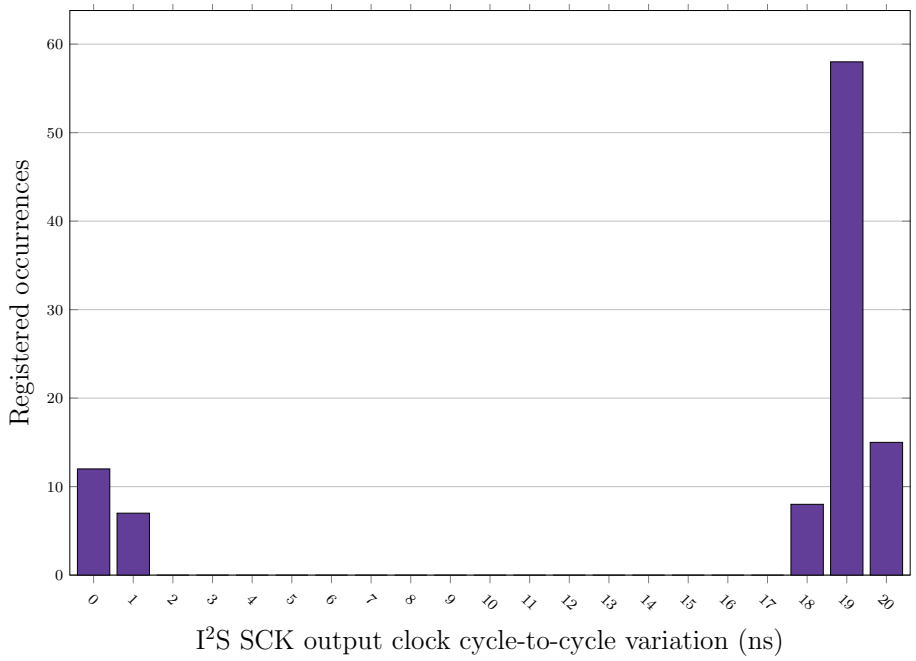


Figure C.9: I²S SCK signal cycle-to-cycle jitter histogram for asynchronous mode USB using the external XO as source clock.

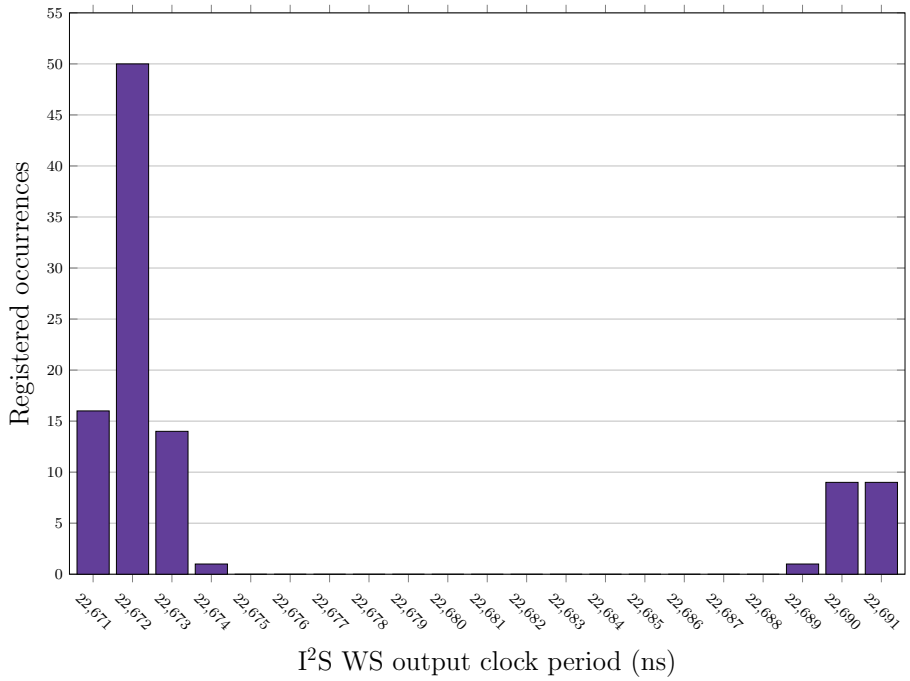


Figure C.10: I²S WS signal period jitter histogram for asynchronous mode USB using the external XO as source clock.

C.1.3 Asynchronous Mode with Fixed Frequency Clock

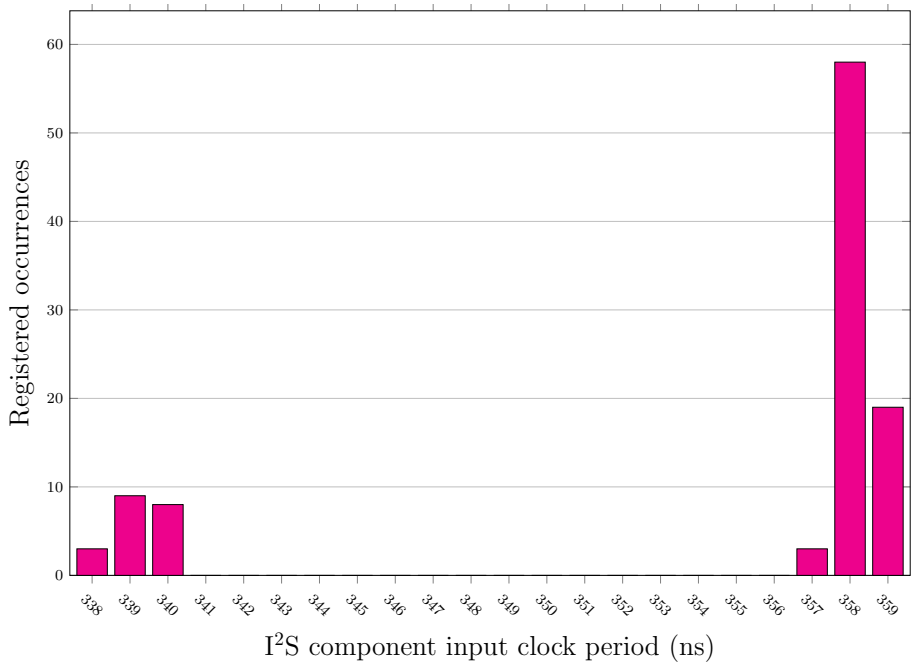


Figure C.11: I²S component input clock period jitter histogram for asynchronous mode USB using the external fixed frequency clock board as source clock.

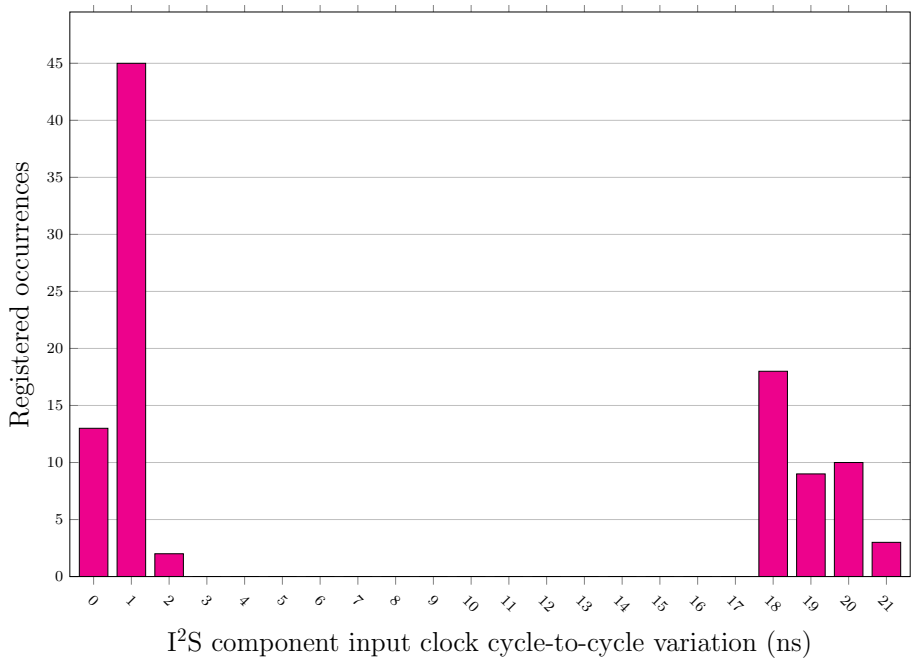


Figure C.12: I²S component input clock cycle-to-cycle jitter histogram for asynchronous mode USB using the external fixed frequency clock board as source clock.

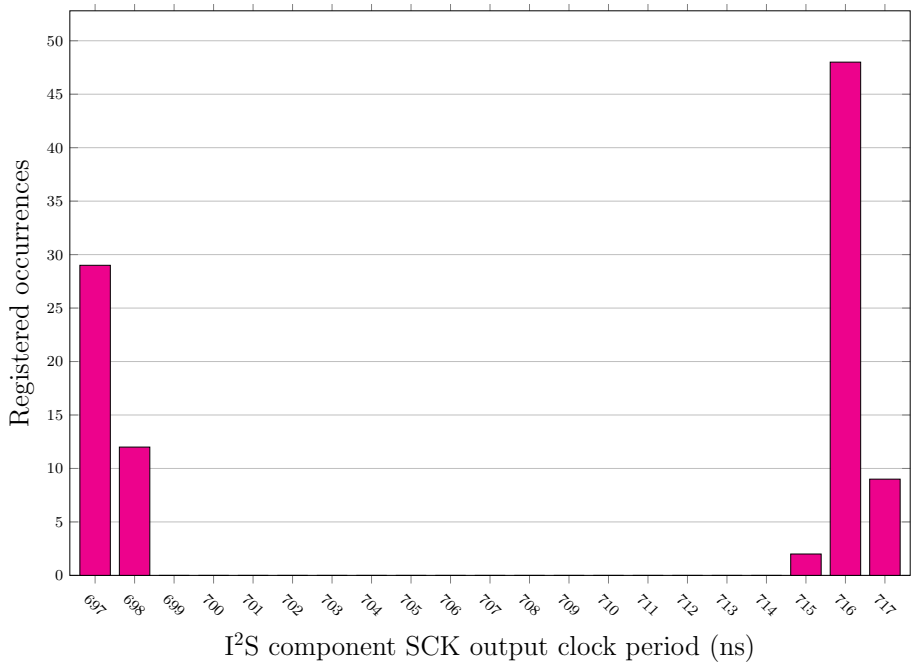


Figure C.13: I²S SCK signal period jitter histogram for asynchronous mode USB using the external fixed frequency clock board as source clock.

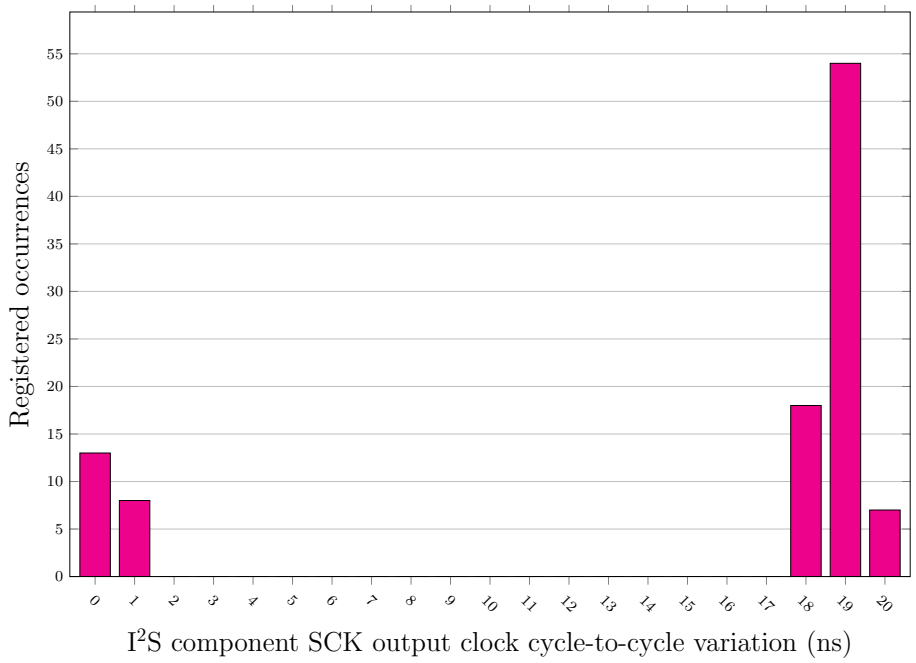


Figure C.14: I²S SCK signal cycle-to-cycle jitter histogram for asynchronous mode USB using the external fixed frequency clock board as source clock.

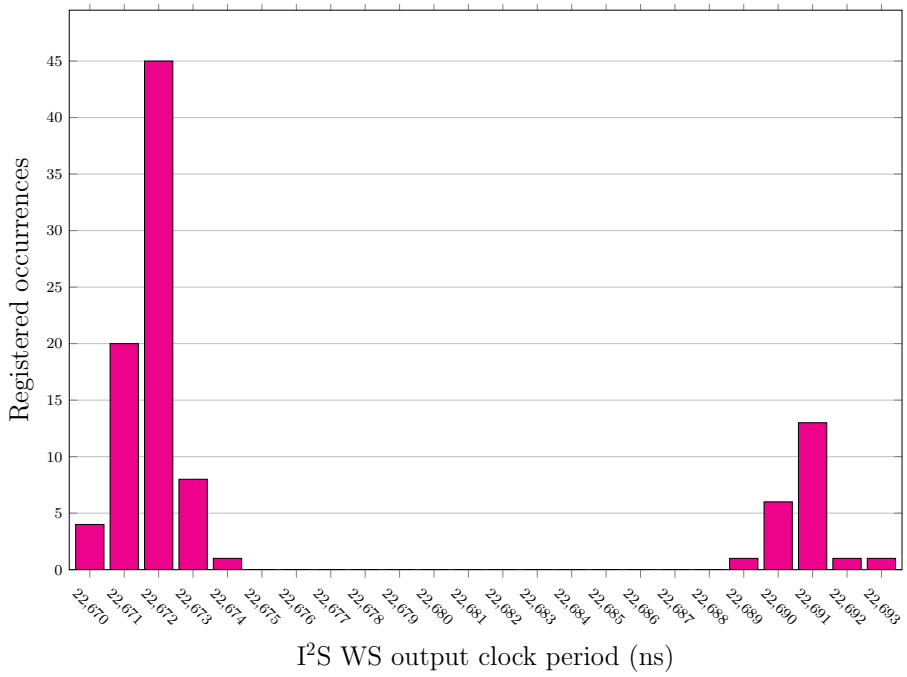


Figure C.15: I²S WS signal period jitter histogram for asynchronous mode USB using the external fixed frequency clock board as source clock.

C.1.4 Asynchronous Mode with Custom Fractional Divider

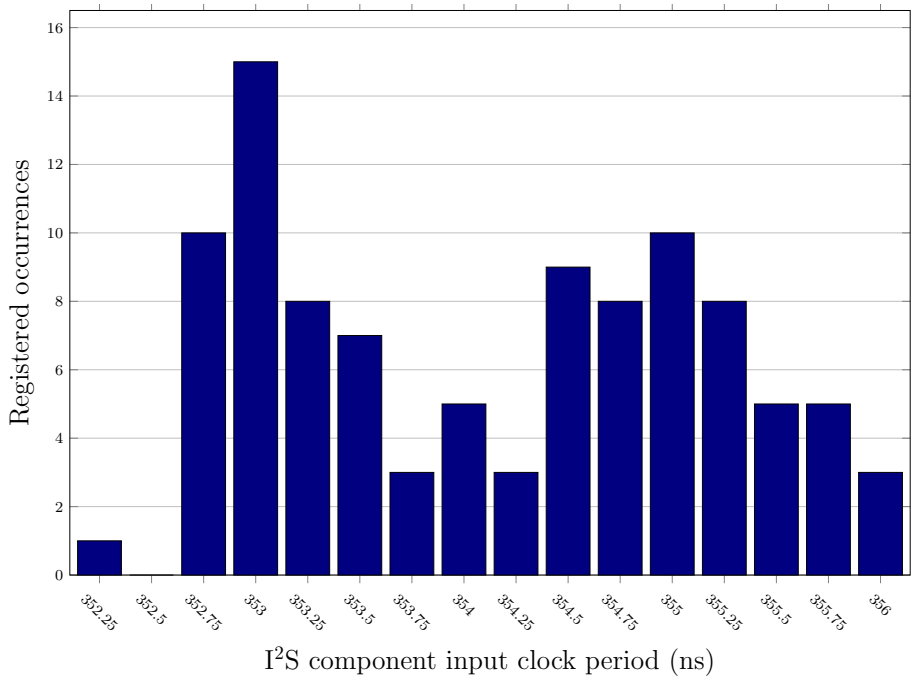


Figure C.16: I²S component input clock period jitter histogram for asynchronous mode USB using the IMO as source clock together with the fractional divider component.

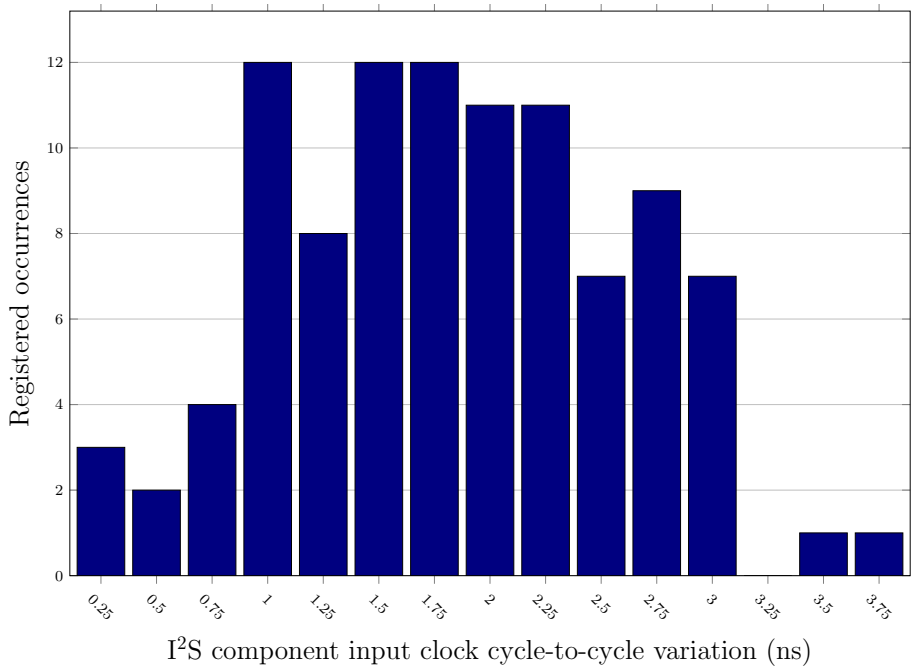


Figure C.17: I²S component input clock cycle-to-cycle jitter histogram for asynchronous mode USB using the IMO as source clock together with the fractional divider component.

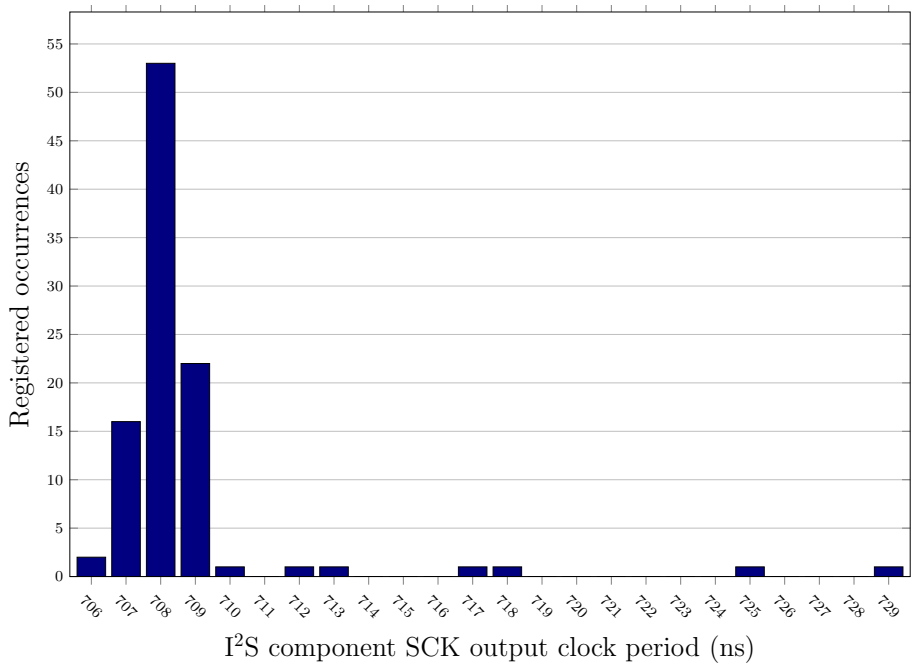


Figure C.18: I²S SCK signal period jitter histogram for asynchronous mode USB using the IMO as source clock together with the fractional divider component.

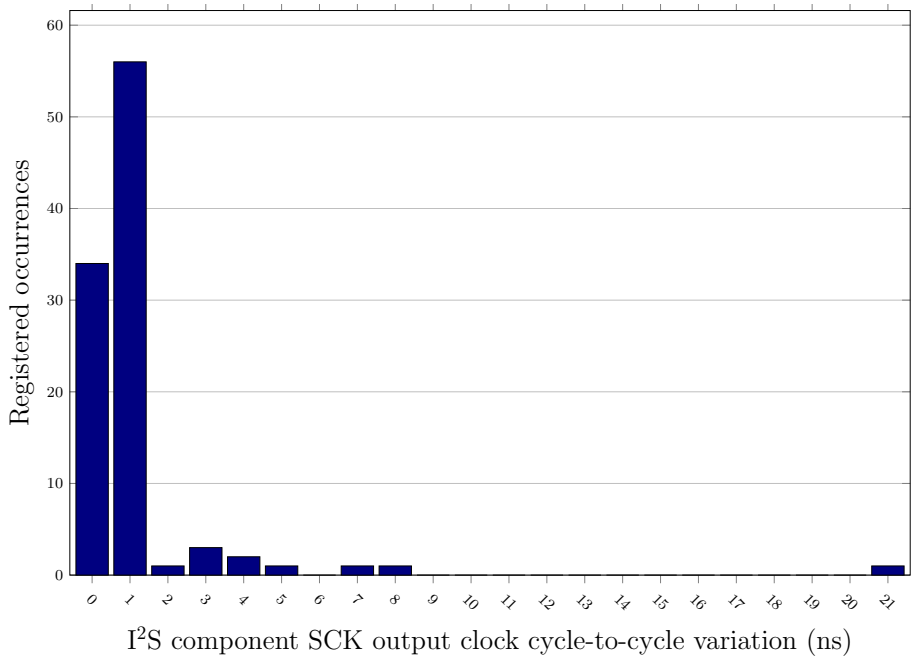


Figure C.19: I²S SCK signal cycle-to-cycle jitter histogram for asynchronous mode USB using the IMO as source clock together with the fractional divider component.

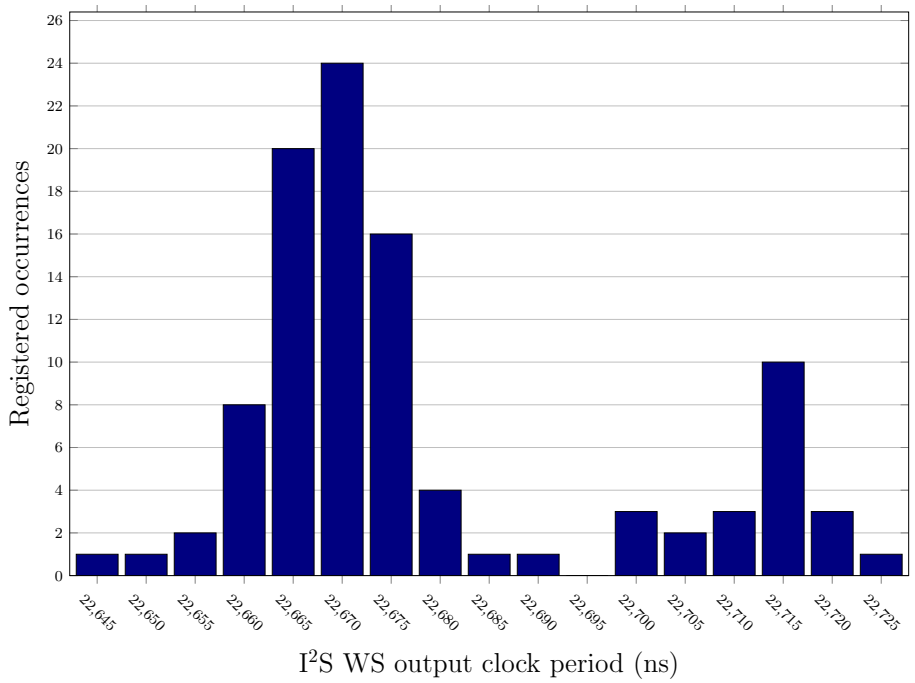


Figure C.20: I²S WS signal period jitter histogram for asynchronous mode USB using the IMO as source clock together with the fractional divider component.

C.1.5 Asynchronous Mode with Si5351 Integer Multisynth

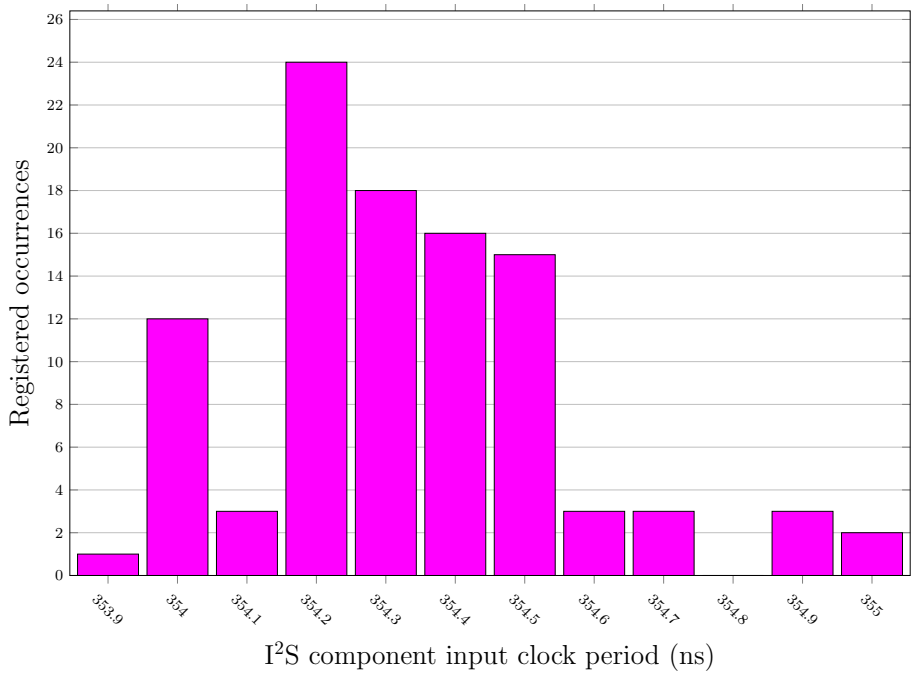


Figure C.21: I²S component input clock period jitter histogram for asynchronous mode USB with the Si5351 multisynth set to integer mode.

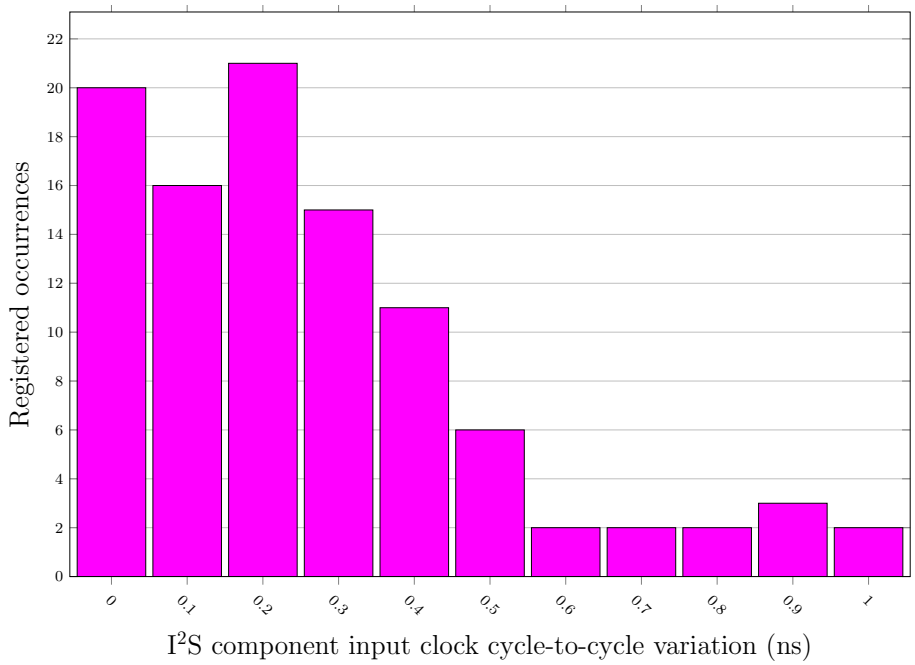


Figure C.22: I²S component input clock cycle-to-cycle jitter histogram for asynchronous mode USB with the Si5351 multisynth set to integer mode.

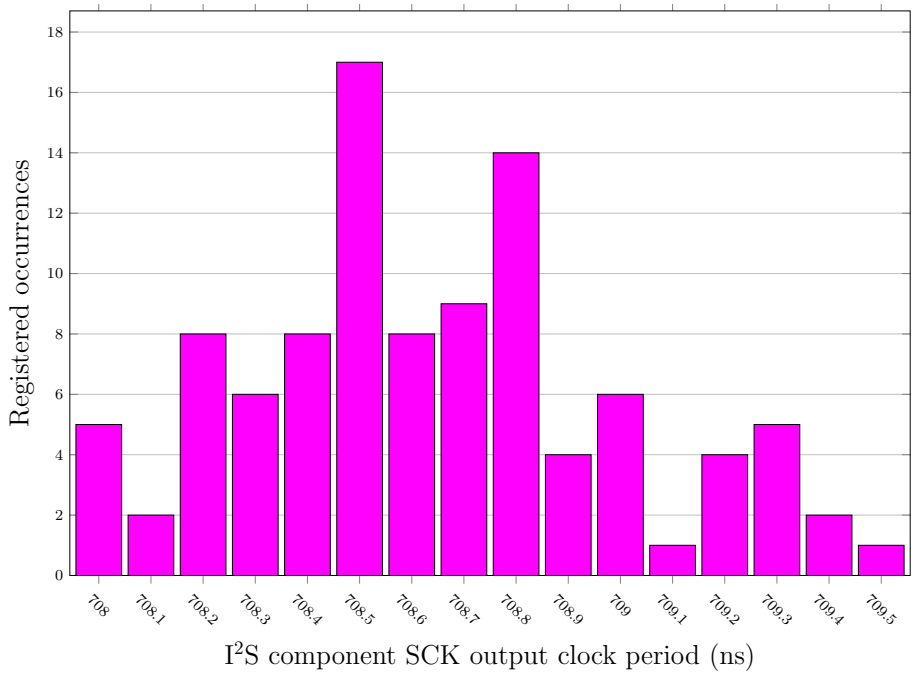


Figure C.23: I²S SCK signal period jitter histogram for asynchronous mode USB with the Si5351 multisynth set to integer mode.

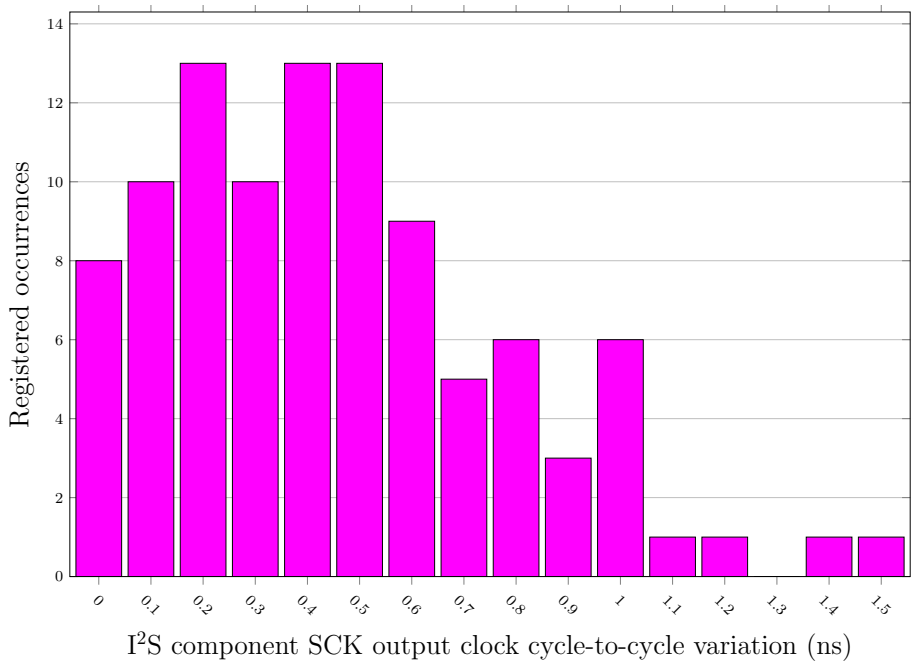


Figure C.24: I²S SCK signal cycle-to-cycle jitter histogram for asynchronous mode USB with the Si5351 multisynth set to integer mode.

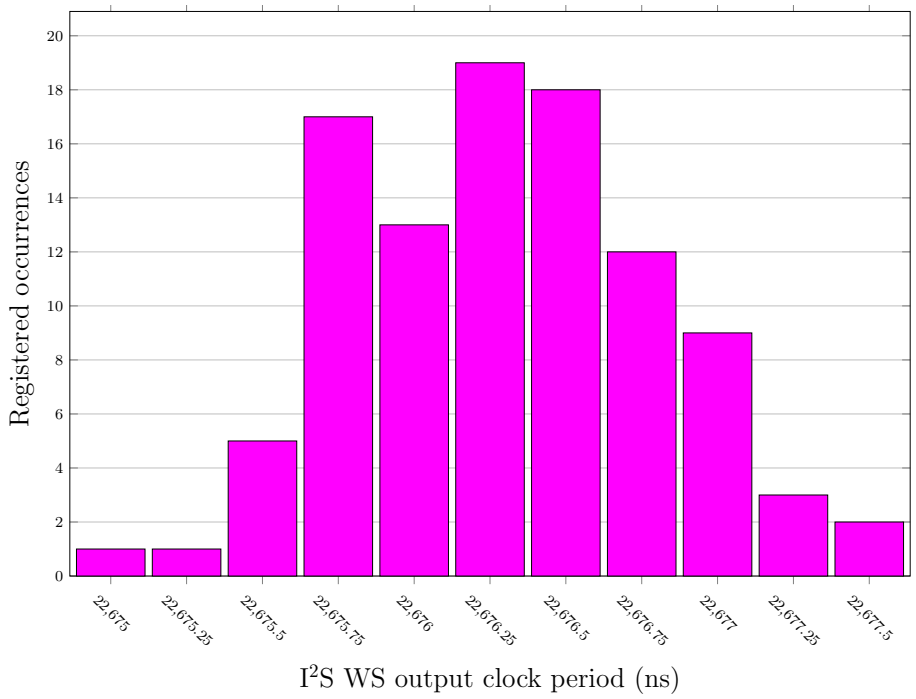


Figure C.25: I²S WS signal period jitter histogram for asynchronous mode USB with the Si5351 multisynth set to integer mode.

C.2 Adaptive Mode Jitter Histograms

C.2.1 Adaptive Mode with Si5351 Integer Multisynth

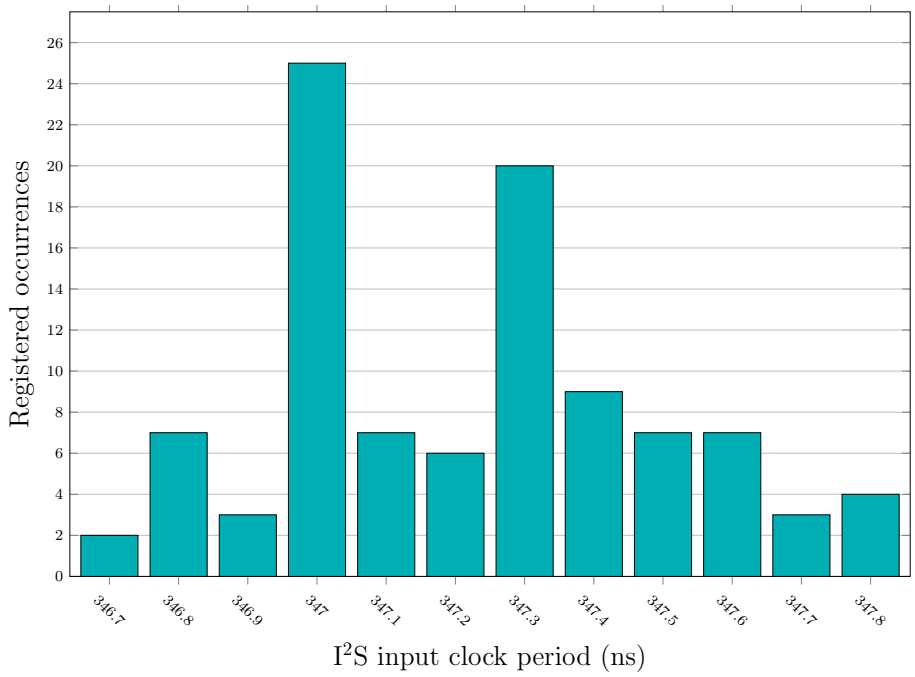


Figure C.26: I²S component input clock period jitter histogram for adaptive mode USB with the Si5351 multisynth set to integer mode.

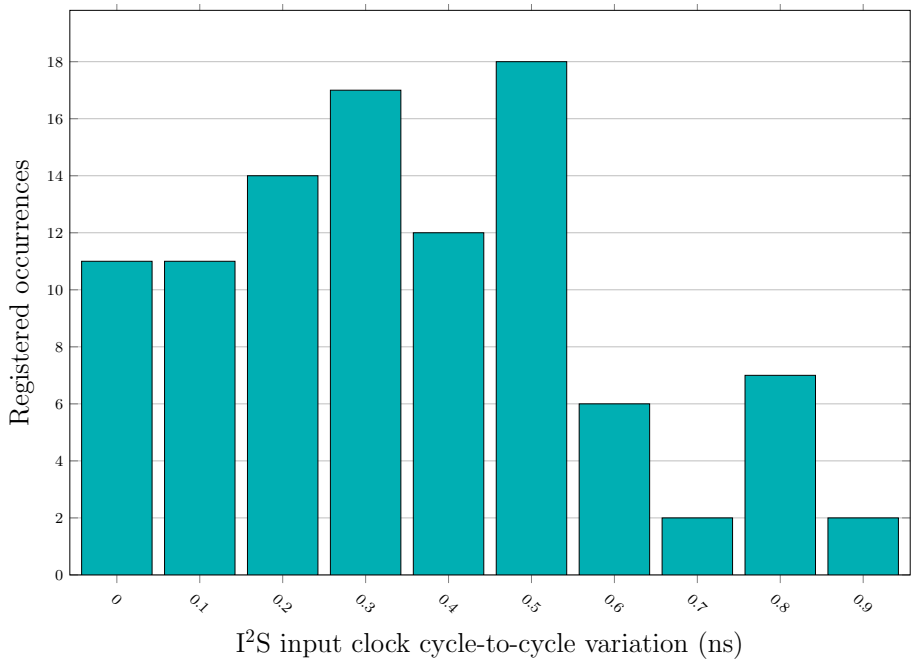


Figure C.27: I²S component input clock cycle-to-cycle jitter histogram for adaptive mode USB with the Si5351 multisynth set to integer mode.

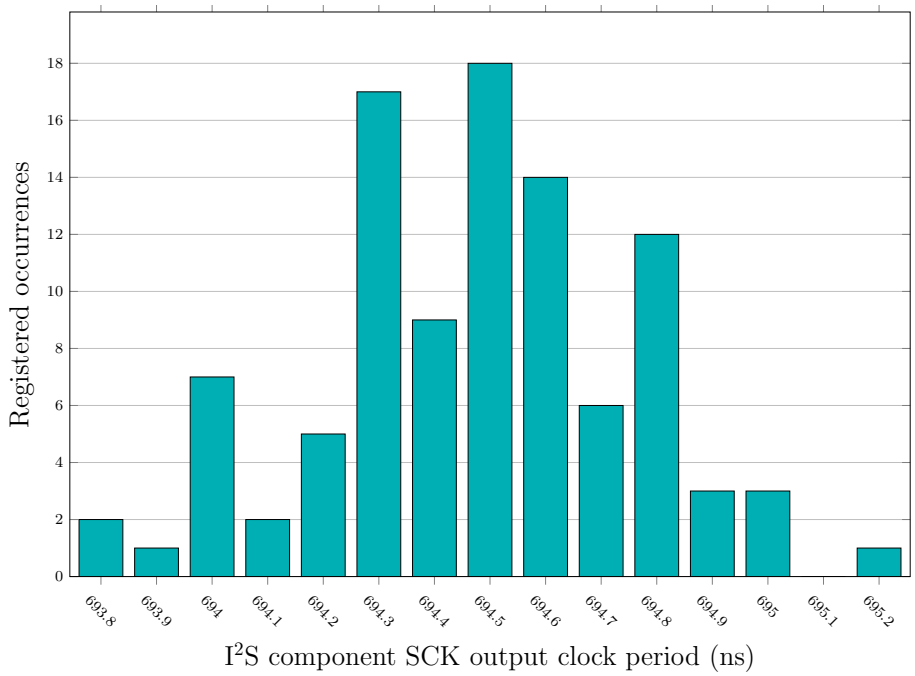


Figure C.28: I²S SCK signal period jitter histogram for adaptive mode USB with the Si5351 multisynth set to integer mode.

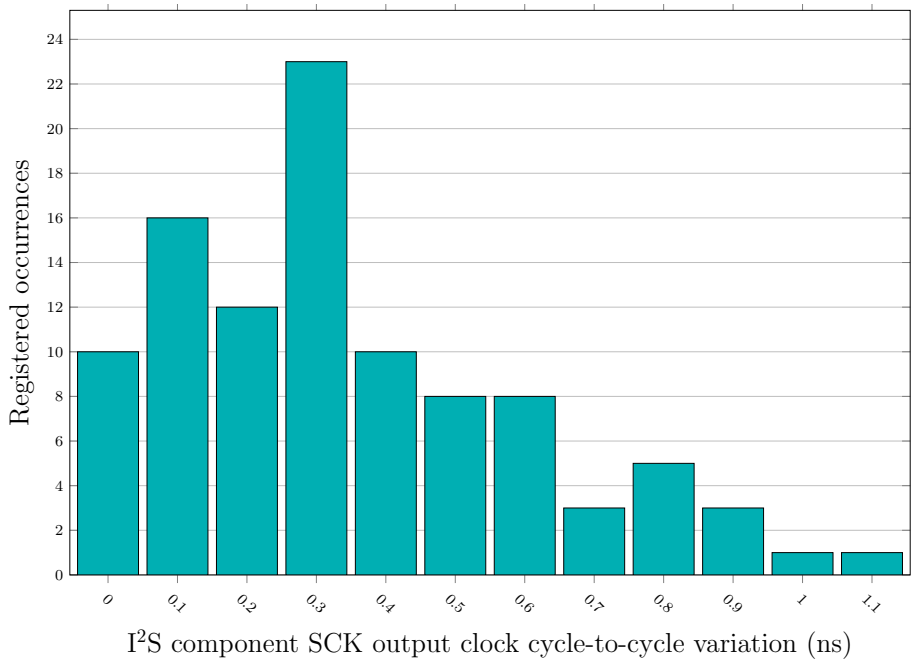


Figure C.29: I²S SCK signal cycle-to-cycle jitter histogram for adaptive mode USB with the Si5351 multisynth set to integer mode.

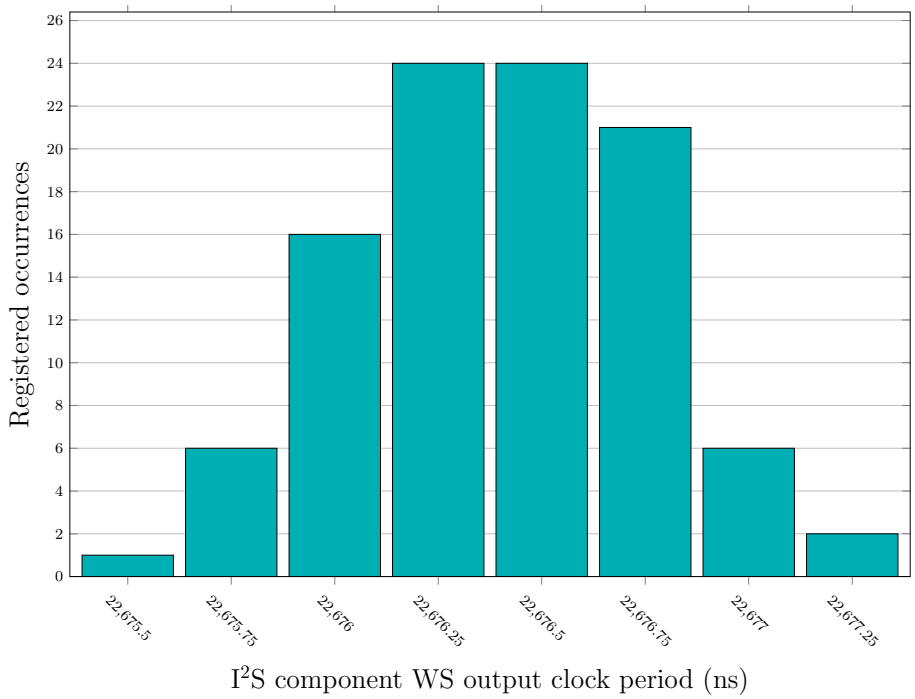


Figure C.30: I²S WS signal period jitter histogram for adaptive mode USB with the Si5351 multisynth set to integer mode.

C.2.2 Adaptive Mode with Si5351 Fractional Multisynth

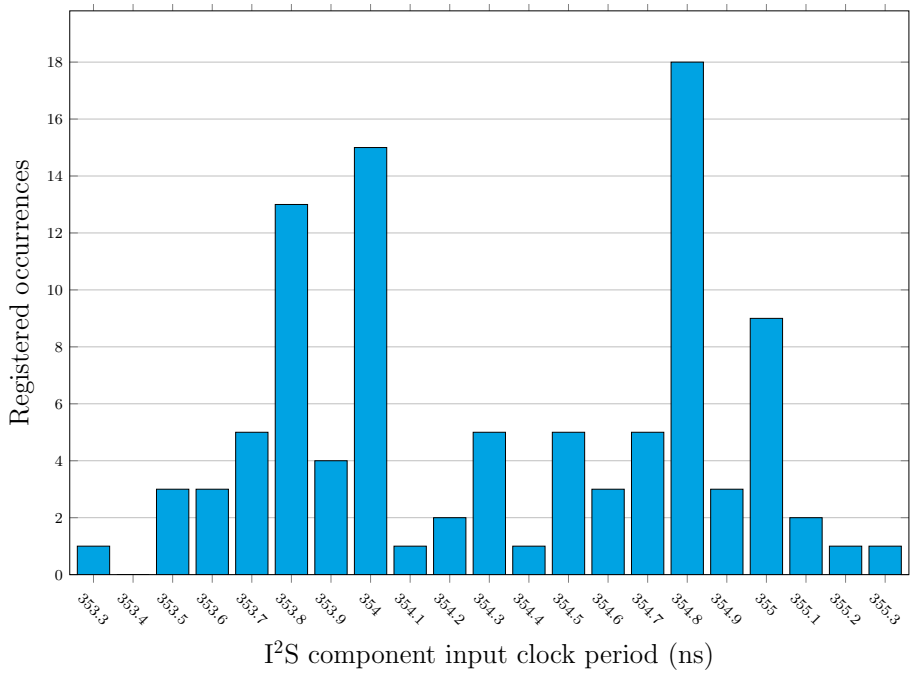


Figure C.31: I²S component input clock period jitter histogram for adaptive mode USB with the Si5351 multisynth set to fractional mode.

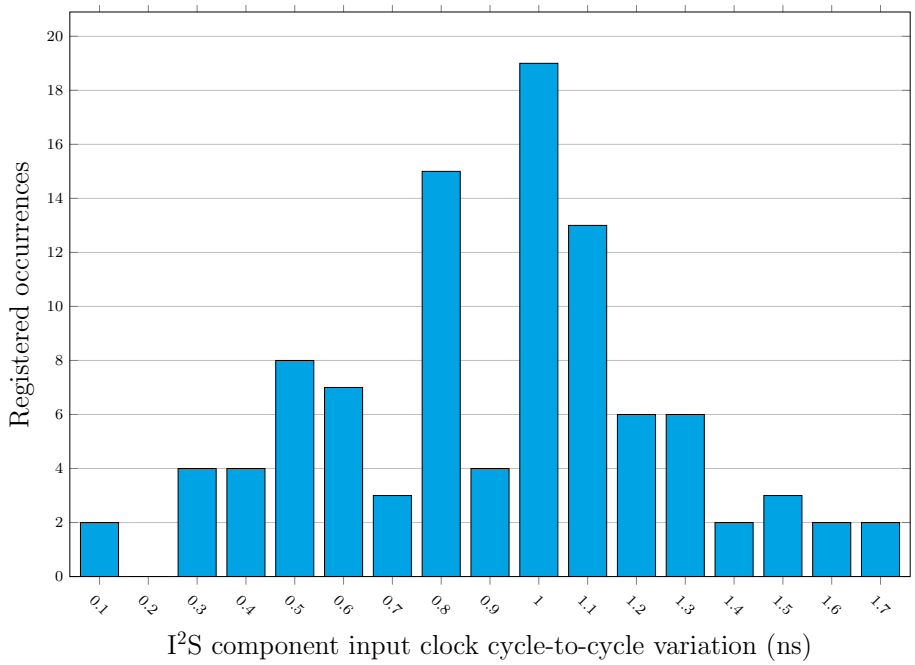


Figure C.32: I²S component input clock cycle-to-cycle jitter histogram for adaptive mode USB with the Si5351 multisynth set to fractional mode.

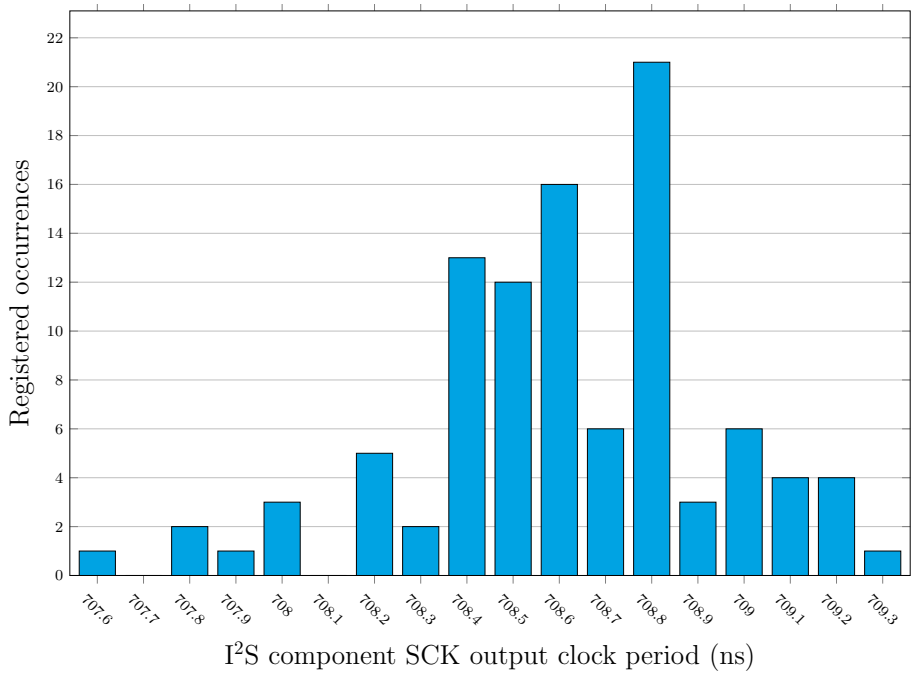


Figure C.33: I²S SCK signal period jitter histogram for adaptive mode USB with the Si5351 multisynth set to fractional mode.

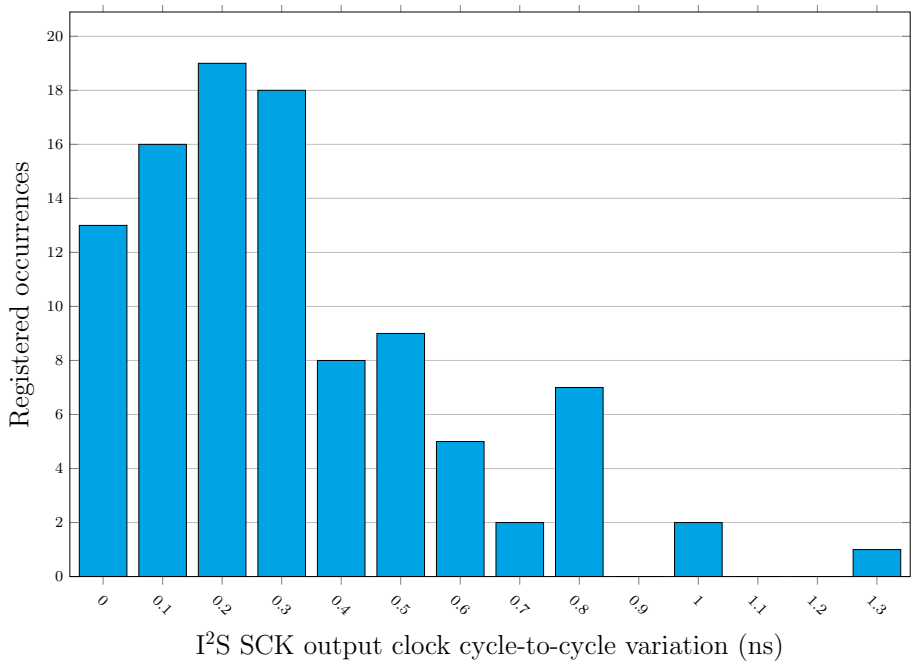


Figure C.34: I²S SCK signal cycle-to-cycle jitter histogram for adaptive mode USB with the Si5351 multisynth set to fractional mode.

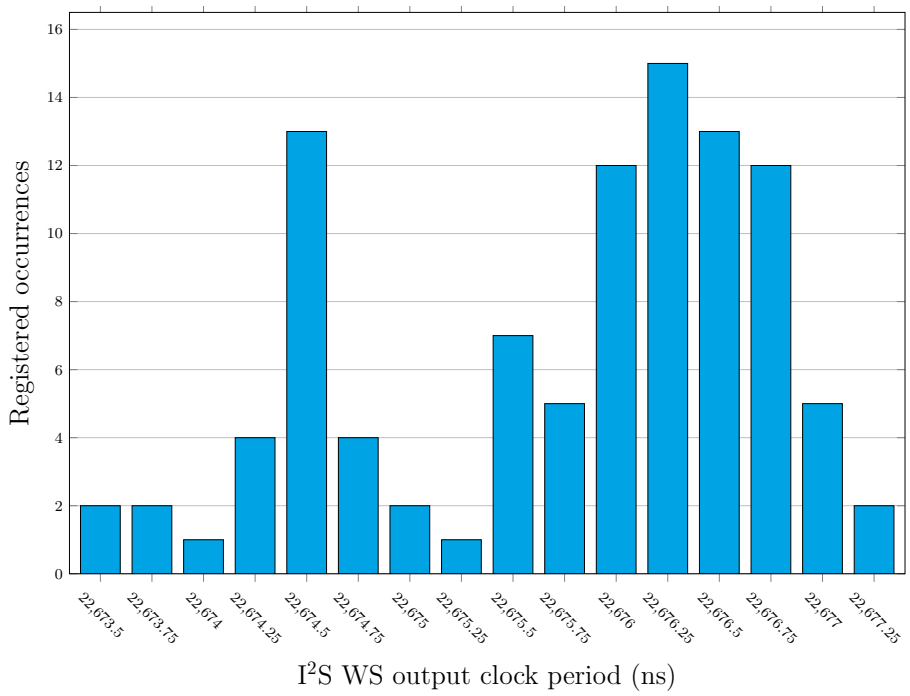


Figure C.35: I²S WS signal period jitter histogram for adaptive mode USB with the Si5351 multisynth set to fractional mode.

C.3 Synchronous Mode Jitter Histograms

C.3.1 Synchronous Mode with Custom Fractional Divider

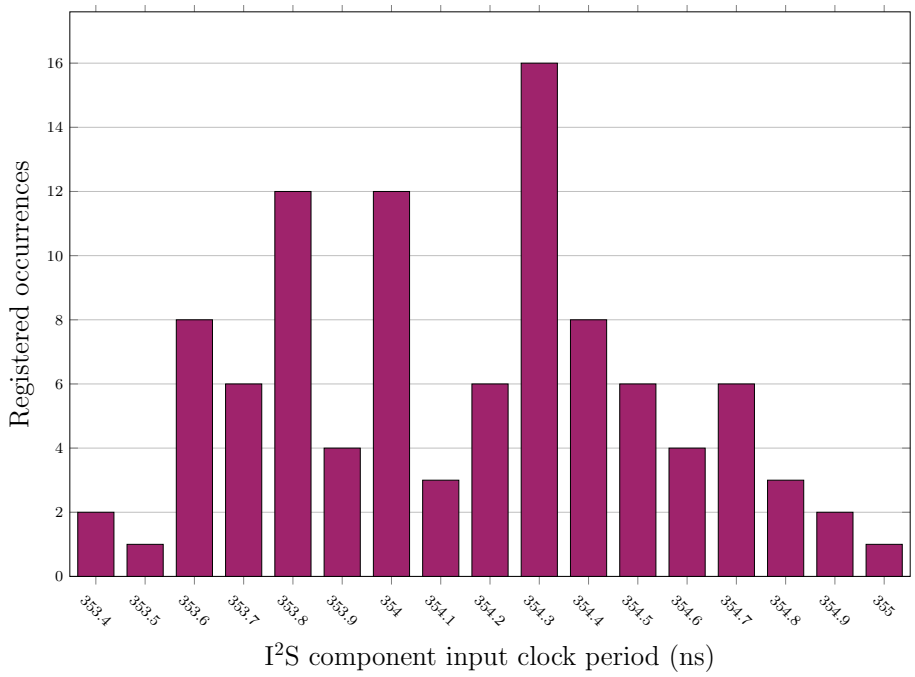


Figure C.36: I²S input clock period jitter histogram for synchronous mode USB using the IMO as source clock with the fractional divider component.

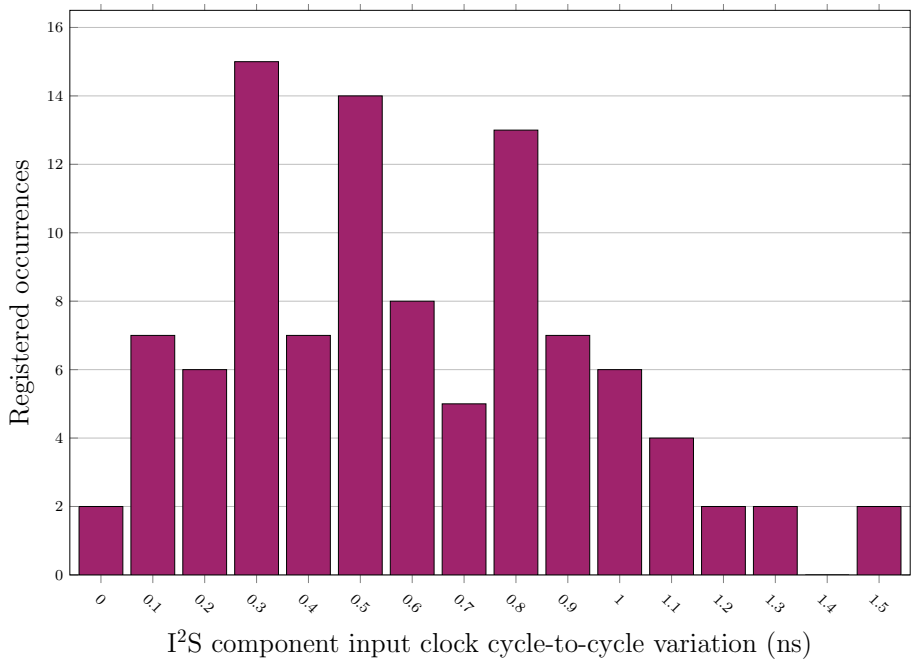


Figure C.37: I²S input clock cycle-to-cycle jitter histogram for synchronous mode USB using the IMO as source clock with the fractional divider component.

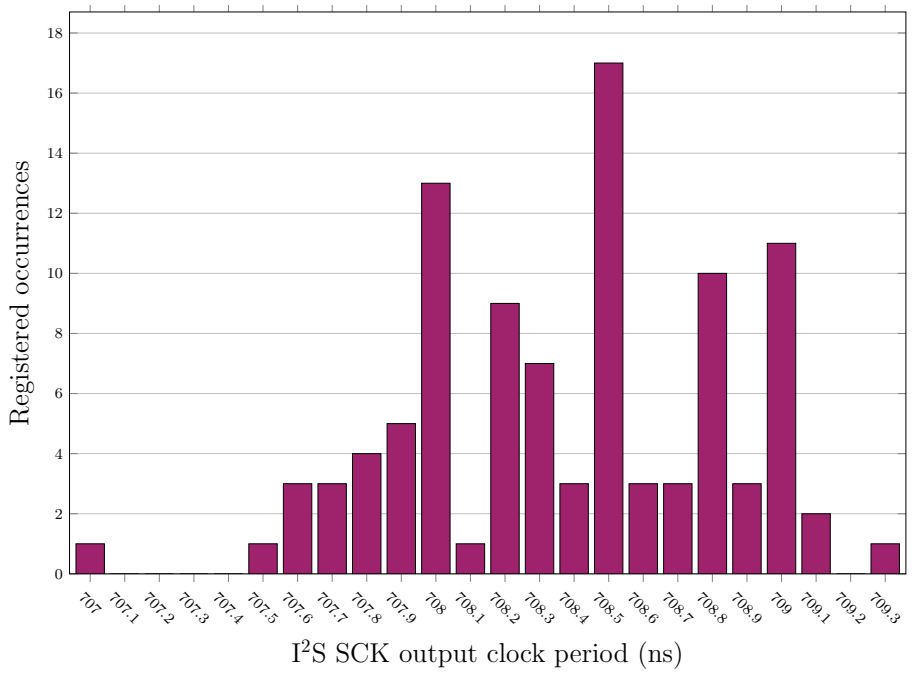


Figure C.38: I²S SCK signal period jitter histogram for synchronous mode USB using the IMO as source clock with the fractional divider component.

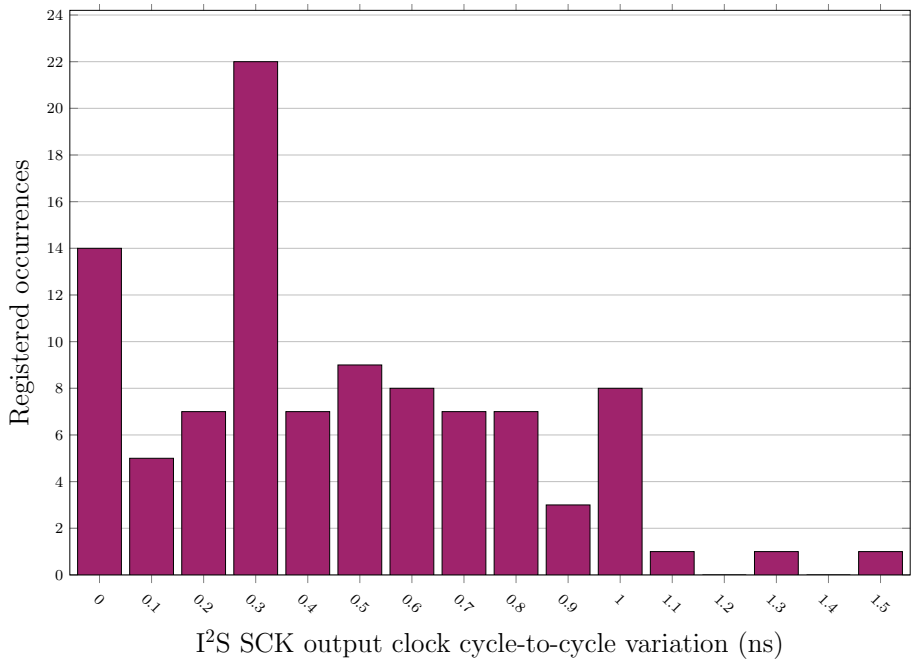


Figure C.39: I²S SCK signal cycle-to-cycle jitter histogram for synchronous mode USB using the IMO as source clock with the fractional divider component.

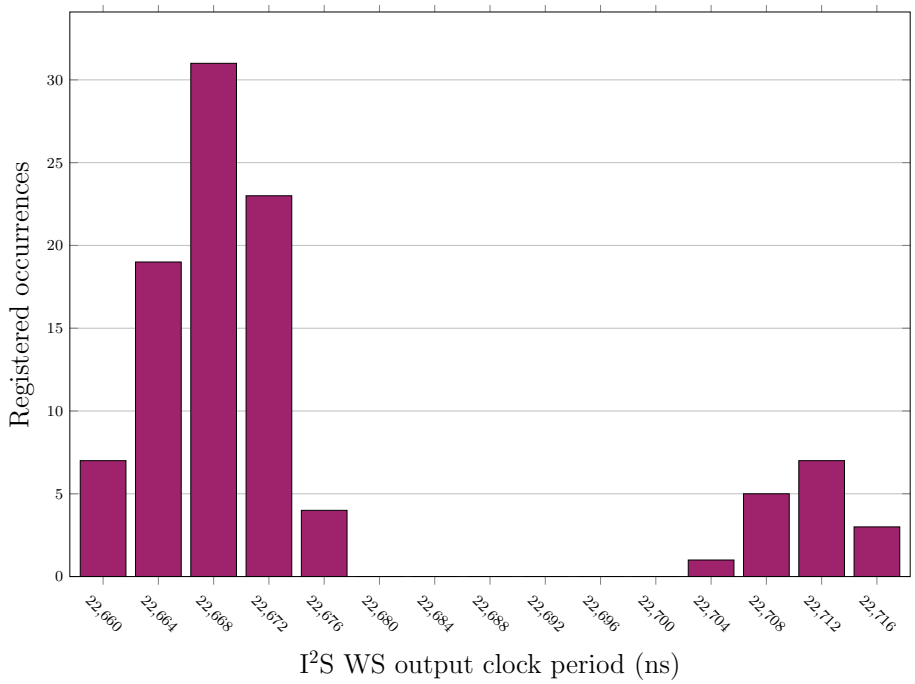


Figure C.40: I²S WS signal period jitter histogram for synchronous mode USB using the IMO as source clock with the fractional divider component.