Real-Time Software-Defined-Radio Implementation of a Two Source Distributed Beamformer

by

James W. McGinley

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

December 2006

APPROVED:

Prof. Donald R. Brown, Major Thesis Advisor	Prof. Fred J. Looft, Head of ECE Department
Prof. John McNeill, Thesis Committee Member	Prof. Richard Vaz, Thesis Committee Member

Abstract

This thesis describes a real-time software-defined-radio implementation of a two source distributed beamformer. The technique in this thesis can be used to synchronize the carriers of two single antenna wireless transmitters (i.e. "sources") with independent local clocks so that their bandpass transmissions arrive in-phase at an intended receiver (i.e. "destination"). Synchronization is achieved via: (i) an unmodulated beacon transmitted by the destination to the sources and (ii) a pair of secondary unmodulated beacons between the sources. No explicit channel state information is exchanged between the sources and/or the destination. Using this method, it is possible to realize a two-source distributed beamformer that provides a reduction in overall transmit energy and increased security due to the directionality of the transmitted signal. System characterization results are provided along with experimental results for both time-invariant and time-varying channels. The experimental results in this thesis confirm the theoretical predictions and also provide explicit guidelines for a real-time implementation of a two-source distributed beamforming system.

Acknowledgments

I would like to express my gratitude to my advisor who provided me with guidance and insight throughout this entire process. My thanks are also to my parents, brother, girlfriend, and friends for always supporting me without limits. Lastly, I would like to thank my thesis committee members and WPI for providing me with the opportunity to have such an experience.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Transmit beamforming [2] is a technique where an array of antennas is used to simultaneously transmit multiple copies of a bandpass transmission, each with a particular phase, in order to control the directivity of the signal. By controlling the phase of the signal at each antenna element in an antenna array, a gain in signal strength can be achieved in one or more desired directions and nulls can be steered to minimize radiation in undesired directions.

Transmit beamforming is an attractive technique for many applications due to its resource efficiency and reduced susceptibility to capture by unintended receivers. The requirement of an antenna array, however, is a physical constraint that precludes the use of beamforming in scenarios where the transmitter must have small size, e.g. cellular handsets. Recently, researchers have considered this problem in the context of multiuser communication systems in which there are multiple single-antenna transmitters. In these types of systems, transmitters can (with appropriate hardware and protocols) form "partnerships" to pool their antenna resources to form a *distributed beamformer*. If the transmitters can be appropriately synchronized, the distributed beamformer can offer many of the gains of "real" antenna arrays to

multiuser communication systems with single-antenna sources.

This thesis considers the problem of how to synchronize single-antenna transmitters in order to facilitate distributed beamforming. Each transmitter in the system is assumed to have an independent clock. Distributed beamforming is possible only if the sources can synchronize the frequency of their carriers *and* adjust the phase of their transmissions so that the signal is steered in a desired direction.

Traditional network synchronization approaches including "mutual synchronization" [3], "master-slave" [4], and hybrid approaches [5], have been shown to be effective at clock synchronization but do not address the problem of phase alignment at the destination. Frequency and phase synchronization for distributed beamforming was considered in [6] where coherent combining at an intended destination is achieved through a master synchronization beacon and precise placement of both the source and destination nodes in order to equalize all round-trip propagation times. Mobility is not permitted in this system.

A distributed beamforming synchronization scheme was also proposed in [7] where a beacon is used to measure round-trip phase delays between each transmitting node and the destination. The destination estimates and quantizes these phase delays and transmits them to the appropriate nodes for local phase pre-compensation. While this system does allow for some node mobility, the amount of mobility is restricted by the time required to estimate, quantize, deliver, and implement the phase pre-compensation estimates.

Recently, a new approach to carrier synchronization for a two-source distributed beamformer was proposed in [1]. The synchronization technique described in [1] requires the destination to transmit a master beacon signal and the sources to each use a pair of phase locked loops to synchronize to the master beacon as well as a secondary beacon signal transmitted by the other source. This method was shown

to offer several advantages with respect to [6] and [8] including not requiring explicit estimation, exchange, and quantization of channel state information as well as successful operation in systems with high rates of source and/or destination mobility.

While [1] analyzed the performance of this two-source distributed beamformer in a variety of channel models, it did not provide implementation details beyond certain guidelines for the design of the PLLs to avoid phase ambiguity. This thesis describes one potential real-time implementation of this distributed beamforming technique using the software-defined-radio paradigm. This technique is implemented, characterized and analyzed using TMS320C6713 DSK boards to create source nodes as well as channels with and without mobility to confirm the analytical results from [1].

Chapter 2

Background

This chapter provides the necessary background knowledge for the information presented in this thesis. This includes concepts such as beamforming, phase-locked loops, software radio and real-time digital signal processing.

2.1 Conventional Beamforming

Conventional beamforming as described in [2] is commonly used in wireless communication systems that contain antenna arrays. It is a technique that modifies the individual phases of the transmissions from two or more antennas in an array so that they align at a desired destination. Using this method it is possible to produce a directional radiation pattern from a given antenna array instead of an omnidirectional radiation pattern that occurs from certain types of antennas (i.e. cellular telephone towers).

Directionality appears because the transmission from each 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 also in the opposite direction. This type of interference increases the overall signal amplitude in

comparison to a single transmission. In all other directions, the individual antenna transmissions cause destructive interference, which decreases the signal amplitude. Figure 2.1 shows a block diagram representation of a conventional beamformer.

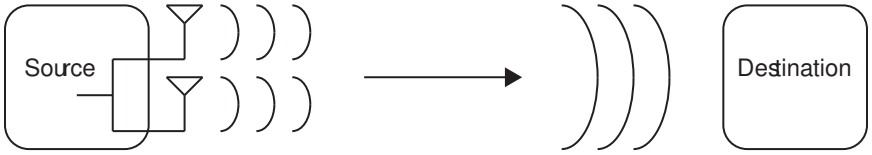

Figure 2.1: Block diagram of a conventional beamformer.

Conventional beamforming is an attractive technique in many applications. It allows for spatial diversity gains, which lead to advantages such as increased energy efficiency and decreased outage probability as shown in [2]. In [9], conventional beamforming is shown to decrease the overall power needed in a given system while maintaining the same coverage area as a single antenna by a factor of $N^{1-n/2}$ where N is the number of antennas and n is the path loss exponent of a simple power law path loss model. The power of an individual antenna is then reduced by a factor of $N^{n/2}$. The overall energy is reduced because the line-of-sight channel is more frequently available and less energy is spent in other channels.

2.2 Overview of Distributed Beamforming

Distributed beamforming as described in [1] and [2] works off of the same principles as conventional beamforming, but does not assume that the transmit antennas are physically located together. Thus, sources that only contain a single antenna (i.e. cellular telephones) could be used together to form a virtual antenna array that would allow for benefits similar to conventional beamforming. Figure 2.2 shows a block diagram representation of a distributed beamformer.

The same overall energy reduction as conventional beamforming can be achieved

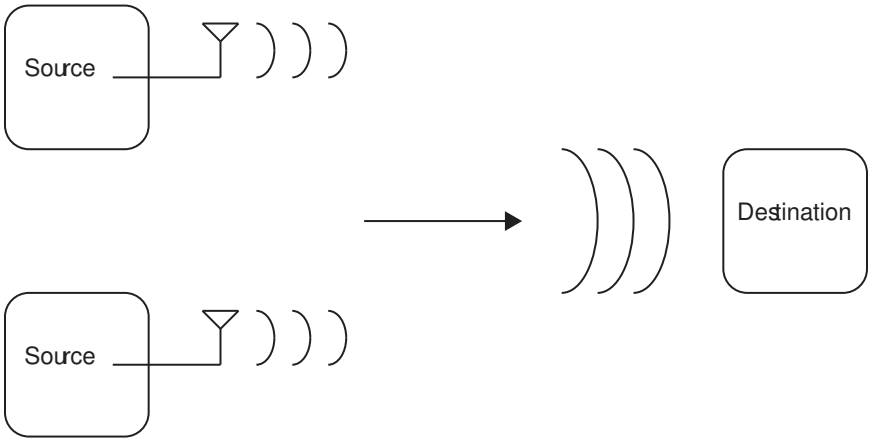

Figure 2.2: Block diagram of a distributed beamformer.

in distributed beamforming as described in [9]. The challenge when using distributed beamforming is in how to force the transmissions from different sources to align their phases at a given destination. The system provided in this thesis uses phase-locked-loops to achieve two source carrier synchronization, which then allows for distributed beamforming of the two sources to occur.

2.3 Phase-Locked-Loops

A phase-locked-loop (PLL) is a device that is used to internally synchronize to the precise frequency and phase of an incoming signal. The input signal is typically a band-limited sinusoidal with an unknown frequency and phase. The major components of a PLL are: a phase detector, a loop filter, and a voltage controlled oscillator (VCO) as shown in Figure 2.3.

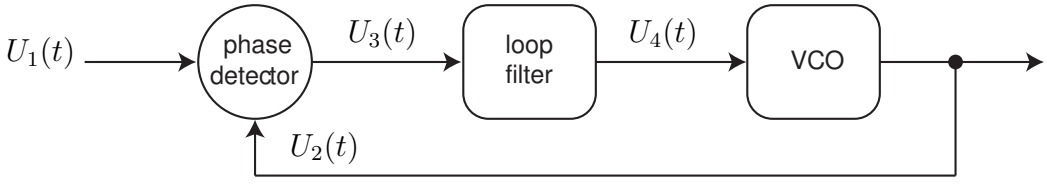

Figure 2.3: Block diagram of a generic PLL.

2.3.1 Phase Detector

The phase detector of a PLL can be defined in several ways. The four main types of phase detectors are: the linear (four-quadrant) multiplier, EXOR gate, edge triggered JK-flipflop, and phase-frequecy detector (PFD). Each detector type performs an operation on the phase difference between the input signal $U_1(t)$ and the feedback signal $U_2(t)$. This thesis considers PLLs that operate using the linear multiplier phase detector.

The linear multiplier phase detector is used solely in linear phase-locked loops (LPLLs). The output of this type of phase detector is simply the product of the two input signals. Thus, the input to the LPLL is

$$U_1(t) = \cos(\Phi_1) \tag{2.1}$$

and the feedback signal is

$$U_2(t) = \cos(\Phi_2) \tag{2.2}$$

where

$$\Phi_1 = \omega_1 t + \phi_1 \tag{2.3}$$

$$\Phi_2 = \omega_2 t + \phi_2 \tag{2.4}$$

Thus, the generalized output equation for the multiplier phase detector is

$$U_3(t) = K_d U_1(t) U_2(t) = \frac{K_d}{2}[\cos(\Phi_1 + \Phi_2) + \cos(\Phi_1 - \Phi_2)] \tag{2.5}$$

where K_d is the gain of the phase detector with units in [rad/V]. When

$$\omega_1 = \omega_2 \tag{2.6}$$

and

$$\phi_1 = \phi_2 + \frac{\pi}{2} \tag{2.7}$$

we say that the LPLL is in a "locked state" and the output of the multiplier phase detector is

$$U_3(t) = \frac{K_d}{2}[1 + \cos(\Phi_1 + \Phi_2)] \tag{2.8}$$

where $U_3(t)$ is also the input to the loop filter.

As seen in (2.7), there is a $\frac{\pi}{2}$ phase offset between the two input signals when the LPLL is in a locked state. This phase offset is due to the cosine function being equal to zero when the phase difference between the two signals is $\pm\frac{\pi}{2}$ as seen in Figure 2.4.

The location where the phase difference is $\frac{\pi}{2}$ is an unstable fixed point whereas the $-\frac{\pi}{2}$ location is a stable fixed point. When the output of the of the phase detector is positive, the VCO speeds up, which causes the phase difference to decrease. When the output is negative, the reverse occurs, thus the phase detector output is as seen in (2.8) when in a "locked state."

2.3.2 Loop Filter

The loop filter of a PLL is typically a low pass filter with a large (ideally infinite) DC gain. For this thesis, a second order active proportional integral (PI) loop filter of the form

$$F(s) = \frac{1 + s(\tau_2 + \tau_3)}{s\tau_1(1 + s\tau_3)} \tag{2.9}$$

is used, where τ_1, τ_2, and τ_3 are chosen to achieve a certain loop bandwidth and phase margin. The active PI filter has a pole at $s = 0$, thus it acts like an integrator, which in theory should provide infinite DC gain. This is necessary to insure that the

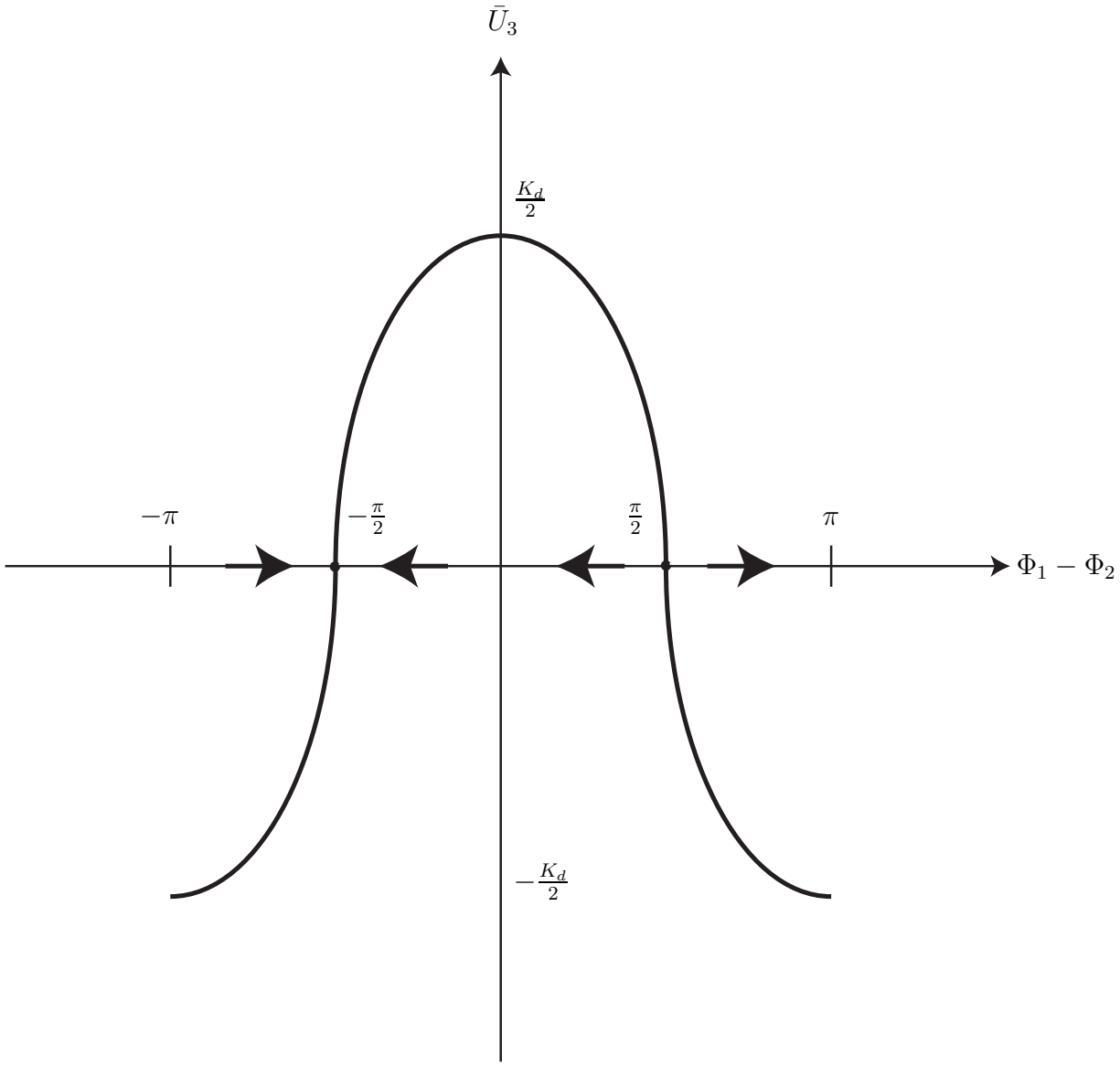

Figure 2.4: Graphical representation of the average phase detector output voltage versus the phase difference of the two input signals.

PLL eliminates the phase error between the input signal and the internal feedback signal.

The loop filter also suppresses the high frequency term in (2.8), thus approximately leaving only the DC term to control the VCO such that

$$U_4(t) \approx \frac{K_d}{2} \tag{2.10}$$

when the PLL is in a "locked state." This means that the input signal to the VCO is constant, which allows the PLL to increment its phase to continue operating in a "locked state."

2.3.3 Voltage Controlled Oscillator (VCO)

A voltage controlled oscillator (VCO) is an oscillator that is controlled by its input voltage. A positive/negative control voltage to the VCO indicates that the input frequency is higher/lower than that of the center frequency of the PLL. The VCO changes its radian frequency in accordance to

$$\omega_{VCO}(t) = \omega_{center} + K_0 U_4(t) \tag{2.11}$$

where K_0 is the gain and ω_{center} is the center frequency of the VCO. This is how the PLL increases or decreases its phase in accordance to the input signal of the PLL.

2.3.4 Software-Defined Frequency Synthesis Phase-Locked Loop (FS-PLL)

Software-defined frequency synthesis phase-locked loops (FS-PLLs) are different in two ways from typical PLLs. The FS-PLLs in this thesis are implemented using pro-

grammable digital signal processors (DSP). Using this method we are able to achieve different PLL designs without changing single purpose hardware components. Also, frequency synthesis is achieved, which refers to a shift in frequency from the input to the output of the PLL, while still remaining in a "locked state."

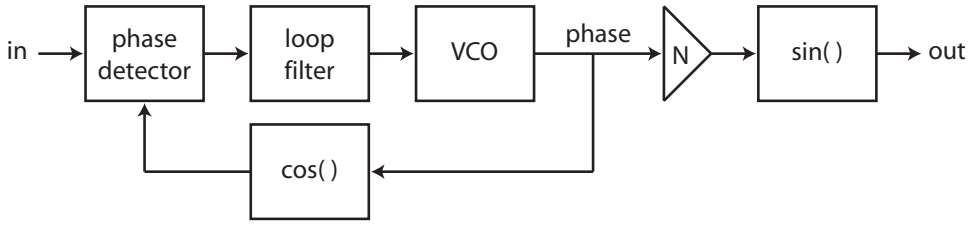

Figure 2.5: Block diagram of a frequency synthesis phase-locked loop.

Figure 2.5 shows a block diagram of a FS-PLL, which now includes a few new components in comparison to Figure 2.3. The frequency synthesis multiplier takes the phase from the output of the VCO and multiplies it by an integer value, which is then converted from a phase to a sinusoidal signal to be output. The phase from the VCO is also converted to a cosine signal, which is then compared to the input signal of the PLL at the phase detector. By using the sine of the VCO phase as the output of the PLL and the cosine of the VCO phase as the feedback signal, we are able to eliminate the $\frac{\pi}{2}$ phase shift induced by the phase detector.

2.4 Real-Time Digital Signal Processing (DSP)

As technology continues to progress, the presence of digital signal processors (DSPs) in everyday life is increasingly apparent. DSPs are used in devices such as cellular telephones, global positioning systems, and computers. These types of devices are constantly receiving, analyzing, and modifying data in real-time to perform their given task. In this thesis, five identical DSPs are used to perform completely different tasks from one another. It is possible to implement the system presented in this

thesis using single purpose hardware, but with decreased versatility to test different designs within seconds.

2.4.1 TMS320C6713 DSK

The system described by this thesis is accomplished using five TMS30C6713 Digital Starter Kits (DSKs) from Spectrum Digital. Each of the DSKs has a TMS320C6713 DSP chip operating at 225 MHz, a TLV320AIC23B codec, 16MB SDRAM, four DIP switches, four LEDs, and four 3.5 mm audio jacks (microphone, line-in, speaker, line-out). For more information on the TMS306713 DSK please see [10].

2.4.2 Code Composer Studio Integrated Developement Environment (IDE) 3.1

The programming of the TMS320C6713 DSK is achieved using the Code Composer Studio (CCS) Integrated Development Environment (IDE) 3.1. The CCS IDE allows a user to connect, program in C, and run the TMS320C6713 DSK through a graphical user interface. Also, a user is able to view the memory contents of the TMS30C6713 and profile the execution time for pieces of their code all in real-time. This is the means by which the system presented in this thesis is implemented in conjunction with the TMS320C6713 DSK.

Chapter 3

System Model

This thesis considers the two-source, one-destination system model shown in Figure 3.1. The destination (D) and the two sources (S_1 and S_2) are assumed to each have independent local clocks and a single antenna. The channels are modeled as linear (possibly time varying) systems with $g_{ij}(t,\tau)$ denoting the response of the channel at time t to an impulse at time $t-\tau$. The impulse response of each channel in the system is assumed to be reciprocal in the forward and reverse directions.

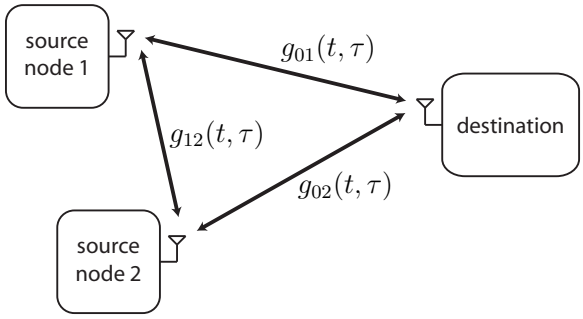

Figure 3.1: Two-source one-destination system model.

While the channels in Figure 3.1 allow for multipath and/or time varying behavior, the intuition behind the carrier synchronization technique described in [1] is best exposed under the temporary assumption that all of the channels in the system

are single-path and time-invariant, so

$$g_{ij}(t) = \delta(t - \tau_{ij}), \quad ij \in \{01, 02, 12\} \tag{3.1}$$

In this case, the total propagation time for the circuit $\mathsf{D} \rightarrow \mathsf{S}_1 \rightarrow \mathsf{S}_2 \rightarrow \mathsf{D}$ can be calculated as

$$\tau_{tot} = \tau_{01} + \tau_{12} + \tau_{02} \tag{3.2}$$

If the destination were to transmit a signal $x(t)$ to S_1, then S_1 relayed this signal to S_2, and S_2 subsequently relayed this signal back to the destination, the signal received at the destination from S_2 could be expressed as

$$r(t) = x(t - \tau_{tot} - \Delta_1 - \Delta_2) \tag{3.3}$$

where Δ_i is the relaying latency of the i^{th} source node. Similarly, since the signal $x(t)$ transmitted by the destination is also received by S_2, it can be relayed by S_2 to S_1, and subsequently relayed by S_1 back to the destination. Recognizing that the total propagation time for the circuit $\mathsf{D} \rightarrow \mathsf{S}_2 \rightarrow \mathsf{S}_1 \rightarrow \mathsf{D}$ is also τ_{tot}, the signal received by the destination from S_1 will be identical to (3.3).

The equivalent round-trip propagation times for both circuits is the key feature of the carrier synchronization technique described in [1]. The destination begins the synchronization process by transmitting a continuous sinusoidal master beacon at frequency ω_0 [rad/s] to both source nodes. The i^{th} source node, as seen in Figure 3.2, receives the continuous master beacon and employs a frequency-synthesis[1] PLL [11] (denoted as FS-PLL $i1$ and tuned to the master beacon frequency ω_0) to synthesize

[1]Frequency synthesis is employed to avoid transmission and reception on the same frequency. This "half-duplex" constraint is commonly imposed due to the limitations of echo cancelers in wireless transceivers.

a secondary sinusoidal beacon that is phase locked to the master beacon. The secondary beacon is at frequency $\omega_i = N_{i1}\omega_0$ where N_{i1} is an integer frequency multiplier. Figure 2.5 shows a block diagram of a typical frequency synthesis PLL. The i^{th} source node simultaneously receives the secondary beacon transmitted by the j^{th} source node and employs a secondary frequency synthesis PLL (denoted as FS-PLL $i2$ and tuned to $\omega_j = N_{j1}\omega_0$). This synthesizes a carrier signal at frequency $\omega_c = N_{i2}\omega_j$ that is phase locked to the received secondary beacon signal. The phase-locked carriers of each source are then used to modulate the baseband signals for bandpass transmission of information to the destination. When the PLLs are locked, both sources' bandpass transmissions arrive with identical phase and coherently combine at the destination.

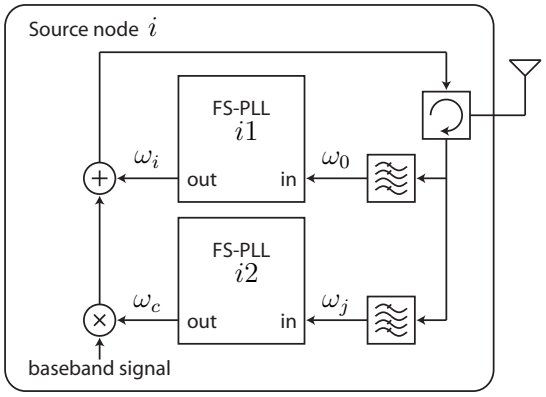

Figure 3.2: Block diagram of i^{th} source node in the distributed beamforming technique described in [1].

While [1] analyzes the performance of this two-source distributed beamformer in both single-path and multi-path channels, it does not provide implementation details beyond certain guidelines for the design of the PLLs to avoid phase ambiguity. The following sections of this thesis describe one potential real-time implementation of this distributed beamforming technique using the software-defined-radio paradigm.

Chapter 4

Implementation

The implementation described in this thesis is achieved by using five TMS320C6713 DSK boards. Three DSK boards are used to model the six channels and two DSK boards are used to model the two sources as shown in Figure 4.1.

The master beacon is simulated using a function generator while the receiver at the destination is an oscilloscope, which allows the user to see the two source nodes' transmissions at the destination. Each TMS320C6713 DSK board has one line-level stereo input and one line-level stereo output that are the means in which the DSK boards are connected to other DSK boards or components. The following sections describe the particular implementation details for the source nodes and channels.

4.1 Source Node Implementation

Each source is implemented using one DSK board that runs two 3^{rd} order software-defined FS-PLLs simultaneously at a sampling frequency (f_s) of 44.1 kHz. One FS-PLL is running on the left channel while the other FS-PLL is operating on the right channel. This choice of f_s allows for real-time execution of the FS-PLL code on the TMS320C6713 DSK while still being well above the Nyquist rate for the

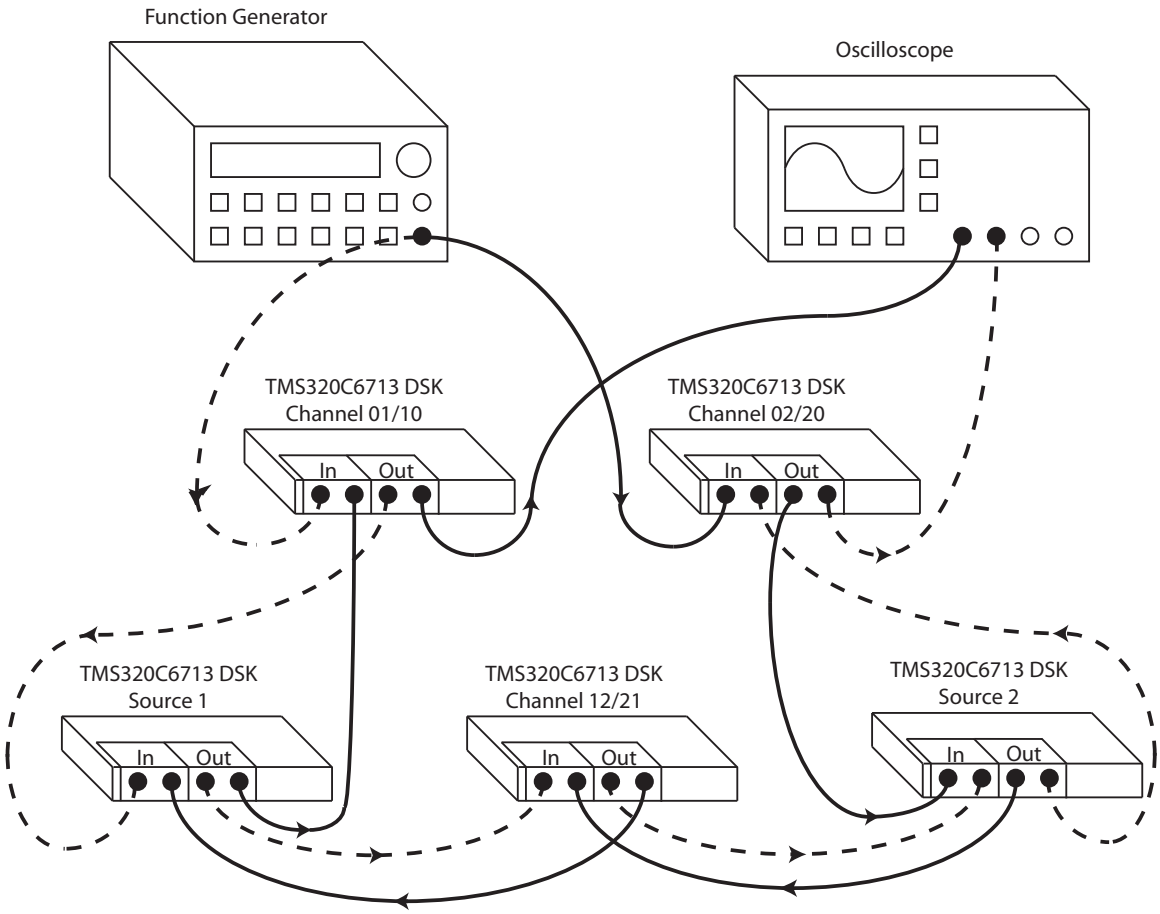

Figure 4.1: Implementation block diagram where the dotted and solid lines each represent a different signal wired path.

frequencies used to test and characterize the system.

A block diagram of the FS-PLL used in this implementation is shown in Figure 2.5 and consists of five major components: (i) a phase detector, (ii) a loop filter, (iii) a voltage controlled oscillator (VCO), (iv) a frequency multiplier and (v) two trigonometric functions. The FS-PLLs were designed by converting the analog linear PLL design described in [11] to C source code to run on the DSK boards.

4.1.1 Phase Detector Implementation

The phase detector we use is a frequency multiplier (type I) with unity gain. When the FS-PLL is locked, this type of phase detector creates a 90^o phase offset between the input to the FS-PLL and the output as discussed in Chapter 2.3.1. The offset can be easily eliminated in a software defined FS-PLL by using the cosine of the VCO phase as a feedback signal and the sine of the VCO phase as an output signal, which can be seen in Figure 2.5. In a programming language such as C, a unity gain type I phase detector can be realized by the multiplication of the input signal and the internal feedback signal as

```
1   FQout_left[n] = fleftchannel[n] * w2_left[n] * Kd;
2   FQout_right[n] = frightchannel[n] * w2_right[n] * Kd;
```

where `n` is the current sample index. The channel of the TMS320C6713 DSK that a given variable is associated with is denoted using `_left` or `_right` for the left and right channels respectivly. The variables `FQout_left[n]` and `FQout_right[n]` store the value of the output of the phase detector where `n` is again the current sample index. The current input signal is stored as `fleftchannel[n]` and `frightchannel[n]`, the phase detector gain as `Kd`, and the feedback signal as `w2_left[n]` and `w2_right[n]`. The two feedback variables are stored as double precision floating point values, while the rest are single precision floating point values.

4.1.2 Loop Filter Implementation

We use a second order low pass active PI loop filter of the form

$$F(s) = \frac{1 + s(\tau_2 + \tau_3)}{s\tau_1(1 + s\tau_3)} \tag{4.1}$$

where τ_1, τ_2, and τ_3 are chosen to achieve a 10 Hz loop bandwidth and a 55^o phase margin to mirror the RF design given in [1]. Using the bilinear z-transform

$$F(z) = F(s)|_{s=\frac{2(z-1)}{T(z+1)}} \tag{4.2}$$

where $T = \frac{1}{44100}$, we are able to create our discrete time loop filter from (4.1) . The loop filter is then implemented using single precision floating point variables as

```
1           if (n==0){
2           LPFout_left[n] = numd_left[0]*FQout_left[0] + numd_left[1]*
3           FQout_left[N-1] + numd_left[2]*FQout_left[N-2] - dend_left[1]*
4           LPFout_left[N-1] - dend_left[2]*LPFout_left[N-2];
5           LPFout_right[n] = numd_right[0]*FQout_right[0] +
6           numd_right[1]*FQout_right[N-1] + numd_right[2]*
7           FQout_right[N-2] - dend_right[1]*LPFout_right[N-1] -
8           dend_right[2]*LPFout_right[N-2];
9           }
10          if (n==1)
11          {
12          LPFout_left[n] = numd_left[0]*FQout_left[1] + numd_left[1]*
13          FQout_left[0] + numd_left[2]*FQout_left[N-1] - dend_left[1]*
14          LPFout_left[0] - dend_left[2]*LPFout_left[N-1];
15          LPFout_right[n] = numd_right[0]*FQout_right[1] +
16          numd_right[1]*FQout_right[0] + numd_right[2]*
17          FQout_right[N-1] - dend_right[1]*LPFout_right[0] -
18          dend_right[2]*LPFout_right[N-1];
19          }
20          if (n>=2)
21          {
22          LPFout_left[n] = numd_left[0]*FQout_left[n] + numd_left[1]*
23          FQout_left[n-1] + numd_left[2]*FQout_left[n-2] - dend_left[1]*
24          LPFout_left[n-1] - dend_left[2]*LPFout_left[n-2];
```

```
25          LPFout_right[n] = numd_right[0]*FQout_right[n] +
26          numd_right[1]*FQout_right[n-1] + numd_right[2]*
27          FQout_right[n-2] - dend_right[1]*LPFout_right[n-1] -
28          dend_right[2]*LPFout_right[n-2];
29          }
```

where `LPFout_left[n]` and `LPFout_right[n]` are the outputs of the loop filters for the current sample index. The numerator and denominator coefficients of the loop filter are stored in `numd` and `dend` respectively.

4.1.3 VCO Implementation

The VCO is controlled by the output of the loop filter $U_4(t)$; if the output of the loop filter is positive or negative, then the VCO increases or decreases its phase respectively. If the output is constant, then the VCO frequency does not change as we are then in a "lock" state where x and $\hat{x}$ are at the same frequency and 90^o out of phase. The VCO phase output (ϕ_{vco}) is governed by

$$f_{vco}(t) = f_0 + \frac{K_0 U_4(t)}{2\pi} \tag{4.3}$$

$$\phi_{vco} = \phi_{vco-previous} + \frac{2\pi f_{vco}}{f_s} \tag{4.4}$$

where K_0 is the VCO gain with units [rad/s $\cdot$ V], and f_0 is the free running frequency of the VCO in [Hz/s]. In (4.3) and (4.4), $\phi_{vco-previous}$ is the phase of the VCO that was calculated during the previous sampling period. Using this definition, the VCO is achieved using

```
1   f2_left = f0_left + (Ko*oneovertwopi) *
2   LPFout_left[n];
3   f2_right = f0_right + (Ko*oneovertwopi) *
4   LPFout_right[n];
5   if(n+1>N-1) {
```

```
6   vco_phase_left[0] = vco_phase_left[N-1] +
7   2*pi*(f2_left*invfs);
8   vco_phase_right[0] = vco_phase_right[N-1] +
9   2*pi*(f2_right*invfs);
10  while(vco_phase_left[0]>2*pi)
11          vco_phase_left[0] += -2*pi;
12  while(vco_phase_left[0]<0)
13          vco_phase_left[0] += 2*pi;
14  while(vco_phase_right[0]>2*pi)
15          vco_phase_right[0] += -2*pi;
16  while(vco_phase_right[0]<0)
17          vco_phase_right[0] += 2*pi;
18  w2_left[0] = sindp(vco_phase_left[0]+pi_half);
19  w2_right[0] = sindp(vco_phase_right[0]+pi_half);
20  w3_left[0] = -sindp(fmult_left*vco_phase_left[0]);
21  w3_right[0] = -sindp(fmult_right*vco_phase_right[0]); }
22  else {
23  vco_phase_left[n+1] = vco_phase_left[n] +
24  2*pi*(f2_left*invfs);
25  vco_phase_right[n+1] = vco_phase_right[n] +
26  2*pi*(f2_right*invfs);
27  while(vco_phase_left[n+1]>2*pi)
28          vco_phase_left[n+1] += -2*pi;
29  while(vco_phase_left[n+1]<0)
30          vco_phase_left[n+1] += 2*pi;
31  while(vco_phase_right[n+1]>2*pi)
32          vco_phase_right[n+1] += -2*pi;
33  while(vco_phase_right[n+1]<0)
34          vco_phase_right[n+1] += 2*pi;
```

where `invfs` and `oneovertwopi` are the inverses of the sampling frequency and the value of 2π respectively. Pre-calculating the inverse of these two values allows for faster execution time of the code in C because there is no requirement to compute the reciprocal in real-time. Lines 1-4 perform the operation seen in (4.3). The VCO sensitivity is denoted as `Ko`, VCO center frequencies as `f0_left` and `f0_right`, and low-pass filter outputs as `LPFout_left` and `LPFout_right`. The feedback signals for the left and right channels are stored as `w2_left` and `w2_right` while the output signals are stored as `w3_left` and `w3_right`. The multipliers used for frequency

synthesis are `fmult_left` and `fmult_right`. Lines 6-9 and 23-26 compute (4.4) while lines 10-17 and 27-34 wrap the phase of the VCO back into the range $[0, 2\pi]$ since the sine function is periodic on that interval. The VCO sensitivity, VCO center frequencies, low-pass filter outputs, and the frequency synthesis multipliers are stored as single precision floating point values, while the rest are stored as double precision floating point values.

4.1.4 FS-PLL Multiplier Implementation

The master beacon frequency for this implementation is chosen as 907 Hz which is approximately of equal wavelength in acoustics to the master beacon RF signal in [1] of 800 MHz. We use acoustic range frequencies due to the limitation of the hardware available. The wavelength λ of a signal at frequency f in Hertz can be calculated as

$$\lambda_{acoustic} = \frac{c}{f} \tag{4.5}$$

where c is 340 [m/s] for acoustic frequencies and 3*10^8 [m/s] for radio frequencies. For our implementation, f_0 is chosen based on the incoming frequency to a given FS-PLL as shown in Table 4.1, where f_0 is shown in Hertz.

FS-PLL	f_0 [Hz]
11	907
12	2721
21	907
22	1814

Table 4.1: Center frequencies (f_0) of each FS-PLL in Hertz.

Source 1 and source 2 have a frequency multiplier of 2 and 3 respectively, which is achieved by a multiplication of their respective VCO phase by the given multiplier

value, which is shown in Figure 2.5. This implementation allows for the master beacon, the two cross beacons, and the transmission back to the destination to operate at different frequencies to avoid interfering with one another.

The input and output of the TMS320C6713 DSK by default have sign inversions, which is strictly due to the hardware and must be corrected for in software since we are interested in the phase of the input signal. Also, we must invert the desired signal at the output so that it is of the correct sign when leaving the TMS320C6713 DSK. This inversion is not an issue for the channel simulators, which are discussed in the following section.

4.2 Channel Implementation

The scope of this thesis is to investigate the implementation of the system in [1] with high SNR single-path channels. The channels may be time-invariant or time-varying depending on the application.

Single-path channels can be modeled as a time delay where

$$\text{delay} = \frac{\text{distance}}{\text{speed of propagation}} \tag{4.6}$$

and the speed of propagation is determined by the medium in which the type waveform travels through. Using air as a medium (at sea level and under normal atmospheric conditions), the speed of propagation for an acoustic (sound) wave is approximately 343 [m/s].

4.2.1 Time-invariant Single-path Channel Implementation

The time-invariant single-path channel simulators operate at a sampling frequency (f_s) of 96 kHz. The TMS320C6713 DSK boards incur approximately a 152 μs

channel delay from their stereo input to output at this sampling frequency, which is shown in Section 5.1.

Delays from approximately 152μs up to 60s (due to the memory limitations of the TMS320C6713 DSK) are be attained by buffering the input signal and delaying its output by an integer number of sampling periods corresponding to the desired delay. This is achieved by using a `j` index to keep track of the current input sample location in the input buffer and then storing the next sample in the `j+1` location. The sample that is output is determined by a second index, `k`, which is equal to `j` minus the desired integer number of samples delay. The desired delay is determined by the user and can be changed on the fly using the DIP switches of the TMS320C6713 DSK.

4.2.2 Time-varying Single-path Channel Implementation

The time-invariant single-path channel simulators operate at a sampling frequency (f_s) of 96 kHz. They differ from the time-invariant channel simulators because they allow for non-integer sample delays to be realized through interpolation. A non-integer sample delay occurs when the desired propagation delay through a channel does not correspond to the exact sampling time of the TMS320C6713 DSK. Cubic interpolation as described in [12] is used in this thesis to realize these delays as

$$f(x_0+p) = \frac{-p(p-1)(p-2)}{6} f_{-1} + \frac{(p^2-1)(p-2)}{2} f_0 - \frac{p(p+1)(p-2)}{2} f_1 + \frac{p(p^2-1)}{6} f_2 \tag{4.7}$$

where p is the fractional sample delay in addition to the integer sample delay caused at index x_0 in relation to the current sample index. More complicated interpolation methods such as spline could be used to achieve a greater accuracy for the non-

integer delay case. Since the channel simulators are sampling at approximately nine times the Nyquist rate, cubic interpolation yields acceptable accuracy.

The time-varying channel simulators incorporate a "mobility script" that allows for a delay to be computed as a function of time based off of a desired distance. The distance $d(t)$ that the channel simulates is based on the current velocity $v(t)$ and acceleration $a(t)$, which is calculated as

$$d(t) = v(t) * t \tag{4.8}$$

$$v(t) = a(t) * t \tag{4.9}$$

The distance is then translated to a propagation delay using (4.6). The delay is realized through sampling the input and storing it in a buffer then outputting the delayed signal at a later sampling period that corresponds to the desired time delay in the same manner as the time-invariant channels.

Two different channel models are considered: piecewise constant position and piecewise constant velocity. For a piecewise constant position channel, the desired distance is calculated using

```
1   distance = distance_script[state]
```

where `distance_script[state]` is the predetermined desired distance as defined in the "mobility script." For a piecewise constant velocity channel, the desired distance is calculated using

```
1   distance += velocity_script[state]*fs_inv;
```

where`fs_inv` is the inverse of the sampling frequency. Once the desired distance is determined, the number of samples delay required is calculated as

```
1   delay = distance*0.00294117647;
2   samples_delay = delay*fs;
3   k = (int)samples_delay;
```

where `0.00294117647` is the inverse of the speed of an acoustic wave (340 [m/s]). The variable `k` is the floor of the desired number of samples delay. These values are then used in conjunction with (4.7) to determine the proper output. The code in its entirety can be seen in Appendix A.3

Using this implementation we can now discuss its performance versus the theoretical predictions in [1]. The next chapter delves into the characterization of this implementation using the TMS320C6713 DSK boards.

Chapter 5

DSK Characterization Results

Practical systems are subject to nonidealities that are not present in theoretical systems. The implementation presented in this thesis is not immune to such drawbacks. Thus, it is important to understand where these nonidealities exist to properly analyze the given system. Both the methods for and the results of several experiments that were conducted to characterize the performance of the TMS320C6713 DSK are given in this chapter and use the following equipment:

- 1 - HP33120A 15 MHz function generator
- 1 - TDS3014 four channel digital phosphor oscilloscope
- 1 - TMS320C6713 DSK board [10]
- 1 - Marantz PMD671/U1B solid state audio recorder (24-bits @ 96 kHz)
- Various RCA and BNC connectors and cables for interconnection between the electronic components' inputs and outputs

5.1 Characterization of Channel DSKs

In order to understand the exact delay from input to output of a given TMS320C6713 DSK running at an arbitrary frequency, one should first know the approximate delay that is expected. An estimate of the time delay from the input of one channel of the TMS320C6713 DSK to the output of the same channel was acquired using a simple loop program that samples both channel inputs then outputs the same data through both of the channel outputs of the TMS320C6713 DSK. The loop program used in this test, is the same one that is used to simulate the single-path time-invariant channels in the overall system presented in this thesis.

While the loop program is running, the "burst" function of an HP33120A function generator is used in conjunction with the "single sequence" function of a Tektronix TDS3014 Oscilloscope. The test setup is shown in Figure 5.1 and is completed using both the 96 kHz and 44.1 kHz sampling frequencies of the TMS320C6713 DSK.

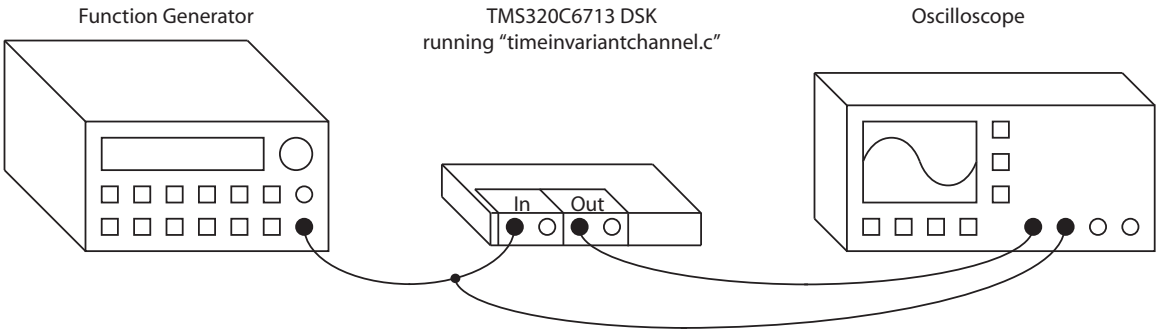

Figure 5.1: Test setup for preliminary characterization.

The "burst" function of the function generator can be used to output a particular number of cycles of a sine, square, or triangle wave with a set amplitude, frequency and starting phase. The "single sequence" function of the oscilloscope is able to record up to four input waveforms for a duration of 4 ns to 10 s in various steps once a predetermined input amplitude threshold (trigger level) is exceeded. For

this experiment, the HP33120A function generator's "burst" function was set up to output a 5.4 kHz sine wave with approximately a 1 volt amplitude and zero starting phase for two cycles. The Tektronix TDS3014 Oscilloscope's "single sequence" function was setup using a threshold of 4 mV with a duration of 1 ms and 2 ms for the 96 kHz (channel simulators) and 44.1 kHz (source nodes) sampling frequencies respectively.

Using the test setup in Figure 5.1, the results shown in Figure 5.2 and Figure 5.3 were acquired.

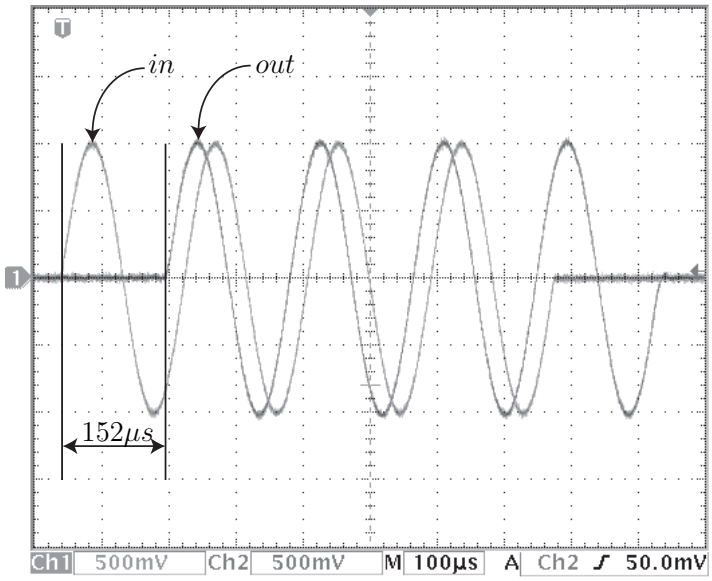

Figure 5.2: Oscilloscope plot to find approximate DSK input/output delay at a sampling frequency of 96 kHz.

When the TMS320C6713 DSK is running a basic stereo loop function at 96 kHz and 44.1 kHz, these results show that there is a propagation delay of approximately 152 μs and 1002 μs respectively. These results can be used as an estimation of the time delay of a given TMS320C6713 DSK running either a channel simulator or source node at a specific sampling frequency, but can not be used for sub-degree

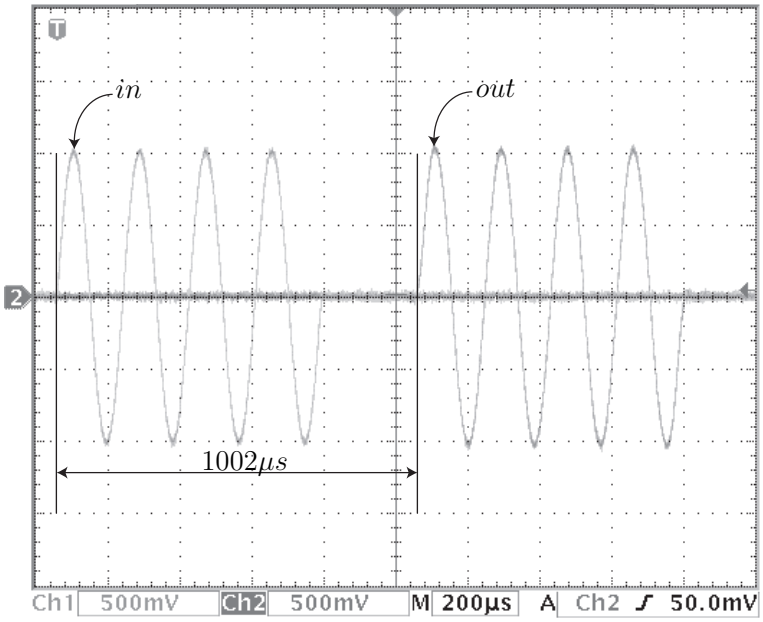

Figure 5.3: Oscilloscope plot to find approximate DSK input/output delay at a sampling frequency of 44.1 kHz.

timing accuracy. Also, these results can not be used to determine propagation delay as a function of frequency. Thus, we will need to conduct more accurate experiments to determine a precise analytical result.

For the next tests, we assume that the TMS320C6713 DSK causes an identical delay from the input to the output of the left channel as it does to the right channel. This is reasonable because the architecture of the hardware for the DSK and the symmetry of the software program. Although all channels are characterized, only one of the three channel simulators in the overall system needs to be discussed, since the other two operate at identical frequencies in each signal path of the overall system, thus induce the same propagation delay. The channel simulator that operates between the two source nodes requires further investigation. In one direction, the channel simulator operates at 1814 Hz and in the opposite direction it operates at 2721 Hz. The test setup in Figure 5.4 is used to determine the time delay for each

channel operating at their given frequency. The results for this test as well as the frequencies used can be found in Table 5.1.

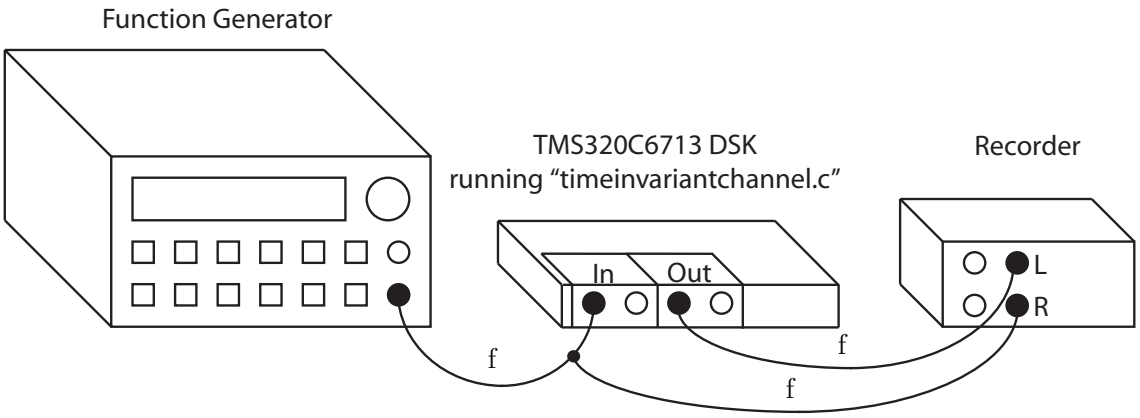

Figure 5.4: Test setup used to find propagation delay difference for the cross beacon channel.

f [Hz]	Delay [μs]
907	170.2
1814	176.2
2721	177.4
5442	177.9

Table 5.1: Results from the test setup shown in Figure 5.4.

The time difference between the two channels is calculated using the maximum likelihood estimation (MLE) for both frequency and phase as described in [13]. This allows us to now have a more accurate measurement for the delay through a given channel than the approximation that the oscilloscope previously yielded. The MLE of the phase of a discrete signal $x[n]$ at frequency ω [rad/sec] and of N samples in length is given as

$$\hat{\phi} = -\arctan * \frac{\sum_{n=0}^{N-1} x[n]\sin(\omega n)}{\sum_{n=0}^{N-1} x[n]\cos(\omega n)} \tag{5.1}$$

where the MLE for the frequency ω can be found by maximizing the likelihood function

$$I(\omega) = \frac{1}{N}\left|\sum_{n=0}^{N-1} x[n]\exp(-j\omega n)\right|^2 \tag{5.2}$$

and then using the ω value that maximizes (5.2) in (5.1). From this, one can easily describe the effective propagation delay τ by

$$\tau = \frac{\hat{\phi} \pm m2\pi}{2\pi} T \tag{5.3}$$

where T is the period of the signal, $\hat{\phi}$ is in radians, and m is an integer that results in the time delay that is closest to the approximations found in Figure 5.2 and Figure 5.3. The integer m is required because the MLE phase estimate is defined between $-\pi$ and π and the propagation delay induced by the TMS320C6713 DSK is multiple periods in time of the incoming signal.

For the test setup in Figure 5.4, the MLE estimation shows that the two signal paths of the distributed beamformer will not have identical propagation times, thus inducing an error into the overall system. This can be seen in Table 5.2 and shows that from the channel delays alone there is a 2.4 degree phase shift at the destination, which is caused by the different propagation delays in the cross beacon channels. Signal path 1 has the cross beacon frequency of 1814 Hz and signal path 2 has the cross beacon frequency of 2721 Hz.

To further characterize the beamforming system, we will next look at the four

Signal path 1 [μs]	Signal path 2 [μs]	Phase difference [deg]
524.3	525.5	2.4

Table 5.2: Overall signal path results from the test setup shown in Figure 5.4.

FS-PLLs to determine their effect on the overall beamforming system.

5.2 Characterization of FS-PLL DSKs

To characterize the FS-PLLs in this system, we need to use different techniques to those used for the cross-beacon channel simulator because the TMS320C6713 DSK input frequency (f_{in}) does not equal the output frequency (f_{out}) as it did in the tests involving the channel simulators. The overall goal is to quantify the propagation delay of each individual FS-PLL in the locked state to determine any potential discrepancies leading to phase offset at the destination.

The delay induced by a given TMS320C6713 DSK is again assumed to be independent from which channel of the DSK's input and/or output is used due to the symmetrical hardware architecture of the DSK and the software running on it. Also, the Marantz solid state recording device is assumed to be ideal. The setup for this test is shown in Figure 5.5 using the values given in Table 5.3.

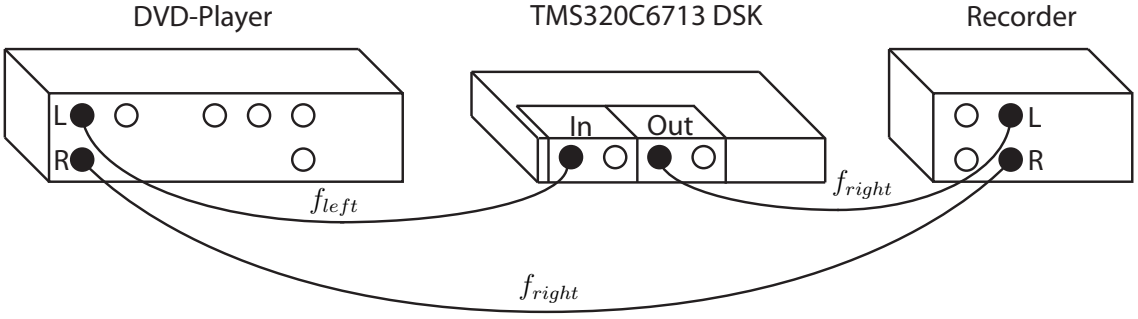

Figure 5.5: Test setup for determination of specific FS-PLL delays.

The four tests using the setup in Figure 5.5 yield four independent equations with

DVD-Player		FS-PLL	
Left channel (f_{left}) [Hz]	Right channel (f_{right}) [Hz]	Center Frequency [Hz]	"N" Multiplier
907	1814	907	2
907	2721	907	3
1814	5442	1814	3
2721	5442	2721	2

Table 5.3: Values used to determine individual FS-PLL delays.

relation to the propagation time of the f_{left} path with respect to the propagation time of the f_{right} path. The propagation delay differences are all calculated using the MLE of the phase as previously discussed and converting the phase $\hat{\phi}$ to a time delay τ using (5.3).

The delay induced by the interconnection cables is negligible, thus the delay in the path starting from the right channel of the DVD-player is approximately zero. The delay induced by the output circuitry of the DVD-player at a given frequency is denoted by τ_{DVD-f} where f is the frequency of the signal being output. The delay of a given FS-PLL is denoted by $\tau_{NxPLL-f}$, where N is the frequency synthesis multiplier and f is its center frequency.

We can then describe the time delay difference between the two paths as a system of equations in matrix form such that

$$\mathbf{Ax} = \mathbf{b} \tag{5.4}$$

where the $\mathbf{A}$ matrix describes the combination of the unknowns in the $\mathbf{x}$ vector while the $\mathbf{b}$ vector contains the actual measurements of the time difference between the f_{left} path in Figure 5.5 and the f_{right} path in milliseconds. For this set of tests, the unknowns are

$$\begin{bmatrix} 1 & -1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \tau_{DVD-907} \\ \tau_{DVD-1814} \\ \tau_{DVD-2721} \\ \tau_{DVD-5442} \\ \tau_{2xPLL-907} \\ \tau_{3xPLL-907} \\ \tau_{3xPLL-1814} \\ \tau_{2xPLL-2721} \end{bmatrix} = \begin{bmatrix} 1.176294 \\ 1.172178 \\ 1.266948 \\ 1.270087 \end{bmatrix} \tag{5.5}$$

It is important to note that there is no unique solution to (5.5). To solve for the propagation delay induced by a given FS-PLL ($\tau_{NxPLL-f}$), more information is needed. One way to solve this problem is to perform additional experiments that involve only the DVD-player and the recorder.

The DVD-player outputs signals with negligible phase offsets between its left and right channels while at the same frequency. This is shown by analyzing the test given in Figure 5.6 where the DVD-player and the recorder are modeled as having both an ideal part and a non-ideal delay that they induce. The results are shown in Table 5.4.

DVD-Player Output Frequency (f_{left} and f_{right}) [Hz]	Phase Difference ($\Delta\phi$) [deg]
907	-0.0573
1814	-0.0525
2721	-0.0615
5442	-0.1026

Table 5.4: DVD-Player Output Frequency vs. Time Delay Difference Measured

The phase difference in degrees between the left and right channels is computed

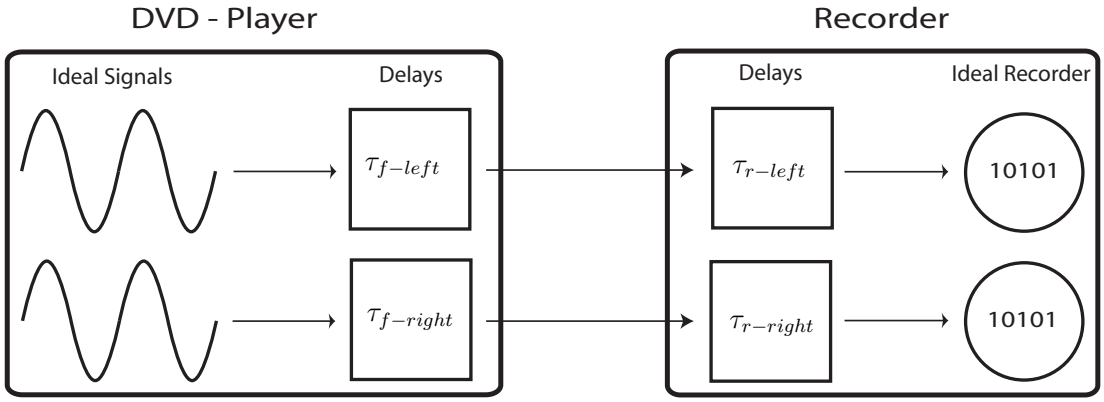

Figure 5.6: Test setup for calculating DVD-player phase offset.

as

$$\Delta\phi = \frac{(\tau_{f-left} + \tau_{r-left}) - (\tau_{f-right} + \tau_{r-right})}{T} * 360 \tag{5.6}$$

using the MLE of the phase where τ_{f-left} and $\tau_{f-right}$ are the time delays caused by the respective channel of the DVD-player at a given frequency and τ_{r-left} and $\tau_{r-right}$ are the time delays caused by the given channel of the recorder. Also, T is the period of the signal at f_{left} and f_{right}. Since the recorder is assumed to be ideal, τ_{r-left} and τ_{right} are both zero and $\Delta\phi$ reduces to

$$\Delta\phi = \frac{\tau_{f-left} - \tau_{f-right}}{T} * 360 \tag{5.7}$$

Although the DVD-player is able to produce identical signals at the output of the two channels while at the same frequency, the same is not true when the two channels are at different frequencies. To determine the time difference between two output waveforms at different frequencies, the test setup in Figure 5.6 was again used, but for the values given in Table 5.5, where a negative time delay is where the left channel is slower than the right channel.

As can be seen in Table 5.5, the propagation delay caused by the DVD-player is

f_{left} [Hz]	f_{right} [Hz]	Time Difference [μ s]
907	1814	-2.428
907	2721	-5.722
907	5442	-4.017
1814	2721	-3.361
1814	5442	-1.691
2721	5442	-1.590

Table 5.5: Left and right channel frequency versus phase difference

substantial when the two channels are at different frequencies. The overall test can then be described in matrix form where $\mathbf{C}$ is the combination of the unknowns in $\mathbf{y}$ and $\mathbf{d}$ contains the actual measurement results from the recordings and the MLE of the phase, which is then converted to a time delay using (5.3).

$$\mathbf{Cy} = \mathbf{d} \tag{5.8}$$

and

$$\begin{bmatrix} 1 & -1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ 1 & 0 & 0 & -1 \\ 0 & 1 & -1 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} \tau_{DVD-907} \\ \tau_{DVD-1814} \\ \tau_{DVD-2721} \\ \tau_{DVD-5442} \end{bmatrix} = \begin{bmatrix} -2.482 \\ -5.722 \\ -4.017 \\ -3.361 \\ -1.691 \\ -1.590 \end{bmatrix} \tag{5.9}$$

The actual measurements in $\mathbf{d}$ are given in microseconds. This is an over determined system, which has no unique solution, but allows us to determine the individual FS-PLL delays by combining (5.4) and (5.8) to obtain

$$\mathbf{Ex} = \mathbf{f} \tag{5.10}$$

where

$$\begin{bmatrix} 1 & -1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 1 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \tau_{DVD-907} \\ \tau_{DVD-1814} \\ \tau_{DVD-2721} \\ \tau_{DVD-5442} \\ \tau_{2xPLL-907} \\ \tau_{3xPLL-907} \\ \tau_{3xPLL-1814} \\ \tau_{2xPLL-2721} \end{bmatrix} = \begin{bmatrix} 1154.834 \\ 1150.547 \\ 1152.745 \\ 1155.810 \\ -2.482 \\ -5.722 \\ -4.017 \\ -3.361 \\ -1.691 \\ -1.590 \end{bmatrix} \tag{5.11}$$

and where the individual measurements in $\mathbf{f}$ are in microseconds. The final results shown in Table 5.6 were attained by solving (5.11):

FS-PLL delay	Time delay in milliseconds
$\tau_{2xPLL-907}$	1.15726
$\tau_{3xPLL-907}$	1.15627
$\tau_{3xPLL-1814}$	1.15444
$\tau_{2xPLL-2721}$	1.15740

Table 5.6: Individual FS-PLL time delay results

Using the individual component delays for each signal path we are able to determine the propagation delay of both entire signal paths (including channels and source nodes) and thus the phase difference between the two paths at the destina-

tion. This is shown in 5.7.

Signal path 1 [μs]	Signal path 2 [μs]	Phase difference [deg]
2836.0	2839.2	6.3

Table 5.7: Overall beamformer characterization results.

These results are critical to the overall system performance because the round trip propagation time between the two paths in the distributed beamformer will no longer be equivalent and a static phase offset of 6.3 degrees between the two sources' transmissions will be present. Thus, the overall system performance is negatively effected, which will be discussed as well as the overall system performance in Chapter 6.

Chapter 6

System Performance Results

This chapter provides analysis and experimental results of the distributed beamformer system presented in this thesis. Results are given for both single-path time-invariant and time-varying channels. The theoretical predictions presented in [1] are also confirmed in this chapter.

6.1 Lock Example

The experiment in this section shows the typical operation of the overall system when going from an "unlocked state" to a "locked state". An "unlocked state" is one in which each of the source nodes' FS-PLLs are operating in a free-running manner. This could be due to the absence of a master beacon signal or the characteristics of the channels affecting the sources. For this case, the reason that the FS-PLLs are considered free-running is that the master beacon signal from the function generator is not present until some time, which is denoted as zero carrier cycles. Once the master beacon signal turns on, the sources' FS-PLLs "lock" to their respective incoming signals and form a distributed beamformer. The coherently combined transmission of the sources at the destination both before and after the master

beacon is present is shown in Figure 6.1.

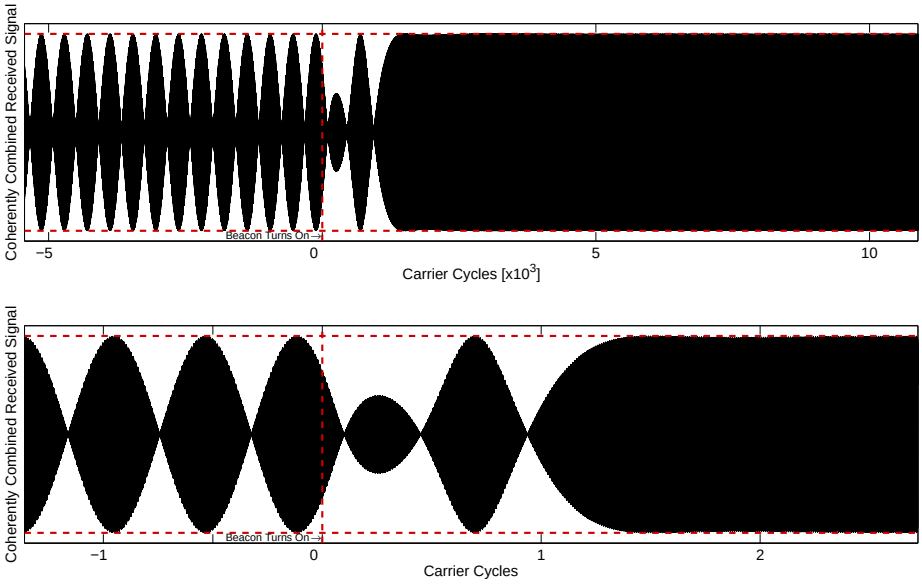

Figure 6.1: Example of two-source carrier synchronization as described in [1].

Another way to see that the two sources are coherently combining is to look at their phase difference over time. Figure 6.2 shows the cosine of the phase difference between the two sources at the destination. Using this method, it is clear that the distributed beamformer is operating to its fullest capacity when the phase difference is zero or a multiple of 2π, thus the cosine of this value is unity.

The phase approximate difference over time is computed in MATLAB using a sliding window of ten samples and the Hilbert transform as

```
phase(n) = -angle(sum(hilbert(source1(n:n+window-1)).*source2(n:window-1)))
```

where `window` is the length of the sliding window and `n` is the current sample index. The vectors `source1` and `source2` are the recorded signals at the destination that have already been normalized to have approximately unit amplitude. The `hilbert` function shifts the positive frequencies of the signal by -90° and the negative frequencies by +90°. For more information on the Hilbert transform see [14].

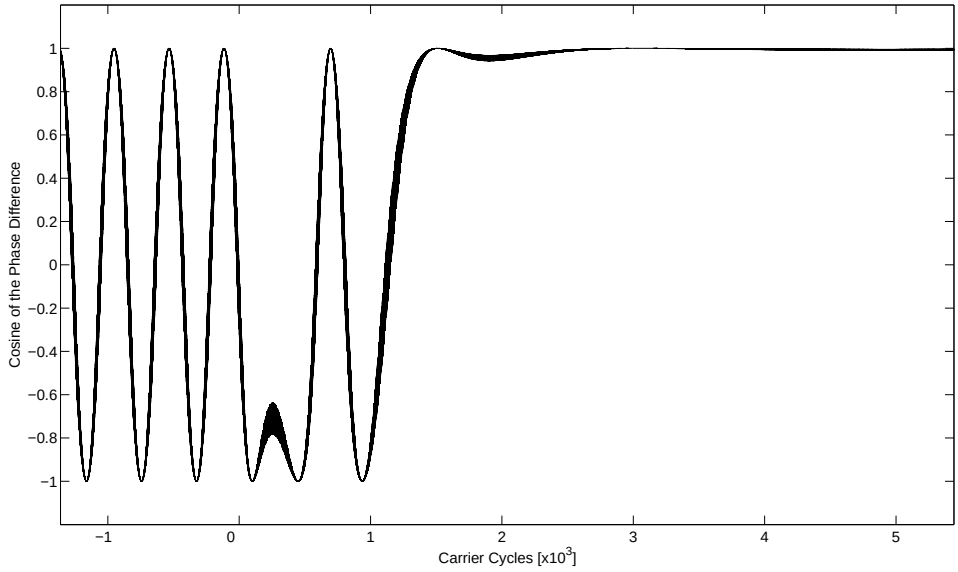

Figure 6.2: The cosine of the phase difference of the carrier transmissions at the destination.

Figure 6.3 shows the phase difference in degrees between the two sources' phases at the destination. This result shows a phase offset between the two sources' transmissions at the destination, which is predicted by the difference in propagation time for the two signal paths in Chapter 5.

The Hilbert transform method is used to find the phase difference of signals at the same frequency. Another way to realize the phase offset regardless of frequency is to find the MLE for both the frequency and phase of each source's transmission. From their respective phases, one could conclude the phase difference between the two signals. This technique is discussed in Chapter 5 and yields a phase difference of 6.4 degrees, which confirms the results found in Section 5.2.

As can be seen in Figure 6.3, the two sources are coherently combining to their full potential in less than five thousand carrier cycles due to converging into a "lock state." Once the system is in a "lock state", it remains in this state unless there is a change in one or more of the channels in the system. Channels of this nature can

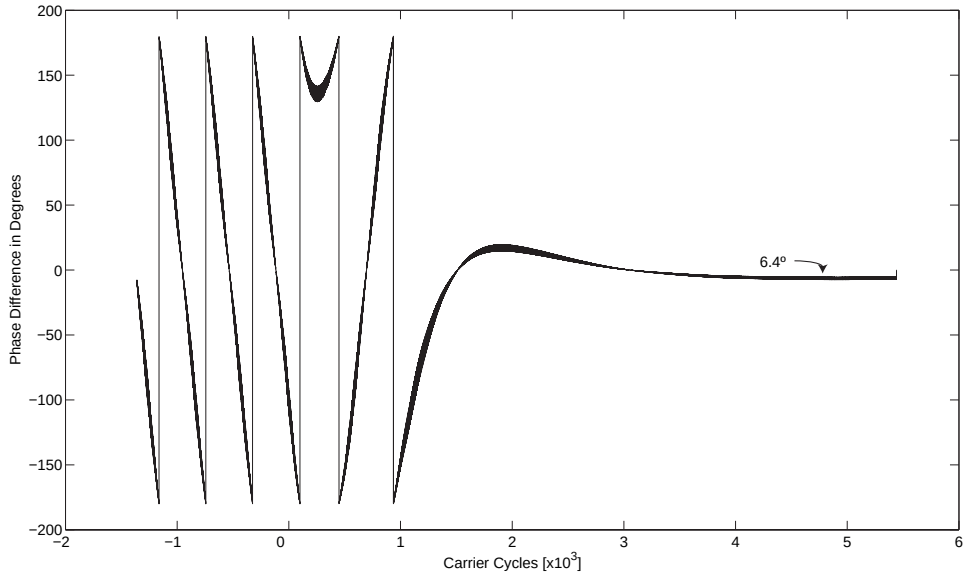

Figure 6.3: Difference between the two sources' phases at the destination.

be considered as time-varying and are discussed in the following sections.

6.2 Piecewise Constant Position Channel

The first type of single path time-varying channel that is investigated is a piecewise constant position channel. This type of channel has a constant delay for a given time, which is equivalent to a constant position that is then instantaneously changed to a different time delay. For this simulation, the position of the g_{01} channel is instantaneously changed periodically while the g_{02} and g_{12} channels are modeled as single-path and time-invariant. The instantaneous position change results in both an infinite velocity and acceleration. The goal of this section is to show that the system presented in this thesis is able to reconverge to a "locked state" when forced into an "unlocked state."

Figure 6.4 shows both the relative position in carrier wavelengths of the g_{01}

channel and the effect on the coherently combined signal at the destination. Relative position refers to the change in position in comparison to the effective length of the channel due to the constant propagation delay of the TMS320C6713 DSK.

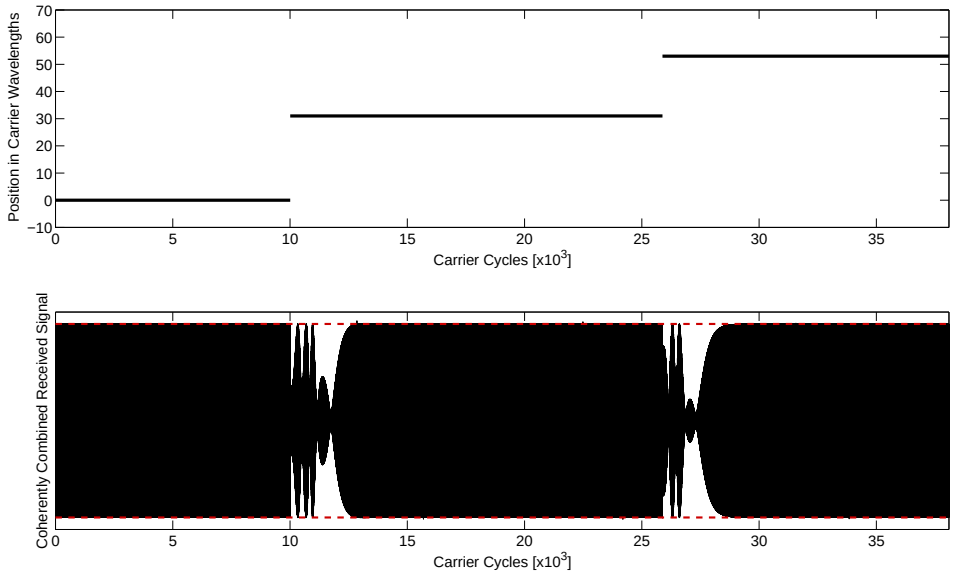

Figure 6.4: Example of a peicewise constant position change in the g_{01} channel.

Similar results to those in Figure 6.4 can be attained by making the g_{02} channel time-varying instead of the g_{01} channel. A variation of the g_{12} channel results in no change in performance on the overall system because any variation in that channel occurs on both source's secondary FS-PLL, thus having no effect in the difference of overall propagation delay between the two signal paths.

Figure 6.4 shows that when an instantaneous position change occurs, the coherently combined signal at the destination is negatively effected for less than five thousand carrier cycles as the system enters a brief "unlock state." This shows how robust this system is even when in an extremely unrealistic situation.

Figure 6.5 shows the cosine of the phase difference between the sources' transmissions over time. It is apparent that the performance of the beamformer is only

degraded for a few thousand carrier cycles before returning to a "lock state."

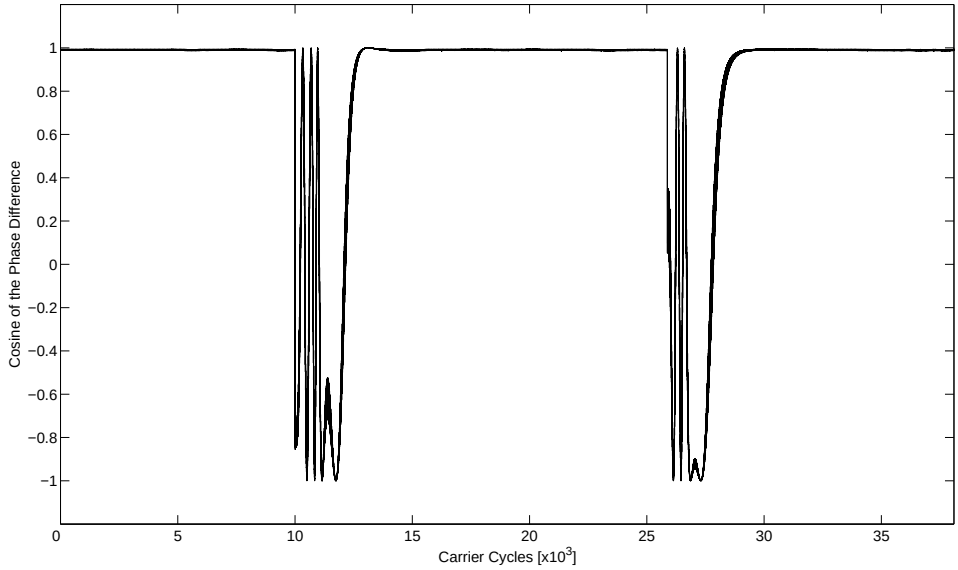

Figure 6.5: The cosine of the phase difference of the carrier transmissions at the destination.

Figure 6.6 shows the phase difference in degrees between the sources' transmission over time. The phase offset between the two sources' transmissions using the MLE is 8.6 degrees, which is not equal to the steady state error as predicted by the characterization results of the system. Thus, we must now look at the delay induced by the time-varying channel based on input/output frequency. Using the test setup shown in Figure 5.4, the single-path time-varying channel is characterized. The results are shown in Table 6.1.

f [Hz]	Delay [μs]
907	169.2
5442	178.2

Table 6.1: Single-path time-varying channel characterization results.

It is apparent that the delay induced based on frequency is not the same for

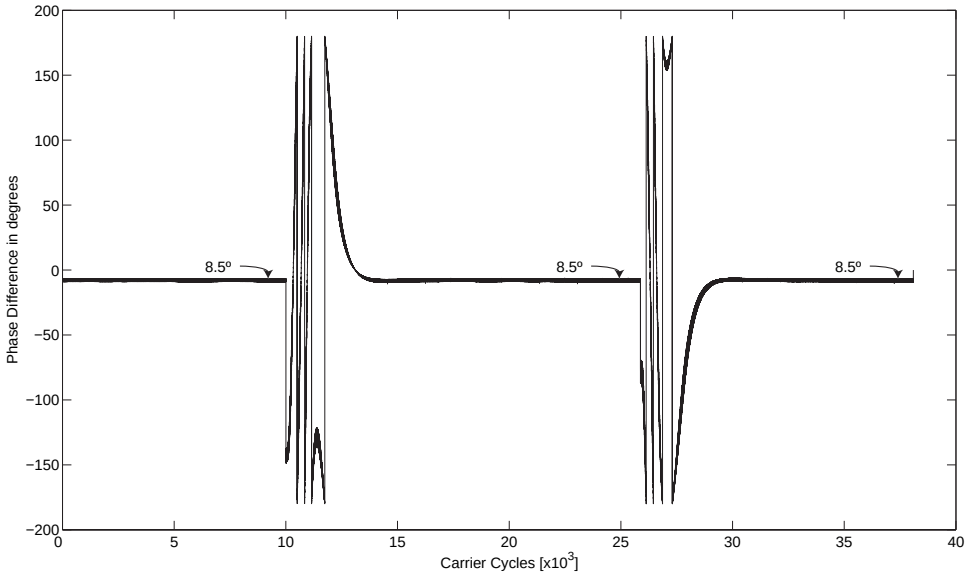

Figure 6.6: Difference between the two sources' phases at the destination.

the single-path time-invariant and time-varying channels. Using the results from Table 6.1 in conjunction with the results presented in Chapter 5, we are now able to accurately predict a static phase offset of 8.7 degrees as shown in Table 6.2, which agrees closely with the MLE method.

Signal path 1 [μs]	Signal path 2 [μs]	Phase difference [deg]
2835.0	2839.4	8.7

Table 6.2: Overall system signal path results while using a piecewise constant position channel.

The reason that the single-path time-invariant and single-path time-varying channel delays are not equal is not currently known and needs to be investigated at a later time. Nevertheless, we are able to predict the static phase offset from the individual component delays.

The results in this section show that the distributed beamformer system can return to a "locked state" when forced to fall out of lock. The next section discusses

the effect of a single-path time-varying velocity channel.

6.3 Piecewise Constant Velocity Channel

The channel models thus far have investigated fixed or instantaneously changing channels. This section discusses the effect of a single-path time-varying velocity channel in the system. The g_{01} channel is used as the time-varying channel, but a similar discussion could be had using the g_{02} channel. The goal of this section is to show that the distributed beamformer is not gravely affected when a time-varying velocity channel is introduced.

Figure 6.7 shows both the relative position in carrier wavelengths of the g_{01} channel and the effect on the coherently combined signal at the destination. For this case, the velocity in the g_{01} channel is either 0 [m/s], 0.3 [m/s], or -0.3 [m/s] and is instantaneously changed at a given time. This instantaneous change in velocity is equivalent to an infinite acceleration.

It is apparent in Figure 6.7 that the performance of the distributed beamformer system is negatively effected at the points where the velocity instantaneously changes, but returns to a "locked state" within a few thousand carrier cycles, which agrees with the results in [1]. The MLE results are shown in Table 6.3.

Velocity [m/s]	Phase Offset [deg]
0.0	8.5
0.3	13.1
-0.3	4.0

Table 6.3: Single-path time-varying velocity channel MLE experimental results.

In order to determine the actual time delay difference caused by a piecewise constant velocity channel we first consider the case of an impulse $\delta(t - t_0)$ through one path of the system. The output is given as

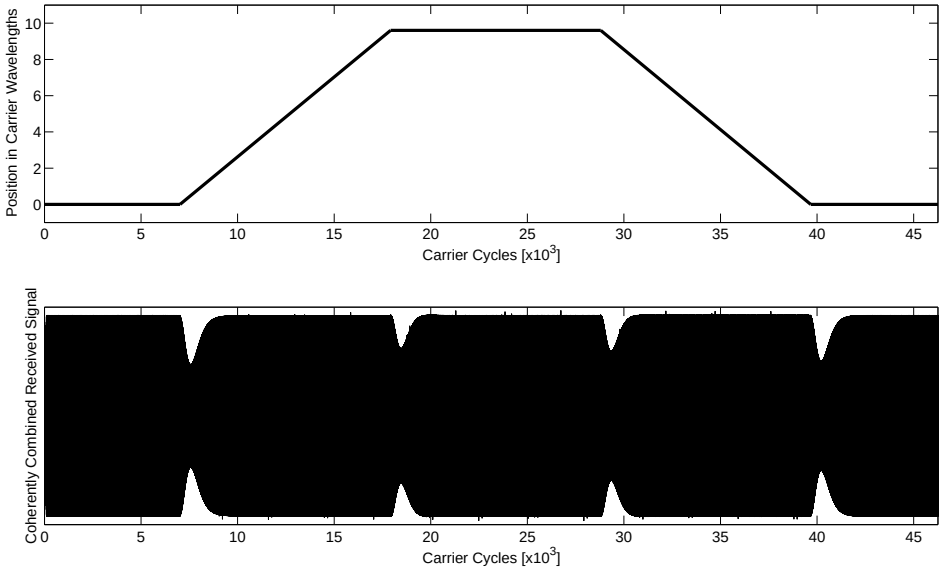

Figure 6.7: Example of the carrier synchronization performance in a single-path time varying velocity channel.

$$y(t) = x(t - \Delta(t)) \tag{6.1}$$

where

$$\Delta(t) = \Delta_0 + \beta t \tag{6.2}$$

and Δ_0 is the delay through channel while β is the slope of the propagation delay due to velocity versus time given by

$$\beta = \frac{v}{c} \tag{6.3}$$

where v is the velocity simulated by the channel in [m/s] and c is the speed of propagation, which is 340 [m/s] for an acoustic wave. We can now write the output as

$$y(t) = \delta((1-\beta)t - t_0 - \Delta_0) \tag{6.4}$$

Solving for the effective delay $(t - t_0)$ through the channel

$$t - t_0 = \frac{\beta t_0 + \Delta_0}{1-\beta} \tag{6.5}$$

The propagation delay offset caused by the velocity channel in the two paths can then be calculated as

$$\Delta_t = \frac{\beta t_L + \Delta_L}{1-\beta} - \frac{\beta t_R + \Delta_R}{1-\beta} \tag{6.6}$$

where β is the slope of the effective propagation delay of the time-varying channel due to velocity versus time. Also, Δ_L and Δ_R are the fixed delays of the left and right velocity channels. Lastly, t_L and t_R are the left and right signal path delays prior to the signal entering the time-varying channel.

Using (6.6) and the previously determined propagation delays, we are now able to calculate the expected overall phase offset in the system. Table 6.4 shows the predicted phase offset of the two sources' transmissions at the destination based on a given velocity in the g_{01} channel.

Velocity [m/s]	Phase Offset [deg]
0.0	8.6
0.3	13.4
-0.3	3.8

Table 6.4: Single-path time-varying velocity channel predicted results using the system characterization results.

The cosine of the phase difference and the phase difference between the two sources' transmissions during this experiment can be seen in Figure 6.8 and Fig-

ure 6.9 respectively. Both show that the two sources do indeed fall into an "unlocked state," but then quickly reconverge into a "locked state."

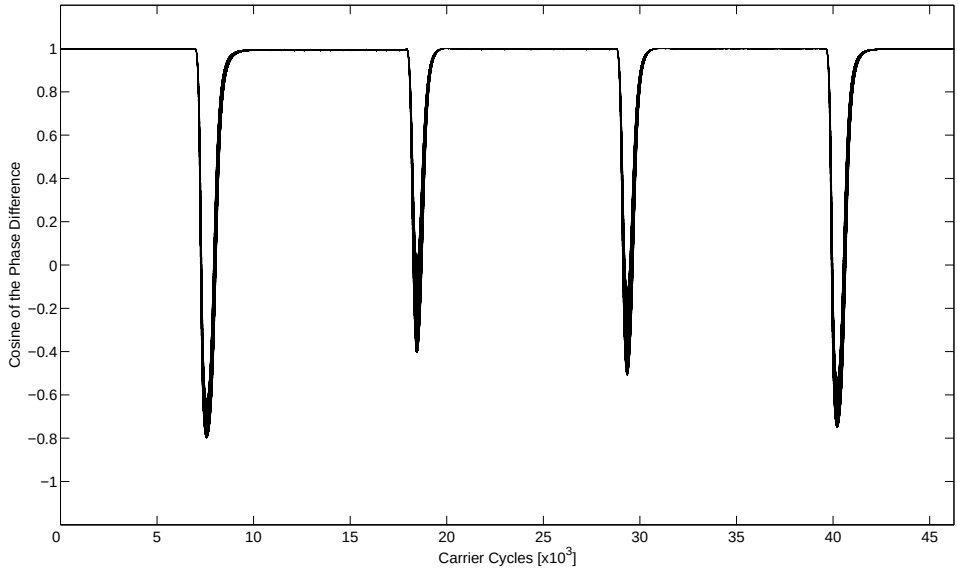

Figure 6.8: The cosine of the phase difference of the carrier transmissions at the destination.

Figure 6.9 shows that velocity does indeed cause the overall beamformer performance to be degraded, which is consistent with the predicted results as shown in Table 6.4. Figure 6.8 shows that even though velocity is present in the g_{01} channel, the effect on the overall beamformer performance is negligible. These results show that this system works even when is mobility present, but also that the amount of mobility must be considered. The current system simulates acoustic signals, but could be implemented in RF, which would lessen the effect of velocity in a channel on the overall beamformer performance.

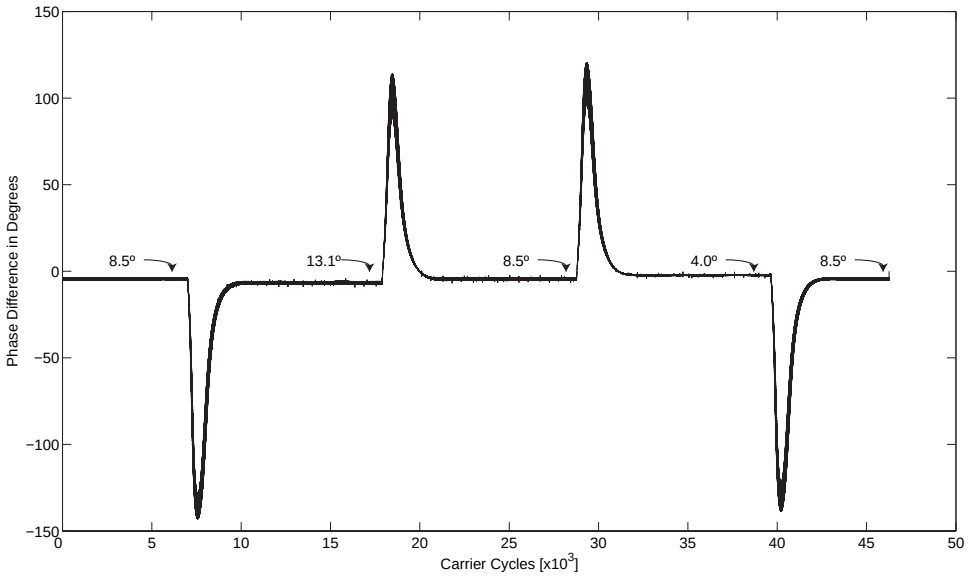

Figure 6.9: Difference between the two sources' phases at the destination.

Chapter 7

Conclusions

This thesis provides specific guidelines, performance analysis, and system characterization of a real-time software-defined-radio implementation of a two source distributed beamformer. This technique can be used to synchronize the carriers of two signal antenna wireless transmitters that each have independent local clocks so that their bandpass transmissions arrive in phase at an intended receiver to create a distributed beamformer.

Two sources nodes were implemented using two TMS320C6713 DSK boards. Each node encompasses two FS-PLLs that are implemented in the C programming language in order to lock to both a master beacon signal and a cross-beacon signal from the other source node.

Three channels nodes were also implemented using three TMS320C6713 DSK boards. Single path time-invariant and two types of single path time-varying (PCP and PCV) channels were investigated. The system performance analysis confirms the theoretical predictions in [1] for both time-invariant and time-varying channels.

System characterization results are also provided to quantify the practical non-idealities that exist in the TMS320C6713 DSK and the experimental equipment.

These results can also be used in an extension of this system or other future research pertaining to the TMS320C6713 DSK.

System performance results are presented that show how the overall beamformer operates in both single-path time-invariant and time-varying channels. These results show that a practical system in both the acoustic and radio frequency range is feasible and what implementation problems may arise.

7.1 Future Research Opportunities

The results of this thesis have revealed several other research opportunities that can be considered in the future. The system provided herein was implemented using wired channels, which could be modified to incorporate wireless channels. This would then lead to the possibility of creating individual nodes that are portable, which could be tested in various real-life environments.

This thesis was implemented using the C programming language, therefore, it could be easily ported to other compatible platforms in addition to DSP chips. Also, other types of carrier synchronization techniques such as time-division multiplexing could be implemented using the presented platform.

Another topic that could be considered is the effect of a single-path time-varying piecewise constant acceleration (PCA) channel. This type of channel is difficult to analyze due to three different components that induce error into the overall beamformer, which include the static system phase offset due to the TMS320C6713 DSKs, the error caused by a frequency ramp on the input of a FS-PLL, and the error from a constantly changing velocity.

The static phase offset has been discussed and characterized throughout this thesis. It has to always be considered when analyzing the presented system. This

offset could possibly be corrected in software if further analysis of an individual TMS320C6713 DSK was conducted. One way of doing such investigation would be to characterize each component of the entire system over every possible frequency below the Nyquist rate.

The second error caused by acceleration in a channel is due to the frequency ramp that is seen at the input of the FS-PLL that operates on that channel. A FS-PLL is able to modify its output frequency and phase based on a static incoming frequency and changing phase, but a phase error occurs when an increasing or decreasing frequency ramp is present at the input. The larger the frequency ramp (positive or negative), the greater the phase error at the output of the FS-PLL as discussed in [1]. Thus, we have a second component that adds error into the overall system when an acceleration channel is present.

The final component adding error into the overall beamformer is the same effect as seen in Table 6.3. The only difference is that the velocity is continuously changing, which means that analysis needs to be completed at every point in time to properly characterize the expected performance.

From the piecewise constant velocity channel results, we can conclude that as velocity in a channel increases, the phase offset of the overall system due to velocity increases. Thus, considering the three components adding to error in a piecewise constant acceleration channel, we can conclude that over time the phase offset of the beamformer will increase indefinitely at any constant acceleration.

Appendix A

System Source Code

This section provides the source code for both the FS-PLLs and the channel simulators that are presented in this thesis.

A.1 Software-Defined Frequency Synthesis Phase-Locked-Loop (FS-PLL)

The following code is used to implement both source nodes in the system. If DIP switch zero is depressed when the program executes, the TMS320C6713 DSK will operate as a source with frequency synthesis multipliers equal to 3. Otherwise, it will operate as a source with frequency synthesis multipliers equal to 2. The center frequencies of the FS-PLLs for a given source assume that the other source is operating at a master beacon frequency of 907 Hz with frequency synthesis multipliers opposite to that of its own.

```
/*************************************************************************
* Interrupt based universal FS-PLL
* James McGinley
* Jan. 5, 2007
* DIP0 = up   --> source 1 (x2 FS-PLLs)
* DIP0 = down --> source 2 (x3 FS-PLLs)
 *************************************************************************/

#define CHIP_6713

#include <stdio.h>
#include <c6x.h>
#include <csl.h>
#include <csl_mcbsp.h>
#include <csl_irq.h>
#include "dsk6713.h"
#include "dsk6713_aic23.h"
#include "stdio.h"
#include "fastmath67x.h"
#define N 10
#define MAX 32768
#define PI        3.14159265358979323846

/*******************************************************
 *  Start of Declare Variables
 *******************************************************/
// Used for codec read/write
union {Uint32 combo; short part[2];} data1;
union {Uint32 combo; short part[2];} data2;

// Index
int n=0;

// Used for scaling
float invMAX = 0.000030517578125;

// Left/right input buffers
float fleftchannel[N]={0.0};
float frightchannel[N]={0.0};

// Left/right FS-PLL center frequencies [Hz]
float f0_left = 907.0;
float f0_right = 2721.0;

// Current VCO frequency left/right [Hz]
double f2_left = 0.0;
double f2_right = 0.0;

// Pi
double pi = PI;

// Pi/2
double pi_half = 0;

// VCO sensitivity [Hz/(V*s)]
float Ko = 15.70797;

// Phase detector gain
float Kd = 1;

// Feedback signal left/right [rad/sec]
double w2_left[N] = {0.0};
double w2_right[N] = {0.0};

// Output signal [rad/sec]
```

```
66  double w3_left[N] = {0.0};
67  double w3_right[N] = {0.0};
68
69  // VCO phase left/right [rad]
70  double vco_phase_left[N] = {0.0};
71  double vco_phase_right[N] = {0.0};
72
73  // Sampling frequency
74  float Fs = 44100.0;
75
76  // Inverse of sampling frequency
77  double invfs = 0;
78
79  // 1/(2*pi)
80  double oneovertwopi = 0;
81
82  // Four quadrant phase detector output left/right
83  float FQout_left[N] = {0.0};
84  float FQout_right[N] = {0.0};
85
86  // Low-pass filter output left/right
87  float LPFout_left[N] = {0.0};
88  float LPFout_right[N] = {0.0};
89
90  // Frequency synthesis multiplier left/right
91  float fmult_left = 2;
92  float fmult_right = 2;
93
94  //PLL with bandwidth of ~ 5 Hz
95  float numd_left[3] = {0.00711163489605, 0.00000101316410, -0.00711062173194};
96  float dend_left[3] = {1.00000000000000, -1.99644443584300, 0.99644443584300};
97
98  float numd_right[3] = {0.00711163489605, 0.00000101316410, -0.00711062173194};
99  float dend_right[3] = {1.00000000000000, -1.99644443584300, 0.99644443584300};
100
101 // Codec configuration with default settings
102 DSK6713_AIC23_CodecHandle hCodec;
103 DSK6713_AIC23_Config config = DSK6713_AIC23_DEFAULTCONFIG;
104 interrupt void serialPortRcvISR(void);
105
106 /*******************************************************
107  *  End of Declare Variables
108  *******************************************************/
109
110
111 void main()
112 {
113         // Initialize index to 0
114         n=0;
115
116         // Compute additional variables
117         invfs = 1/(float)44100;
118         oneovertwopi = 1/(2*pi);
119         pi_half = pi/2;
120
121         // Initialize output to 0
122         data2.combo = 0;
123
124         // Initialize the board support library, must be called first
125         DSK6713_init();
126
127         // open codec and get handle
128         hCodec = DSK6713_AIC23_openCodec(0, &config);
129
130         // initialize the DIP switches
```

```
131         DSK6713_DIP_init();
132
133         // initialize the LEDs
134         DSK6713_LED_init();
135
136         // Configure buffered serial ports for 32 bit operation
137         MCBSP_FSETS(SPCR1, RINTM, FRM);
138         MCBSP_FSETS(SPCR1, XINTM, FRM);
139         MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);
140         MCBSP_FSETS(XCR1, XWDLEN1, 32BIT);
141
142         // set codec sampling frequency
143         DSK6713_AIC23_setFreq(hCodec, DSK6713_AIC23_FREQ_44KHZ);
144
145         // interrupt setup
146
147         // Globally disables interrupts
148         IRQ_globalDisable();
149
150         // Enables the NMI interrupt
151         IRQ_nmiEnable();
152
153         // Maps an event to a physical interrupt
154         IRQ_map(IRQ_EVT_RINT1,15);
155
156         // Enables the event
157         IRQ_enable(IRQ_EVT_RINT1);
158
159         // Globally enables interrupts
160         IRQ_globalEnable();
161
162
163         // Check DIP0 to determine which source to operate as
164         // Check if DIP0 is up
165         if (DSK6713_DIP_get(0) == 0)
166                 {
167                 // turn LED 0 on
168                 DSK6713_LED_on(0);
169
170                 // Center frequency (right)
171                 f0_right = 1814.0;
172
173                 // Define VCO phase multiplier (left)
174                 fmult_left = 3;
175
176                 // Define VCO phase multiplier (right)
177                 fmult_right = 3;
178                 }
179
180         // enter infinite "while" loop and wait for interrupt
181         while(1)
182         {
183         }
184 }
185
186 interrupt void serialPortRcvISR()
187 {
188         // Read input from left/right channels
189         data1.combo = MCBSP_read(DSK6713_AIC23_DATAHANDLE);
190
191         // Store input to float, scale, and negate for inversion correction
192         fleftchannel[n] = -((float)data1.part[1])*invMAX;
193         frightchannel[n] = -((float)data1.part[0])*invMAX;
194
195 //=============================================================================================================
```

```
196 //      START: Four Quadrant Phase Detector
197 //=============================================================================================================
198
199
200         FQout_left[n] = fleftchannel[n] * w2_left[n] * Kd;
201         FQout_right[n] = frightchannel[n] * w2_right[n] * Kd;
202
203 //=============================================================================================================
204 //      END: Four Quadrant Phase Detector
205 //=============================================================================================================
206 //=============================================================================================================
207 //      START: Loop Filter (LF)
208 //=============================================================================================================
209
210         if (n==0)
211         {
212         LPFout_left[n] = numd_left[0]*FQout_left[0] + numd_left[1]*FQout_left[N-1] + numd_left[2]*
213         FQout_left[N-2] - dend_left[1]*LPFout_left[N-1] - dend_left[2]*LPFout_left[N-2];
214         LPFout_right[n] = numd_right[0]*FQout_right[0] + numd_right[1]*FQout_right[N-1] + numd_right[2]*
215         FQout_right[N-2] - dend_right[1]*LPFout_right[N-1] - dend_right[2]*LPFout_right[N-2];
216         }
217         if (n==1)
218         {
219         LPFout_left[n] = numd_left[0]*FQout_left[1] + numd_left[1]*FQout_left[0] + numd_left[2]*
220         FQout_left[N-1] - dend_left[1]*LPFout_left[0] - dend_left[2]*LPFout_left[N-1];
221         LPFout_right[n] = numd_right[0]*FQout_right[1] + numd_right[1]*FQout_right[0] + numd_right[2]*
222         FQout_right[N-1] - dend_right[1]*LPFout_right[0] - dend_right[2]*LPFout_right[N-1];
223         }
224         if (n>=2)
225         {
226         LPFout_left[n] = numd_left[0]*FQout_left[n] + numd_left[1]*FQout_left[n-1] + numd_left[2]*
227         FQout_left[n-2] - dend_left[1]*LPFout_left[n-1] - dend_left[2]*LPFout_left[n-2];
228         LPFout_right[n] = numd_right[0]*FQout_right[n] + numd_right[1]*FQout_right[n-1] +
229         numd_right[2]*FQout_right[n-2] - dend_right[1]*LPFout_right[n-1] - dend_right[2]*LPFout_right[n-2];
230         }
231
232 //=============================================================================================================
233 //      END: Loop Filter (LF)
234 //=============================================================================================================
235 //=============================================================================================================
236 //      START: Voltage Controlled Oscillator (VCO)
237 //=============================================================================================================
238
239         // Calculate VCO frequency left/right
240         f2_left = f0_left + (Ko*oneovertwopi) * LPFout_left[n];
241         f2_right = f0_right + (Ko*oneovertwopi) * LPFout_right[n];
242
243         // Increment VCO phase angle left/right and handle phase wrap
244         if(n+1>N-1)
245         {
246                 vco_phase_left[0] = vco_phase_left[N-1] + 2*pi*(f2_left*invfs);
247                 vco_phase_right[0] = vco_phase_right[N-1] + 2*pi*(f2_right*invfs);
248
249                 while(vco_phase_left[0]>2*pi)
250                         vco_phase_left[0] += -2*pi;
251                 while(vco_phase_left[0]<0)
252                         vco_phase_left[0] += 2*pi;
253                 while(vco_phase_right[0]>2*pi)
254                         vco_phase_right[0] += -2*pi;
255                 while(vco_phase_right[0]<0)
256                         vco_phase_right[0] += 2*pi;
257
258                 // Feedback signal left/right
259                 w2_left[0] = sindp(vco_phase_left[0]+pi_half);
260                 w2_right[0] = sindp(vco_phase_right[0]+pi_half);
```

```
261
262                 // Output signal left/right, negate for inversion correction
263                 w3_left[0] = -sindp(fmult_left*vco_phase_left[0]);
264                 w3_right[0] = -sindp(fmult_right*vco_phase_right[0]);
265         }
266         else
267         {
268                 vco_phase_left[n+1] = vco_phase_left[n] + 2*pi*(f2_left*invfs);
269                 vco_phase_right[n+1] = vco_phase_right[n] + 2*pi*(f2_right*invfs);
270
271                 while(vco_phase_left[n+1]>2*pi)
272                         vco_phase_left[n+1] += -2*pi;
273                 while(vco_phase_left[n+1]<0)
274                         vco_phase_left[n+1] += 2*pi;
275                 while(vco_phase_right[n+1]>2*pi)
276                         vco_phase_right[n+1] += -2*pi;
277                 while(vco_phase_right[n+1]<0)
278                         vco_phase_right[n+1] += 2*pi;
279
280                 // Feedback signal left/right
281                 w2_left[n+1] = sindp(vco_phase_left[n+1]+pi_half);
282                 w2_right[n+1] = sindp(vco_phase_right[n+1]+pi_half);
283
284                 // Output signal left/right, negate for inversion correction
285                 w3_left[n+1] = -sindp(fmult_left*vco_phase_left[n+1]);
286                 w3_right[n+1] = -sindp(fmult_right*vco_phase_right[n+1]);
287
288         }
289
290         // Set data2 to zero (output)
291     data2.combo=0;
292
293     // Setup left/right channel outputs (including scaling)
294         data2.part[1] = (short)(MAX*w3_left[n]*0.99);
295         data2.part[0] = (short)(MAX*w3_right[n]*0.99);
296
297 //=================================================================================================
298 //      END: Voltage Controlled Oscillator (VCO)
299 //=================================================================================================
300         // now output the left/right channels to the codec
301         MCBSP_write(DSK6713_AIC23_DATAHANDLE,data2.combo);
302
303         // increment buffer index and handle wraparound
304         n++;
305         if (n>=N) n=0;
306 }
307
```

A.2 Single-Path Time-Invariant Channel Simulator

The following code is used to implement single-path time-invariant channel simulators. The user can change the delay induced by the channel at any point by depressing any combination of the four DIP switches. Each DIP switch induces twice the delay of the previous starting with DIP switch zero that increases the delay in the channel by 50 μs. The channel delay can be no less than approximately 152 μs as described in Section 5.1.

```
1  /*************************************************************************
2  * Interrupt based time-invariant channel simulator
3  * James McGinley
4  * Dec. 1, 2006
5  * All DIP switches up = 152 microsecond delay
6  * DIP0 = down  --> +50 microseconds delay
7  * DIP1 = down  --> +100 microseconds delay
8  * DIP2 = down  --> +200 microseconds delay
9  * DIP3 = down  --> +400 microseconds delay
10  *************************************************************************/
11
12 #define CHIP_6713
13 #define N 1024                          // buffer size
14 #define DELAY_0 5                       // Delay = 50  microseconds
15 #define DELAY_1 10              // Delay = 100 microseconds
16 #define DELAY_2 21              // Delay = 200 microseconds
17 #define DELAY_3 42              // Delay = 400 microseconds
18 #define MAX 32768
19 #include <stdio.h>
20 #include <c6x.h>
21 #include <csl.h>
22 #include <csl_mcbsp.h>
23 #include <csl_irq.h>
24 #include "dsk6713.h"
25 #include "dsk6713_aic23.h"
26
27 // ------------------------------------------------------------
28 // Global declarations
29 // ------------------------------------------------------------
30 // index
31 Uint16 i=0;
32
33 // index
34 Uint16 j=0;
35
36 // index
37 Uint16 k=0;
38
39 // for codec read
40 union {Uint32 combo; short part[2];} data1;
```

```
41
42  // for codec write
43  union {Uint32 combo; short part[2];} data2;
44
45  // Used for scaling
46  float invMAX = 0.000030517578125;
47
48  // test string to see if DIP was previously on
49  short test[4] = {0, 0, 0, 0};
50
51  // delay string
52  short DELAY[4] = {DELAY_0, DELAY_1, DELAY_2, DELAY_3};
53
54  // buffer
55  short leftchannel[N]={0};
56
57  // buffer
58  short rightchannel[N]={0};
59  // codec handle
60  DSK6713_AIC23_CodecHandle hCodec;
61
62  // codec configuration with default settings
63  DSK6713_AIC23_Config config = DSK6713_AIC23_DEFAULTCONFIG;
64  interrupt void serialPortRcvISR(void);
65
66  void main()
67  {
68          // Initialize the board support library
69          DSK6713_init();
70
71          // initialize the DIP switches
72          DSK6713_DIP_init();
73
74          // initialize the LEDs
75          DSK6713_LED_init();
76
77      // open codec and get handle
78          hCodec = DSK6713_AIC23_openCodec(0, &config);
79
80          // configure buffered serial ports for 32 bit operation
81          MCBSP_FSETS(SPCR1, RINTM, FRM);
82          MCBSP_FSETS(SPCR1, XINTM, FRM);
83          MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);
84          MCBSP_FSETS(XCR1, XWDLEN1, 32BIT);
85
86          // set sampling frequency
87          DSK6713_AIC23_setFreq(hCodec, DSK6713_AIC23_FREQ_96KHZ);
88
89          // globally disables interrupts
90          IRQ_globalDisable();
91
92          // enables the NMI interrupt
93          IRQ_nmiEnable();
94
95          // maps an event to a physical interrupt
96          IRQ_map(IRQ_EVT_RINT1,15);
97
98          // enables the event
99          IRQ_enable(IRQ_EVT_RINT1);
100
101         // globally enables interrupts
102         IRQ_globalEnable();
103
104         // enter infinite "while" loop and wait for interrupt
105         while(1)
```

```
106         {
107         }
108 }
109
110 interrupt void serialPortRcvISR()
111 {
112         // read left/right channels from codec
113         data1.combo = MCBSP_read(DSK6713_AIC23_DATAHANDLE);
114
115         // test DIP switches
116         for (i=0; i<=3; i++)
117         {
118                 // DIP switch is on
119                 if (DSK6713_DIP_get(i) == 0)
120                 {
121                         if (test[i] == 0)
122                         {
123                                 DSK6713_LED_on(i);
124                                 j=j + DELAY[i];
125                                 test[i] = 1;
126                         }
127                 }
128                 // DIP switch is off
129                 if (DSK6713_DIP_get(i) == 1)
130                 {
131                         if (test[i] == 1)
132                         {
133                                 DSK6713_LED_off(i);
134                                 j = j - DELAY[i];
135                                 test[i] = 0;
136                         }
137                 }
138         }
139         // Store input to float, scale, and negate for inversion correction
140         fleftchannel[j] = ((short)data1.part[1])*invMAX;
141         frightchannel[j] = ((short)data1.part[0])*invMAX;
142
143         // clear output variable
144         data2.combo=0;
145
146         // Setup left/right channel outputs (including scaling)
147         data2.part[1] = (short)(MAX*w3_left[k]*0.99);
148         data2.part[0] = (short)(MAX*w3_right[k]*0.99);
149
150         // increment buffer index and handle wraparound
151         j++;
152         if (j==N) j=0;
153         k++;
154         if (k==N) k=0;
155
156         // write left/right channels to codec
157         MCBSP_write(DSK6713_AIC23_DATAHANDLE,data2.combo);
158 }
159
```

A.3 Single-Path Time-Varying Channel Simulator

The follow code is used to implement single-path time-varying channel simulators. At the start of the program, the user is able to determine which of the three time-varying channels (PCP, PCV, or PCA) that will be implemented by using DIP switches zero and one as described in the code documentation.

```
1  /************************************************************************
2  * Interrupt based universal time-varying channel simulator
3  * James McGinley
4  * Jan. 5, 2007
5  * DIP0 = down,          DIP1 = up       --> piecewise constant position channel
6  * DIP0 = up,            DIP1 = up       --> piecewise constant velocity channel
7  * DIP0 = up,            DIP1 = down     --> piecewise constant acceleration channel
8   ************************************************************************/
9
10 #include <c6x.h>
11 #include <csl.h>
12 #include <csl_mcbsp.h>
13 #include <csl_irq.h>
14 #include "dsk6713.h"
15 #include "dsk6713_aic23.h"
16 #include "stdio.h"
17 #include "math.h"
18 #define N 1000
19 #define T 95999
20 #define MAX 32768
21 // ---------------------------------------------------------------------------
22 // Global declarations
23 // ---------------------------------------------------------------------------
24
25 // Indexes
26 int i=0;
27 int j=0;
28 int k=0;
29 int t=0;
30
31 // Determine channel mode (PCV is default)
32 int mode = 1;
33
34 // Determines where in the "mobility" script we are
35 int state = 0;
36
37 // Check to see if mode has been selected
38 int check = 0;
39
40 // Stores desired delay
41 float delay = 0.0;
42
43 // Desired distance [m]
44 double distance = 0.0;
45
```

```
46  // Desired velocity [m/s]
47  float velocity = 0.0;
48
49  // Used for wrap around
50  float counter = 0.0;
51
52  // Length of mobility
53  float times = 1.0;
54
55  // Desired delay in samples
56  float samples_delay = 0;
57
58  // Cubic interpolation variables
59  int M1,M2,M3,M4;
60  float kfrac = 0.0;
61  float lslope, linterp, rslope, rinterp;
62  float w1 = 0.0;
63  float w2 = 0.0;
64  float p = 0.0;
65  float psm1 = 0.0;
66  float lc[4] = {0.0,0.0,0.0,0.0};
67  float rc[4] = {0.0,0.0,0.0,0.0};
68
69  // Sampling frequency and inverse
70  float fs = 96000.0;
71  float fs_inv = 0.0000104166666666666;
72
73  // Input/output gain scale factor
74  float invMAX = 0.000030517578125;
75
76  // Mobility scripts
77  float distance_script[8] = {0, 0, 1.9538773, 1.9538773,
78                              3.3404998, 3.3404998, 4.97923558, 4.97923558};
79  float velocity_script[8] = {0.0, 0.3, 0.3, 0.0, 0.0, -0.3, -0.3, 0.0};
80  float acceleration_script[8] = {0 , 5, -5, -5, 5, 0, 0, 0};
81
82  // Codec read/write variables
83  union {Uint32 combo; short part[2];} data1;
84  union {Uint32 combo; short part[2];} data2;
85
86  // Left/right channel input buffers
87  float leftchannel[N]={0};
88  float rightchannel[N]={0};
89
90  // Codec configuration with default settings
91  DSK6713_AIC23_CodecHandle hCodec;
92  DSK6713_AIC23_Config config = DSK6713_AIC23_DEFAULTCONFIG;
93  interrupt void serialPortRcvISR(void);
94
95
96  // ------------------------------------------------------------------------
97  // End of Declare Variables
98  // ------------------------------------------------------------------------
99
100
101 void main()
102 {
103         // Initialize the board support library
104         DS6713_init();
105
106         // Open codec and get handle
107         Codec = DSK6713_AIC23_openCodec(0, &config);
108
109         // Set codec sampling frequency
110         DSK6713_AIC23_setFreq(hCodec, DSK6713_AIC23_FREQ_96KHZ);
```

```
111
112         // Configure buffered serial ports for 32 bit operation
113         MCBSP_FSETS(SPCR1, RINTM, FRM);
114         MCBSP_FSETS(SPCR1, XINTM, FRM);
115         MCBSP_FSETS(RCR1, RWDLEN1, 32BIT);
116         MCBSP_FSETS(XCR1, XWDLEN1, 32BIT);
117
118         // Check DIP switches to determine which channel to operate as
119         while(check==0)
120         {
121                 // DIP0 is down, DIP1 is up, turn on LED0, PCP channel
122                 if (DSK6713_DIP_get(0) == 0 && DSK6713_DIP_get(1)==1)
123                 {
124                         DSK6713_LED_on(0);
125                         mode = 1;
126                         check = 1;
127                 }
128                 // DIP0 is up, DIP1 is up, turn on LED1, PCV channel
129                 if (DSK6713_DIP_get(0) == 1 &&  DSK6713_DIP_get(1)==1)
130                 {
131                         DSK6713_LED_on(1);
132                         mode = 2;
133                         check = 1;
134                 }
135                 // DIP0 is up, DIP1 is down, turn on LED2, PCA channel
136                 if (DSK6713_DIP_get(0) == 1 &&  DSK6713_DIP_get(1)==0)
137                 {
138                         DSK6713_LED_on(2);
139                         mode = 3;
140                         check = 1;
141                 }
142         }
143
144         // Setup interrupt
145         IRQ_globalDisable();
146         IRQ_nmiEnable();
147         IRQ_map(IRQ_EVT_RINT1,15);
148         IRQ_enable(IRQ_EVT_RINT1);
149         IRQ_globalEnable();
150
151         // Main loop, do nothing and wait for interrupt
152         while(1)
153         {
154         }
155 }
156
157 interrupt void serialPortRcvISR()
158 {
159         // Read left/right channels from codec
160         data1.combo = MCBSP_read(DSK6713_AIC23_DATAHANDLE);
161
162         // Store left/right input to buffers
163         leftchannel[j]=(float)(data1.part[1])*invMAX;
164         rightchannel[j]=(float)(data1.part[0])*invMAX;
165
166         // We are in state 0
167         if(counter < times)
168                 state = 0;
169
170         // We are in state 1
171         if(counter > times - 1 && counter <= 2*times-1)
172                 state = 1;
173
174         // We are in state 2
175         if(counter > 2*times-1 && counter <= 3*times-1)
```

```
176                 state = 2;
177
178         // We are in state 3
179         if(counter > 3*times-1 && counter <= 4*times-1)
180                 state = 3;
181
182         // We are in state 4
183         if(counter > 4*times-1 && counter <= 5*times-1)
184                 state = 4;
185
186         // We are in state 5
187         if(counter > 5*times-1 && counter <= 6*times-1)
188                 state = 5;
189
190         // We are in state 6
191         if(counter > 6*times-1 && counter <= 7*times-1)
192                 state = 6;
193
194         // We are in state 7
195         if(counter > 7*times-1)
196             state = 7;
197
198         // piecewise constant position channel
199         if (mode==1)
200         {
201           distance = distance_script[state];
202         }
203
204         // piecewise constant velocity channel
205         if (mode==2)
206         {
207           distance += velocity_script[state]*fs_inv;
208         }
209
210         // piecewise constant acceleration channel
211         if (mode==3)
212         {
213                 velocity += acceleration_script[state]*fs_inv;
214                 distance += velocity*fs_inv + 0.5*acceleration_script[state]*
215                 fs_inv*fs_inv;
216         }
217
218         // Calculate desired delay
219         delay = distance*0.00294117647;
220         samples_delay = delay*fs;
221         k = (int)samples_delay;
222
223         // -------------------------------------------------------
224         // START: Cubic Interpolation
225         // -------------------------------------------------------
226         p = (k+1)-samples_delay;
227         psm1 = p*p-1;
228         M1 = j-(k+2);
229         M2 = j-(k+1);
230         M3 = j-k;
231         M4 = j-(k-1);
232
233         while (M1<0) M1 += N;
234         while (M1>N-1) M1 -= N;
235         while (M2<0) M2 += N;
236         while (M2>N-1) M2 -= N;
237         while (M3<0) M3 += N;
238         while (M3>N-1) M3 -= N;
239         while (M4<0) M4 += N;
240         while (M4>N-1) M4 -= N;
```

```
241
242         for (i=0;i<=3;i++)
243         {
244                 if (M1+i>N-1) M1 -= N;
245                 lc[i] = (float)leftchannel[M1+i];
246                 rc[i] = (float)rightchannel[M1+i];
247         }
248
249         linterp = -p*(p-1)*(p-2)*(float)leftchannel[M1]/6 + (p*p-1)*(p-2)*
250         (float)leftchannel[M2]/2 - p*(p+1)*(p-2)*(float)leftchannel[M3]/2 +
251         p*(p*p-1)*(float)leftchannel[M4]/6;
252         rinterp = -p*(p-1)*(p-2)*(float)rightchannel[M1]/6 + (p*p-1)*(p-2)*
253         (float)rightchannel[M2]/2 - p*(p+1)*(p-2)*(float)rightchannel[M3]/2 +
254         p*(p*p-1)*(float)rightchannel[M4]/6;
255
256         // -------------------------------------------------------
257         // END: Cubic Interpolation
258         // -------------------------------------------------------
259
260         // Set data2 to zero
261         data2.combo=0;
262
263         // Setup left/right channel output
264         data2.part[1] = (short)(MAX*linterp*0.99);
265         data2.part[0] = (short)(MAX*rinterp*0.99);
266
267         // increment buffer index and handle wraparound
268         j++;
269         if (j>=N)
270         {
271                 j-=N;
272         }
273         t++;
274         if(t>=T)
275         {
276                 t-=T;
277                 counter = counter + 1;
278                 if(counter == 8*times)  counter = 0;
279         }
280
281         // Output left/right channels to codec
282         MCBSP_write(DSK6713_AIC23_DATAHANDLE,data2.combo);
283 }
284
```

Appendix B

Maximum Likelihood Estimation

The following MATLAB code is used to determine the maximum likelihood estimation of frequency and phase for both channels of a stereo *.wav file recording as described in [13].

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Maximum likelihood estimation (MLE) for frequency and phase
% James McGinley
% Dec. 1, 2006
% Example for a *.wav file with a left/right channel frequency
% of 1814/2721 respectively
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% clear all variables and the command window
clear all;
clc;

% read in a *.wav file
[x,fs,nbits] = wavread('z:\Thesis\Phase Stuff\1290.wav');

% estimate left/right channel frequency
f_est_left = 1814;
f_est_right = 2721;

% define range of frequencies to check
range_left = 0.001;
range_right = 0.001;

% define desired precision
frequency_per_left = 0.0000005;
frequency_per_right = 0.0000005;

% separate left/right channels and clip
x1r = x(2.0e5 : 2.8e5, 1)';
x2r = x(2.0e5 : 2.8e5, 2)';

```

```
% scale left/right to have equal amplitudes
x1r = x1r-mean(x1r);
x2r = x2r-mean(x2r);
x1r = x1r/sqrt(sum(x1r.*x1r))*sqrt(sum(x2r.*x2r));

% setup vectors
N = length(x1r);
n = 0:N-1;

% setup test vectors
ftest_left = f_est_left*(1+[-range_left:frequency_per_left:range_left]);
I_left = zeros(size(ftest_left));

ftest_right = f_est_right*(1+[-range_right:frequency_per_right:range_right]);
I_right = zeros(size(ftest_right));

% Find value that maximizes I (left and right)
wb = waitbar(0,'Please wait as I calculate frequency, phase, and time delay...');
for k = 1:length(ftest_left),
    I_left(k) = (1/N)*abs(x1r*exp(-j*2*pi*ftest_left(k)/fs*n(:)));
    waitbar(k/(length(ftest_left)+length(ftest_right)),wb)
end

for k = 1:length(ftest_right),
    I_right(k) = (1/N)*abs(x2r*exp(-j*2*pi*ftest_right(k)/fs*n(:)));
    waitbar((k+length(ftest_left))/(length(ftest_left)+length(ftest_right)),wb)
end
close(wb)

% find index where I is maximized (left and right)
[value_left,index_left] = max(I_left);
[value_right,index_right] = max(I_right);

% find MLE frequency (left and right)
frequency_left = ftest_left(index_left)
frequency_right = ftest_right(index_right)

% find MLE phase (left and right), convert to degrees, and output to command window
phi_hat_left = -atan(sum(x1r.*sin(2*pi*frequency_left/fs*n))/sum(x1r.*cos(2*pi*frequency_left/fs*n)));
phase_estimate_left = phi_hat_left/2/pi*360

phi_hat_right = -atan(sum(x2r.*sin(2*pi*frequency_right/fs*n))/sum(x2r.*cos(2*pi*frequency_right/fs*n)));
phase_estimate_right = phi_hat_right/2/pi*360

% output the phase difference between the two channels to the command window
phase_difference = phase_estimate_left-phase_estimate_right

% determine time delay difference between channels
t_left = phase_estimate_left/360*(1/frequency_left);
t_right = phase_estimate_right/360*(1/frequency_right);

% output the time difference between the two channels to the command window
time_difference_us = (t_left - t_right)*1e6

```

Bibliography

[1] D. R. Brown III, Gregory Prince, and John McNeill. A method for carrier frequency and phase synchronization of two autonomous cooperative transmitters. In *Proceedings of the 5th IEEE Signal Processing Advances in Wireless Communications*, pages 278–282, June 5-8 2005.

[2] Lal C. Godara. Application of antenna arrays to mobile communications, part ii: Beam-forming and direction-of-arrival considerations. In *Proceedings of the IEEE*, volume 85, pages 1195–1245, August 1997.

[3] A. Gersho and B. J. Karafin. Mutual synchronization of geographically separated oscillators. *Bell Systems Technical Journal*, 45:1689–1704, December 1966.

[4] T. Egawa, M. Makimo, and Y. Inoue. A study on master-slave synchronization technique. In *Proceedings of the International Switching Symposium*, page 441, 1976.

[5] D. Mitra. Network synchronization: Analysis of a hybrid of master-slave and mutual synchronization. *IEEE Trans. on Communications*, 28(8):1245–1259, August 1980.

[6] G. Barriac, R. Mudumbai, and U. Madhow. Distributed beamforming for information transfer in sensor networks. In *Information Processing in Sensor Networks (IPSN), Third International Workshop*, pages 81–88, Berkeley, CA, April 26-27 2004.

[7] Y. S. Tu and G. J. Pottie. Coherent cooperative transmission from multiple adjacent antennas to a distant stationary antenna through awgn channels. In *Proceedings of the IEEE Vehicular Technology Conference (VTC)*, volume 1, pages 130–134, Spring 2002.

[8] Yung Szu Tu and Gregory J. Pottie. Coherent cooperative transmission from multiple adjacent antennas to a distant stationary antenna through awgn channels. In *Proceedings of the IEEE Vehicular Technology Conference (VTC)*, volume 1, pages 130–134, Spring 2002.

[9] Simon R. Saunders. *Antennas and Propagation for Wireless Communication Systems*. John Wiley and Sons, LTD, Chichester, 1999.

[10] *TMS30C6713 DSK Technical Reference*, 2003.

[11] Roland E. Best. *Phase-Locked Loops: Design, Simulation, and Applications*. McGraw-Hill, New York, 2003.

[12] Milton Abramowitz and Irene Stegun. *Handbook of Mathematical Functions*. Dover Publications Inc., New York, 1972.

[13] Steven M. Kay. *Fundamentals of Statistical Signal Processing Estimation Theory*. Prentice Hall PTR, New Jersey, 1993.

[14] Alan V. Oppenheim and Ronald W. Schafer. *Discrete-Time Signal Processing*. Prentice Hall, New Jersey, 1989.