An Optimized Software-Defined-Radio Implementation of Time-Slotted Carrier Synchronization for Distributed Beamforming

by

Min Ni

A Thesis
Submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Master of Science
in
Electrical and Computer Engineering
by

Feb 2011

APPROVED:

Professor Donald. Richard Brown III, Major Advisor

Professor John A. McNeill, Committee Member

Professor Andrew G. Klein, Committee Member
Abstract

This thesis describes the development of an optimized software-defined-radio implementation of a distributed beamforming system and presents experimental results for two-source and three-source wired-channel and acoustic-channel distributed beamforming using the time-slotted round-trip carrier synchronization protocol. The frequency and phase synthesizer used in this system is based on an optimized “hybrid” phase locked loop (PLL) with averaging window which is shown to have high frequency estimation accuracy and consistency. For the wired-channel experiments, each source node was implemented by a TMS320C6713DSK while for the acoustic experiments, each source node in the system was built using commercial off-the-shelf parts including TMS320C6713DSK, microphone, speaker, audio amplifier, and battery. The source node functionality including phase locked loops and the logic associated with the time-slotted round-trip carrier synchronization protocol was realized through real-time software independently running on each source node’s C6713 digital signal processor. Experimental results for two-source and three-source realizations of the wired-channel and acoustic-channel distributed beamforming system are presented. The results show that near-ideal beamforming performance can be consistently achieved at acoustic wavelengths equivalent to common radio frequency wavelengths.
Foremost, I would like to express my deep and sincere gratitude to my advisor, Professor Rick Brown, for the continuous support of my study and research, for his patience, motivation, and enthusiasm. As an advisor, his endless pursuit of perfection in his work and rigorous attitude toward research set up a good example for me. As a professor and electrical engineer, he taught me how to build a system and do research “brick by brick” and never give up. The things that I learnt from him will benefit me all through my life.

Besides my advisor, I am thankful to the rest of my committee members: Professor John A. McNeill and Professor Andrew G. Klein who asked me challenging questions and gave professional and insightful comments on my work.

I would also like to thank my graduate labmates, Yizheng, Qingxiong, Radu, Boyang and Boris. Thank you for taking the time to discuss my research problems and help me with the experiments.

Thank you, Mom and Dad. You were always there for me, pushing me and supporting me to be where I am today. I am forever in debt to you for your unconditional love.

Most importantly, I want to say thanks for Yunxing, for always encouraging me and caring about me.

Special thanks should be given to all my friends accompanying and helping me during my study at WPI.

Last, but not least, I am grateful for the generous support of the Texas Instruments for donating the equipment and the financial support of Worcester Polytechnic Institute.
Contents

List of Figures vi
List of Tables viii

1 Introduction 1

2 Background 4
2.1 Conventional beamforming 4
2.2 Distributed beamforming 5
2.2.1 Overview of Carrier Synchronization Techniques 6
2.2.2 Time-slotted Round-Trip Carrier Synchronization Protocol 7
2.3 Phase-locked loop 12
2.3.1 PLL basics 13
2.3.2 Examples of PLL behavior in the unlocked state 14
2.3.3 Third order analog PLL design 16
2.3.4 Software-based discrete-time PLL design 21
2.4 Real-time digital signal processing (DSP) 23
2.4.1 TMS320C6713DSK 24
2.4.2 Code Composer Studio 3.1 24

3 Optimization of PLL unlocked state performance 25
3.1 PLL behaviour in the unlocked state 26
3.2 A statistical study of PLL with MUL, XOR and PFD 31
3.2.1 Optimization of \( \omega_{3dB} \) for multiplier phase detector 32
3.2.2 Optimization of \( \omega_{3dB} \) for XOR phase detector 34
3.2.3 Optimization of \( \omega_{3dB} \) for PFD 35
3.3 Non-satisfactory performance of the optimized PLLs 37
3.3.1 Non-satisfactory frequency MSE 38
3.3.2 Inconsistent performance of PLL with MUL and XOR 38
3.4 Hybrid PLL with averaging window 42
3.5 Final optimized Hybrid PLL design 48
3.6 Experimental demonstration 49
List of Figures

2.1 Block diagram of a conventional beamformer. ........................................ 5
2.2 Block diagram of a distributed beamformer. ........................................ 6
2.3 Time-slotted round-trip carrier synchronization system with two source nodes. 8
2.4 Summary of the two-source time-slotted round-trip carrier synchronization protocol where PB and SB denote primary and secondary beacon synchronization timeslots, respectively. ................................................................. 9
2.5 Time-slotted round-trip carrier synchronization system with three source nodes. 10
2.6 Summary of the three-source time-slotted round-trip carrier synchronization protocol where PB and SB denote primary and secondary beacon synchronization timeslots, respectively. ................................................................. 12
2.7 A general PLL block diagram. ................................................................. 13
2.8 Input/output function of the VCO. ........................................................... 14
2.9 Phase step applied to the reference input of a PLL at t = 0.4s. ................. 15
2.10 Transient response of PLL to a phase step applied at t = 0.4s. ............... 15
2.11 Frequency step applied to the reference input of a PLL at t = 0.4s. ......... 16
2.12 Transient response of PLL to a frequency step applied at t = 0.4s. ......... 17
2.13 Bode diagram of a 2nd order active PI filter. ......................................... 19
3.1 11 transient responses of PLL to frequency and phase step applied at t=0. . 27
3.2 Overall view of frequency errors of PLL with different loop filter bandwidths. 28
3.3 Zoomed-in view of frequency errors between t = 0.45 to t = 0.55 of PLL with different loop filter bandwidths. Note that the PLL with ω_{3dB} = 0.01ω_o has converged but the PLL with ω_{3dB} = 0.005ω_o is still converging at t = 0.5. . 28
3.4 ω_{3dB} effects on settling time and output ripples. .................................. 30
3.5 Histograms of ω_1 - ω_2 at t = 0.5 seconds shown in Figure 3.4. ............ 30
3.6 Input signals of the multiplier phase detector.(a) Signal u_1(t) is a sine wave. Dashed line: θ_1 = 0; solid line: θ_1 > 0. (b) Signal u_2(t) is a cosine wave. Dashed line: θ_2 = 0; solid line: θ_2 > 0. .......................................................... 33
3.7 The multiplier phase detector output signal \bar{u}_d versus phase error \theta_e. .... 33
3.8 Waveforms of the signals for the XOR phase detector. .......................... 34
3.9 State diagram for the PFD. This drawing shows the events causing the PFD to change its current state. .......................................................... 36
3.10 Waveforms for positive phase error. The PFD output signal is pulsed to the
1 state. 36
3.11 \( \text{MSE}(\omega_1 - \omega_2)|_{t=0.5} \) for PLLs with MUL, XOR and PFD. 37
3.12 Outliers of frequency error at \( t = 0.5 \) seconds with optimal \( \omega_{\text{opt}} = 0.0061\omega_0 \). 40
3.13 Comparison of frequency errors with outlier and normal cases. 41
3.14 Three special cases of transient responses of the optimized PLL with multi-
plier phase detector. 42
3.15 Plot of the averaged PFD output signal \( \bar{u}_d \) versus phase error \( \theta_e \). 43
3.16 An example of PLL convergence behavior with MUL and PFD. 44
3.17 The zoom-in version of the PLL convergence behavior shown in Figure 3.16
with MUL and PFD between \( t = 0.45 \) to \( t = 0.55 \) seconds. 45
3.18 Block diagram of the PLL with averaging window and holdover function. 46
3.19 Implementation block diagram of an unsynchronized system and a synchro-
nized system. 50
3.20 The frequency offset of two unsynchronized signals by using the local oscillator
on the DSK boards. 51
3.21 The frequency offset of two synchronized signals by using the optimized hybrid PLL with averaging window. 52

4.1 Implementation block diagram of the two-source time-slotted round-trip carrier
synchronization system where the blue and green lines each represents a
different signal wired path. 54
4.2 Test setup for inherent delay of DSK board where the solid and the dashed
lines each represents a different signal wired path. 57
4.3 A snapshot of oscilloscope to find approximate DSK input/output delay without
processing at sample frequency 44.1 kHz. 58
4.4 A snapshot of oscilloscope to find approximate DSK input/output delay with
1500 processing cycles inserted between read/write operations at sample fre-
quency 44.1 kHz. 58
4.5 Block diagram of an ACCENT acoustic source node. 64
4.6 ACCENT acoustic source node hardware in a plastic enclosure. 65
4.7 Two-source acoustic distributed beamforming experiment configuration. 65
4.8 Effect of multipath on PLL tracking and holdover. 67
4.9 Amplitude estimation in the \( n^{th} \) test of a two-source round-trip distributed
beamforming experiment. 68
4.10 Two-source power ratio distribution for an acoustic experiment with \( N = 100 \)
tests. 69

5.1 Implementation block diagram of the three-source time-slotted round-trip
carrier synchronization system where the blue, green and red lines each re-
resents a different signal wired path. 72
5.2 Three-source acoustic distributed beamforming experiment configuration. 75
5.3 Three-source power ratio distribution for an acoustic experiment with \( N = 100 \) tests. 77
List of Tables

3.1 Simulation parameter values .......................................................... 26
3.2 Mean squared error of the frequency offset at \( t = 0.5 \) seconds shown in Figure 3.4 and the corresponding convergence behavior. .................. 31
3.3 Nominal PLL simulation parameter values ........................................ 32
3.4 MSE(\( \omega_1 - \omega_2 \))\(_{t=0.5}\) of PLLs with different PD. ......................... 38
3.5 MSE(\( \omega_1 - \omega_2 \))\(_{t=0.5}\) of PLL with multiplier phase detector under the same condition as in Table 3.3. .................................................. 39
3.6 MSE(\( \omega_1 - \omega_2 \))\(_{t=0.5}\) of hybrid PLL with averaging window under the same condition as in Table 3.3. .................................................. 48
3.7 MSE(\( \omega_1 - \omega_2 \))\(_{t=0.5}\) of optimized hybrid PLL with averaging window under the same condition .................................................. 49

4.1 Two-source round-trip synchronization protocol timing. After detection of the primary beacon, each node keeps time using its sample clock running at 44.1 kHz. .......................................................... 55
4.2 Experimental results of two-source wired-channel tests. Each experiment consisted of 100 distributed beamforming tests. ................................. 69

5.1 Three-source experiment approximate node separations in meters ............ 75
5.2 Experimental results of three-source wired test .................................... 77

A.1 Three-source round-trip synchronization protocol timing .......................... 83
Chapter 1

Introduction

Transmit beamforming is an energy-efficient wireless communication technique in which an information source transmits a signal over two or more antennas and aligns the phases of the transmissions across the antennas such that, after propagation, the signals combine constructively at the intended direction [1]. Compared to single-antenna transmission, transmit beamforming can achieve increased range, increased rate or increased power efficiency since more power is directed in the desired direction and less is scattered in undesired directions.

Recently, the idea of transmit beamforming has been extended to a network of cooperating single-antenna sources which behave in a distributed fashion. Unlike conventional transmit beamforming, the carriers of each single-antenna transmitter in a distributed beamformer are synchronized from its independent and imperfect oscillator. Therefore, carrier phase and frequency synchronization among the transmitters is necessary to ensure that a beam is aimed in the desired direction.

As discussed in [1], carrier synchronization approaches can be mainly categorized into two groups: closed-loop synchronization and open-loop synchronization. In closed-loop systems, since the destination coordinates the synchronization process based on a feedback mechanism, interaction among the sources can be minimal. Both the master-slave approach described in [2] and the one-bit feedback synchronization approach discussed in [3, 4, 5] belong to the closed-loop type. Unlike closed-loop, the emphasis of open-loop systems is on using local interactions between the sources to minimize interaction with the distant desti-
nation. Inspired by master-slave frequency synchronization, an open-loop synchronization approach is proposed in [6]. The primary difference between the close-loop and open-loop master-slave synchronization is that the feedback is from the master source node to the slave nodes and does not involve the destination. But no matter what synchronization approach is applied, the beamforming gains are proved to be quite robust to the moderate errors in phase alignment. For instance, as described in [4, 6], 90% of an ideal two-antenna beamforming power gain can be attained even with phase offsets on the order of $30^\circ$.

Based on the equivalence of round-trip propagation delays through a multihop chain of source nodes, a new open-loop carrier synchronization scheme called round-trip carrier synchronization was proposed recently in [7, 8]. Compared with the closed-loop approaches discussed above, the round-trip carrier synchronization was proven to offer several advantages including not requiring explicit estimation, exchange, and quantization of channel state information as well as successful operation in systems with high rates of sources and/or destination mobility.

While almost all of the research on distributed transmit beamforming are theoretical, relatively little has been published on prototypes and/or experimental studies of distributed transmit beamforming except the one-bit feedback carrier synchronization system described in [4, 9] and the round-trip carrier synchronization system in [10].

This thesis extends the work in [10] by optimizing the phase-locked loops. It reports on the development of an acoustic proof-of-concept prototype for time-slotted round-trip carrier synchronization protocol. Prior to performing the acoustic experiments, the real-time implementation of the round-trip protocol was tested over wired channels and the wired-channel experiments consistently resulted in power ratios greater than 98.00% and standard deviations on the order of $10^{-3}$. These results confirmed that the round-trip carrier synchronization protocol can consistently offer near-ideal performance over “perfect” channels.

The development of building an acoustic system is motivated by the observation that the electromagnetic radio-frequency (RF) communications at most common carrier wavelengths can be easily replicated acoustically by scaling all frequencies in the RF systems by the ratio $340.3/(3 \cdot 10^8)$. This implies that results obtained through acoustic proof-of-concept
prototypes can provide guidelines for the design and development of RF systems. The use of acoustic communications is also appealing due to the fact that acoustic transducers are simple and inexpensive and the inherently low data rates allow for real-time operation with low-cost hardware. All the acoustic source nodes used in the experimental study were built by using commercial off-the-shelf parts which include a Texas Instruments floating-point digital signal processor, microphone, AKG speaker, audio amplifier and battery. The functionality of each Acoustic Cooperative Communication Experimental Network Testbed (ACCENT) node including the hybrid phase-locked loops and the logic associated with the time-slotted round-trip carrier synchronization protocol was realized through real-time software independently running on each source node’s C6713 digital signal processor. Experimental results for two-source and three-source acoustic distributed beamforming system in a room with multipath channels suggest that mean power gains of approximately 97.45% and 94.07%, respectively, of the ideal predictions which are better than the results in [10].
Chapter 2

Background

This chapter covers general background relevant to the entire thesis. It begins with a brief discussion of conventional beamforming and distributed beamforming. Then a description of the synchronization technique implemented in this thesis, i.e., time-slotted round-trip carrier synchronization protocol is provided in the following section. Necessary background knowledge and concepts in this thesis such as phase-locked loops and real-time digital signal processing is also discussed in this chapter.

2.1 Conventional beamforming

A conventional beamformer, also known as the delay-and-sum beamformer, is a simple beamformer with all its weights of equal magnitudes [11]. As it is shown in Figure 2.1, the transmit antennas of conventional beamforming are often physically located together. The individual phases of the transmissions from the transmit antennas can be modified so that they align at a desired destination. By using this method, it is possible to produce a directional radiation pattern from a given antenna array instead of an omnidirectional radiation pattern.

Compared to single-antenna transmission, conventional beamforming offers benefits in terms of directionality, interference reduction and security since individual antenna in the array is set up to be phase aligned at a specific destination, which causes constructive interference in the direction of the destination and destructive interference in all other
directions. In general, an ideal $n$-antenna beamformer will have a power gain of $n^2$ if the total input power is held constant. Also, since the energy efficiency is increased, the transmission range can also be extended with constant total input power.

### 2.2 Distributed beamforming

As discussed in the last section, the advantages of conventional transmit beamforming are manifold. The increased energy efficiency, for instance, is particularly appealing in wireless communication systems with energy constrained nodes such as sensor networks. In these types of systems, nodes are typically too small to allow for the use of conventional antenna arrays. Thus distributed beamforming is a powerful technique that offers the potential power gains of conventional antenna arrays to wireless communication systems composed of multiple single-antenna users.

As can be seen in Figure 2.2, in a distributed beamforming system, two or more separate transmitters with common information work together to emulate an antenna array and focus their bandpass transmissions toward an intended direction. A key distinguishing feature of distributed beamforming with respect to conventional beamforming is that the carriers of the single-antenna transmitters are each synthesized from independent and imperfect local oscillators. This feature leads to problems including coordinating the sources for information
sharing and timing synchronization and, most crucially, distributed carrier synchronization.

![Block diagram of a distributed beamformer.](image)

**Figure 2.2: Block diagram of a distributed beamformer.**

### 2.2.1 Overview of Carrier Synchronization Techniques

Based on the interaction between the sources and the destination, we can put carrier synchronization approaches into two basic categories: closed-loop carrier synchronization and open-loop carrier synchronization.

As an example of closed-loop system, the master-slave approach described in [2] uses the intended destination as the master node. After the destination broadcasts a common master beacon to all source nodes, the source nodes bounce the beacons back to the destination and the destination estimates and quantizes the phase delay in each channel. The quantized phase delays are then transmitted to the appropriate source nodes for local phase precompensation prior to beamforming. Recently, a randomized gradient search technique with only one bit of feedback for carrier phase synchronization and distributed beamforming was proposed in [4]. The simplicity and scalability of the one-bit feedback synchronization system make it an attractive candidate for practical implementation where closed-loop carrier
from the destination is possible.

In open-loop carrier synchronization systems, the interaction between the source nodes and the destination can be minimized by increasing the level of inter-source interactions. Inspired by master-slave frequency synchronization, a hierarchical technique for multiuser carrier synchronization and distributed beamforming was recently proposed in [6]. Prior to beamforming, the phase synchronization among source nodes is achieved through a closed-loop method similar to [2] except that a TDD protocol is used between the master source node and the slave source nodes. The key difference in this case is that the feedback is from the master source node to the slave nodes and does not involve the destination. As another example of open-loop synchronization, a time-slotted round-trip carrier synchronization protocol was proposed in [7, 8]. This scheme eliminates the need for digital signaling during synchronization; carrier phase and frequency synchronization is established through the continuously transmitted unmodulated beacons and a pair of phase locked loops at each source node. This thesis focuses on the time-slotted round-trip carrier synchronization technique.

2.2.2 Time-slotted Round-Trip Carrier Synchronization Protocol

Time-slotted round-trip carrier synchronization is based on the equivalence of round-trip propagation delays through a multihop chain of source (single-antenna transmitter) nodes. A two-source round-trip model is shown in Figure 2.3. The basic idea is that an unmodulated carrier transmitted by the destination node and “bounced” around the solid (clockwise) circuit shown in Figure 2.3 will incur the same total phase shift as an unmodulated carrier transmitted by the destination node “bounced” around the dashed (counterclockwise) circuit shown in Figure 2.3. In practice, the “bouncing” of carriers can be performed actively by having each source node track the signals received by other nodes using phase locked loops (PLLs) and then using the PLL in “holdover mode” to transmit a periodic extension of the signal received in a previous timeslot. Coherent combining is achieved since the destination is receiving the sum of two carriers, modulated by the common message, after they have propagated through circuits with identical phase shifts. The rest of this section will describe the time-slotted round-trip carrier synchronization protocol in
Two-source synchronization

In the two-source case, there are four timeslots in the time-slotted round-trip carrier synchronization protocol. Among them, the first three timeslots are used for synchronization and the final timeslot is used for beamforming. The activity in each timeslot is summarized below:

**TS\(^{(0)}\):** The destination transmits the sinusoidal primary beacon to both sources. Both sources generate phase and frequency estimates from their local observations, denoted as \(\hat{\omega}_{0i}\) and \(\hat{\phi}_{0i}\), respectively, at \(S_i\) for \(i \in \{1, 2\}\).

**TS\(^{(1)}\):** \(S_1\) transmits a sinusoidal secondary beacon to \(S_2\). This secondary beacon is transmitted as a periodic extension of the beacon received in \(TS\(^{(0)}\). \(S_2\) generates local phase and frequency estimates from this observation which are denoted by \(\hat{\omega}_{12}\) and \(\hat{\phi}_{12}\), respectively.

**TS\(^{(2)}\):** \(S_2\) transmits a sinusoidal secondary beacon to \(S_1\). This secondary beacon is trans-
mitted as a periodic extension of the beacon received in \( \text{TS}^{(0)} \), with initial phase extrapolated from the phase and frequency estimates obtained by \( S_2 \) in \( \text{TS}^{(0)} \). \( S_1 \) generates local phase and frequency estimates from this observation denoted as \( \hat{\omega}_{21} \) and \( \hat{\phi}_{21} \), respectively.

\( \text{TS}^{(3)} \): Both sources transmit simultaneously to the destination as a distributed beamformer. The carrier frequency of each source is based on both local frequency estimates obtained in the prior timeslots. The initial phase of the carrier at each source is extrapolated from the local phase and frequency estimates from the secondary beacon observation.

Figure 2.4 summarizes the time-slotted round-trip carrier synchronization protocol and shows how the protocol is repeated in order to avoid unacceptable phase drift between the sources during beamforming.

![Figure 2.4: Summary of the two-source time-slotted round-trip carrier synchronization protocol where PB and SB denote primary and secondary beacon synchronization timeslots, respectively.](image)

Assuming temporarily that all of the channels in the system are single-path, it can easily be seen that the aggregate propagation times of the \( D \rightarrow S_1 \rightarrow S_2 \rightarrow D \) and the \( D \rightarrow S_2 \rightarrow S_1 \rightarrow D \) circuits are identical. As each source transmits periodic extensions of beacons it received in prior timeslots, each source is essentially “bouncing” the signal around the respective circuits. Beamforming is achieved since the destination is receiving the sum of two primary beacons after they have propagated through circuits with identical propagation times.

The principles developed in the case of single-path channels can also be applied to the case with multipath channels. However, the transient components caused by the multipath
channels must be accounted for in the protocol. This will be discussed in Section 4.3.

**Three or M-source synchronization**

In a distributed beamforming system with \( M > 2 \) sources, the time-slotted round-trip carrier synchronization protocol has a total of \( 2M \) timeslots denoted as \( S_0, \ldots, S_{2M-1} \). As with \( M = 2 \) sources, the first timeslot is used for the transmission of the primary beacon and the last timeslot is used for beamforming. The \( 2M - 2 \) timeslots in the middle are used for the transmission of synchronization beacons among the source nodes. The basic concepts of two-source synchronization apply here with additional synchronization timeslots and minor modifications in the calculation of the transmission phase for source nodes \( S_2, \ldots, S_{M-1} \).

![Diagram of time-slotted round-trip carrier synchronization system with three source nodes.](image)

In a three-source system shown in Figure 2.5, the activity in each timeslot is summarized below:

**TS\(^{(0)}\):** The destination transmits the sinusoidal primary beacon to all 3 sources. Each source generates local frequency and phase estimates from its observation, denoted as \( \hat{\omega}_{0i} \) and \( \hat{\phi}_{0i} \), respectively, at \( S_i \) for \( i \in \{1, 2, 3\} \).
\( \text{TS}^{(1)} \): \( S_1 \) transmits a sinusoidal secondary beacon to \( S_2 \). This secondary beacon transmitted by \( S_1 \) in \( \text{TS}^{(1)} \) is a periodic extension of the beacon received in \( \text{TS}^{(0)} \). \( S_2 \) generates local phase and frequency estimates from this observation, denoted as \( \hat{\omega}_{12} \) and \( \hat{\phi}_{12} \).

\( \text{TS}^{(2)} \): \( S_2 \) transmits a sinusoidal secondary beacon to \( S_3 \). This secondary beacon is transmitted as a periodic extension of the beacon received in \( \text{TS}^{(1)} \), with initial phase extrapolated from the phase and frequency estimates obtained by \( S_2 \) in \( \text{TS}^{(1)} \). \( S_3 \) generates local phase and frequency estimates from this observation denoted as \( \hat{\omega}_{30} \) and \( \hat{\phi}_{30} \), respectively.

\( \text{TS}^{(3)} \): \( S_3 \) transmits a sinusoidal secondary beacon back to \( S_2 \). This secondary beacon is transmitted as a periodic extension of the beacon received in \( \text{TS}^{(0)} \). The frequency and phase of this secondary beacon are equal to \( \hat{\omega}_{03} \) and \( \hat{\phi}_{03} \). \( S_2 \) generates local phase and frequency estimates from this observation denoted as \( \hat{\omega}_{32} \) and \( \hat{\phi}_{32} \).

\( \text{TS}^{(4)} \): \( S_2 \) transmits a sinusoidal secondary beacon to \( S_1 \). This secondary beacon is transmitted as a periodic extension of the beacon received in \( \text{TS}^{(3)} \). The frequency and phase of this secondary beacon are equal to \( \hat{\omega}_{32} \) and \( \hat{\phi}_{32} \). \( S_1 \) generates local phase and frequency estimates from this observation denoted as \( \hat{\omega}_{10} \) and \( \hat{\phi}_{10} \).

\( \text{TS}^{(5)} \): All of the sources transmit simultaneously to the destination as a distributed beamformer. The carrier frequency of each source is based on both local frequency estimates obtained in the prior timeslots. The local estimates of \( S_1 \) and \( S_3 \) are \( \hat{\omega}_{10}, \hat{\phi}_{10} \) and \( \hat{\omega}_{30}, \hat{\phi}_{30} \) respectively. The local estimates of \( S_2 \) denoted as \( \hat{\omega}_{20} \) and \( \hat{\phi}_{20} \), however, are generated as a combination of all the local estimates in previous timeslots which can be expressed as

\[
\hat{\omega}_{20} = \hat{\omega}_{12} + \hat{\omega}_{32} - \hat{\omega}_{02}, \quad (2.1)
\]

\[
\hat{\phi}_{20} = \hat{\phi}_{12} + \hat{\phi}_{32} - \hat{\phi}_{02}. \quad (2.2)
\]

Figure 2.6 summarizes the time-slotted round-trip carrier synchronization protocol with three sources.
Since, like the two-source case, the total phase shift of the $D \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow D$ and the $D \rightarrow S_3 \rightarrow S_2 \rightarrow S_1 \rightarrow D$ circuits are identical, distributed beamforming between source nodes $S_1$ and $S_3$ can be achieved by following the round-trip protocol and transmitting secondary beacons as periodic extensions of previously received beacons. The only difference is that instead of direct propagation in the two-source case, the secondary beacons propagate between $S_1$ and $S_3$ with a hop when $M = 3$. The node $S_2$ must also derive appropriate transmission phases to participate in the distributed beam former. Similarly, in the case of $M > 3$, the secondary beacons propagate between $S_1$ and $S_M$ with multiple hops. Nodes $S_1, \cdots, S_{M-1}$ must also derive appropriate transmission phases. Details of the $M$-source synchronization is provided in [7].

### 2.3 Phase-locked loop

Almost all the carrier synchronization techniques require some form of carrier phase and frequency estimation. Maximum likelihood estimation (MLE) is one of the carrier phase and frequency estimation techniques, however it may be computationally infeasible for real-time operation. As a classic approach of tracking the phase and frequency of a sinusoidal signals, phase-locked loops (PLL) are an appealing phase and frequency estimation technique in theory and implementation.

The PLL can help keep parts of the distributed beamforming orderly. It causes a particular system to track with another one. More precisely, it synchronizes an output signal with a reference or input signal in frequency as well as in phase. If a phase error builds up, a control mechanism acts on the PLL’s oscillator in such a way that the phase error is
again reduced to a minimum. The operating principle of the PLL will be explained by the following sections.

### 2.3.1 PLL basics

The most basic block diagram of PLL is shown in Figure 2.7. A PLL must at least have the following components:

- **Phase detector (PD):** This is a nonlinear device whose output contains information about the phase difference between the input signal $u_1(t)$ and the output signal $u_2(t)$. $K_d$ is called detector gain and $C_d$ is an adjustable gain of the phase detector which will be discussed in Section 2.3.3.

- **Loop filter (LF):** The output signal $u_d(t)$ of the phase detector consists of signals with different frequency components. In most PLL designs, a low-pass filter is used to filter out the unwanted higher frequencies.

- **Voltage controlled oscillator (VCO):** This is another nonlinear device which produces an oscillation whose frequency is controlled by $u_f(t)$ which is a lower frequency voltage. It generates a periodic oscillation with its frequency $\omega_2$ being controlled by $u_f(t)$. The control voltage $u_f(t)$ is the filtered form of the phase error $\theta_e(t)$. In response to this, the VCO adjusts its frequency $\omega_2$ so that the phase error is driven towards 0 or a constant. The angular frequency is $\omega_2$ given by

$$\omega_2(t) = \omega_o + K_o u_f(t)$$

(2.3)

where $\omega_o$ is the center frequency of the VCO and $K_o$ is the VCO gain in rad·s$^{-1}$·V$^{-1}$.

Figure 2.8 shows the input/output function of the VCO.
• Holdover (optional): The holdover is a buffer that keeps sending the same loop filter output $u_f(t)$ to the VCO. It is not a necessary function unit in PLL but since often times the reference signal only lasts for a short period of time, a holdover is needed so that the PLL can hold the signal with fixed frequency and keeps updating phase when the reference signal vanishes.

2.3.2 Examples of PLL behavior in the unlocked state

Since distributed beamforming typically involves receiving beacons and locking to them from the unlocked state, we are particularly interested in the PLL’s behaviour in the unlocked state. To see how PLL response to phase and frequency change, we will show the PLL’s answer to a phase step and a frequency step applied to its reference input.

Assume the input signal $u_1(t)$ is a sine wave with a phase step applied at $t = 0.4s$, as indicated in Figure 2.9. The sudden phase shift in the reference input will switch the PLL from locked state to unlocked state. The PLL is expected to respond to this phase change and get back to the locked state again. As will be discussed in Section 2.3.3, when the PLL uses a multiplier phase detector (MUL) and the PLL is locked, $\theta_1 \approx \theta_2$, i.e. $\theta_e \approx 0$.

Figure 2.8 shows the transient response of PLL. As can be seen, the PLL is locked before
Figure 2.9: Phase step applied to the reference input of a PLL at $t = 0.4$s.

Figure 2.10: Transient response of PLL to a phase step applied at $t = 0.4$s.
0.4s since $\cos(\theta_e(t)) \approx 1$. Then the phase shift at 0.4s breaks the balance and forces the PLL to leave the locked state. Now the PLL starts to track the “new” reference signal. Finally, the PLL successfully becomes locked again at about 0.6s. Notice that this acquisition process of the PLL is nonlinear. A mechanical analogy is given in [12].

Figure 2.11 shows a reference signal performing a frequency step at time $t = 0.4s$. Similarly, when a sudden frequency step applied to the reference input, the PLL is forced to the unlocked state at the beginning. It then gets back to the locked state again at about 0.7s. This process can be seen in Figure 2.12. Hence, the PLL can track both phase and frequency steps.

![Figure 2.11](image_url)

Figure 2.11: Frequency step applied to the reference input of a PLL at $t = 0.4s$.

### 2.3.3 Third order analog PLL design

In this section, we will study a basic third order PLL assuming an ideal noiseless input signal. The model in Figure 2.7 depends somewhat on the types of phase detectors and loop filters that are used in a particular PLL configuration. For the following analysis, we assume
that the PLL contains a multiplier phase detector and a second order PI (proportional integrated) active loop filter.

The multiplier phase detector is used exclusively in linear PLLs. The input signal $u_1(t)$ is a sinusoidal signal, given by

$$u_1(t) = U_{10} \sin(\omega_1 t + \theta_1).$$  

(2.4)

The second input signal $u_2(t)$, is the feedback given by

$$u_2(t) = U_{20} \cos(\omega_2 t + \theta_2).$$  

(2.5)
The phase detector output signal \( u_d(t) \) can be written as

\[
 u_d(t) = u_1(t)u_2(t) \\
= C_d \{ U_{10} \sin(\omega_1 t + \theta_1)U_{20} \cos(\omega_2 t + \theta_2) \} \\
= C_d \frac{U_{10}U_{20}}{2} \left[ \sin(\omega_1 t + \theta_1) - (\omega_2 t + \theta_2) + \sin(\omega_2 t + \theta_2) + (\omega_1 t + \theta_1) \right] \\
= C_d \frac{U_{10}U_{20}}{2} \left[ \sin((\omega_1 - \omega_2) t + (\theta_1 - \theta_2)) + \sin((\omega_2 + \omega_1) t + (\theta_2 + \theta_1)) \right] \\
= C_d \frac{U_{10}U_{20}}{2} \left[ \sin((\omega_1 - \omega_2) t + \theta_e) + \sin((\omega_2 + \omega_1) t + (\theta_2 + \theta_1)) \right] \\
\tag{2.6}
\]

where \( \theta_e := \theta_1 - \theta_2 \). \( C_d \) is a constant which controls the gain of the phase detector. Since \( \omega_1 \approx \omega_2 \) and \( \theta_1 \approx \theta_2 \), i.e. \( \theta_e \approx 0 \), we have \( \sin(\theta_e) \approx \theta_e \). \( u_d(t) \) can be written as

\[
 u_d(t) \approx C_d \frac{U_{10}U_{20}}{2} \left[ \sin(\theta_e) + \sin(2\omega_1 t + (\theta_1 + \theta_2)) \right] \\
\approx C_d \frac{U_{10}U_{20}}{2} \left[ \theta_e + \sin(2\omega_1 t + (\theta_1 + \theta_2)) \right]. \\
\tag{2.7}
\]

The first term in this equation is the desired “dc” term that contains the information about the phase offset, whereas the latter term is the higher frequency component to be eliminated by the loop filter. Hence Equation (2.7) can be further simplified as

\[
 u_d(t) \approx C_d \frac{U_{10}U_{20}}{2} \theta_e \\
= K_d \theta_e \\
\tag{2.8}
\]

where \( K_d := C_d \frac{U_{10}U_{20}}{2} \) is called phase detector gain with dimension radians per volt (rad/V). It is defined as the slope of the phase detector which is computed as follows: the average value of the output divided by the phase difference of the input. Since usually \( U_{20} = 1 \), the multiplier phase detector gain is \( K_d = \frac{C_d U_{10}}{2} \).

Since the loop filter in the PLL must pass the lower frequencies and suppress the higher, it must be a low pass filter. Another desired feature of the loop filter is that it must have large dc gain so that small phase error between the input signal and the feedback signal can be eliminated.

An active PI filter, in which PI stands for “proportional+integral”, is a type of filter that meets the needs. Its transfer function can be stated as

\[
 F(s) = \frac{1 + s(\tau_2 + \tau_3)}{s\tau_1(1 + s\tau_3)}. \\
\tag{2.9}
\]
Since there is a pole at $s = 0$, it has infinite dc gain. This is a nice feature in PLL design, because it ensures that even slightest $u_d$ is sufficient to pull in the loop. Figure 2.13 shows the bode diagram of a 2nd order active PI filter with $f_{3dB} = 34.8337$ Hz and $54.9^\circ$ phase margin. Based on the mathematical model in Figure 2.7, we can get the open-loop gain of the third-order PLL:

$$G(s) = \frac{K_o K_d}{s} F(s) = \frac{K_o K_d}{s} \frac{1 + s(\tau_2 + \tau_3)}{s\tau_1(1 + s\tau_3)} \quad (2.10)$$

where the $\frac{K_o}{s}$ term corresponds to the transfer function of the VCO. Because we must determine the corner frequencies of the Bode plot, it is more convenient to write $G(s)$ in the standardized form as

$$G(s) = \frac{K_o K_d}{s} \frac{1 + sT_2}{sT_1(1 + sT_3)} \quad (2.11)$$

In order to strike a balance between the settling time and level of ripples, the parameters

![Figure 2.13: Bode diagram of a 2nd order active PI filter.](image)
of the PLL should be carefully designed. Thus according to the above equation, we need to pick appropriate $K_o, K_d, T_1, T_2$ and $T_3$.

In [12], a detailed PLL design procedure is provided so that all the key parameters can be determined.

**step 1.** Normally the designer knows the desired bandwidth of the PLL. Hence it is reasonable to start by choosing a value for $\omega_{3dB}$. In [12], a default value

$$\omega_{3dB} = 0.05\omega_o$$

is recommended, where $\omega_o$ is the center frequency of the PLL.

**step 2.** Since it is the low frequency components that contain most of the desired information rather than the higher harmonics, it is reasonable to shift our focus to the bandwidth of the PLL. The poles and zeros of the system are placed with reference to the transition frequency $\omega_T$. This is the frequency where the magnitude curve crosses 0dB line. In [12], the mathematical relation between $\omega_{3dB}$ and $\omega_T$ is

$$\omega_T \approx \frac{\omega_{3dB}}{1.33}.$$  

(2.14)

**step 3.** To get sufficient phase margin, the corner frequencies $\omega_2$ and $\omega_3$ are recommended to set

$$\omega_2 = \frac{\omega_T}{\sqrt{10}} \quad \text{and} \quad \omega_3 = \omega_T\sqrt{10.}$$

(2.15)

**step 4.** Parameter $T_1$ is chosen such that the open-loop gain is 1 at $\omega = \omega_T$. This leads to

$$T_1 = \frac{K_oK_d}{\omega_2^2\sqrt{10}}.$$  

(2.16)

**step 5.** The time constants $T_2$ and $T_3$ are computed from $T_2 = 1/\omega_2$ and $T_3 = 1/\omega_3$.

**step 6.** Given $T_1, T_2$ and $T_3$ we can calculate the filter time constants $\tau_1, \tau_2$ and $\tau_3$ as

$$\tau_1 = T_1,$$

$$\tau_2 = T_2 - T_3, \text{ and}$$

$$\tau_3 = T_3.$$  

(2.17)
So far we have introduced all the parameters we need in designing a PLL. It is impossible to try each of them to see their influences on the system. Therefore, analysing their interconnections beforehand may help us simplify the design.

By substituting \( T_1, T_2 \) and \( T_3 \) by \( \omega_T \), we have

\[
G(s) = \frac{K_o K_d}{s} \frac{1 + sT_2}{sT_1(1 + sT_3)}
\]

\[
= \frac{K_o K_d}{s} \frac{1 + s\sqrt{\frac{10}{\omega_T}}}{\frac{K_o K_d \sqrt{10}}{\omega_T} (1 + \frac{s}{\omega_T \sqrt{10}})}
\]

\[
= \frac{10\omega_T^2 s + \sqrt{10}\omega_T^3}{\sqrt{10}s^3 + 10\omega_T s^2}
\]

\[
= \frac{10(\frac{\omega_{3dB}}{1.33})^2 s + \sqrt{10}(\frac{\omega_{3dB}}{1.33})^3}{\sqrt{10}s^3 + 10(\frac{\omega_{3dB}}{1.33})^2 s^2}
\]

\[
= \frac{13.3\omega_{3dB}^2 s + \sqrt{10}\omega_{3dB}^3}{1.33^3 \sqrt{10}s^3 + 10 \times 1.33^2 \omega_{3dB}^2 s^2}.
\]  

(2.18)

From the above equation, we can see that the closed loop transfer function \( G(s) \) is only a function of \( \omega_{3dB} \) which indicates that other parameters such as \( K_o, K_d \), etc. have nothing to do with the PLL performance.

2.3.4 Software-based discrete-time PLL design

Based on the analog PLL design procedure, a software-based discrete-time PLL design can be derived by using the bilinear transform. The design procedure can be stated as follows:

**step 1.** As discussed in Section 2.3.3, when the PLL uses a multiplier phase detector and is locked, there is a \( \frac{\pi}{2} \) phase offset between the input to the phase detector and the output of the VCO. A multiplier phase detector can also be used in the PLL.

**step 2.** The design procedure of the loop filter of the PLL is almost the same with the analog PLL except that bilinear transform is applied to translate the loop filter to discrete time. From previous sections, we have the active PI loop filter with a transfer function in Laplace domain

\[
F(s) = \frac{1 + s(\tau_2 + \tau_3)}{s\tau_1(1 + s\tau_3)}.
\]  

(2.19)
The bilinear transform specifies

\[ F(z) = F(s)\bigg|_{s=\frac{2(z-1)}{T(z+1)}} \]  

(2.20)

where \( T = 1/44100 \). Thus the discrete time loop filter can be derived as

\[ F(z) = \frac{(T^2 + 2T(\tau_2 + \tau_3))z^2 + 2T^2z + (T^2 - 2T(\tau_2 + \tau_3))}{2\tau_1(T + 2\tau_3)z^2 - 8\tau_1\tau_3z + 2\tau_1(2\tau_3 - T)}. \]  

(2.21)

The \( \tau_1, \tau_2 \) and \( \tau_3 \) were chosen to achieve a required bandwidth \( (\omega_{3dB}) \) as described in Section 2.3.3.

**step 3.** As discussed in Section 2.3.1, the output frequency and phase of the VCO can be computed as

\[ \omega_2(n) = 2\pi f_o + K_0 u_f(n) \text{ and} \]  

(2.22)

\[ \phi_{vco} = \phi_{vco-previous} + \frac{\omega_2(n)}{f_s}. \]  

(2.23)

Therefore, the VCO can be discretized by updating the phase.

The **MATLAB code of a software PLL implementation is given as below**

```matlab
for i = 1:npts-1,

    %*************** Phase detector ***************
    pd_out(i) = Cd*sig_in(i)*vco_out(i);

    %*************** Loop filter ***************
    if i==1,
        uf(1) = b(1)*pd_out(1);
    elseif i==2,
        uf(2) = b(1)*pd_out(2)+b(2)*pd_out(1)-a(2)*uf(1);
    else
        uf(i) = b(1)*pd_out(i)+b(2)*pd_out(i-1)+...
                b(3)*pd_out(i-2)-a(2)*uf(i-1)-a(3)*uf(i-2);
    end
```
\[
%\text{*************} \ VCO \ %\text{*************}
\]
\[
\text{w_vco}(i) = 2\pi f_{\text{vco}} + k_o u_f(i);
\]
\[
\phi_{\text{vco}}(i+1) = \phi_{\text{vco}}(i) + w_{\text{vco}}(i)/f_s;
\]
\[
\text{while } \phi_{\text{vco}}(i+1)>2\pi, \quad \phi_{\text{vco}}(i+1)=\phi_{\text{vco}}(i+1)-2\pi;
\]
\[
\text{vco\_out}(i+1) = \sin(\phi_{\text{vco}}(i+1));
\]

**step 1.** Line 4 implements the multiplier phase detector where \(i\) is the current sample index.

**step 2.** Line 7-14 implement the transfer function (2.21) under two special situations in which the last two samples' indexes of the loop filter's input and output buffers cannot be directly accessed by subtracting 1 or 2 from the current sampling index. This inherent delay might cause the PLL to converge much slower to certain input which is not “tolerated” as described in [13]. Line 12-13 implement (2.21) under normal situation in which previous two indexes can be directly accessed by subtracting 1 or 2 from the current sampling index.

**step 3.** Line 17-18 perform the operation in (2.22) and (2.23) while Line 19-21 wrap the phase of the VCO back into the range \([0, 2\pi]\) since the sine function is periodic on that interval.

## 2.4 Real-time digital signal processing (DSP)

Digital signal processing (DSP) is concerned with the digital representation of signals and the use of digital systems to analyze, modify, store, or extract information from these signals. In recent years, much research has been done to develop DSP algorithms and systems for real-world applications. DSPs are now used in a wide range of applications, such as in digital communications, audio signal processing, RADAR, SONAR, and so on. In many of these applications, data receiving, analyzing and modifying are required to be
processed in real-time. In this thesis, up to three identical DSPs are used to achieve different tasks and all of the experiments are implemented in real-time.

2.4.1 TMS320C6713DSK

All the systems in this thesis are implemented by Texas Instrument’s TMS320C6713DSK Digital Signal Processing Starter Kit development board. The synchronization functionality is realized in software running in real-time on the floating point DSP operating at 255MHz. Other key features of TMS320C6713DSK include: an TLV320AIC stereo codec, 2M×32 on board SDRAM, 4 user accessible LEDs and DIP switches and four 3.5mm audio jacks (microphone input, line input, line output and headphone output). More information about TMS320C6713DSK can be found in [14].

2.4.2 Code Composer Studio 3.1

In this thesis, the DSK works under the Code Composer Studio (CCS) Integrated Development Environment (IDE) 3.1. The CCS IDE has extended traditional DSP code generation tools by integrating a set of editing, emulating, debugging, and analyzing capabilities in one entity. The DSK communicates with CCS via its onboard universal serial bus (USB) JTAG emulator. It allows a user to connect, program in C and run the DSK through a graphical user interface (GUI). On top of that, checking memory contents, values of variables and profiling execution time can be accomplished with memory window, debugging settings and profiler tools provided by CCS IDE. This is the means by which the system presented in this thesis is implemented in conjunction with the TMS320C6713DSK. More technical information can be found in [15].
Chapter 3

Optimization of PLL unlocked state performance

In 2009, a prototype of the time-slotted round-trip carrier synchronization system described in [10] was built by using nominal parameters from [12]. The experimental results confirmed the theoretical predictions but the performance of the system was not optimized. Based on the PLL model in Figure 2.7, the optimization of a PLL that gives best performance for distributed beamforming is discussed in this chapter. Since distributed beamforming requires the PLLs to acquire signals that will have random frequency and phase offsets, we focus our analysis on the behavior of the PLL in the unlocked state, so that the key parameters in a PLL design can be determined. Notice that the linear PLL model is no longer suitable for analysis of a PLL in the unlocked state. We must resort to simulations to better understand the statistical behavior of PLLs in the unlocked state. Three types of phase detectors, including multiplier phase detector (MUL), XOR and phase frequency detector (PFD) are investigated and evaluated by comparing their frequency mean square error (MSE) with corresponding optimal loop filter bandwidths since the frequency MSE can best reflect the frequency accuracy of the PLL-based estimator. As will be seen in Section 3.3, none of these PLLs are satisfactory for distributed beamforming since the mean squared frequency error is no better than a typical unsynchronized local oscillator and/or the frequency MSE is inconsistent. The reason for the inconsistency revealed by the multi-
plier phase detector is discussed in detail in Section 3.3.2. To overcome these problems, a new PLL design called “hybrid” PLL with averaging window is proposed. While the hybrid PLL acquires with PFD, it tracks with the multiplier. Also, the mean squared frequency error is significantly improved by using the averaging window technique. The final optimized PLL design is summarized in Section 3.5 and an experimental demonstration in Section 3.6 is shown to confirm that the optimized hybrid PLL with averaging window can achieve high frequency estimation accuracy and consistency in the context of distributed beamforming.

### 3.1 PLL behaviour in the unlocked state

In a distributed beamforming system, the received beacons have random frequency and phase offsets. This indicates that the acquisition process of PLL is a dynamic process with respect to the local oscillator. The linear PLL model as we have seen in Section 2.3.3 is no longer valid in the unlocked state. Hence, simulations are necessary in studying PLL behavior in the unlocked state.

In order to better understand the behavior of the PLL with multiplier phase detector in the unlocked state, a simulation is performed using the parameter values given in Table 3.1.

<table>
<thead>
<tr>
<th>Frequency/Bandwidth</th>
<th>Gain/Amplitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_s=44.1$ kHz</td>
<td>$K_o=10$</td>
</tr>
<tr>
<td>$f_o=1021$ Hz</td>
<td>$K_d=1$</td>
</tr>
<tr>
<td>$f_{in}=1020.5639$ Hz</td>
<td>$A=0.95$</td>
</tr>
<tr>
<td>$\omega_{3dB}=0.0061\omega_o$</td>
<td>$C_d=\frac{2+K_d}{A}$</td>
</tr>
</tbody>
</table>

Table 3.1: Simulation parameter values.

The initial phases ($\theta_1$) of the input sinusoidal beacon are given in Figure 3.1. The performance metric considered in this simulation is the frequency error defined as $\omega_1 - \omega_2$, where $\omega_1 = 2\pi f_{in}$ is the angular frequency of the reference signal and $\omega_2$ is the output of VCO in Equation (2.3). Figure 3.1 shows the PLL’s response to different phase steps at $t=0$. As can be seen, the frequency errors become small as $t$ goes to infinity. Also note that the input signals with initial phase $\theta_1$ close to $\pm \pi$ take a little longer for the DT-PLL
Figure 3.1: 11 transient responses of PLL to frequency and phase step applied at \( t=0 \).

to settle. This can be explained by the unstable equilibrium point brought about by the multiplier phase detector. We will discuss this problem in more detail in Section 3.3.2.

To have a closer look at the PLL behavior after convergence and the influence of \( \omega_{3dB} \), two more simulations are performed in Figure 3.2 by using the parameters in Table 3.1 except \( \omega_{3dB1} = 0.005\omega_o \) for the blue curve and \( \omega_{3dB2} = 0.01\omega_o \) for the red curve and initial phase \( \theta_1 = -0.35\pi \) for both tests.

It can be shown in Figure 3.2 that the PLL with comparatively wider loop filter bandwidth (\( \omega_{3dB1} = 0.01\omega_o \)) converges faster than the PLL with narrower loop filter bandwidth (\( \omega_{3dB2} = 0.005\omega_o \)). But fast convergence is not the only factor affecting the frequency error performance.

Assume that the frequency error is sampled at time \( t = 0.5 \) seconds (The sampling time used here can also be extended to arbitrary \( t \) depending on different requirements.) where the PLL will enter holdover so that the frequency accuracy must be good. Inspection of Figure 3.3 indicates that the frequency error of the PLL does not settle exactly to zero even when PLL becomes locked. This can be explained by the double frequency term produced
Figure 3.2: Overall view of frequency errors of PLL with different loop filter bandwidths.

Figure 3.3: Zoomed-in view of frequency errors between $t = 0.45$ to $t = 0.55$ of PLL with different loop filter bandwidths. Note that the PLL with $\omega_{3dB} = 0.01\omega_o$ has converged but the PLL with $\omega_{3dB} = 0.005\omega_o$ is still converging at $t = 0.5$. 
by the multiplier phase detector which cannot be completely eliminated by the loop filter. It oscillates around zero which produces ripples as shown in Figure 3.3. It is also shown that the PLL with wide loop filter bandwidth suffers much larger ripples than its narrow bandwidth counterpart. Thus the level of the output ripples after the PLL becomes locked is another factor needs to be considered.

Based on these observations, it can be concluded that a tradeoff between the level of ripples after convergence and settling time must be evaluated when design a PLL system. Both factors affect the ability to estimate the phase/frequency of the input signal. The goal is to achieve a balance between the settling time and the level of output ripples. Intuitively, a PLL design that can consistently converge before $t = 0.5$ seconds with reasonably small ripples after convergence is desirable for distributed beamforming. So in the example shown in Figure 3.2 and 3.3, the PLL with $\omega_{3dB2} = 0.005\omega_o$ is likely to be better than the PLL with $\omega_{3dB2} = 0.01\omega_o$.

More convincing simulation results are provided by using the parameter in Table 3.1 except that an even narrower ($\omega_{3dB1} = 0.002\omega_o$) and wider ($\omega_{3dB3} = 0.05\omega_o$) loop filter bandwidths are chosen, and 20 evenly distributed points between $[-\pi, \pi]$ are selected as the initial phases. In Figure 3.4, it can be seen that both PLLs with $\omega_{3dB2} = 0.0061\omega_o$ and $\omega_{3dB3} = 0.05\omega_o$ can converge before $t = 0.5$ seconds whereas $\omega_{3dB1} = 0.002\omega_o$ converges slower than its counterparts. Comparing the level of ripples, we can see that the PLL with $\omega_{3dB3} = 0.05\omega_o$ apparently has much large ripples after convergence. More evidence can be found in Figure 3.5.

In Figure 3.5, the corresponding histograms of the frequency offsets at $t = 0.5$ seconds are shown. Since the histograms are displayed on the same x-axis, it is easy to see that the $\omega_{3dB2}$ case has the narrowest distribution while the other two cases are loosely distributed around 0. The corresponding mean squared error of the frequency offsets at $t = 0.5$ seconds in Table 3.2 clearly show that the PLL with $\omega_{3dB2} = 0.0061\omega_o$ outperforms its counterparts. Not only can it track frequency variations of the input signal quickly, but also suppress high frequency components efficiently. It strikes a good tradeoff between settling time and ripple through the loop filter. It is also shown that when utilizing smaller $\omega_{3dB}$, it barely converges to 0 at $t = 0.5$ seconds, and larger $\omega_{3dB}$ leads to unacceptable ripples. Thus, conclusion can
Figure 3.4: $\omega_{3dB}$ effects on settling time and output ripples.

Figure 3.5: Histograms of $\omega_1 - \omega_2$ at $t = 0.5$ seconds shown in Figure 3.4.
be made that the PLL with proper $\omega_{3dB}$ can achieve a balance between the settling time and level of ripples after convergence.

Up to this point, all discussions and simulations are based on deterministic assumptions. But statistical analysis is necessary and even more important for distributed beamforming since the frequency and phase offset that the PLLs acquire are typically random. A good metric for judging the performance of PLLs is the MSE (mean squared error) of the frequency error at $t = 0.5$ seconds which is defined as

$$\text{MSE}(\omega_1 - \omega_2)_{t=0.5} = E \left[ |\omega_1 - \omega_2|^2 \right]_{t=0.5}$$

(3.1)

where the expectation is taken over the random frequencies and phases of the beacons.

Hence, the goal is to minimize the MSE of the frequency error at $t = 0.5$ seconds. This goal is consistent with what matters in distributed beamforming since the beamformer will not work for very long if there is significant frequency offset. While the analysis and simulations in this chapter focus on holdover at $t = 0.5$ seconds, it should be clear to the reader that other $t$ values could be selected without changing the technique.

### 3.2 A statistical study of PLL with MUL, XOR and PFD

In Section 2.3.3, it is stated that only $\omega_{3dB}$ and the phase detector type may have effects on the PLL performance. The first part of this statement is shown in (2.18) since there is no other parameter except for $\omega_{3dB}$ appears in the equation.

As discussed in the previous section, the MSE of the frequency error serves as a good indicator for PLL performance in the distributed beamforming context. Thus a comparison between statistical performance of PLLs with different phase detectors is needed. In the

<table>
<thead>
<tr>
<th>$\omega_{3dB}$</th>
<th>freq MSE</th>
<th>convergence behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0.002\omega_0$</td>
<td>7.9662</td>
<td>slow convergence</td>
</tr>
<tr>
<td>$0.0061\omega_0$</td>
<td>0.0200</td>
<td>medium convergence</td>
</tr>
<tr>
<td>$0.05\omega_0$</td>
<td>104.3850</td>
<td>fast convergence</td>
</tr>
</tbody>
</table>

Table 3.2: Mean squared error of the frequency offset at $t = 0.5$ seconds shown in Figure 3.4 and the corresponding convergence behavior.
following sections, three types of phase detectors including multiplier phase detector, XOR and PFD are discussed. Simulations are performed to find the smallest $\text{MSE}(\omega_1 - \omega_2) |_{t=0.5}^t$ and the corresponding $\omega_{3dB}$. All the statistical simulations are performed by using the same nominal PLL parameter set as shown in Table 3.3, with sinusoidal signals as input signals.

<table>
<thead>
<tr>
<th>Frequency/Bandwidth</th>
<th>Gain/Amplitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f_s=44.1$ kHz</td>
<td>$K_o=10$</td>
</tr>
<tr>
<td>$f_o=1021$ Hz</td>
<td>$K_d=1$</td>
</tr>
<tr>
<td>$f_{in}$: i.i.d randomly generated between 1020 and 1022 Hz,</td>
<td>$A=0.95$</td>
</tr>
<tr>
<td>$\theta_1$: i.i.d randomly generated between 0 and $2\pi$</td>
<td>$C_d=\frac{2}{K_d}$</td>
</tr>
</tbody>
</table>

Table 3.3: Nominal PLL simulation parameter values.

3.2.1 Optimization of $\omega_{3dB}$ for multiplier phase detector

The multiplier phase detector, also referred to as four-quadrant multiplier, is the first phase detector in the history of the PLL. As discussed in Section 2.3.3, the output signal of the multiplier phase detector is obtained by multiplying the signals $u_1(t)$ and $u_2(t)$. These input signals are shown in Figure 3.6 where the dashed curves in (a) and (b) are sine/cosine waves having $\theta_1 = 0$ and $\theta_2 = 0$, and the solid lines have non zero phases $\theta_1$ and $\theta_2$. For simplicity, we assume here that the frequency offset is zero, i.e., $\omega_1 = \omega_2$ and phase is constant over time. The output signal of the multiplier then can be written as

$$u_d(t) = K_d \sin(\theta_e) + \text{double frequency component}.$$  \hspace{1cm} (3.2)

Since the double frequency component is almost entirely suppressed by the loop filter, there remains one AC term in (3.2). Note that the average of this AC signal $u_d(t)$ as shown in Figure 3.7 is not zero since it is an asymmetric “sine wave”; i.e., the durations of the positive and negative half waves are different. Consequently there is a nonzero DC component that pulls the average output frequency of the VCO up or down until lock is acquired. It will be demonstrated later in Section 3.2.3 that PFD enables much faster acquisition, because its output signal is not only phase sensitive, but also frequency sensitive in the unlocked state.
Figure 3.6: Input signals of the multiplier phase detector. (a) Signal $u_1(t)$ is a sine wave. Dashed line: $\theta_1 = 0$; solid line: $\theta_1 > 0$. (b) Signal $u_2(t)$ is a cosine wave. Dashed line: $\theta_2 = 0$; solid line: $\theta_2 > 0$.

Figure 3.7: The multiplier phase detector output signal $\bar{u}_d$ versus phase error $\theta_e$. 
In order to find the optimal $\omega_{3dB}$ for multiplier phase detector, a statistical simulation is performed by using the nominal parameters given in Table 3.3. A total of $10^4$ iterations are conducted each with i.i.d randomly generated $f_{in}$ and $\theta_1$ and $\text{MSE}(\omega_1 - \omega_2)|_{t=0.5}$ for each $\omega_{3dB}$ is computed. Figure 3.11 shows the relationship between $\omega_{3dB}$ and $\text{MSE}(\omega_1 - \omega_2)|_{t=0.5}$. In this figure we can see that there exists a $\omega_{3dB}$ that produces the minimum value of frequency $\text{MSE}=0.0266$. This $\omega_{3dB} = 0.0061\omega_o$ is thus the optimum value that results the optimal PLL design.

3.2.2 Optimization of $\omega_{3dB}$ for XOR phase detector

The operation of XOR phase detector is similar to that of the multiplier. Since the signals in the digital PLLs are always binary, it might be more desirable to work with square wave signals rather than sinusoidal signals.

Figure 3.8 depicts the waveform of the XOR phase detector for phase error $\theta_e$. When $\theta_e = \frac{\pi}{2}$, $u_1$ and $u_2$ are out of phase by exactly $\frac{\pi}{2}$. As can be seen from the figure, the output signal $u_d$ is a square wave whose frequency is twice the input signal. Since the higher frequency component will be filtered out by the loop filter, we consider only the average value of $u_d$, as shown in the dashed line in Figure 3.8. $\bar{u}_d$ is the arithmetic mean of the two logical levels. One advantage of this phase detector is that the loop gain is now

\[ u_{1}(t) \quad \theta_e = 0 \]

\[ u_{2}(t) \]

\[ u_{d}(\bar{u}_d = 0) \]

Figure 3.8: Waveforms of the signals for the XOR phase detector.
independent of input signal amplitude. Furthermore, the XOR phase detector’s response can have a larger linear range than a multiplier. The disadvantage is that the linearity of the baseband response is affected by the relative duty cycles of the input and VCO signals.

To find the optimal $\omega_{3dB}$ for XOR phase detector, the simulation parameters are the same with the multiplier phase detector in Table 3.3. Figure 3.11 shows the relationship between $\omega_{3dB}$ and $\text{MSE}(\omega_1 - \omega_2)|_{t=0.5}$. Like the multiplier, there exists a $\omega_{3dB}$ that produces the minimum frequency $\text{MSE}=0.1363$. This $\omega_{3dB} = 0.0072\omega_0$ is thus the optimum value that results the optimal PLL design.

### 3.2.3 Optimization of $\omega_{3dB}$ for PFD

The PFD is a very popular phase detector. It differs greatly from the multiplier and XOR phase detectors. As its name implies, its output signal depends not only on phase error $\theta_e$ but also on frequency error $\Delta \omega = \omega_1 - \omega_2$, when the PLL has not yet acquired lock. The PFD is built from two D-flipflops, whose outputs are denoted UP and DN, respectively. The PFD can be in one of three states:

- $\text{DN} = 1, \text{UP} = 0 \rightarrow \text{state} = -1$
- $\text{DN} = 0, \text{UP} = 0 \rightarrow \text{state} = 0$
- $\text{DN} = 0, \text{UP} = 1 \rightarrow \text{state} = 1$

The actual state of the PFD is determined by the positive edge of the signals $u_1$ and $u_2$, as explained by the state diagram shown in Figure 3.9. As Figure 3.9 shows, a positive transition of $u_1$ drives the PFD to go into its next higher state, unless it is already in the highest state. In analogy, a positive edge of $u_2$ forces the PFD into its next lower state, unless it is already in the lowest state. The output of PFD $u_d$ is a ternary signal, it is either in 1 state, -1 state or in 0 state.

To see the bonus offered by using PFD, we must assume that the PLL is unlocked initially. If we further assume that the the reference frequency $\omega_1$ is higher than the output frequency $\omega_2$, we can predict that the PFD only toggles between the states 0 and 1 but will never go to -1 state or vice versa. This can be seen in Figure 3.10. We can conclude that the average output signal $u_d$ of the PFD depends on phase error in the locked state of the PLL and on the frequency error in the unlocked state. Therefore, a PLL that uses PFD
Figure 3.9: State diagram for the PFD. This drawing shows the events causing the PFD to change its current state.

\[ \theta_e > 0 \]

Figure 3.10: Waveforms for positive phase error. The PFD output signal is pulsed to the 1 state.
will lock under any condition, irrespective of the type of loop filter used. For this reason, the PFD is often the preferred phase detector in PLLs.

![Figure 3.11: MSE(ω₁ - ω₂)|_{t=0.5} for PLLs with MUL, XOR and PFD.](image)

To find the optimal ω_{3dB} for PFD phase detector, the simulation parameters are the same with multiplier phase detector and XOR as indicated in Table 3.3. Figure 3.11 shows the relationship between ω_{3dB} and MSE(ω₁ - ω₂)|_{t=0.5}. The minimum value of frequency MSE in this case is 0.0131 with corresponding ω_{3dB} = 0.0059ω_o.

### 3.3 Non-satisfactory performance of the optimized PLLs

In the previous section, the PLLs with multiplier, XOR phase detector and PFD are optimized respectively. Table 3.4 shows the corresponding optimal ω_{3dB}s and the minimum MSE(ω₁ - ω₂)|_{t=0.5}.

While it appears that the best choice is a PLL with the PFD and ω_{3dB} = 0.0059ω_o, it turns out that none of these results are satisfactory for the following reasons:

- The standard deviation of the frequency error is no better than that of the typical unsynchronized local oscillator that would be present in standard communications equipment.
<table>
<thead>
<tr>
<th>PD type</th>
<th>MUL</th>
<th>XOR</th>
<th>PFD</th>
</tr>
</thead>
<tbody>
<tr>
<td>optimal $\omega_{3dB}$</td>
<td>0.0061$\omega_o$</td>
<td>0.0072$\omega_o$</td>
<td>0.0059$\omega_o$</td>
</tr>
<tr>
<td>MSE($\omega_1 - \omega_2$)</td>
<td>0.0266</td>
<td>0.1363</td>
<td>0.0131</td>
</tr>
</tbody>
</table>

Table 3.4: MSE($\omega_1 - \omega_2$)$_{t=0.5}$ of PLLs with different PD.

- The MSE($\omega_1 - \omega_2$)$_{t=0.5}$ of the PLLs with multiplier phase detector and XOR is inconsistent.

These two problems limit the PLL performance in distributed beamforming. In the following sections, sources of the inconsistent behavior of the PLLs are discussed. The frequency MSE is analyzed and compared with the unsynchronized local oscillator which paves the way for the new technique.

### 3.3.1 Non-satisfactory frequency MSE

Recall the Table 3.5, the smallest MSE($\omega_1 - \omega_2$)$_{t=0.5}$ so far is obtained from the PLL with PFD which is 0.0131 (rad/s)$^2$. Hence, the frequency MSE of the best software-based DT-PLL design is computed as

$$\text{std of frequency error (ppm)} = \frac{\sqrt{\text{MSE}(\omega_1 - \omega_2)|_{t=0.5}}}{2 \times \pi \times f_o(\text{rad/s})^2}$$

$$= \frac{\sqrt{0.0131(\text{rad/s})^2}}{2 \times \pi \times 1021(\text{rad/s})}$$

$$\approx 1.7841 \times 10^{-5}$$

$$\approx 18 \text{ ppm.}$$

Comparing with the frequency drift of an inexpensive oscillator which is around $\pm 30$ ppm to $\pm 100$ ppm, this design only gives similar performance. Therefore, the frequency accuracy of the optimized PLL-based phase/frequency estimator is not much better than that of the unsynchronized local oscillator. Section 3.4 considers extensions to the PLL design that can improve the frequency accuracy of the PLL.

### 3.3.2 Inconsistent performance of PLL with MUL and XOR

In Table 3.4, it is shown that the optimized PLL with multiplier phase detector can produce MSE($\omega_1 - \omega_2$)$_{t=0.5}$ comparable with the PFD and thus become a candidate for the
optimal PLL design. However, it might be surprising to find that the MSEs for the optimized PLL are not always consistent. Table 3.5 shows 5 simulation results of the optimized PLLs ($\omega_{3dB} = 0.0061\omega_o$) with multiplier phase detector using the same simulation parameters in Table 3.3.

| Experiment No. | $\text{MSE}(\omega_1 - \omega_2)|_{t=0.5}$ of PLL with MUL |
|---------------|-------------------------------------------------|
| 1             | 0.0312                                          |
| 2             | 0.3038                                          |
| 3             | 0.4855                                          |
| 4             | 0.0266                                          |
| 5             | 0.0221                                          |

Table 3.5: $\text{MSE}(\omega_1 - \omega_2)|_{t=0.5}$ of PLL with multiplier phase detector under the same condition as in Table 3.3.

The table shows that the $\text{MSE}(\omega_1 - \omega_2)|_{t=0.5}$ after $10^4$ iterations varies from time to time. The worst $\text{MSE}(\omega_1 - \omega_2)|_{t=0.5}$ obtained from the PLL with multiplier is over 20 times the best case. This indicates that the PLL with multiplier phase detector does not perform consistently.

As we look into the frequency errors at $t = 0.5$ seconds, there are often some large outliers. These outliers are problematic for distributed beamforming. This can be seen in Figure 3.12. In this statistical simulation, the parameters are set according to Table 3.3 except that the $f_{in} = 1021$ Hz and $\omega_{3dB} = 0.0061\omega_o$. Therefore, only the initial phase is random.

As can be seen from Figure 3.12, after $10^4$ iterations, two frequency errors (test 8164 and 8632) are significantly larger than the other frequency errors. These outliers significantly affect the MSE and are the source of the inconsistent results. Furthermore, the inaccurate acquisition would likely lead to destructive combining of the beamformers at the destination. Therefore, the outliers are disastrous in a distributed beamforming.

To understand the importance of consistency, a deterministic simulation is done with the simulation parameters described in Table 3.3 except that $f_{in} = 1021$ Hz, $\omega_{3dB} = 0.0061\omega_o$, and while the initial phases $\theta_1$ of 10 normal cases (blue curves) are generated uniformly distributed between $[-0.35\pi, 0.35\pi]$, the initial phases of the outliers (red and purple curves) are directly extracted from the test 8164 and 8632 of simulation in Figure 3.12. Inspection
Figure 3.12: Outliers of frequency error at \( t = 0.5 \) seconds with optimal \( \omega_{3dB} = 0.0061\omega_o \).

of Figure 3.13 indicates that the frequency error of the outliers at \( t = 0.5 \) seconds does not converge to zero. Since the only different parameter between the normal cases and the outlier is the initial phase, it can be concluded that the PLL designed in this procedure takes a long time to converge from certain initial phases (such as \( \theta_1 \) of test 8164 and test 8632). Hence, even though the PLL is “optimized”, it has inconsistent behaviour that may be problematic in the distributed beamforming application.

The simulation results above confirm that certain “bad” phases can cause the PLL to take a long time to converge. In this case, the initial phase \( \theta_1 \) of tests 8164 and 8632 are 3.1585 and 3.1358 which is very close to \( \pi \approx 3.1415 \). Since we assumed \( \theta_2 = 0 \), the corresponding phase offset is \( \theta_e = \theta_1 - \theta_2 = \pi \) which is an unstable equilibrium. The stable equilibrium of the PLL with multiplier phase detector, as we have discussed in Section 2.3.2, is 0. To confirm the two special cases, another deterministic simulation is done by using parameters in Table 3.3 except that \( f_{in} = 1021 \text{ Hz}, \omega_{3dB} = 0.0061\omega_o \), and the initial phases are set to \( \theta_1 = 0 \) and \( \pm\pi \).

Figure 3.14 shows the transient responses of these three cases. In the case of \( \theta_1 = \pm\pi \),
Figure 3.13: Comparison of frequency errors with outlier and normal cases.

i.e., $\theta_e = \pm \pi$, the PLL converges so slowly that it barely approaches zero at 0.5s while in the case of $\theta_1 = 0$, i.e., $\theta_e = 0$, the PLL converges almost at the beginning of the acquisition process. If we plot the average signal $\bar{u}_d$ versus phase error $\theta_e$, we get a sine function as shown in Figure 3.7. Inspection of this figure suggests that $\theta_e = 0$ is a stable equilibrium and $\theta_e = \pm \pi$ is an unstable equilibrium in $[-\pi, \pi]$. Since if $\theta_e$ happens to be in the vicinity of $+\pi$ and $\bar{u}_d < 0$, the VCO will update negatively towards 0 which may take a long time to reach the equilibrium. On the contrary, if $\theta_e$ is already in the vicinity of 0, it is reasonable to take much less time to reach the equilibrium. Hence, the unstable equilibrium of the multiplier phase detector is one of the sources cause the inconsistencies.

Thus, we come to the conclusion that the optimized PLL with multiplier PD seems to work well most of the time, however, the inconsistencies are problematic. The PLL with XOR has the same problem but the unstable equilibrium is $\theta_e = -\frac{\pi}{2}$ when $\theta_e \in [\pi, \pi]$. A better approach is needed to achieve the original goal of phase and frequency estimation for distributed beamforming.
3.4 Hybrid PLL with averaging window

Due to the non-satisfactory performance of PLLs discussed in the previous section, an improved PLL design is proposed in this section to solve both the inconsistency and insufficient frequency MSE problems. In order to overcome the inconsistency with the PLL, the unstable equilibria inherent in the multiplier phase detector should be avoided by using other types of phase detector. To improve the frequency MSE at holdover, one approach is to introduce an averaging window in the system. In the rest of this section, a “hybrid” PLL using a PFD phase detector for initial convergence and a MUL phase detector with averaging window for steady state performance is proposed and statistical simulation is provided to demonstrate its efficacy.

Recall the PFD discussed in Section 3.2.3. It is easily seen from the waveforms in Figure 3.10 that in the case where $u_1$ leads, the PFD toggles between the states 0 and 1. The $u_d$ becomes largest when the phase error $\theta_e$ is positive and approaches $2\pi$. Similarly, if $u_2$ leads, the PFD toggles between the states 0 and -1. $u_d$ becomes smallest when $\theta_e$ is

Figure 3.14: Three special cases of transient responses of the optimized PLL with multiplier phase detector.
negative and approaches $-2\pi$. If we plot the average signal $\bar{u}_d$ versus $\theta_e$, we get a sawtooth function as shown in Figure 3.15.

Figure 3.15 also shows the average detector output signal for phase errors greater than $2\pi$ or smaller than $-2\pi$. When the phase error $\theta_e$ exceeds $2\pi$, the PFD behaves as if the phase error recycled at zero; hence characteristic curve of the PFD becomes periodic with period $2\pi$. Thus, unlike the multiplier phase detector, there is no unstable equilibria in PFD. The only stable equilibrium is 0. But the occasional transients that cannot be fully suppressed by the loop filter make it extremely hard for the PFD to have smaller $\text{MSE}(\omega_1 - \omega_2)|_{t=0.5}$ even by using an averaging window. This can be shown by the deterministic simulation in Figure 3.16 and 3.17. The simulation is done by using parameters in Table 3.3 except that $f_{in} = 1021$ Hz, $\theta_1 = -0.35\pi$, $\omega_{3dB} = 0.0061\omega_o$ for the multiplier phase detector and $\omega_{3dB} = 0.0059\omega_o$ for the PFD. In Figure 3.16, it shows that the frequency error of both PLLs becomes small 0 before $t = 0.5$ seconds. In Figure 3.17, it is clearly shown that while the multiplier PLL produces ripples oscillating at frequency $2f_{in}$ around 0, the PFD, however, produces occasional spiky transients with aperiodic behavior. Therefore, it is not hard to predict that the PLL with multiplier phase detector can outperform the PLL with PFD if the averaging window technique is applied.
Based on the discussion above, a tradeoff can be made so that the PLL can avoid outliers as well as produce smallest $\text{MSE}(\omega_1 - \omega_2)|_{t=0.5}$. Since the advantages of PFD and multipliers are complementary, it is natural to think that the combination of the two together with the averaging window technique might achieve the goal. The basic idea of the “hybrid” PLL is to use PFD at the beginning of the acquisition process in order to move the PLL away from an unstable equilibrium, then switch to the multiplier phase detector with averaging window to retain the advantage of small MSE after convergence. To maintain the consistency in the system, the phase detector gain $K_d$ should be the same all the time. This is because the loop filter has memory of the system. If $K_d$ changes during the switch, the loop filter changes and so does the memory. In order to have the same slope for both phase detectors, we set $C_d = \frac{2}{4}$ for the multiplier phase detector. Thus, the phase detector gain for the multiplier phase detector $K_d = 1$. This can also be seen in Equation (2.8), where $U_{10} = A$. To make the PFD have the same $K_d$ as the multiplier phase detector, the output of PFD should be multiplied by $2\pi$. Then we have $K_d = 1$ for the PFD. The MATLAB code of the hybrid phase detector implementation is given as below.
Figure 3.17: The zoom-in version of the PLL convergence behavior shown in Figure 3.16 with MUL and PFD between \( t = 0.45 \) to \( t = 0.55 \) seconds.

```matlab
1 %************************* Phase detector *************************
2 if i<iswitch && i>1,
3    % PFD during startup
4    sig_in(i) = (sign(sig_in(i))+1)/2;
5    vco_outi(i) = (sign(vco_outi(i))+1)/2;
6
7    up1 = (i>1)&&(¬sig_in(i-1)&&sig_in(i));
8    up2 = (i>1)&&(¬vco_outi(i-1)&&vco_outi(i));
9
10   status(1) = (i>1)&&(sig_in(i-1)==1)&&(up1&&¬up2);
11   status(2) = (i>1)&&(sig_in(i-1)==-1)&&(¬up1&&up2);
12   status(3) = (i>1)&&(¬status(1)&&¬status(2));
13
14   if status(1)
15      pd_out(i) = pd_out(i-1)+2*pi;
```
elseif status(2)
    pd\_out (i) = pd\_out (i-1)-2*\pi i;
elseif status(3)
    pd\_out (i) = pd\_out (i-1);
end

% regular multiplier phase detector
else
    pd\_out (i) = Cd*sig\_in (i)*vco\_outq (i);
end

Lines 2-20 implement the PFD where \(i\) is the current sample index and \(iswitch\) is the index of the switching time. Lines 2-4 change the sinusoidal waves to square waves. Lines 7-8 check the positive edges of \(u_1\) and \(u_2\) respectively. Lines 10-20 implement the state transitions where \(status(1)\) forces the PFD into its next higher state; \(status(2)\) forces the PFD into its next lower state and \(status(3)\) let the PFD stay in the current state. Notice that the \(pd\_out\) is multiplied by \(2\pi\) in order to make the \(K_d\) the same as the multiplier phase detector. Line 24 implements the multiplier phase detector.

The idea of introducing an averaging window is based on the PLL model with holdover function. Since if \(t = 0.5\) seconds is the given time that the PLL goes into holdover, the \(u_f(t = 0.5)\) will be continuously sent to the VCO before the next beacon is detected. Therefore, instead of only using the last output of the loop filter, it might be better to use the average of more samples of \(u_f(t)\). Figure 3.18 shows the functional block digram of the PLL with averaging window and holdover function.

![Figure 3.18: Block diagram of the PLL with averaging window and holdover function.](image-url)
As can be seen in Figure 3.18, the holdover is followed by an \( M \)-samples averaging window which can attenuate the ripples of \( u_f(t) \) when the PLL is in the locked state. Once the holdover switch is on at \( t = 0.5 \) seconds, the PLL stops tracking and the values stored in the holdover buffer is averaged and sent to the VCO.

To confirm the hybrid PLL with averaging window does help avoid the unstable equilibria and improve the frequency MSE, several MATLAB simulations are conducted. The simulation parameters applied are shown in Table 3.3. Other parameters that need to be considered for this averaging window hybrid PLL are: the loop filter bandwidth \( \omega_{3dB} \), the switching time from PFD to MUL \( T_{shift} \), and the averaging window length \( M \).

For a quick test, \( \omega_{3dB} \) is set to 0.01\( \omega_0 \) which drives PLL to converge faster than the PLLs in Table 3.5. \( T_{shift} \) is set to 0.3 seconds which means that the PLL uses PFD in the first 0.3 seconds and switches to multiplier phase detector for the remainder of the tracking period up to \( t = 0.5 \) seconds. Since the final acquisition process is completed by the multiplier phase detector, \( N \) should be selected to average an integral number of periods so that the double frequency term can be suppressed as much as possible. Since \( f_s = 44.1 \) kHz, \( f_o = 1021 \) Hz, assuming that \( N \) periods of the double frequency output of the multiplier phase detector are to be averaged, then the window length can be computed as

\[
M = \frac{f_s}{2 \times f_o} \times N \\
\approx 21.6 \times N.
\]

When \( N = 10 \), then \( M = 216 \) samples of \( u_f(t) \) are averaged. A total of \( 10^4 \) iterations are performed for 5 statistical simulations. The results are shown in Table 3.6.

The MSE\((\omega_1 - \omega_2)|_{t=0.5}\) in Table 3.6 are very consistent and much smaller compared with the PLL in Table 3.5. As can be seen, the MSE\((\omega_1 - \omega_2)|_{t=0.5}\) is at least 40 times better than the optimal PLL with PFD and the standard deviation of frequency error is 6 times better than that of PFD. Note that the hybrid PLL with averaging window has not been optimized yet. Therefore, there is good reason to believe that the hybrid PLL with averaging window can successfully solve the problems in the PLLs discussed in Section 3.2. It not only avoids the unstable equilibrium inherent in multiplier phase detector but also
Table 3.6: MSE\((\omega_1 - \omega_2)|_{t=0.5}\) of hybrid PLL with averaging window under the same condition as in Table 3.3.

dramatically improves the frequency MSE of the PLL.

### 3.5 Final optimized Hybrid PLL design

In the last section, it has been shown that the hybrid PLL with averaging window is a promising PLL design that can solve the inconsistency problem and improve the frequency MSE of the basic PLL design. The performance of the hybrid PLL is governed by three key parameters:

- \(\omega_{3dB}\): the bandwidth of the loop filter.
- \(T_{shift}\): the time to switch from the PFD to the multiplier phase detector.
- \(M\): the length of the averaging window.

The optimization of switching time \(T_{shift}\) and averaging window length \(M\), similar to the optimization of \(\omega_{3dB}\), is a process of comparing the MSEs of all the possible switching time and \(M\). Hence for the averaging window hybrid PLL, a joint optimization with respect to all three parameters needs to be performed. Simulation parameters in Table 3.3 are used with 20 equally spaced points between \([0.0034, 0.0214] \times 2\pi f_o\) being generated as \(\omega_{3dB}\), 20 uniformly distributed values between \([0.0029, 0.2879]\) being generated as \(T_{shift}\) and 20 evenly distributed \(M\) between \([7425, 10476]\) being generated as the window length. For each triplet of \([\omega_{3dB}, T_{shift}, M]\), \(10^4\) iterations are computed for the MSE\((\omega_1 - \omega_2)|_{t=0.5}\). The minimum MSE=3.0846 \times 10^{-7} with corresponding \(\omega_{3dB} = 0.0138 \times 2\pi f_o\), \(T_{shift} = 0.0629\) and \(M = 9288\) is obtained. This \([\omega_{3dB}, T_{shift}, M]\) is thus the optimum value of the hybrid...
DT-PLL. The corresponding time constants of the optimized averaging window hybrid PLL are $T_1 = 0.0071$, $T_2 = 0.0475$ and $T_3 = 0.0048$.

In order to confirm that this PLL design achieves consistent performance, another simulation similar to Table 3.6 was performed with the optimal $[\omega_{3dB}, T_{shift}, M]$ triplet. $10^4$ iterations are conducted for each test. The simulation results are shown in Table 3.7.

| Experiment No. | $\text{MSE}(\omega_1 - \omega_2)|_{t=0.5}$ of optimized hybrid PLL with averaging window | ppm         |
|---------------|------------------------------------------------|-------------|
| 1             | 3.1347e-07                                      | 8.7276e-02  |
| 2             | 3.1313e-07                                      | 8.7228e-02  |
| 3             | 3.1781e-07                                      | 8.7878e-02  |
| 4             | 3.1467e-07                                      | 8.7442e-02  |
| 5             | 3.2179e-07                                      | 8.8426e-02  |
| Average       | 3.1617e-07                                      | 8.7651e-02  |
| Std           | 1.3264e-017                                      | 0.0504      |

Table 3.7: $\text{MSE}(\omega_1 - \omega_2)|_{t=0.5}$ of optimized hybrid PLL with averaging window under the same condition.

As shown in Table 3.7, the mean squared frequency error is extremely consistent and the $\text{MSE}(\omega_1 - \omega_2)|_{t=0.5}$ is 40,000 times better than the optimized PFD. The frequency standard deviation now is approximately 0.087 ppm which is much more stable than typical temperature compensated or oven-controlled local oscillators.

### 3.6 Experimental demonstration

A wired experiment was conducted to show that the optimized hybrid PLL with averaging window can give better frequency synchronization than the unsynchronized local oscillators on TMS320C6713DSK board.

Figure 3.19 shows the block diagram of the experiment configuration including two Texas Instruments TMS320C6713DSK floating point DSP starter kits, a portable CD player, a mixer, a Marantz solid state recorder (or an oscilloscope for real-time monitoring), various RCA, BNC connectors and cables. The source nodes in the wired experiment are implemented by using the TMS320C6713DSK boards at a sampling frequency 44.1 kHz.

For the unsynchronized wired experiment, both source nodes transmit a 1021 Hz simu-
soidal beacon with unit amplitude. The CD player and the mixer are not used. The beacons are recorded by using different channels of the recorder for approximately 213.2 seconds. For the synchronized experiment, a primary beacon was transmitted by the CD player to both source nodes. Since the primary beacon detection and the optimized hybrid PLL functionalities were implemented by programming the DSKs in C, each source node runs on identical software and tracks the primary beacon for 0.5 seconds. After the acquisition process is completed, both source nodes transmit the signal in holdover and the beacons are recorded by using separate channels for approximately 3.14 hours (11310 seconds).

After the conclusion of the experiment, the uncompressed .wav recording of the experiment was transferred to a PC and analyzed in MATLAB. In order to quantify the frequency offset of the unsynchronized/synchronized signals, the product of the beacons is performed and the high frequency components are removed by using a low-pass filter. According to the trigonometric identity, the product of two sinusoidal signals can be written as

\[
\sin(\omega_1 t + \theta_1) \sin(\omega_2 t + \theta_2) = \frac{1}{2} \{\cos[(\omega_1 - \omega_2)t + \theta_1 - \theta_2] - \cos[(\omega_1 + \omega_2)t + \theta_1 + \theta_2]\}. \quad (3.5)
\]

Since the high frequency components are eliminated by a low-pass filter, only the first low frequency term remains in the equation.
The experimental result of the unsynchronized case is shown in Figure 3.20. As can be seen in this figure, the low frequency component is periodic with the period approximately equals to 62.4 seconds which indicates that the frequency offset is approximately 15.7 ppm.

![Figure 3.20: The frequency offset of two unsynchronized signals by using the local oscillator on the DSK boards.](image)

The result of the synchronized experiment is shown in Figure 3.21. Unlike the unsynchronized case, no obvious periodicity is revealed in the low frequency components in the 11310-second observation time. However, according to (3.5), there is always a low frequency term in the product of two sinusoidal signals. The only reason for not seeing the periodicity in Figure 3.21 is because that the observation time is not long enough. Thus a sinusoid with matching frequency and phase can be generated based on the result of the synchronized experiment. According to Figure 3.21, one half period can be calculated as approximately 9588 seconds which indicates that the period in this case is approximately 19177 seconds. Hence, the matching signal can be generated as

$$0.21 \cos \left( \frac{2\pi}{19177} t + \frac{\pi}{20} \right)$$

(3.6)

in which the amplitude and the initial phase are picked to match the experimental result. As can be seen in Figure 3.21, the period of the matching signal which can also be considered as
the period of the synchronized experiment is 19177 seconds. The corresponding frequency offset is 0.0511 ppm which is better than the MATLAB simulation results.

![Graph showing frequency offset of two synchronized signals](image)

Figure 3.21: The frequency offset of two synchronized signals by using the optimized hybrid PLL with averaging window.

This experiment demonstrates that the hybrid PLL can synchronize the sources to achieve approximately 307 times better performance than the unsynchronized DSKs. This result is consistent with MATLAB simulations.
Chapter 4

Two-Source Time-Slotted Round-Trip Carrier Synchronization System

In this chapter, an experimental study of a distributed beamforming system with two-source, one-destination will be presented by using the optimized hybrid DT-PLL proposed in Section 3.4. We achieve synchronization here via the time-slotted round-trip carrier synchronization protocol. Experimental results of both wired and acoustic tests will be analyzed and discussed in the following sections.

4.1 Experimental methodology for two-source wired test

Each source in the wired two-source time-slotted round-trip carrier synchronization system is implemented by using one TMS320C6713DSK board at a sampling frequency 44.1 kHz. Except for the DSK boards, the system is composed of a CD player (SONY CD walkman D-CJ01), a Behringer EURORACK UB1202 Mixer, a Tektronix TDS 3014 oscilloscope, various RCA, BNC connectors and cables for interconnection between the electronic components’ inputs and outputs. The whole system setup is depicted in Figure 4.1.

The primary beacon detection and round-trip protocol functionalities were implemented
Figure 4.1: Implementation block diagram of the two-source time-slotted round-trip carrier synchronization system where the blue and green lines each represents a different signal wired path.

by programming the TMS320C6713DSKs in C using Texas Instrument’s Code Composer Studio integrated development environment. Each source node runs identical software and determines its identity by polling a bank of DIP switches upon initialization. When DIP switch 1 is pressed, the DSK board performs as source node 1, while DIP switch 2 is pressed the DSK board performs as source node 2. Detailed software design and implementation of the source nodes will be discussed in Section 4.1.2.

The “destination node” was realized by using a portable CD player for primary beacon generation, as well as a Marantz PMD661 solid state recorder for recording of the signals. An oscilloscope was also connected to the output of the Marantz digital recorder for real-time monitoring.

Each wired/acoustic experiment consisted of $N = 100$ “tests” where a test is a complete execution of the $2M - 1$ timeslots of the round-trip protocol. Upon initialization, each node enters into a state where it listens for a primary beacon from the destination node. When the start of the primary beacon is detected, the nodes execute the round-trip protocol according to the schedule in Table 4.1 where each node keeps time by counting samples
received from the codec onboard the TMS320C6713DSK sampling at a rate of 44.1 kHz.

<table>
<thead>
<tr>
<th>Timeslot</th>
<th>Time</th>
<th>S₁</th>
<th>S₂</th>
</tr>
</thead>
<tbody>
<tr>
<td>TS0</td>
<td>0.00s</td>
<td>detect primary beacon</td>
<td>detect primary beacon</td>
</tr>
<tr>
<td></td>
<td>0.00-0.10s</td>
<td>wait</td>
<td>wait</td>
</tr>
<tr>
<td></td>
<td>0.10-0.1629s</td>
<td>track PLL1 with PFD</td>
<td>track PLL1 with PFD</td>
</tr>
<tr>
<td></td>
<td>0.1629-0.60s</td>
<td>track PLL1 with MUL</td>
<td>track PLL1 with MUL</td>
</tr>
<tr>
<td></td>
<td>0.6s</td>
<td>compute holdover for PLL1</td>
<td>compute holdover for PLL1</td>
</tr>
<tr>
<td></td>
<td>0.60-1.25s</td>
<td>holdover PLL1</td>
<td>holdover PLL1</td>
</tr>
<tr>
<td>TS1</td>
<td>1.25-1.35s</td>
<td>transmit holdover PLL1</td>
<td>holdover PLL1</td>
</tr>
<tr>
<td></td>
<td>1.35-1.4129s</td>
<td>transmit holdover PLL1</td>
<td>holdover PLL1 and track PLL2 with PFD</td>
</tr>
<tr>
<td></td>
<td>1.4129-1.85s</td>
<td>transmit holdover PLL1</td>
<td>holdover PLL1 and track PLL2 with MUL</td>
</tr>
<tr>
<td></td>
<td>1.85s</td>
<td>transmit holdover PLL1</td>
<td>compute holdover for PLL2 and holdover PLL1</td>
</tr>
<tr>
<td></td>
<td>1.85-2.25s</td>
<td>transmit holdover PLL1</td>
<td>holdover PLL1 and PLL2</td>
</tr>
<tr>
<td></td>
<td>2.25-2.50s</td>
<td>wait</td>
<td>holdover PLL1 and PLL2</td>
</tr>
<tr>
<td>TS2</td>
<td>2.50-2.60s</td>
<td>wait</td>
<td>transmit holdover PLL1; also holdover PLL2</td>
</tr>
<tr>
<td></td>
<td>2.60-2.6629s</td>
<td>track PLL2 with PFD</td>
<td>transmit holdover PLL1; also holdover PLL2</td>
</tr>
<tr>
<td></td>
<td>2.6629s-3.10s</td>
<td>track PLL2 with MUL</td>
<td>transmit holdover PLL1; also holdover PLL2</td>
</tr>
<tr>
<td></td>
<td>3.10</td>
<td>compute holdover for PLL2</td>
<td>transmit holdover PLL1; also holdover PLL2</td>
</tr>
<tr>
<td></td>
<td>3.10-3.50s</td>
<td>holdover PLL2</td>
<td>transmit holdover PLL1; also holdover PLL2</td>
</tr>
<tr>
<td></td>
<td>3.50-3.80s</td>
<td>holdover PLL2</td>
<td>holdover PLL2</td>
</tr>
<tr>
<td>TS3</td>
<td>3.80-5.80s</td>
<td>transmit holdover PLL2</td>
<td>transmit holdover PLL2</td>
</tr>
<tr>
<td></td>
<td>5.80-6.80s</td>
<td>clear state</td>
<td>clear state</td>
</tr>
<tr>
<td></td>
<td>6.80s</td>
<td>re-arm primary beacon detector</td>
<td>re-arm primary beacon detector</td>
</tr>
</tbody>
</table>

Table 4.1: Two-source round-trip synchronization protocol timing. After detection of the primary beacon, each node keeps time using its sample clock running at 44.1 kHz.

Note that Table 4.1 corresponds to the timing for a two-source experiment; the three-source experiment has similar timing but require more timeslots to exchange the beacons as discussed in Section 5.1. In all of the experiments reported in this paper, the duration of each beacon was one second with a 0.25 second guard time between timeslots. After the final beacon, a guard time of 0.3 seconds occurs before beamforming. The experiments were
automated by creating a compact disk with the one second primary beacon signal repeating every 7 seconds for the two-source experiments and every 10 seconds for the three-source experiments.

4.1.1 Channel characterization

All practical systems are subject to nonidealities which might not have been considered in theoretical analysis. Hence, it is necessary to understand where the nonidealities exist so that we can analyze the system in a proper way. Therefore, before we run the two-source time-slotted round-trip synchronization protocol on this system, we should first know the approximate delay in the wire channels. There are mainly two sources of channel delays: the transmission delay in the line-level stereo cables, and the inherent delay of the ADC/DAC operations on the DSK boards.

Given the signal transmission speed in the wires and the length of wires, the propagation delay through wire channels can be calculated as

$$t_{delay} = \frac{\text{length of cables}}{\text{signal speed}} = \frac{5m}{3 \times 10^8 \text{m/s}} \approx 0.0167 \mu s.$$  

Since the carrier periods of audio frequencies 20 Hz to 20 kHz range from 50 $\mu$s to 50 ms, the propagation delay through wires is negligible with respect to these carrier periods.

To find out the inherent delay of the ADC/DAC operations on DSK boards, a characterization test needs to be performed. As described in [16], the setup is shown in Figure 4.2.

The sampling frequency of the DSK board is set to 44.1 kHz. The Tektronix AFG 3021 function generator was configured to the “burst” function and output a 1021 Hz sine wave with approximately 1V amplitude and zero starting phase for two cycles. We first measured the time delay between the input signal and the output signal of the DSK board by using the “single sequence” function on the Tektronix TDS 2004B digital storage oscilloscope. During this test, the code on DSK board only read and wrote samples without any processing. The test result captured by the oscilloscope can be found in Figure 4.3. Another test was conducted with exactly the same setup except that 1500 cycle loops are inserted between
Figure 4.2: Test setup for inherent delay of DSK board where the solid and the dashed lines each represents a different signal wired path.

reading and writing of samples to the codec in the code running on the DSK board. The
test result captured is shown in Figure 4.4.

Comparing Figure 4.3 and Figure 4.4, we can see that the measured delays are 1.06ms
which are approximately the same. Since the delay 1.06ms ≥ tc ≈ 0.97ms, where tc is the
carrier period at 1021 Hz, we can conclude that the inherent delays are determined mainly
by the ADC/DAC operations on DSK boards, rather than the processing between samples.
Although the ADC/DAC delays is not negligible with respect to most audio frequency
carrier periods, this delay however is identical in each source node in the system. Thus,
the total channel delays of the clockwise and counterclockwise circuits D → S₁ → S₂ → D
and D → S₂ → S₁ → D are identical. Based on these analysis, it is fair to say that
although the time-slotted round-trip carrier synchronization system automatically corrects
phase offsets caused by both the channel and local oscillator, the experimental results of the
system shown in Figure 4.1 are focused on correcting oscillator offsets rather than correcting
channel offsets.

The channel noise is another factor that needs to be considered. Since all signals are
transmitted by wires, the channel noise are primarily caused by the Analog-to-Digital Con-
verter (ADC) and the Digital-to-Analog Converter (ADC) in the TLV320AIC23B features
Figure 4.3: A snapshot of oscilloscope to find approximate DSK input/output delay without processing at sample frequency 44.1 kHz.

Figure 4.4: A snapshot of oscilloscope to find approximate DSK input/output delay with 1500 processing cycles inserted between read/write operations at sample frequency 44.1 kHz.
third-order multi-bit architecture with up to 90 dB signal-to-noise (SNR) at sampling rates up to 96 kHz, while the DAC sigma-delta modulator features a second-order multi-bit architecture with up to 100 dB SNR at sampling rates up to 96 kHz [17]. Therefore, the channels in this system can be regarded as essentially noise-free due to signals’ high SNR.

4.1.2 Source node functionality and PLLs

As discussed in [7], since each source transmission in the time-slotted round-trip carrier synchronization protocol is intended to be a periodic extension of a beacon received in a previous timeslot, each node in the system could realize its phase and frequency estimation function by using the hybrid PLLs with averaging window as described in 3.4.

The software implementation of each source node is achieved by converting the hybrid PLL design to C source code running on the DSKs. In order to prevent false detection and noise, an IIR filter with peak frequency 1021 Hz and bandwidth of 100 Hz is added before the signal enters PLLs. Depending on the number of nodes and the node number, as many as two independent phase locked loops are implemented on a source node. The PLL loop filter is realized by following the analog active-PI loop filter design procedure in [12] with 3 dB bandwidth of approximately 14 Hz and then using the bilinear transform to convert the analog loop filter to discrete time. The PLL’s “voltage controlled oscillator” is implemented in software as a numerically controlled oscillator (NCO) centered at the nominal frequency of 1021 Hz. All processing was performed on the DSP in floating point.

The following sections will give detailed discussion of the function units.

Phase detector implementation

The phase detector in each hybrid PLL is implemented in two stages. In the “rough acquisition” stage (the first 0.0629 seconds of tracking), the PLL uses a phase-frequency detector (PFD). The reasons for using PFD have been discussed in Section 3.3.2 and 3.4 which can be briefly summarized as: First, unlike most other phase detectors, e.g. the multiplier, the PFD does not possess any unstable equilibria and convergence times are predictable. Second, the PFD output is independent of the input amplitude. Hence, the PLL can perform rough acquisition without automatic gain control. The PFD output
after convergence, however, has occasional transients that are not fully suppressed by the
loop filter which can lead to inconsistent beamforming performance. Hence, after rough
acquisition, the PLL switches to “fine acquisition” for the remainder of the tracking period
by changing the phase detector to a standard multiplier. The multiplier phase detector does
not have output transients like the PFD after convergence, but is not suitable for rough
acquisition due to its sensitivity to input amplitude and unpredictable convergence times
caused by the presence of unstable equilibria. During fine acquisition, the input signal is
normalized to unity amplitude by using a local estimate of the signal amplitude obtained
during rough acquisition so that the PFD and the standard multiplier can share the same
phase detector gain. Inconsistent phase detector gain may result slow convergence for the
multiplier phase detector.

The C code for the hybrid PLL implementation is listed as:

```c
1 /**************Phase detector***************/
2 if (PhaseDetector == 0)
3 /**************PFD***************/
4 {
5   if (input > maxScale[PLLnum])
6     maxScale[PLLnum] = input;
7
8   if (input >= 0)
9     sig_in[PLLnum][0] = 1;
10  else if (input < 0)
11     sig_in[PLLnum][0] = 0;
12
13  if (vco_out_sin[PLLnum][0] >=0)
14     vco_out_sin[PLLnum][0] = 1;
15  else if (vco_out_sin[PLLnum][0] <0)
16     vco_out_sin[PLLnum][0] = 0;
17```
\[
\begin{align*}
up1 &= ((\text{sig}_i[n][PLLnum][1]) \& \& \text{sig}_i[n][PLLnum][0]); \\
up2 &= ((\text{vco}_o[n][PLLnum][1]) \& \& \text{vco}_o[n][PLLnum][0]); \\
\text{status}[PLLnum][0] &= (\text{sig}_i[n][PLLnum][1] != 1) \& \& (up1 \& \& (!up2)); \\
\text{status}[PLLnum][1] &= (\text{sig}_i[n][PLLnum][1] != -1) \& \& ((!up1) \& \& up2); \\
\text{status}[PLLnum][2] &= (!\text{status}[PLLnum][0]) \& \& (!\text{status}[PLLnum][1]); \\
\text{if} (\text{status}[PLLnum][0]) \\
\quad \text{pd}_o[PLLnum][0] &= \text{pd}_o[PLLnum][1] + \text{twopi}; \\
\text{else if} (\text{status}[PLLnum][1]) \\
\quad \text{pd}_o[PLLnum][0] &= \text{pd}_o[PLLnum][1] - \text{twopi}; \\
\text{else if} (\text{status}[PLLnum][2]) \\
\quad \text{pd}_o[PLLnum][0] &= \text{pd}_o[PLLnum][1]; \\
\text{else if} (\text{PhaseDetector} == 1) \\
\quad /***************MUL***************/ \\
\{ \\
\quad \text{Cd}[PLLnum] &= 2*\text{Kd}/\maxScale[PLLnum]; \\
\quad \text{pd}_o[PLLnum][0] &= \text{Cd}[PLLnum]*\text{input}*\text{vco}_o[n][\cos[PLLnum][0]]; \\
\quad /***************END Phase detector***************/
\end{align*}
\]

This code is just the C version of the MATLAB code listed in Section 3.4 except that the maximum amplitude of the sine signal is based on the measured data (lines 5-6) rather than setting \( A = 0.95 \) directly. Hence, in order to have the same slope for both phase detectors, we have \( C_d \) be calculated as line 36. Lines 2-31 implement PFD while lines 33-38 implement multiplier phase detector.
Loop filter implementation

As discussed in Section 2.3.4, the transfer function of the second-order low-pass active proportional integral (PI) loop filter has the form as Equation (2.19). Since $T_1 = 0.0071$, $T_2 = 0.0475$, $T_3 = 0.0048$, the corresponding time constants can be calculated by Equation (2.17) and thus $\tau_1 = 0.0071$, $\tau_2 = 0.0428$ and $\tau_3 = 0.0048$. Using bilinear transform, we can arrive at

$$F(z) = \frac{(T^2 + 2T(\tau_2 + \tau_3))z^2 + 2T^2z + (T^2 - 2T(\tau_2 + \tau_3))}{2\tau_1(T + 2\tau_3)z^2 - 8\tau_1\tau_3z + 2\tau_1(2\tau_3 - T)} = \frac{(T^2 + 2T(0.0428 + 0.0048))z^2 + 2T^2z + (T^2 - 2T(0.0428 + 0.0048))}{2 \times 0.0071(T + 2 \times 0.0048)z^2 - 8 \times 0.0071 \times 0.0048z + 2 \times 0.0071(2 \times 0.0048 - T)} \tag{4.2}$$

where $T = 1/44100$. The corresponding C code can be found in Appendix B and C.

VCO implementation

Based on the working principle of voltage-controlled oscillators, the VCO oscillates at the center frequency 1021 kHz, which is determined by the output signal $u_f$ of the loop filter. In C, the VCO is implemented as a numerically controlled oscillator (NCO) and it is made into a function by the following code:

```c
void VCO(unsigned short PLLnum, double holdover_value,
unsigned int mode)
{
    w_vco[PLLnum] = twopi*f_vco+Ko*holdover_value;
    phi_vco[PLLnum] = phi_vco[PLLnum]+ w_vco[PLLnum]*inv_fs;
    while (phi_vco[PLLnum] > twopi)
    while (phi_vco[PLLnum]< 0)
        phi_vco[PLLnum]= phi_vco[PLLnum]+ twopi;
}
```
where line 4 computes frequency in (2.3) and line 5 computes the corresponding phase. Lines 7-10 wrap the phase of the VCO back into the range \([0, 2\pi]\) since the sine function is periodic on that interval. All variables are stored as single precision floating point values.

**Holdover implementation**

The holdover function as described in Section 3.4 is an average-value low-pass filter which can attenuate the ripples in the outputs of the filter. When the holdover switch is on, the latest \(M\) \(u_f\)'s are averaged out and input to VCO. According to the MATLAB simulation in Section 3.5, the optimal length of the averaging window is \(M=430\). When the holdover switch is off, the \(u_f\) is directly sent to VCO. The holdover switches on at 0.5s of each 1-second-long input beacon, and switches off when the desired secondary beacon or beamforming beacon has been transmitted. The following C code shows the software implementation of holdover:

```c
1   holdover[PLLnum][index] = uf[PLLnum][0];
2   if (index < (m−1))
3     {
4       running_sum[PLLnum]= running_sum[PLLnum] + holdover[PLLnum][index]
5         − holdover[PLLnum][index +1];
6       holdover_index[PLLnum]++;
7     }
8   else
9     {
10    running_sum[PLLnum] = running_sum[PLLnum] + holdover[PLLnum][index]
11       − holdover[PLLnum][0];
12    holdover_index[PLLnum] = 0;
13  }
```

When the PLL stops tracking and goes into the holdover, the following code computes the averaged value of the “running_sum”:

```c
1   running_avg[0] = running_sum[0]*invM;
```
and this value will be used as $\bar{u}_f$ to update the $\omega_2$.

4.2 Experimental methodology for two-source acoustic test

The two-source acoustic test shares the same round-trip synchronization protocol with the wired-channel test discussed in Section 4.1, as well as the software implementation. The difference mainly lies in the experimental setup. Instead of just using a TMS320C6713DSK, the acoustic source nodes used in the experimental study were developed as part of an Acoustic Cooperative Communication Experimental Network Testbed (ACCENT). All of the hardware components of the ACCENT node are low-cost off-the-shelf parts. Figure 4.5 shows a block diagram of the major components of the source node including a Texas Instruments TMS320C6713DSK floating point DSP starter kit, microphone, power amplifier, speaker, and battery. As shown in Figure 4.6, the components are mounted in an plastic enclosure with the microphone and speaker placed in close proximity to approximate a single transducer. Note that each source node operates independently using its own local oscillator; there are no wires or signals shared among the source nodes other than the acoustic signals generated during the round-trip protocol.

![Figure 4.5: Block diagram of an ACCENT acoustic source node.](image)

Photographs of the experiment environment configured for an acoustic experiment with two source nodes is shown in Figures 4.7. The room in which the acoustic experiments were performed was a typical carpeted conference room with dimensions approximately 7.5 meters by 7.5 meters. In the two-source experiment, the nodes were placed in an approximately equilateral triangle configuration with approximately 4 meters of separation.
Figure 4.6: ACCENT acoustic source node hardware in a plastic enclosure.

Figure 4.7: Two-source acoustic distributed beamforming experiment configuration.
between the source nodes and between each source node and the destination.

4.3 Data analysis methodology

The assumption of single-path channels in the prior development of the two-source round-trip carrier synchronization protocol was used for clarity of exposition but is not necessary to enable beamforming. Note that the beacons exchanged in the round-trip carrier synchronization system are all at the same frequency as the carrier. Hence, each bidirectional channel between a pair of source nodes (and between individual source nodes and the destination) is a time-division-duplex (TDD) channel that is reciprocal in both directions. The principles developed in the case of single-path channels can then be applied to the case with multipath channels with the difference being that it is now the phase shift, rather than the propagation delay, of each channel that is identical in both directions. Denoting the phase of the channel between node $i$ and $j$ as $\theta_{i,j}$, it is easy to see that the aggregate round trip phase shifts of the $D \to S_1 \to S_2 \to D$ circuit and the $D \to S_2 \to S_1 \to D$ circuit are identical and equal to

$$\theta_{rt} = \theta_{0,1} + \theta_{1,2} + \theta_{2,0}.$$ 

Although the steady-state phase shift of each channel is identical in both directions, the multipath channels also cause the finite-duration beacons received by $S_1$ and $S_2$ to have transient components that must accounted for in the protocol. In a system with multipath channels, each source node should delay tracking the beacon with the appropriate PLL until the transient effects of the channel have become negligible. The source then tracks the beacon with the appropriate PLL during the steady-state portion of the beacon observation and puts the PLL into holdover mode prior to the conclusion of the steady-state portion of the beacon. This is summarized in Figure 4.8.

The important thing here is that each source uses only the steady-state portion of its noisy observation in each timeslot for PLL tracking and subsequent computation of local estimates of the received frequency and phase. The initial and final transient portions of the observation are ignored. As with single-path channels, the phase estimates at each source are extrapolated for transmission of the secondary beacons and carriers as periodic
extensions of the steady state portion of the primary beacon observations.

In order to ensure that some portion of the observation is steady-state observation, the duration of each secondary beacon must exceed the delay spread of the channel in which the beacon is transmitted. Guard times may also be added between timeslots to allow for the transients in a previous timeslot to vanish before a new beacon is transmitted. No other modifications to the synchronization protocol are necessary.

At the conclusion of an experiment consisting of $N$ wired/acoustic beamforming tests, the uncompressed .wav recording of the experiment was transferred to a PC and analyzed in MATLAB to generate the statistical results presented in Section 4.4. To quantify the efficacy of the distributed beamformer, the “power ratio” $\rho$ of the beamformer is calculated by estimating the power received during beamforming and computing its ratio with respect to the ideal prediction if the carriers were received in perfect phase alignment. A power ratio of one corresponds to an ideal beamformer with perfect phase alignment. A power ratio of zero corresponds to the case where the carriers completely cancel at the destination.

To understand how power ratio is computed from the recordings, Figure 4.9 shows a figurative example of a typical recording for a two-source round-trip beamforming test. Since the secondary beacons in $\text{TS}^{(1)}$ and $\text{TS}^{(2)}$ are transmitted at the same amplitude as the carriers in $\text{TS}^{(3)}$, the power ratio of the $n^{\text{th}}$ test can be computed by estimating the
amplitudes of the signals recorded in timeslots $\text{TS}^{(1)}$, $\text{TS}^{(2)}$, and $\text{TS}^{(3)}$ and calculating

$$\rho[n] = \left( \frac{\hat{a}_{bf}[n]}{\hat{a}_{10}[n] + \hat{a}_{20}[n]} \right)^2.$$  

The amplitude estimates in each test are obtained via the MLE FFT technique described in [18] using an 0.2 second window of the steady state portion of each signal in timeslots $\text{TS}^{(1)}$, $\text{TS}^{(2)}$, and $\text{TS}^{(3)}$. In the general $M$-source case, the power ratio of the $n^{th}$ test can be calculated as

$$\rho[n] = \left( \frac{\hat{a}_{bf}[n]}{\hat{a}_{10}[n] + \cdots + \hat{a}_{M0}[n]} \right)^2.$$  

As with two nodes, a power ratio of one corresponds to an ideal beamformer with perfect phase alignment.

### 4.4 Experimental results

Several wired-channel experiments were performed and the experimental results can be found in Table 4.2. The mean power ratio over $N = 100$ tests of the time-slotted round-trip carrier synchronization protocol was consistently greater than 0.99 and standard deviations on the order of $10^{-3}$. Hence, the wired-channel experiments confirmed that the round-trip
carrier synchronization protocol can consistently offer near-ideal performance over “perfect” channels.

<table>
<thead>
<tr>
<th></th>
<th>Experiment1</th>
<th>Experiment2</th>
<th>Experiment3</th>
<th>Experiment4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min(ρ)</td>
<td>0.9970</td>
<td>0.9960</td>
<td>0.9970</td>
<td>0.9970</td>
</tr>
<tr>
<td>Max(ρ)</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
<td>1.0000</td>
</tr>
<tr>
<td>Mean(ρ)</td>
<td>0.9980</td>
<td>0.9983</td>
<td>0.9986</td>
<td>0.9984</td>
</tr>
<tr>
<td>Std(ρ)</td>
<td>8.8335e-004</td>
<td>1.1000e-003</td>
<td>9.5796e-004</td>
<td>9.8711e-004</td>
</tr>
</tbody>
</table>

Table 4.2: Experimental results of two-source wired-channel tests. Each experiment consisted of 100 distributed beamforming tests.

Figure 4.10 shows a histogram of the two-source power ratios over $N = 100$ tests of the time-slotted round-trip carrier synchronization protocol with one second beacons and 0.25 second guard times. The mean power ratio of the distributed beamformer was computed to be 0.9745 and the standard deviation was computed to be approximately 0.010.

![Figure 4.10: Two-source power ratio distribution for an acoustic experiment with $N = 100$ tests.](image)

This result shows that the time-slotted round-trip carrier synchronization protocol was consistently effective at synchronizing the phase of the carriers of the ACCENT nodes and
that the synchronization errors lead to only a small loss in performance with respect to the ideal prediction. The factors that contribute to the non-ideal performance observed in the acoustic experiments will be discussed in Chapter 6.
Chapter 5

Three-Source Time-Slotted Round-Trip Carrier Synchronization System

Based on the discussion and experimental results in Chapter 4, this chapter extends the two-source round-trip carrier synchronization system to three-source scenario. Similar to the two-source case, both wired-channel and acoustic experiments were conducted. The experimental results were analyzed via the MLE FFT technique. The three-source beamforming system is also a proof-of-concept for the beamforming system with sources $M > 3$, i.e., 4, 5, ..., $M$ since the functionalities of the middle nodes in the “synchronization chain” are very similar to the functionality of the source node 2 in the three-source beamforming system. Experimental results are analyzed and discussed in the following sections.

5.1 Experimental methodology for three-source wired test

Prior to performing the three-source acoustic experiments, the real-time implementation of the three-source round-trip protocol was tested over wired channels by connecting the line-level outputs of the DSKs to the Behringer EURORACK UB1202 mixer and connecting the line-level inputs of the TMS320C6713DSKs to the output of the mixer. The CD player
used for primary beacon generation was also connected to the audio mixer. The whole system setup is depicted in Figure 5.1.

![Implementation block diagram of the three-source time-slotted round-trip carrier synchronization system where the blue, green and red lines each represents a different signal wired path.](image)

Figure 5.1: Implementation block diagram of the three-source time-slotted round-trip carrier synchronization system where the blue, green and red lines each represents a different signal wired path.

Like the two-source case, the primary beacon detection and round-trip protocol functionalities were implemented by programming the TMS320C6713DSKs in C using 6713 DSK CCStudio v3.1. Each source node runs identical software and determines its identity by polling a bank of DIP switches upon initialization. DIP switch 1-3 corresponds to the source node 1-3 respectively. When DIP M is pressed, the DSK board performs as source node M. Again, a portable CD player was used for primary beacon generation, and a Marantz recorder for recording of the signals. The experiments were automated by creating a compact disk with the one second primary beacon signal repeating every 10 seconds. An oscilloscope was also connected to the main output of the mixer for real-time observation.

While the source node functionality and PLLs are almost the same as the two-source case, the middle node denoted as source node 2, is not just the local estimate in $\mathbf{TS}^{(3)}$. According to the protocol [7], the local estimates of the source node 2, namely $\omega_{20}$ and $\phi_{20}$ are generated as a combination of all the local estimates in $\mathbf{TS}^{(0)}$, $\mathbf{TS}^{(1)}$ and $\mathbf{TS}^{(3)}$, which
could be expressed as

\[
\begin{align*}
\omega_{20} &= \omega_{12} + \omega_{32} - \omega_{02} \\
\phi_{20} &= \phi_{12} + \phi_{32} - \phi_{02}.
\end{align*}
\]  

(5.1)

By using the \(\omega_{20}\) and \(\phi_{20}\) above, the total phase shift in the carrier signal of source node 2 is approximately the same as in the corner nodes.

Experiments consisted of \(N = 100\) “tests” were conducted following Table A.1 in Appendix A, where each node keeps time by counting samples received from the codec onboard the TMS320C6713DSK sampling at a rate of 44.1 kHz.

5.1.1 Channel characterization

It seems that the experimental setup of the wired three-source and two-source time-slotted round-trip carrier synchronization system are almost the same except that a mixer is used in the three-source system so that the beamforming beacons can be combined and collected. Hence, the propagation delay introduced by the mixer might also need to be taken into consideration. In [10], the propagation delay of the mixer is proved to be negligible by using the same setup in Section 4.1.1 except that the DSK board is replaced by the mixer. The delay between the input and output of the mixer is almost 0 with respect to the test beacon period in [10].

Since we have shown in Section 4.1.1 that the only non-negligible delay in this system is the propagation delay of the ADC/DAC on DSK6713, the total channel delays of the two circuits \(D \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow D\) and \(D \rightarrow S_3 \rightarrow S_2 \rightarrow S_1 \rightarrow D\) are identical. Therefore, it is fair to say that although the time-slotted round-trip carrier synchronization system can automatically correct phase offset introduced by both the channel and local oscillator, the experimental results of the system are focused on oscillator offsets.

5.1.2 Source node functionality

Similar to the two-source system, the primary beacon detection and round-trip protocol functionalities were implemented by programming the TMS320C6713DSKs in C using Texas Instrument’s Code Composer Studio integrated development environment. Each source
node runs identical software and determines its identity by polling a bank of DIP switches upon initialization.

The discrete-time phase locked loops in each source node are implemented in software. Depending on the number of nodes and the node number, as many as three independent phase locked loops are implemented on a source node. The PLL loop filter is realized by following the analog active-PI loop filter design procedure in [12] with 3 dB bandwidth of approximately 14 Hz and then using the bilinear transform to convert the analog loop filter to discrete time. The PLL’s “voltage controlled oscillator” is implemented in software as a numerically controlled oscillator (NCO) centered at the nominal frequency of 1021 Hz. All processing was performed on the DSP in floating point.

The only difference between the two-source system and three-source system is that, instead of only having the corner nodes, the three-source system also has a middle node which bounces the secondary beacons of the corner nodes to each other at appropriate timeslots. Hence there are three PLLs running on the middle node, each tracking the beacons in $\text{TS}^{(0)}$, $\text{TS}^{(1)}$ and $\text{TS}^{(3)}$. During the beamforming timeslot, namely $\text{TS}^{(5)}$, the local estimates of the source node 2 will be computed by the following C code:

```c
1   w_vco_m = w_vco[1] + w_vco[2] - w_vco[0];
3       while (phi_vco_m > twopi)
4           phi_vco_m = phi_vco_m - twopi;
5       while (phi_vco_m < 0)
6           phi_vco_m = phi_vco_m + twopi;
```

Line 1 and line 2 perform the operation in Equation (5.1) while Line 3-6 wrap the phase of the VCO back into the range $[0, 2\pi]$. Other function units, such as the hybrid phase detector, loop filter, VCO and holdover implementation are the same as the two-source system.
5.2 Experimental methodology for three-source acoustic test

The three-source acoustic test shares the same round-trip synchronization protocol with the wired test, as well as the software implementation. The experiment was conducted in the carpeted conference room described in Section 4.2. Photographs of the test environment configured for the acoustic experiment with three source nodes is shown in Figure 5.2. The approximate node separations are given in Table 5.1.

![Figure 5.2: Three-source acoustic distributed beamforming experiment configuration.](image)

<table>
<thead>
<tr>
<th></th>
<th>dest</th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
</tr>
</thead>
<tbody>
<tr>
<td>dest</td>
<td>0</td>
<td>2.4</td>
<td>2.1</td>
<td>2.4</td>
</tr>
<tr>
<td>S1</td>
<td>2.4</td>
<td>0</td>
<td>2.1</td>
<td>3.8</td>
</tr>
<tr>
<td>S2</td>
<td>2.1</td>
<td>2.1</td>
<td>0</td>
<td>1.9</td>
</tr>
<tr>
<td>S3</td>
<td>2.4</td>
<td>3.8</td>
<td>1.9</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.1: Three-source experiment approximate node separations in meters.

Three ACCENT nodes were used as the source nodes. The “destination node” was realized by using a portable CD player and a self-amplified loudspeaker for primary beacon generation, as well as a microphone and a Marantz digital recorder for recording of the signals. An oscilloscope was also connected to the output of the Marantz digital recorder for real-time monitoring.
Each acoustic experiment consisted of \( N = 100 \) “tests” where a test is a complete execution of the 5 timeslots of the round-trip protocol. Upon initialization, each node enters into a state where it listens for a primary beacon from the destination node. When the start of the primary beacon is detected, the nodes execute the round-trip protocol according to the schedule in Table A.1 where each node keeps time by counting samples received from the codec onboard the TMS320C6713DSK sampling at a rate of 44.1 kHz. The duration of each beacon was one second with a 0.25 second guard time between timeslots. After the final beacon, a guard time of 0.3 seconds occurs before beamforming. The experiments were automated by creating a compact disk with the one second primary beacon signal repeating every 10 seconds for the three-source tests.

### 5.3 Experimental results

To statistically analyze the experimental results, the uncompressed .wav recordings of \( N \) wired and acoustic beamforming tests were transferred to a PC and analyzed in MATLAB. The same data analysis methodology in Section 4.3 is used for the three-source case. The “power ratio” \( \rho \) of the beamformer is calculated as

\[
\rho[n] = \left( \frac{\hat{a}_{bf}[n]}{\hat{a}_{10}[n] + \hat{a}_{20}[n] + \hat{a}_{30}[n]} \right)^2
= \left( \frac{\hat{a}_{bf}[n]}{\hat{a}_{10}[n] + (\frac{\hat{a}_{12}[n] + \hat{a}_{22}[n]}{2}) + \hat{a}_{30}[n]} \right)^2.
\]

The amplitude estimates in each test are obtained via the MLE FFT technique described in [18] using an 0.2 second window of the steady state portion of each signal from timeslots TS\(^{(1)}\) to TS\(^{(5)}\). As with three nodes, a power ratio of one corresponds to an ideal beamformer with perfect phase alignment.

Table 5.2 shows the experimental results of three-source wired tests. As can be seen in this table, four wired-channel experiments were performed and these experiments consistently resulted in power ratios greater than 0.98 and standard deviations on the order of \( 10^{-3} \). Hence, the wired-channel experiments confirmed that the round-trip carrier synchronization protocol with \( M \) source nodes can consistently offer near ideal performance over “perfect” channels.
Table 5.2: Experimental results of three-source wired test.

<table>
<thead>
<tr>
<th></th>
<th>Experiment1</th>
<th>Experiment2</th>
<th>Experiment3</th>
<th>Experiment4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min($\rho$)</td>
<td>0.9910</td>
<td>0.9920</td>
<td>0.9890</td>
<td>0.9880</td>
</tr>
<tr>
<td>Max($\rho$)</td>
<td>0.9960</td>
<td>0.9960</td>
<td>0.9960</td>
<td>0.9950</td>
</tr>
<tr>
<td>Mean($\rho$)</td>
<td>0.9937</td>
<td>0.9937</td>
<td>0.9919</td>
<td>0.9920</td>
</tr>
<tr>
<td>Std($\rho$)</td>
<td>0.0015</td>
<td>0.0015</td>
<td>0.0018</td>
<td>0.0017</td>
</tr>
</tbody>
</table>

Figure 5.3 shows a histogram of the three-source power ratios over 100 tests of the time-slotted round-trip carrier synchronization protocol with one second beacons and 0.25 second guard times. The mean power ratio of the distributed beamformer was computed to be 0.9407 and the standard deviation was computed to be approximately 0.0168.

These results show that the time-slotted round-trip carrier synchronization protocol with three source nodes was consistently effective at synchronizing the phase of the carriers of the ACCENT nodes and that the synchronization errors lead to only a small loss in performance with respect to the ideal prediction.
Chapter 6

Conclusions and discussions

The two-source acoustic results shown in Figure 4.10 are very similar to the results obtained over wired channels in the sense of mean power ratio whereas the average power ratio of the three-source results shown in Figure 5.3 is somewhat lower than the average ratios observed in the wired experiments. One important factor in the acoustic experiments is that the microphone and speaker at each source node (and at the destination) are separate transducers in slightly different locations with different radiation patterns. Hence, the channel reciprocity between each pair of nodes required by the round-trip protocol is only approximate.

It is worth emphasizing that the two-source and three-source power ratio results over acoustic channels are nevertheless consistent with the wired channel results in that the standard deviation of the acoustic experiments is similar to the standard deviation of the wired channel results. The consistency of these results confirms that the source node PLLs are converging consistently and that the round-trip protocol can be used to realize a distributed beamformer with near-ideal performance and low computational complexity even in noisy multipath channels.

This thesis summarizes the results of an experimental study on the achievable performance of wired and acoustic distributed beamforming system with carrier wavelengths equivalent to 900MHz electromagnetic propagation using time-slotted round-trip carrier synchronization. While recent research in this area has focused primarily on theoretical
studies and the development of efficient synchronization protocols, relatively little has been published on prototypes and/or experimental studies of distributed transmit beamforming. This thesis serves as an extension of the work in [10]. The experimental results presented in this thesis not only confirmed the theoretical predictions of the time-slotted round-trip carrier synchronization using wireless acoustic propagation in noisy multipath channels, but also confirmed that a distributed beamformer based on the optimized hybrid PLL with averaging window can consistently achieve a large fraction of the power gains of an ideal conventional beamformer.

There are many potential research opportunities following this thesis. One of them is the hybrid PLL performance in the presence of noise. Since one of the most intriguing properties of the PLLs is its ability to extract signals from an extremely noisy environment. Therefore, it is worth studying the impact of noise on the hybrid PLL performance. Also, no particular channels or noises are implemented in our acoustic experiments. The system performance under noisy environment is likely to be another research topic. On top of that, a more controllable experimental setup needs to be built so that more accurate phase alignment can be achieved in the system.
Appendix A

Three-source round-trip synchronization protocol timing.

In three-source acoustic experiment, when the start of the primary beacon is detected, the nodes execute the round-trip protocol according to the schedule in Table A.1. The duration of each beacon was one second with a 0.25 seconds guard time between timeslots. After the final beacon a guard time of 0.3 seconds occurs before beamforming. The experiments were automated by creating a compact disk with the one second primary beacon signal repeating every 10 seconds.
<table>
<thead>
<tr>
<th>TS</th>
<th>Time</th>
<th>S1</th>
<th>S2</th>
<th>S3</th>
</tr>
</thead>
<tbody>
<tr>
<td>TS0</td>
<td>0.00s</td>
<td>detect primary beacon</td>
<td>detect primary beacon</td>
<td>detect primary beacon</td>
</tr>
<tr>
<td></td>
<td>0.00-0.10s</td>
<td>wait</td>
<td>wait</td>
<td>wait</td>
</tr>
<tr>
<td></td>
<td>0.10-0.16s</td>
<td>track PLL1 with PFD</td>
<td>track PLL1 with PFD</td>
<td>track PLL1 with PFD</td>
</tr>
<tr>
<td></td>
<td>0.16-0.60s</td>
<td>track PLL1 with MUL</td>
<td>track PLL1 with MUL</td>
<td>track PLL1 with MUL</td>
</tr>
<tr>
<td></td>
<td>0.60s</td>
<td>compute holdover for PLL1</td>
<td>compute holdover for PLL1</td>
<td>compute holdover for PLL1</td>
</tr>
<tr>
<td></td>
<td>0.60-1.25s</td>
<td>holdover PLL1</td>
<td>holdover PLL1</td>
<td>holdover PLL1</td>
</tr>
<tr>
<td>TS1</td>
<td>1.25-1.35s</td>
<td>transmit holdover PLL1; also holdover PLL1</td>
<td>holdover PLL1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.35-1.41s</td>
<td>transmit holdover PLL1; also holdover PLL1 and track</td>
<td>holdover PLL1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.41-1.85s</td>
<td>transmit holdover PLL1; also holdover PLL1 and track</td>
<td>holdover PLL1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.85s</td>
<td>transmit holdover PLL1; also compute holdover for PLL2</td>
<td>holdover PLL1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.85-2.25s</td>
<td>transmit holdover PLL1; also holdover PLL1 and PLL2</td>
<td>holdover PLL1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.25-2.50s</td>
<td>wait</td>
<td>holdover PLL1 and PLL2</td>
<td>holdover PLL1</td>
</tr>
<tr>
<td>TS2</td>
<td>2.50-2.60s</td>
<td>wait</td>
<td>transmit holdover PLL1; also holdover PLL1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.60-2.66s</td>
<td>wait</td>
<td>transmit holdover PLL1; also holdover PLL1 and track</td>
<td>holdover PLL2</td>
</tr>
<tr>
<td></td>
<td>2.66s-3.10s</td>
<td>wait</td>
<td>transmit holdover PLL1; also holdover PLL1 and track</td>
<td>PLL2 with PFD</td>
</tr>
<tr>
<td></td>
<td>3.10</td>
<td>wait</td>
<td>transmit holdover PLL1; also transmit holdover PLL1</td>
<td>holdover PLL1 and PLL2</td>
</tr>
<tr>
<td></td>
<td>3.10-3.50s</td>
<td>wait</td>
<td>transmit holdover PLL1; also transmit holdover PLL1</td>
<td>holdover PLL1 and PLL2</td>
</tr>
<tr>
<td></td>
<td>3.50-3.75s</td>
<td>wait</td>
<td>transmit holdover PLL1; also transmit holdover PLL2</td>
<td>holdover PLL1 and PLL2</td>
</tr>
<tr>
<td>TS</td>
<td>Time</td>
<td>$S_1$</td>
<td>$S_2$</td>
<td>$S_3$</td>
</tr>
<tr>
<td>-----</td>
<td>-----------</td>
<td>----------------</td>
<td>------------------------------------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>TS3</td>
<td>3.75-3.85s</td>
<td>wait</td>
<td>holdover PLL1 and PLL2</td>
<td>transmit PLL1; also holdover PLL2</td>
</tr>
<tr>
<td></td>
<td>3.85-3.91s</td>
<td>wait</td>
<td>holdover PLL1, PLL2 and PLL3</td>
<td>transmit PLL1; also holdover PLL2</td>
</tr>
<tr>
<td></td>
<td>3.91-4.35s</td>
<td>wait</td>
<td>track PLL3 with PFD</td>
<td>transmit PLL1; also holdover PLL2</td>
</tr>
<tr>
<td></td>
<td>4.35s</td>
<td>wait</td>
<td>compute holdover for PLL3</td>
<td>transmit PLL1; also holdover PLL2</td>
</tr>
<tr>
<td></td>
<td>4.35-4.75s</td>
<td>wait</td>
<td>holdover PLL1, PLL2 and PLL3</td>
<td>transmit PLL1; also holdover PLL2</td>
</tr>
<tr>
<td></td>
<td>4.75-5.00s</td>
<td>wait</td>
<td>holdover PLL1, PLL2 and PLL3</td>
<td>holdover PLL2</td>
</tr>
<tr>
<td></td>
<td>5.00-5.10s</td>
<td>wait</td>
<td>transmit PLL3; also holdover PLL1</td>
<td>holdover PLL2</td>
</tr>
<tr>
<td></td>
<td>5.10-5.16s</td>
<td>track PLL2 with PFD</td>
<td>transmit PLL3; also holdover PLL1</td>
<td>holdover PLL2</td>
</tr>
<tr>
<td></td>
<td>5.16-5.60s</td>
<td>track PLL2 with MUL</td>
<td>transmit PLL3; also holdover PLL1</td>
<td>holdover PLL2</td>
</tr>
<tr>
<td></td>
<td>5.60s</td>
<td>compute holdover for PLL2</td>
<td>transmit PLL3; also holdover PLL1</td>
<td>holdover PLL2</td>
</tr>
<tr>
<td></td>
<td>5.60-6.00s</td>
<td>holdover PLL2</td>
<td>transmit PLL3; also holdover PLL1</td>
<td>holdover PLL2</td>
</tr>
<tr>
<td></td>
<td>6.00-6.30s</td>
<td>holdover PLL2</td>
<td>holdover PLL1, PLL2 and PLL3</td>
<td>holdover PLL2</td>
</tr>
<tr>
<td>TS5</td>
<td>6.30s</td>
<td>holdover PLL2</td>
<td>compute $\omega_{20}$ and $\phi_{20}$</td>
<td>holdover PLL2</td>
</tr>
<tr>
<td></td>
<td>6.30-8.30s</td>
<td>transmit PLL2</td>
<td>transmit the combo of the local estimates</td>
<td>transmit PLL2</td>
</tr>
<tr>
<td>TS</td>
<td>Time</td>
<td>$S_1$</td>
<td>$S_2$</td>
<td>$S_3$</td>
</tr>
<tr>
<td>----</td>
<td>--------</td>
<td>----------------</td>
<td>----------------</td>
<td>----------------</td>
</tr>
<tr>
<td></td>
<td>8.30s</td>
<td>re-arm primary beacon detector</td>
<td>re-arm primary beacon detector</td>
<td>re-arm primary beacon detector</td>
</tr>
</tbody>
</table>

Table A.1: Three-source round-trip synchronization protocol timing.
Appendix B

Source Code of the TMS320C6713 source node in the two-source system

The following C code is the software implementation of the two-source time-slotted round-trip carrier synchronization system. When DIP switch 1 is pressed, the DSK board performs as source node 1, whereas DIP switch 2 corresponds to source node 2.
#define CHIP_6713

/* Header Files */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <c6x.h>
#include <cs1.h>
#include <cs1_mcbsp.h>
#include <cs1_irq.h>

#include "dsk6713.h"
#include "dsk6713_aic23.h"

/* Codec Configurations */

DSK6713_AIC23_CodecHandle hCodec;
DSK6713_AIC23_Config config = DSK6713_AIC23_DEFAULTCONFIG;

// Codec configuration with default settings

// Defines

#define MAX 32767     //max value of ADC Output
#define PI 3.141592654
#define N 2 //Number of Node
#define M 9288 //length of LF averaging window for holdover
#define LENVEDT 88 // length of averaging window for envelope detector

// Constant

//*******************************

// Time Table for Each Time Slot*******************************

//For TS0, TS1, TS2

const unsigned int start_slot = 0;   //start of time slot
const unsigned int start_listen = 4410; //start to listen @ 0.1s
const unsigned int shift_time = 7183; //shift from PFD to MUL @ (0.0629+0.1)s
const unsigned int stop_listen = 26460; //stop to listen @ 0.6s
const unsigned int end_slot = 55125; // end of time slot @ 1.25s
const unsigned int start_play = 0; // start to play
const unsigned int stop_play = 44100; // stop to play @ 1s

// For TS3
const unsigned int start_slot_last = 0; // start of time slot
const unsigned int start_play_last = 2205; // wait 0.05s before playing
const unsigned int stop_play_last = 90405; // play 2s
const unsigned int end_slot_last = 125685; // wait 0.8s for end of time

const unsigned int Kd = 1;
const unsigned int f_vco = 1021;
const unsigned int Ko = 10;
const double fs = 44100;

const double num[3] = {0.015851338604076, 0.000007564091554, 
                      -0.015843774512522};
const double den[3] = {1.0000000000000000, -1.995238330480629, 
                       0.995238330480629};
const double iir_num[3] = {0.01380680781, 0, -0.01380680781};
const double iir_den[3] = {1, -1.951554418, 0.9723863602};
const double threshold = 0.01;
const double pi = PI;

const unsigned int m = M;

// Optimal parameter set

// Variables

// For each PLL

double vco_out[N][2];  // Initialize VCO output
double vco_out_sin[N][2];  // Initialize VCO output — sin
double vco_out_cos[N][2];  // Initialize VCO output — cos
double pd_out[N][3];  // Initialize phase detector output
double uf[N][3];  // Initialize loop filter output
#pragma DATA_SECTION(holdover,".EXTRAM");

double holdover[N][M]; // for both PLL1 and PLL2
unsigned int holdover_index[N];
unsigned int status[N][3];
double sig_in[N][2]; // save status for PFD
double maxScale[N];
double Cd[N];
double running_sum[N];

// For Detector
double iir_buffer[3];
unsigned short detector_index;
double detector_sum;
double detector_buffer[LENVEEDT];

// Holdover Parameters
double w_vco[N]; // Initialize VCO frequency
double phi_vco[N]; // Initialize VCO phase
double running_avg[N]; // running sum average

// Other Parameters
double inv_fs;
unsigned int node;
double inv_max;
unsigned short TS; // time slot
unsigned short start_flag = 0; // start flag
unsigned int counter;
double invM;
double twopi;

interrupt void serialPortRcvISR(void);

void PLL(double input, unsigned short PLLnum, unsigned int PhaseDetector);
void PLL_init(unsigned short PLLnum);
double PLL_play(unsigned short PLLnum);
void VCO(unsigned short PLLnum, double holdover_value, unsigned int mode);
void main()
{
    unsigned int i, j;
    DSK6713_init();  // Initialize the board support library
    DSK6713_DIP_init();  // Initializes DIP switches
    DSK6713_LED_init();  // Initializes LEDs
    hCodec = DSK6713_AIC23_openCodec(0, &config);  // Open the codec
    // Configure buffered serial port for 32-bit operation
    // This allows transfer of both right and left channels in one read/write
    MCBSP_FSETS(SPCR1, RINTM, FRM);
    MCBSP_FSETS(SPCR1, XINTM, FRM);
    MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);
    MCBSP_FSETS(XCR1, XWDLEN1, 32BIT);
    DSK6713_AIC23_setFreq(hCodec, DSK6713_AIC23_FREQ_44KHZ);
    // set the sampling rate
    inv_fs = (double)(1/(double)fs);
    inv_max = (double)(1/(double)32767);
    invM = (double)(1/(double)m);
    twopi = pi*2;
    node = 0;
    counter = 0;
    TS = 0;
    start_flag = 0;
    for (j = 0; j < 2; j++)
    {
        w_vco[j] = 0;
        maxScale[j] = -0.1;
        Cd[j] = 1;
        holdover_index[j] = 0;
running_sum[j] = 0;
phi_vco[j] = 0;
running_avg[j] = 0;

vco_out[j][0] = 0;
vco_out[j][1] = 0;
vco_out_sin[j][0] = 0;
vco_out_sin[j][1] = 0;
vco_out_cos[j][0] = 0;
vco_out_cos[j][1] = 0;
sin[j][0] = 0;
sin[j][1] = 0;

pd_out[j][0] = 0;
pd_out[j][1] = 0;
pd_out[j][2] = 0;
uf[j][0] = 0;
uf[j][1] = 0;
uf[j][2] = 0;
status[j][0] = 0;
status[j][1] = 0;
status[j][2] = 0;

for (i = 0; i < m; i++)
{
    holdover[j][i] = 0;
}
Pll_init(0);
Pll_init(1);

detector_index = 0;
detector_sum = 0;

for (i = 0; i < LENVEDT; i++)
{
detector_buffer[i] = 0;
}
iir_buffer[0] = 0;
iir_buffer[1] = 0;
iir_buffer[2] = 0;

// Interrupt setup
IRQ_globalDisable(); // Globally disables interrupts
IRQ_nmiEnable(); // Enables the NMI interrupt
IRQ_map(IRQ_EVT_RINT1, 15); // Maps an event to a physical interrupt
IRQ_enable(IRQ_EVT_RINT1); // Enables the event
IRQ_globalEnable(); // Globally enables interrupts

// Get Node information

//show LED
if (DSK6713_DIP_get(1) == 0)
{
    node = 1;
}
else if (DSK6713_DIP_get(2) == 0)
{
    node = 2;
}
DSK6713_LED_on(node);

// Infinite Loop
while (1)
{
    // Do nothing here...
}

interrupt void serialPortRevISR()
{
    // Uses the union construct to have the same
    // memory referenced by two different variables
    union {Uint32 combo; short channel[2];} data;
    double iir_output = 0;
double input = 0;
double output = 0;

data.combo = MCBSP_read(DSK6713_AIC23_DATAHANDLE);

// Read L+R channels

input = (double)data.channel[0] * inv_max;

// IIR

iir_buffer[2] = iir_buffer[1];
iir_buffer[1] = iir_buffer[0];

if (start_flag == 0)
{

// ********** Signal Detector **********

// Detection

detector_buffer[detector_index] = iir_output * iir_output;

if (detector_index < (LENVEDT - 1))
{

detector_sum = detector_sum + detector_buffer[detector_index] -
detector_buffer[detector_index+1];
detector_index++;
}
else
{

detector_sum = detector_sum + detector_buffer[detector_index] -
detector_buffer[0];
detector_index = 0;
}

if (detector_sum > threshold)
{

counter = 0;
start_flag = 1;
}
TS = 0;
PLL_Init(0);
PLL_Init(1);

/**************************** Detection END****************************/

} else if (start_flag == 1)
{
input = iir_output;
if (TS == 0)
{
if ((counter < start_listen) && (counter >= start_slot))  // 0.1s wait
{
if (counter < 88)
{
detector_buffer[counter] = 0;
}
counter++;
}
else if ((counter >= start_listen) && (counter < shift_time))
// 0.0629s PLL,PFD
{
PLL(input, 0, 0);  // first PLL,PFD
counter++;  
}
else if ((counter >= shift_time) && (counter < stop_listen))
{
PLL(input, 0, 1);  // first PLL, MUL

counter++;  
}
else if (counter == stop_listen)
{
running_avg[0] = running_sum[0] * invM;
VCO(0, running_avg[0], 0);
counter++;  
}
else if ((counter > stop_listen) && (counter < end_slot))

{ 
  VCO(0, running_avg[0], 0);
  counter ++;
}
else if (counter == end_slot)
{
  VCO(0, running_avg[0], 0);
  TS++;
  counter = 0;
}
}

else if (TS == 1)
{
  if (node == 1)
  {
    if ((counter>=start_play) && (counter <= stop_play))
    {
      VCO(0, running_avg[0], 0);
      output = PLLplay(0);
      if (counter < m)
      {
        holdover[0][counter] = 0;
        holdover[1][counter] = 0;
      }
    }
    data.channel[0] = (short)(output*32000);
    data.channel[1] = data.channel[0];
    counter++;
    MCBSP_write(DSK6713_AIC23_DATAHANDLE, data.combo);  // Write L+R channels
  }
  else if ((counter > stop_play) && (counter < end_slot))
  {
    counter++;
  }
  else if (counter == end_slot)
328 {  
329     counter = 0;  
330     TS++;  
331 }  
332 }  
333 else if (node ==2)  
334 {  
335     if ((counter < start_listen)&&(counter>=start_slot)) //0.1s wait  
336     {  
337         VCO(0,running_avg[0],0);  
338         counter++;  
339     }  
340     else if ((counter >= start_listen) && (counter < shift_time)) //0.5s PLL  
341     {  
342         VCO(0,running_avg[0],0);  
343         PLL(input,1,0); //second PLL  
344         counter++;  
345     }  
346     else if ((counter >= shift_time) && (counter < stop_listen))  
347     {  
348         VCO(0,running_avg[0],0);  
349         PLL(input,1,1); //first PLL, MUL  
350         counter++;  
351     }  
352     else if (counter == stop_listen)  
353     {  
354         running_avg[1] = running_sum[1]*invM;  
355         VCO(1,running_avg[1],0);  
356         VCO(0,running_avg[0],0); //update first PLL holdover  
357         counter++;  
358     }  
359     else if ((counter > stop_listen) && (counter < end_slot))  
360     {  
361         VCO(1,running_avg[1],0);  
362         VCO(0,running_avg[0],0); //update first PLL holdover  
363         counter ++;  
364     }
else if (counter == end_slot)
{
VCO(1, running_avg[1], 0);
VCO(0, running_avg[0], 0); //update first PLL holdover
TS++;
counter = 0;
}

else if (TS == 2)
{
if (node == 1)
{
if ((counter < start_listen)&&(counter>=start_slot)) //0.1s wait
{
    counter++;
}
else if ((counter >= start_listen) && (counter < shift_time)) //0.5s PLL
{
    PLL(input, 1, 0); //second PLL
    counter++;
}
else if ((counter >= shift_time) && (counter < stop_listen))
{
    PLL(input, 1, 1); //first PLL, MUL
    counter++;
}
else if (counter == stop_listen)
{
    running_avg[1] = running_sum[1]*invM;
    VCO(1, running_avg[1], 0);
    counter++;
}
else if ((counter > stop_listen) && (counter < end_slot))
{
    VCO(1, running_avg[1], 0);
        counter ++;
    } else if (counter == end_slot)
    {
        VCO(1,running_avg[1],0);
        TS++;
        counter = 0;
    }
    else if (node == 2)
    {
        if ((counter >= start_play) && (counter <= stop_play))
        {
            VCO(1,running_avg[1],0);
            VCO(0,running_avg[0],0); //update first PLL holdover
            output = PLL_play(0);
            if (counter < m)
            {
                holdover[0][counter] = 0;
                holdover[1][counter] = 0;
            }
            data.channel[0] = (short)(output*32000);
            data.channel[1] = data.channel[0];
            counter++;
            MCBSP_write(DSK6713_AIC23_DATAHANDLE, data.combo); // Write L+R channels
        }
        else if ((counter > stop_play) && (counter < end_slot))
        {
            VCO(1,running_avg[1],0);
            counter++;
        }
        else if (counter == end_slot)
        {
            VCO(1,running_avg[1],0);
            counter = 0;
            TS++;
        }
elseif (TS == 3)
{
    if ((counter >= start_slot_last) && (counter < start_play_last))
    {
        VCO(1,running_avg[1],0);
        counter++;
    }
    else if ((counter >= start_play_last) && (counter < stop_play_last))
    {
        VCO(1,running_avg[1],0);
        output = PLL_play(1);
        counter++;
        data.channel[0] = (short)(output*32000);
        data.channel[1] = data.channel[0];
        MCBSP_write(DSK6713_AIC23_DATAHANDLE, data.combo);  // Write L+R channels
    }
    else if ((counter >= stop_play_last) && (counter < end_slot_last))
    {
        counter++;
    }
else if (counter == end_slot_last)
{
    PLL_init(0);
    PLL_init(1);
    TS = 0;
    start_flag = 0;
    counter = 0;
    detector_index = 0;
    detector_sum = 0;
    iir_buffer[0] = 0;
    iir_buffer[1] = 0;
    iir_buffer[2] = 0;
void PLL(double input, unsigned short PLLnum, unsigned int PhaseDetector)
{
    double up1 = 0.0, up2 = 0;

    unsigned int index = holdover_index[PLLnum];

    /*****************************************************************************
    Phase detector**************************************************************************/
    if (PhaseDetector == 0)
    /*****************************************************************************
    PFD**************************************************************************/
    {
        if (input > maxScale[PLLnum])
            maxScale[PLLnum] = input;
        if (input <= 0)
            sig_in[PLLnum][0] = 1;
        else if (input < 0)
            sig_in[PLLnum][0] = 0;

        if (vco_out_sin[PLLnum][0] >= 0)
            vco_out_sin[PLLnum][0] = 1;
        else if (vco_out_sin[PLLnum][0] < 0)
            vco_out_sin[PLLnum][0] = 0;

        up1 = (!sig_in[PLLnum][1] & & sig_in[PLLnum][0]);
        up2 = (!vco_out_sin[PLLnum][1] & & vco_out_sin[PLLnum][0]);

        status[PLLnum][0] = (sig_in[PLLnum][1] != 1) & & (up1 & & !up2);
        status[PLLnum][1] = (sig_in[PLLnum][1] != -1) & & (!up1 & & up2);
        status[PLLnum][2] = (!status[PLLnum][0]) & & (!status[PLLnum][1]);

        if (status[PLLnum][0])
        { ... }
99

513 pd_out[PLLnum][0] = pd_out[PLLnum][1] + twopi;
514 else if (status[PLLnum][1])
515 pd_out[PLLnum][0] = pd_out[PLLnum][1] - twopi;
516 else if (status[PLLnum][2])
517 pd_out[PLLnum][0] = pd_out[PLLnum][1];
518 }
519 else if (PhaseDetector == 1)
520 /***************************************************************************/
521 {
522 Cd[PLLnum] = 2*Kd/maxScale[PLLnum];
523 pd_out[PLLnum][0] = Cd[PLLnum]*input*vco_out*cos[PLLnum][0];
524 }
525 /***************************************************************************/
526 /**********END Phase detector**********/
527 /***************************************************************************/
528 uf[PLLnum][0] = num[0] * pd_out[PLLnum][0] + num[1] * pd_out[PLLnum][1] + num
530 holdover[PLLnum][index] = uf[PLLnum][0];
531
532 if (index< (m-1))
533 {
535 [PLLnum][index+1];
536 holdover_index[PLLnum]++;
537 }
538 else
539 {
541 [PLLnum][0];
542 holdover_index[PLLnum]= 0;
543 }
544 /***************************************************************************/
545 VCO(PLLnum, uf[PLLnum][0], 0);
546 //exchange variable
void VCO(unsigned short PLLnum, double holdover_value, unsigned int mode)
{
    if (mode == 0) // tracking
    {
        w_vco[PLLnum] = twopi*f_vco+Ko*holdover_value;
    }
    else if (mode == 1) // update
    {
        w_vco[PLLnum] = 1021.0 * twopi;
    }
    phi_vco[PLLnum] = phi_vco[PLLnum] + w_vco[PLLnum]*inv_fs;
    while (phi_vco[PLLnum] > twopi)
    while (phi_vco[PLLnum] < 0)
    phi_vco[PLLnum] = phi_vco[PLLnum] + twopi;
    vco_out_sin[PLLnum][1] = vco_out_sin[PLLnum][0];
    vco_out_sin[PLLnum][0] = sin(phi_vco[PLLnum]);
    vco_out_cos[PLLnum][1] = vco_out_cos[PLLnum][0];
    vco_out_cos[PLLnum][0] = cos(phi_vco[PLLnum]);
}

void PLL_init(unsigned short PLLnum)
{
    w_vco[PLLnum] = 0;
maxScale[PLLnum] = -0.1;
Cd[PLLnum] = 1;
holdover_index[PLLnum]= 0;
running_sum[PLLnum] = 0;
phi_vco[PLLnum]= 0 ;
running_avg[PLLnum] = 0;

vco_out[PLLnum][0] = 0;
vco_out[PLLnum][1] = 0;
vco_out_sin[PLLnum][0] = 0;
vco_out_sin[PLLnum][1] = 0;
vco_out_cos[PLLnum][0] = 0;
vco_out_cos[PLLnum][1] = 0;
sig_in[PLLnum][0] = 0;
sig_in[PLLnum][1] = 0;
pd_out[PLLnum][0] = 0;
pd_out[PLLnum][1] = 0;
pd_out[PLLnum][2] = 0;
uf[PLLnum][0] = 0;
uf[PLLnum][1] = 0;
uf[PLLnum][2] = 0;
status[PLLnum][0] = 0;
status[PLLnum][1] = 0;
status[PLLnum][2] = 0;
}

do\_le\_PLL\_play(unsigned\_short\ PLLnum)
{
  double output;

  output = sin(phi_vco[PLLnum]);

  return output;
}
Appendix C

Source Code of the TMS320C6713 source node in the three-source system

The following C code is the software implementation of the three-source time-slotted round-trip carrier synchronization system. When DIP switch 1 is pressed, the DSK board performs as source node 1, whereas DIP switch 2 and DIP switch 3 corresponds to source node 2 and 3 respectively.
```c
#define CHIP_6713

/* Header Files */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <c6x.h>
#include <csl.h>
#include <csl_mcbsp.h>
#include <csl_irq.h>

#include "dsk6713.h"
#include "dsk6713_aic23.h"

/* Codec Configurations */
DSK6713_AIC23_CodecHandle hCodec;
DSK6713_AIC23_Config config = DSK6713_AIC23_DEFAULTCONFIG;

// Codec configuration with default settings

// Defines

#define MAX 32767 //max value of ADC Output
#define PI 3.14159265358979
#define N 3 //Number of Node
#define M 9288 //length of LF averaging window for holdover
#define LENVEDT 88 //length of averaging window for envelope detector

// Constant

// Time Table for Each Time Slot*************/
// For TS0, TS1, TS2, TS3, TS4
const unsigned int start_slot = 0;  // start of time slot @ 0s
const unsigned int start_listen = 4410; // start to listen @ 0.1s
```
const unsigned int shift_time = 7183; //shift from PFD to MUL @ (0.34+0.1)s
const unsigned int stop_listen = 26460; //stop to listen @ 0.6s
const unsigned int end_slot = 55125; //end of time slot @ 1.5s
const unsigned int start_play = 0; //start to play @ 0s
const unsigned int stop_play = 44100; //stop to play @ 1s

//For TS5
const unsigned int start_slot_last = 0; //start of time slot @ 0s
const unsigned int start_play_last = 2205; //start to play @ 0.05s
const unsigned int stop_play_last = 90405; //stop to play for @2.05s
const unsigned int end_slot_last = 125685; wait 1.2s for end of time slot

const unsigned int Kd = 1; //Gain of Phase Detector
const unsigned int f_vco = 1021; //central frequency of VCO
const unsigned int Ko = 10; //Gain of VCO
const double fs = 44100; //sampling frequency
const double num[3] = {0.015851338604076, 0.000007564091554, -0.015843774512522};
const double den[3] = {1.0000000000000000, -1.995238330480629, 0.995238330480629};
const double iir_num[3] = {0.01380680781, 0.001380680781};
//NUM of IIR Peak Filter
const double iir_den[3] = {1, -1.951554418, 0.9723863602};
//DEN of IIR Peak Filter
const double threshold = 0.01; //threshold of Signal Detector
const double pi = PI;
const unsigned int m = M; //length of LF averaging window for holdover

//-------------------------------
//Variables
//-------------------------------
//For each PLL

double vco_out[N][2]; //Initialize VCO output
double vco_out_sin[N][2]; //Initialize VCO output — sin
double vco_out_cos[N][2]; //Initialize VCO output — cos
double pd_out[N][3]; //Initialize phase detector output
double uf[N][3]; //Initialize loop filter output
#pragma DATA SECTION(holdover",".EXTRAM");
double holdover[N][M]; // for both PLL1 and PLL2
unsigned int holdover_index[N]; //Holdover Window
unsigned int status[N][3]; //save status for PFD
double sig_in[N][2];
double maxScale[N]; //maximum of input
double Cd[N]; //Gain of MUL
double running_sum[N]; //running sum for Holdover

//For Detector
double iir_buffer[3]; //IIR Filter Buffer
unsigned short detector_index; //Signal Detector Index
double detector_sum; //Signal Detector sum
double detector_buffer[LENVEDT]; //Signal Detector Buffer

//Holdover Parameters
double w_vco[N]; //Initialize VCO frequency
double phi_vco[N]; //Initialize VCO phase
double running_avg[N]; //running sum average
//for the middle node
double w_vco_m; //VCO frequency for middle node w_vco_m = w12+w32-w02;
double phi_vco_m; //VCO phase for middle node phi_vco_m = theta12+theta32-
theta02;

//Other Parameters
double inv_fs; //inv_fs = 1/fs;
unsigned int node; //node number (1 or 2)
double inv_max; //inv_max = 1/32768 (32768 is the maxscale of 6713 Codec)
unsigned short TS; //time slot
unsigned short start_flag = 0; //start flag
unsigned int counter; //Global Counter
double invM; //invM = 1/M;
double twopi; // twopi = 2*pi

interrupt void serialPortRcvISR(void); //Interrupt Function
void PLL(double input, unsigned short PLLnum, unsigned int PhaseDetector);
//Phase Locked Loop update function
//Parameter
//input = read value from Codec (between −1 and 1)
//PLLnum = flag of PLL indicate (1st PLL: PLLnum = 0; 2nd PLL: PLLnum = 1)
//PhaseDetector = flag of Phase detector (PFD: PhaseDetector = 0; MUL:
//                   PhaseDetector = 1);
void PLL_init(unsigned short PLLnum);
//Initialize PLL variables
//Parameter
//PLLnum = flag of PLL indicate (1st PLL: PLLnum = 0; 2nd PLL: PLLnum = 1)
double PLL_play(unsigned short PLLnum);
//Generate output value
void VCO(unsigned short PLLnum, double holdover_value, unsigned short mode);
//Update frequency and phase
//Parameter
//mode: 0 is tracking mode and 1 is update mode
void main(void)
{
    unsigned int i, j;

    DSK6713_init();
    // Initialize the board support library
    DSK6713_DIP_init(); // Initializes DIP switches
    DSK6713_LED_init(); // Initializes LEDs

    hCodec = DSK6713_AIC23_openCodec(0, &config); //Open the codec

    // Configure buffered serial port for 32-bit operation
    // This allows transfer of both right and left channels in one read/write
    MCBSP_FSETS(SPCR1, RINTM, FRM);
    MCBSP_FSETS(SPCR1, XINTM, FRM);
    MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);
    MCBSP_FSETS(XCR1, XWDLEN1, 32BIT);
DSK6713_AIC23_setFreq(hCodec, DSK6713_AIC23_FREQ_44KHZ);  
"set the sampling rate"

\[ \text{inv}_fs = (\text{double})(1/(\text{double})fs); \]

\[ \text{inv}_\text{max} = (\text{double})(1/(\text{double})32767); \]

\[ \text{invM} = (\text{double})(1/(\text{double})\text{m}); \]

two\(\pi\) = \(\pi\times2;\)

\[ \text{node} = 0; \]

\[ \text{counter} = 0; \]

\[ \text{counter_temp} = 0; \]

\[ \text{TS} = 0; \]

\[ \text{start_flag} = 0; \]

\[ \text{for} \ (j = 0; j < \text{N}; j++) \{ \]

\[ \text{w_vco}[j] = 0; \]

\[ \text{maxScale}[j] = -0.1; \]

\[ \text{Cd}[j] = 1; \]

\[ \text{holdover_index}[j] = 0; \]

\[ \text{running_sum}[j] = 0; \]

\[ \text{phi_vco}[j] = 0; \]

\[ \text{running_avg}[j] = 0; \]

\[ \text{vco_out}[j][0] = 0; \]

\[ \text{vco_out}[j][1] = 0; \]

\[ \text{vco_out_sin}[j][0] = 0; \]

\[ \text{vco_out_sin}[j][1] = 0; \]

\[ \text{vco_out_cos}[j][0] = 0; \]

\[ \text{vco_out_cos}[j][1] = 0; \]

\[ \text{sig_in}[j][0] = 0; \]

\[ \text{sig_in}[j][1] = 0; \]

\[ \text{pd_out}[j][0] = 0; \]

\[ \text{pd_out}[j][1] = 0; \]

\[ \text{pd_out}[j][2] = 0; \]

\[ \text{uf}[j][0] = 0; \]
uf[j][1] = 0;
uf[j][2] = 0;
status[j][0] = 0;
status[j][1] = 0;
status[j][2] = 0;

for (i = 0; i < m; i++)
{
    holdover[j][i] = 0;
}

PLL_init(0);
PLL_init(1);
PLL_init(2);
w_vcom = 0;
phi_vcom = 0;
detector_index = 0;
detector_sum = 0;
for (i = 0; i < LENVDT; i++)
{
    detector_buffer[i] = 0;
}

iir_buffer[0] = 0;
iir_buffer[1] = 0;
iir_buffer[2] = 0;

// Interrupt setup
IRQ_globalDisable(); //Globally disables interrupts
IRQ_nmiEnable(); //Enables the NMI interrupt
IRQ_map(IRQ_EVT_RINT1, 15); //Maps an event to a physical interrupt
IRQ_enable(IRQ_EVT_RINT1); //Enables the event
IRQ_globalEnable(); //Globally enables interrupts

//Get Node information

//show LED
if (DSK6713_DIP_get(1) == 0) {
    node = 1;
}
else if (DSK6713_DIP_get(2) == 0) {
    node = 2;
}
else if (DSK6713_DIP_get(3) == 0) {
    node = 3;
}
DSK6713_LED_on(node);

// Infinite Loop
while(1) {
    // Do nothing here...
}

interrupt void serialPortRcvISR() {
    union {Uint32 combo; short channel[2];} data;
double iir_output = 0;
double input=0;
double output = 0;

data.combo = MCBSP_read(DSK6713_AIC23_DATAHANDLE);
//Read L+R channels
input = (double)data.channel[0]*inv_max;

//IIR
iir_buffer[2] = iir_buffer[1];
\begin{verbatim}
    iir_buffer[1] = iir_buffer[0];

    if (start_flag == 0)
        {
            //**********Signal Detector**********/ 
            //Detection
            detector_buffer[detector_index] = iir_output*iir_output;
        
            if (detector_index < (LENVEDT - 1))
                {
                    detector_sum = detector_sum + detector_buffer[detector_index] - detector_buffer[detector_index+1];
                    detector_index++;
                }
            else
                {
                    detector_sum = detector_sum + detector_buffer[detector_index] - detector_buffer[0];
                    detector_index = 0;
                }

        } 
        else
            {
                detector_sum = detector_sum + detector_buffer[detector_index] - detector_buffer[0];
            }

    if (detector_sum > threshold)
        { 
            counter = 0;
            counter_temp = 0;
            start_flag = 1;
            TS = 0;
            PLL_init(0);
            PLL_init(1);
            PLL_init(2);
            holdover_index = 0;
        }

        //**********Detection END**********/

    else if (start_flag == 1)
\end{verbatim}
\begin{verbatim}
288 { 
289     input = iir_output; 
290     if (TS == 0) 
291     {
292         if ((counter < start_listen) && (counter >= start_slot))  // 0.1s wait 
293             {
294                 if (counter < 88) 
295                     {
296                         detector_buffer[counter] = 0; 
297                     }
298                 counter++; 
299             }
300         else if ((counter >= start_listen) && (counter < shift_time))  // 0.34s PLL, PFD 
301             {
302                 PLL(input, 0, 0); // first PLL, PFD 
303                 counter++; 
304             }
305         else if ((counter >= shift_time) && (counter < stop_listen)) 
306             {
307                 PLL(input, 0, 1); // first PLL, MUL 
308                 counter++; 
309             }
310         else if (counter == stop_listen) 
311             {
312             running_avg[0] = running_sum[0] * invM; 
313             VCO(0, running_avg[0], 0); 
314             counter++; 
315         }
316         else if ((counter > stop_listen) && (counter < end_slot)) 
317             {
318             VCO(0, running_avg[0], 0); // update first PLL holdover 
319             counter++; 
320         }
321         else if (counter == end_slot) 
322             {
323             VCO(0, running_avg[0], 0); // update first PLL holdover 
324             TS++; 
\end{verbatim}
counter = 0;
}

else if (TS == 1)
{
    if (node == 1)
    {
        if ((counter>=start_play) && (counter <= stop_play))
        {
            VCO(0,running_avg[0],0); // update first PLL holdover
            output = PLL_play(0);
        }
        if (counter < m)
        {
            holdover[0][counter] = 0;
            holdover[1][counter] = 0;
            holdover[2][counter] = 0;
        }
        data.channel[0] = (short)(output*32000);
        data.channel[1] = data.channel[0];
        counter++;
    }
    else if ((counter > stop_play) && (counter < end_slot))
    {
        counter++;
    }
    else if (counter == end_slot)
    {
        counter = 0;
        TS++;
    }
}
else if (node == 2)
{
    if ((counter < start_listen) && (counter >= start_slot)) // 0.1s wait
    {
        VCO(0, running_avg[0], 0); // update first PLL holdover
        counter++;
    }
    else if ((counter >= start_listen) && (counter < shift_time)) // 0.5s PLL
    {
        VCO(0, running_avg[0], 0); // update first PLL holdover
        PLL(input, 1, 0); // second PLL
        counter++;
    }
    else if ((counter >= shift_time) && (counter < stop_listen))
    {
        VCO(0, running_avg[0], 0); // update first PLL holdover
        PLL(input, 1, 1); // first PLL, MUL
        counter++;
    }
    else if (counter == stop_listen)
    {
        VCO(1, running_avg[1], 0);
        VCO(0, running_avg[0], 0); // update first PLL holdover
        counter++;
    }
    else if ((counter > stop_listen) && (counter < end_slot))
    {
        VCO(1, running_avg[1], 0);
        VCO(0, running_avg[0], 0); // update first PLL holdover
        counter ++;
    }
    else if (counter == end_slot)
    {
        VCO(1, running_avg[1], 0);
        VCO(0, running_avg[0], 0); // update first PLL holdover
        TS++;
counter = 0;
}
}

else if (node == 3)
{
if ((counter < end_slot) && (counter >= start_slot))
{
VCO(0, running_avg[0], 0); // update first PLL holdover
counter++;
}
else if (counter == end_slot)
{
VCO(0, running_avg[0], 0); // update
counter = 0;
TS++;   
}
}

else if (TS == 2)
{
if (node == 1)
{
if ((counter < end_slot) && (counter >= start_slot))
{
counter++;
}
else if (counter == end_slot)
{
counter = 0;
TS++;   
}
else if (node == 2)
{
if ((counter >= start_play) && (counter <= stop_play))
{


if (counter < m)
{
    holdover[0][counter] = 0;
    holdover[1][counter] = 0;
    holdover[2][counter] = 0;
}

data.channel[0] = (short)(output+32000);
data.channel[1] = data.channel[0];
counter++;
MCBSP_write(DSK6713_AIC23_DATAHANDLE, data.combo);

else if ((counter > stop_play) && (counter < end_slot))
{
    counter++;
    VCO(1,running_avg[1],0);
    VCO(0,running_avg[0],0); //update first PLL holdover
}

else if (counter == end_slot)
{
    counter = 0;
    TS++;
    VCO(1,running_avg[1],0);
    VCO(0,running_avg[0],0); //update first PLL holdover
}
else if (node == 3)
{
    if ((counter < start_listen)&&(counter>=start_slot)) //0.1s wait
    {
        VCO(0,running_avg[0],0); //update first PLL holdover
        counter++;
    }
}
else if ((counter >= start_listen) && (counter < shift_time)) //0.5 s PLL
{
  VCO(0, running_avg[0], 0); //update first PLL holdover
  PLL(input, 1, 0); //second PLL
  counter++;
}
else if ((counter >= shift_time) && (counter < stop_listen))
{
  VCO(0, running_avg[0], 0); //update first PLL holdover
  PLL(input, 1, 1); //first PLL, MUL
  counter++;
}
else if (counter == stop_listen)
{
  VCO(0, running_avg[0], 0); //update first PLL holdover
  running_avg[1] = running_sum[1]*invM;
  VCO(1, running_avg[1], 0);
  counter++;
}
else if ((counter > stop_listen) && (counter < end_slot))
{
  VCO(0, running_avg[0], 0); //update first PLL holdover
  VCO(1, running_avg[1], 0); //update second PLL holdover
  counter ++;
}
else if (counter == end_slot)
{
  VCO(0, running_avg[0], 0); //update first PLL holdover
  VCO(1, running_avg[1], 0); //update second PLL holdover
  TS++;
  counter = 0;
}
else if (TS == 3)
{

if (node == 1) {
  if ((counter < end_slot) && (counter >= start_slot))
    { counter++; }
  else if (counter == end_slot)
    { counter = 0; TS++; }
}
else if (node == 2) {
  if ((counter < start_listen)&& (counter>=start_slot)) //0.1s wait
    { VCO(0,running_avg[0],0); //update first PLL holdover
      VCO(1,running_avg[1],0);
      counter++; }
  else if ((counter >= start_listen) && (counter < shift_time)) //0.5s PLL
    { VCO(0,running_avg[0],0); //update first PLL holdover
      VCO(1,running_avg[1],0);
      PLL(input,2,0); //third PLL
      counter++; }
  else if ((counter >= shift_time) && (counter < stop_listen))
    { VCO(0,running_avg[0],0); //update first PLL holdover
      VCO(1,running_avg[1],0);
      PLL(input,2,1); //first PLL, MUL
      counter++; }
  else if (counter == stop_listen)
    { VCO(0,running_avg[0],0); //update first PLL holdover

VCO(1, running_avg[1], 0);
VCO(2, running_avg[2], 0);
counter++;
} else if ((counter > stop_listen) && (counter < end_slot)) {
VCO(0, running_avg[0], 0); // update first PLL holdover
VCO(1, running_avg[1], 0); // update second PLL holdover
VCO(2, running_avg[2], 0);
counter ++;
} else if (counter == end_slot) {
VCO(0, running_avg[0], 0); // update first PLL holdover
VCO(1, running_avg[1], 0); // update second PLL holdover
VCO(2, running_avg[2], 0);
TS++;
counter = 0;
} else if (node == 3) {
if ((counter >= start_play) && (counter <= stop_play)) {
VCO(1, running_avg[1], 0);
VCO(0, running_avg[0], 0); // update first PLL holdover
output = PLL_play(0);
if (counter < m) {
holdover[0][counter] = 0;
holdover[1][counter] = 0;
holdover[2][counter] = 0;
}
data.channel[0] = (short)(output * 32000);
data.channel[1] = data.channel[0];
counter++;  
MCBSP_write(DSK6713_AIC23_DATAHANDLE, data.combo); //Write L+R channels
}

} else if ((counter > stop_play) && (counter < end_slot))
{
    counter++;  
    VCO(1,running_avg[1],0);
    VCO(0,running_avg[0],0); //update first PLL holdover
}

} else if (counter == end_slot)
{
    counter = 0;
    TS++;
    VCO(1,running_avg[1],0);
    VCO(0,running_avg[0],0); //update first PLL holdover
}

} else if (TS == 4)
{
    if (node == 1)
    {
        if ((counter < start_listen) && (counter >= start_slot)) //0.1s wait
            counter++;  
    }

} else if ((counter >= start_listen) && (counter < shift_time)) //0.5s PLL
{
    PLL(input,1,0); //second PLL
    counter++;  
}

} else if ((counter >= shift_time) && (counter < stop_listen))
{
    PLL(input,1,1); //first PLL, MUL
    counter++;  
}

} else if (counter == stop_listen)


```c
{ 
  running_avg[1] = running_sum[1]*invM;
  VCO(1, running_avg[1], 0);
  counter++;
}
else if ((counter > stop_listen) && (counter < end_slot))
{
  VCO(1, running_avg[1], 0); //update second PLL holdover
  counter ++;
}
else if (counter == end_slot)
{
  VCO(1, running_avg[1], 0); //update second PLL holdover
  TS++;
  counter = 0;
}
else if (node == 2)
{
  if ((counter >= start_play) && (counter <= stop_play))
  {
    VCO(1, running_avg[1], 0);
    VCO(0, running_avg[0], 0); //update first PLL holdover
    VCO(2, running_avg[2], 0);
    output = PLL_play(2);
    if (counter < m)
    {
      holdover[0][counter] = 0;
      holdover[1][counter] = 0;
      holdover[2][counter] = 0;
    }
    data.channel[0] = (short)(output*32000);
    data.channel[1] = data.channel[0];
    counter++;
  }
}
```
MCBSP_write(DSK6713_AIC23_DATAHANDLE, datacombo);

//Write L+R channels
}

else if ((counter > stopplay) && (counter < endslot))
{
    counter++;
    VCO(1, running_avg[1], 0);
    VCO(0, running_avg[0], 0);  //update first PLL holdover
    VCO(2, running_avg[2], 0);
}

else if (counter == end_slot)
{
    counter = 0;
    TS++;
    VCO(1, running_avg[1], 0);
    VCO(0, running_avg[0], 0);  //update first PLL holdover
    VCO(2, running_avg[2], 0);
}

else if (node == 3)
{
    if ((counter < end_slot) && (counter >= start_slot))
    {
        counter++;
    }
    else if (counter == end_slot)
    {
        VCO(1, running_avg[1], 0);
        counter = 0;
        TS++;
    }
}

else if (TS == 5)
{
    if (node != 2)
if (counter >= start_slot_last) && (counter < start_play_last))
{
    VCO(1, running_avg[1], 0);
    counter++;
}
else if ((counter >= start_play_last) && (counter <= stop_play_last))
{
    VCO(1, running_avg[1], 0);
    output = PLL_play(1);

data.channel[0] = (short)(output*32000);
data.channel[1] = data.channel[0];
    counter++;
    MCBSP_write(DSK6713_AIC23_DATAHANDLE, data.combo);
    //Write L+R channels
}
else if ((counter > stop_play_last) && (counter < end_slot_last))
{
    counter++;
}
else if (counter == end_slot_last)
{
    PLL_init(0);
    PLL_init(1);
    PLL_init(2);
    TS = 0;
    start_flag = 0;
    counter = 0;
    detector_index = 0;
    detector_sum = 0;
    iir_buffer[0] = 0;
    iir_buffer[1] = 0;
    iir_buffer[2] = 0;
    w_vco_m = 0;
    phi_vco_m = 0;
}
if ((counter >= start_slot_last) && (counter < start_play_last))
{
    VCO(1, running_avg[1], 0); //update second PLL holdover
    VCO(0, running_avg[0], 0); //update first PLL holdover
    VCO(2, running_avg[2], 0);
    counter++;
}
else if ((counter >= start_play_last) && (counter <= stop_play_last))
{
    VCO(1, running_avg[1], 0); //update second PLL holdover
    VCO(0, running_avg[0], 0); //update first PLL holdover
    VCO(2, running_avg[2], 0);
    output = PLL_play(3); //for the middle node in last slot
    if (counter < m)
    {
        holdover[0][counter] = 0;
        holdover[1][counter] = 0;
        holdover[2][counter] = 0;
    }
    data.channel[0] = (short)(output*32000);
    data.channel[1] = data.channel[0];
    counter++;
}
else if ((counter > stop_play_last) && (counter < end_slot_last))
{
    counter++;
}
else if (counter == end_slot_last)
{
    PLL_init(0);
    PLL_init(1);
    PLL_init(2);
TS = 0;
start_flag = 0;
counter = 0;
detector_index = 0;
detector_sum = 0;
iir_buffer[0] = 0;
iir_buffer[1] = 0;
iir_buffer[2] = 0;
w_vco_m = 0;
phi_vco_m = 0;
}
}
}
}
}
}
}
}
}
}
}
}
}
}
}

void PLL(double input, unsigned short PLLnum, unsigned int PhaseDetector)
{
double up1 = 0, up2 = 0;

unsigned int index = holdover_index[PLLnum];

/***************Phase detector*******************/
if (PhaseDetector == 0)
/***************PFD***************************/
{
    if (input > maxScale[PLLnum])
        maxScale[PLLnum] = input;
    if (input >= 0)
        sig_in[PLLnum][0] = 1;
    else if (input < 0)
        sig_in[PLLnum][0] = 0;

    if (vco_out_sin[PLLnum][0] >= 0)
        vco_out_sin[PLLnum][0] = 1;
    else vco_out_sin[PLLnum][0] = 0;
up1 = (!sig_in[PLLnum][1]) & & sig_in[PLLnum][0];
up2 = (!vco_out_sin[PLLnum][1]) & & vco_out_sin[PLLnum][0];

status[PLLnum][0] = (sig_in[PLLnum][1] != 1) & & (up1 && (!up2));
status[PLLnum][1] = (sig_in[PLLnum][1] != -1) & & ((!up1) && up2);
status[PLLnum][2] = (!status[PLLnum][0]) && (!status[PLLnum][1]);

if (status[PLLnum][0])
pd_out[PLLnum][0] = pd_out[PLLnum][1] + twopi;
else if (status[PLLnum][1])
pd_out[PLLnum][0] = pd_out[PLLnum][1] - twopi;
else if (status[PLLnum][2])
pd_out[PLLnum][0] = pd_out[PLLnum][1];
else

Cd[PLLnum] = 2*Kd/maxScale[PLLnum];
pd_out[PLLnum][0] = Cd[PLLnum]*input*vco_out*cos[PLLnum][0];


// Update running_sum
holdover[PLLnum][index] = uf[PLLnum][0];

if (index < (m−1))
{
running_sum[PLLnum] = running_sum[PLLnum] + holdover[PLLnum][index] -
holdover[PLLnum][index+1];
holdover[index][PLLnum]++;}
```c
running_sum[PLLnum] = running_sum[PLLnum] + holdover[PLLnum][index] - holdover[PLLnum][0];
holdover_index[PLLnum] = 0;
}

VCO(PLLnum, uf[PLLnum][0], 0);

// exchange variable
sig_in[PLLnum][1] = sig_in[PLLnum][0];
pd_out[PLLnum][2] = pd_out[PLLnum][1];
pd_out[PLLnum][1] = pd_out[PLLnum][0];
uf[PLLnum][2] = uf[PLLnum][1];
uf[PLLnum][1] = uf[PLLnum][0];
}

void VCO(unsigned short PLLnum, double holdover_value, unsigned short mode)
{
  if (mode == 0)
  {
    w_vco[PLLnum] = twopi * f_vco + K0 * holdover_value;
  }
  else if (mode == 1)
  {
    w_vco[PLLnum] = f_vco * twopi;
  }
  while (phi_vco[PLLnum] > twopi)
  while (phi_vco[PLLnum] < 0)
    phi_vco[PLLnum] = phi_vco[PLLnum] + twopi;
  vco_out_sin[PLLnum][1] = vco_out_sin[PLLnum][0];
```
```c
void PLL_init(unsigned short PLLnum)
{
    w_vco[PLLnum] = 0;
    maxScale[PLLnum] = -0.1;
    Cd[PLLnum] = 1;
    holdover_index[PLLnum] = 0;
    running_sum[PLLnum] = 0;
    phi_vco[PLLnum] = 0;
    running_avg[PLLnum] = 0;
    vco_out[PLLnum][0] = 0;
    vco_out[PLLnum][1] = 0;
    vco_out_sin[PLLnum][0] = 0;
    vco_out_sin[PLLnum][1] = 0;
    vco_out_cos[PLLnum][0] = 0;
    vco_out_cos[PLLnum][1] = 0;
    sig_in[PLLnum][0] = 0;
    sig_in[PLLnum][1] = 0;
    pd_out[PLLnum][0] = 0;
    pd_out[PLLnum][1] = 0;
    pd_out[PLLnum][2] = 0;
    uf[PLLnum][0] = 0;
    uf[PLLnum][1] = 0;
    uf[PLLnum][2] = 0;
    status[PLLnum][0] = 0;
    status[PLLnum][1] = 0;
    status[PLLnum][2] = 0;
}

double PLL_play(unsigned short PLLnum)
```
{ 
    double output;

    if (PLLnum != 3) {
        output = sin(phi_vco[PLLnum]);
    }

    else {
        w_vco_m = w_vco[1] + w_vco[2] - w_vco[0];
        while (phi_vco_m > twopi) 
            phi_vco_m = phi_vco_m - twopi;
        while (phi_vco_m < 0) 
            phi_vco_m = phi_vco_m + twopi;
        output = sin(phi_vco_m);
    }

    return output;
}
Appendix D

Maximum likelihood estimation

Matlab source code

The following MATLAB code is used to analyze the recordings in *.WAV format by using the MLE estimation described in [18]. By using this code, the amplitude estimate in each timeslot can be calculated and thus the power ratio $\rho$ of the beamformer can be obtained.
clear all;
clc;

% USER PARAMETERS

M = 2; \%number of nodes
nfft = 2^26; \%number of points in estimator FFT
tvec = [0:99]*7+0.73; \%start time of first signal in each trial
D = 1.25; \%time difference between successive signals
T = 0.2; \%duration of signal used for estimates
j = sqrt(-1);

format compact
[y,fs] = wavread('cut1.wav');
y = y(:,1);

ahat = zeros(2*M,length(tvec));
omegahat = zeros(2*M,length(tvec));
thetahat = zeros(2*M,length(tvec));
theta_ext = zeros(1,length(tvec));
deltatheta = zeros(1,length(tvec));
delta_omega = zeros(1,length(tvec));

tic
for i=1:length(tvec), \%test index
   for k=1:2*M, \%signal index in each test
      t0 = tvec(i)+(k-1)*D;
      if(k == 4)
         t0 = t0+.05;
      end
      startindex = floor(t0*fs);
      endindex = ceil((t0+T)*fs);
      s = y(startindex:endindex);
      N = endindex-startindex+1;
      A = fft(s,nfft)/N; \%do oversampled fft
      [ahat(k,i),index] = max(2*abs(A(1:nfft/2))); \%get amplitude estimate
      omegahat(k,i) = (index-1)/nfft*fs*2*pi; \%get frequency estimate
      thetahat(k,i) = angle(exp(-j*omegahat(k,i)*t0)*A(index)); \%get phase estimate
ampratio = ahat(end,:) ./ (ahat(2,:) + ahat(3,:));

powratio = ampratio.^2;
Bibliography


