Improvements to an Optical Coherence Microscope Through Digital Signal Processing

Tom Driscoll

Senior Thesis

4 May 2001

Physics Department, Harvey Mudd College

Advisors:

Richard Haskell

Dan Petersen.

Abstract

Our current optical coherence microscope (OCM) has been designed for imaging biological samples. We wish to improve the speed, dynamic range, and signal-to-noise ratio of our current instrument. We can achieve these goals by replacing the existing analog signal processing circuitry with a digital processing (DSP) system. We describe the design and performance of our DSP solution. As an added bonus, our solution offers flexibility in the tradeoff between acquisition time and signal-to-noise ratio. The system we implemented utilizes a 2.5 MHz 16-bit analog to digital converter and a 60 MHz Innovative Integration processing board on which C language code runs. The essence of our digital processing is a partial discrete Fourier transform performed by the C-code. The finished digital system integrates very nicely with the existing OCM, and is a valuable improvement to the instrument.

Introduction

History has shown us time and again that advancements in imaging technology have foreshadowed improvements in medical and biological technique. Early in this century, the invention and development of the x-ray allowed doctors to more accurately judge bone breaks and later, with the development of computed axial tomography (CAT) x-ray scans, to create 3-D images of the entire body. The latter half of this century has been witness to an explosion of new concepts in imaging. Of the many recent new techniques, those that are not harmful to a living sample, and can penetrate into an object to produce a 3-D image are of the most interest in medical and biological areas. Magnetic resonance imaging (MRI) is already very widely used in medical and developmental biology fields. Optical coherence microscopy (OCM) is quickly attracting interest due to some of its advantages over MRI. While MRI allows for much larger scan volumes and deeper scan depth than OCM, its restrictive costs and longer scan time are making OCM a preference in many areas. OCM can also achieve higher resolutions than MRI can. Doctors are using OCM to give them information that once required invasive surgery $^{(1-3)}$. Because the scan time for a typical 1 mm^3 image is on the order of minutes, developmental biologists can view cell divisions and other changes inside a living sample⁽⁴⁻⁶⁾.

Our optical coherence microscope [Fig. 1] has been designed as a tool in developmental biology. The microscope's resolution is fine enough to distinguish individual cells of size 5 μ m to 10 μ m. By focusing our 843 nm superluminescent diode (SLD) source beam down to a full-width-half-maximum (FWHM) of 5 μ m, we are able

to achieve 5 µm lateral resolution across the XY plane. Depth (Z-direction) resolution is constrained to 10 µm in tissue by the coherence length of our SLD. A modulating (at 122.3 kHz) reference mirror is used to produce fringes at the output of our interferometer ⁽¹⁰⁾. We are able to circumvent the slow phase drifts inherent in an optical fiber interferometer by summing the powers in the fringe signal at the fundamental (122.3 kHz) and first harmonic (244.6 kHz). Scan times for our OCM are around 6 minutes for a 1 million voxel image.



Figure 1 – Schematic of the HMC OCM

The current OCM setup experiences limitations from the analog filter and rootmean-square (RMS) circuitry. As well as limiting scan times, the analog setup offers very little flexibility in scan options. We have designed and installed a digital signal processing setup based on the following hardware and software components. A high-

speed analog-to-digital (A/D) chip gives us 16-bit sampling at up to 2.5 MHz. After sampling, we perform a discrete Fourier transform (DFT). This DFT, as well as all signal processing and filtering, is then done digitally via software. To aid with computational power, processing is done on a dedicated DSP board manufactured by Innovative Integration. This board includes our A/D chip as well as a 60 MHz Texas Instruments microprocessor. By replacing the old analog section with a digital system, we should be able to improve scan times for a 1 million voxel image to below 1 minute. Digital implementation will also drastically increase the dynamic range of our instrument by eliminating the need for an analog root-mean-square chip. As an added bonus, the new digital setup offers us flexibility in our scan parameters. By choosing to sample over different durations, we can adjust the frequency bandpass of our digital filter. Longer scans offer tighter frequency bandpass, and consequently, less electronic and photon noise is allowed through the filter. This allows us to quickly and easily choose between a rapid scan speed versus a narrow bandpass resulting in a better signal-to-noise ratio. This is an invaluable option for a versatile instrument.

Section 1 of this report discusses the details of Optical Coherence Microscopy as they pertain to our setup. Section 2 looks at the existing analog processing, and discusses its limitations. Section 3 presents our digital solution to these problems. In addition, the reader may find the information presented in the appendixes a valuable tool in reading this report. The appendixes are intended to document the details of technical phenomena important to the design of our instrument. Especially key is Appendix 1, which familiarizes the reader with natural aspects of digitization. Relevant appendixes are referenced throughout the paper at the appropriate points.

Section 1 – An Overview of Optical Coherence Microscopy

Optical coherence microscopy is a technique that uses photon backscattering to gain information about the structure of a sample. In any situation, photons incident on a medium can do one of three things. The photons can be absorbed by the medium, they can be scattered by the medium, or they can be transmitted completely through the medium with no interaction [Figure 2]. Photons can be scattered in any direction, including directly backwards (as is of particular interest to us). We can collect and measure the intensity of these backscattered photons to find the backscattering cross section at any given point (voxel) in the sample.





Figure 2 – The Nature of Backscattering

To collect backscattered photons, we construct a setup based upon a Michelson interferometer [Figure 3]. Light from a superluminescent diode (SLD) source is split by a beam splitter into a reference arm path and a sample arm path. The light in the sample arm is directed through a focusing lens down onto the sample. By translating the position of our focused beam waist relative to the sample (via galvoscanners), we can select a specific sample point in the XY plane. The focusing lens also translates in the Z direction (toward and away from the sample) to set the depth of the focus beam spot. Our resolution in the XY plane is limited by the diameter of the focused beam, and is around 5 μ m (FWHM). Depth resolution is determined by the coherence length [see Appendix 2 for details] of our SLD light source, which is around 15 μ m in air, or 10 μ m in tissue. The 3-D resolution volume around our sample point is called a voxel. We can create a 3-D image of a sample by compiling measurements at many voxels. In our setup, we sweep the x-position to make a row, then increment the y-position to make a plane of rows, then step down in Z to take data in successive planes.



Figure 3 – Michelson Interferometer

The light in the reference arm of the Michelson interferometer is reflected from a mirror mounted on a piezo stack in a cat's-eye retroreflector [Figure 4]. The driven movement of this mirror causes the path length of the light to change as a 122.3 kHz sine with peak-to-peak amplitude 350 nm. This modulated light is then recombined with the backscattered light from the sample that passes back up through the sample arm. The two photon-streams interfere at the photodetector either constructively or destructively, depending on the relative phase between the two - which is changing at 122.3 kHz due to the modulation in the reference arm. The resultant photodetector output is a 122.3 kHz fringe sine wave [see Figure 5], with amplitude proportional to the square root of the intensity of the backscattered light from the sample. We convert this fringe amplitude into a DC voltage via analog circuitry, and this voltage is the output of our instrument. Unfortunately, the actual photodetector signal is not simply a 122.3 kHz sine wave. There is noise in the signal from different sources. We also have to take into consideration sidebands [see Appendix 3] around the 122.3 kHz sine-wave which arise due to amplitude modulation. These sidebands contain true signal information, and must be included in our measurement. However, we want to exclude all frequencies aside from our signal and sidebands, as it contains only white noise. This exclusion is done by means of bandpass filtering. In the next section, I shall talk about the existing analog noise filtering method – especially its design in relation to the nature of signal sidebands.

Phase Modulation at 125 KHz



Figure 4 - Phase Modulation

Fringes at output of Michelson Interferometer



Figure 5 - 122.3 kHz Output fringes (for initial relative phase of zero)

Section 2. – Analog Signal Processing.

To get a good measurement of the backscattering cross-section at our sample point (voxel), we must separate our signal from the background white noise. The simplest way to do this is to construct a twin-peak RLC bandpass filter with its peaks centered around 122.3 kHz and 244.6 kHz. The first harmonic (244.6 kHz) is included to reduce problems associated with phase wander. Thermal drifts of our interferometer lead to undesirable phase wander via expansion and contraction in the optical fibers. Whittier Myers (HMC '99) and others $(^{7,8)}$ found that the sum of the powers in the fundamental and first harmonics would be constant regardless of phase wander². This twin peak filter lets through only those frequencies that contain signal, albeit it does let through the noise at those frequencies. Rather than construct a filter with a very narrow bandpass, as might seem appropriate, we want our filter bandwidth to also include the sidebands. The locations of the sidebands are dependent upon the speed at which we scan [see Appendix 3]. We chose to construct our filter with a bandwidth of 9.5 kHz in the fundamental and 7.5 kHz in the first harmonic ⁽⁹⁾. After filtering, our signal passes through a root-meansquared (RMS) chip. The resultant voltage is the square root of the summed power in the fundamental and first harmonic, and represents the square root of the backscattering coefficient at our sample point.

This analog setup has several limitations. The dynamic range of RMS chips is small (input voltages need to be in the range of roughly 50 mV to 2 V) ⁽⁹⁾ due to the nature of analog multiplication operations. The fixed bandpass circuit also restricts several aspects of operation. Because the total bandwidth has been built to be 17 kHz, we just include sidebands for a scan speed of 2 minutes. If we scan slower than this, the

sidebands are closer to the center frequency, and our filter passes more noise than it has to and we get a poorer signal-to-noise ratio. If we scan faster, the sidebands spread out, and our filter begins cutting out sideband signal. When this happens, we begin to lose information about changes in the backscattering cross-section. The restriction to a single scan speed is a shortcoming of our analog filtering design. We found that changing to digital signal processing alleviates problems with dynamic range and static filter bandwidth.

Section 3. - Digital Signal Processing.

The heart of our digital solution is the Innovative Integration M44 board with its AIX module. The M44 board itself has a dedicated 60 MHz Texas Instruments TMS320C44 processor. The AIX module contains a high-speed (2.5 MHz) 2 channel 16-bit analog-to-digital chip. The amplified output from our photodector is sent directly to the channel 0 digitizing input on the AIX module. We sample our input at 8 times our signal modulating frequency, or 978.4 kHz.

This digital signal is stored in a hardware buffer on the AIX module, which is read out on command into a memory array. From here, processing is done by software. C code is executed on the M44 board processor. The code first extracts the channel-0 16bit word from the paired 2-channel 32-bit word read from the hardware buffer. Then a discrete Fourier transform (DFT) extracts those components of the signal around 122.3 kHz and 244.6 kHz [see Appendix 6 for C code].

The bandwidth of the DFT is primarily dependent on the total duration time of sampling [see Appendix 1], which can be expressed as 1/NT where N is the total number of samples and T is the sample interval. Because the position of sidebands [see Appendix 3] is also dependent on the scan speed, it is possible to set the total sampling duration such that our digital bandpass will always just include the sidebands regardless of scan speed. This is a remarkable improvement over the fixed bandwidth analog setup. In our setup we have a sampling rate of 978.4 kHz, and 32 samples per 50 µs voxel. A voxel dwell time of 50 µs means that we will have amplitude modulation of period 100 µs.

 $\pm 1 / 100 \ \mu s. = \pm 10 \ kHz.$

around both the fundamental and first harmonic. The bandwidth of our digital filter can be found by dividing the nyquist frequency by the number of frequency information points as

489.2 kHz / 16 points = 30.6 kHz.

This bandwidth is more than enough to include our sidebands.[see Figure 6].



Figure 6 - DFT bandwidth and signal sidebands. 50 µs voxels.

After the DFT has extracted the 122.3 and 244.6 kHz Fourier components, our software squares the result from each frequency, sums the two together, and takes the square root of the result. This result is the fringe amplitude as found from the power in the fundamental and first harmonic. This is directly proportional to the square root of the backscattering coefficient at our sample point (voxel). Because this RMS calculation is

done in digital, there are none of the dynamic range problems associated with an analog RMS chip.

Timing.

The master timing and control of the instrument is ultimately controlled by a LabVIEW VI running on the host Pentium III computer. LabVIEW controls the AT-MIO-16XE-10 multifunction I/O board, which creates a number of waveforms used for timing and control of the various aspects of the instrument. The steering of the focused beam across the X-Y plane is controlled by galvoscanners, whose angular orientation is determined by waveforms from the AT-MIO-16XE-10 board for X and the AT-AO-6 board for Y. The positions of the reference arm mirror and the focusing lens are controlled by dc-motor-translators. LabVIEW-controlled pulses from the AT-MIO-16XE-10 board also mediate the interaction between the host computer and the operation of the M44 board. Because the software running on the M44 board controls when the AIX module is sampling the input signal, it is important that the M44 software knows when the LabVIEW VI is beginning each voxel. This is accomplished by means of an external interrupt signal sent from the AT-MIO-16XE-10 board to the AIX module on connector P2 [see Figure 7 of the M44 board]. The AT-MIO-16XE-10 board generates a short TTL-like pulse at the beginning of every voxel (called SCANCLK) [see Figure 8]. The software is constantly monitoring this SCANCLK, and when one is received, the software reads out the data in the AIX buffer (from the last voxel) into an array, clears the AIX buffer, and begins sampling data for the current voxel. While the AIX is sampling for the current voxel, the software is performing the DFT and RMS calculations on the

previous voxel's data. The result of these calculations is the voxel fringe amplitude, and it is stored in an array in SRAM on the M44 board. During the processing, the software continues monitoring the external interrupt. If another external interrupt is received before the software has completed its calculations, the next voxel has already started before the software is ready. In this event, an error is logged.



Figure 7 – M44 board and ATX module.

This is the essential speed limitation of our digital processing system – can the software finish bleeding and clearing the AIX buffer, formatting the data array, and calculating the DFT of that data array before the next voxel starts? Because it's so important to be sure that we're not introducing voxel errors at whatever scan speed we choose, a great deal of time was spent looking at the timing of the code and its interaction with instrument timing. Figure 8 shows the analog out waveforms produced by the AT-

MIO-16XE-10 board that are used to control the x and y galvoscanners. Figure 9 shows both the external interrupt waveform as well as how long the sections of the code take. Both figures are for 32 samples per voxel, with 50 μ s voxels – but the hardware and software configurations are easily variable for longer voxel dwell times.



Figure 8 – Timing Analog Waveforms.



Figure 9. DFT version 3.3t Timing Diagram

Scan Speed Flexibility.

We have seen that a voxel dwell time of 50 μ s allows us to perform a 32 sample DFT within the time constraints. If we want, we can also spend longer than 50 μ s per voxel and average out some of the photon noise. This allows us to take more than 32 samples per voxel. The runtime for the software scales linearly with the number of samples - as is apparent in Figure 10.



Figure 10- runtimes for different (samples / voxel)

The position of the sidebands is also inversely linearly dependent on our voxel dwell time. So if we decrease our scanning speed by the same factor we increase our number of samples, our filter bandpass retains its proportionality with respect to our sidebands. The net effect of scanning slower is that we have sidebands closer to our center frequencies, and we have a tighter bandpass that allows in less noise – giving us a better signal-to-noise ratio. The flexibility to select the tradeoff between scan speed and signal-to-noise ratio is a significant advantage of our digital processing setup.

Appendix 1. Aspects of Digital Signal Processing.

Nyquist frequency. For a given sampling rate R, the highest frequency that can be resolved in the digital information is $\frac{1}{2}R$. This is known as the Nyquist frequency [see Figure 11]. This means that for us to just resolve our 244.6 kHz first harmonic, we must sample at 489.2 kHz or higher. To include the first upper sideband on the first harmonic, we must sample faster than 489.2 kHz. Our sampling frequency is 978.4 kHz, more than covering the Nyquist requirements. Sampling at higher frequencies also reduces the noise error associated with digitization.



Figure 11 - Nyquist frequency = $\frac{1}{2}$ sampling frequency.

Digital frequency combs and DFT bandwidth

The nature of a Fourier transform is such that two points in time space provide the necessary information for one point in the frequency space. Thus when we take our set of discrete information in the time domain and perform a DFT on it, the result is a set of discrete information in the frequency domain. The set of frequency information points

spans from 0 Hz (DC) to the Nyquist frequency. The points of this frequency comb are evenly spaced, so the distance between each frequency point can be found as:

 $\nabla v = 1/NT = \frac{1/T}{N} = \frac{Sampling Frequency}{Number of Samples}$

In our setup, this gives us a comb bandwidth of 30.6 kHz. The result is a division of frequency space like the one shown below [see Figure 12]. Our filtering is done by including the information in certain bands, and excluding others. The grey areas show the bandwidth that is allowed through filtering. Included in red are the sidebands for a 50µs voxel dwelltime.



Figure 12 - DFT bandwidth with sidebands

Appendix 2. On Coherence Length.

Light arrives in wave packets whose temporal duration is inversely proportional to their frequency bandwidth as.

$$\Delta v \approx 1/\Delta t^{(11)}$$

This is a phenomenon we expect from knowledge of Fourier series. We know that the atoms responsible for the electron transitions that emit our photons are undergoing random thermal motion. When these atoms collide with other atoms, they suffer interruptions that lead to discontinuities in the wavetrain. Thus, successive wave packets will not necessarily have reliably similar phases. Thus, if the average duration of a single wave packet is t_c, two points on a waveform that lie greater than t_c away from each other must be on separate wave packets, and cannot be assumed to have definite relative phase. This coherence time t_c directly relates to a coherence length through the speed of the wave as:

$l_{c} = c t_{c}^{(11)}$

Thus in our interferometer, if the path difference between the reference and sample arm is greater than the coherence length of our SLD source, the relative phase of wave packets arriving from one arm will be statistically uncorrelated to the phase of wave packets arriving from the other arm, and the two will not coherently interfere to produce fringes. The bandwidth of our SLD source is approximately 20.7 nm FWHM. This results in a coherence length of approximately 15 μ m in air. This coherence length is the primary determining factor in our depth resolution. Once the beam waist is set to a certain sample

depth, the reference arm retroreflector is translated such that there is zero path length difference between the two arms. Thus photons backscattering from more than the coherence length before or after the beam waist will not interfere with the light from the reference arm, and so will not contribute to fringe amplitude.

Appendix 3. Sidebands

If our backscattering cross-section is constant across a scan, the only components of our signal we need be concerned with are at 122.3 and 244.6 kHz. However, if the backscattering index varies, the amplitude of the OCM output fringe signal varies. In frequency space, a constant amplitude sine wave has components only at its center frequency. But if the sine wave's amplitude starts changing in time, it introduces new frequency components called sidebands. The location in frequency space of these sidebands depends on how quickly the amplitude of the sine wave is varying. In our case, we know the maximum rate of change of our sine wave's amplitude. For purposes of sideband calculation, we assume that in our sample we will alternate between sample points of high backscattering and sample points of low backscattering. This is a worst case scenario leading to highest frequency sidebands. The position of these primary sidebands is ± the frequency of amplitude modulation [see Figure 13].

To give an example [Figure 13], we have chosen our voxel dwell time to be 50 μ s. This corresponds to an amplitude modulation period of 100 μ s, which is an amplitude modulation frequency of 10 kHz. We will thus find our primary sidebands at 122.3 kHz ± 10 kHz, and also at 244.6 kHz ± 10 kHz – as is illustrated below.



B

Figure 13 – Sidebands

Appendix 4: Testing Setup

The test setup for debugging and timing the code simulated the instrument inputs to the computer. I used two Hewlett Packard function generators, one to create a simulated SCANCLK external interrupt, and one to create a 122.3 kHz pure sine wave to simulate a fringe signal. I input the 122.3 kHz simulated signal in pin 9 and pin 1(ground) of P1, one of two fifteen pin connectors on the back of the M44 board [see Figure 7]. The external interrupt was wired to P2 between pins 7 and 15 (digital ground) in a similar manner. By increasing the speed of the simulated external interrupt coming from my function generator, I was able to determine the fastest external interrupt (at a given number of samples) for which the code would not yet begin to report that it was receiving the next external interrupt before it completed running [see Appendix 5 for timing diagrams]. Also of interest is the fact that after reading and clearing the AIX FIFO, it takes longer to read in the new samples than it does to complete the rest of the software processing. This means that as we steadily increase the external interrupt frequency we will notice a drop in the mean voxel intensity before we begin to see voxel overtime errors. This is because we are cutting short the AIX sampling, leaving 0-values as several of the last samples. In my timing, I measured both the time it takes to complete each section of the signal processing code as well as the point at which we start cutting short the AIX sampling. We can be sure then that if we are scanning with voxel dwell times longer than the runtimes indicated in Appendix 5, we will avoid voxel overtime errors, and ensure ample time to complete sampling.

















Appendix 6 – Software Code.

// DSP33t.c

// This program controls the M44 digital signal processing board including // its AIX analog-to-digital conversion module. It initializes all relevant // hardware devices and begins sampling, processing, and transferring data to // the host PC upon recognition of the receipt of a hardware-generated interrupt // (SCANCLK) from the National Instruments AT-MIO-16XE-10 board. $\ensuremath{\prime\prime}\xspace$ // This program must be compiled for the Texas Instruments TMS320C44 processor // using the Texas Instruments C compiler version 5.1 or higher. The compiled // and linked COFF???? output file must be downloaded to the M44 board using // the program Terminal.exe supplied by Innovative Integration // or using the Download_to_M44.vi LabView VI. // Version 1.0 created 7/8/1999 by Whittier Myers // Version 1.1 revised 7/9/1999 by Aaron Boyer // Version 1.2 revised 7/12/1999 by Aaron Boyer // Version 2.0 revised 7/15/1999 by Aaron Boyer and Whittier Myers // Version 2.1 revised 7/15/1999 by Aaron Boyer and Whittier Myers // This version is partially optimized for speed and works. // Version 3.0 revised 7/22/1999 by Aaron Boyer // This version is partially optimized for speed, and includes // working host-target communication. // Version 3.1 revised 7/26/1999 by Andrew Harrington // This version puts g_plane_of_voxel_intensities[] into M44 Global RAM // and fixes target-to-host transfer of the array. // Aaron's note on version 3.1: making g_plane_of_voxel_intensities // a pointer and assigning it to the first address in global memory // caused it to overwrite other variables in global memory when data // was directly stored in that, or subsequent addresses (see memory // notes below). // Version 3.2 revised $7/27/1999\ by Aaron Boyer$ // This version features fully optimized DFT code, and makes // g_plane_of_voxel_intensities an array again. // Version 3.3 revised March and April, 2001 by Tom Driscoll and Richard Haskell // This version fine tunes verion 3.2, eliminating erratic behavior // during testing and timing of the code. Comments were cleaned up, // and some historical debris was eliminated. Some corrections were // made to the statistical function which is useful during testing. // We note that during timing of the code the first indication that the $\ensuremath{{\prime}}\xspace$ in time is that a voxel value // is reported as (oddly) low. This is because there is insufficient time // to acquire the last AIX sample - instead the sample is taken as zero $\ensuremath{{\prime}}\xspace$ // because the FIFO was cleared during the previous readout. When the // external interrupts are even more closely spaced in time, the DFT does // not have time to execute and the failure counter is incremented and // reported. // A few notes on style: functions are CapitalizedUnspaced, error codes are // CapitalizedUnspaced, variables are lowercase_underscore_spaced, $\ensuremath{\prime\prime}$ global variables begin with "g_", function input variables begin with // "in_", and function output variables begin with "out_". // Memory notes: // Because this version uses a large value for MAX_SAMPLES_PER_VOXEL, it must be // linked with a large stack size, like 0x2000 (hex 2000 = decimal 8192). // ** This version is linked with all variables placed in global memory. ** #include "c:\m44cc\include\target\stdio.h" #include "c:\m44cc\include\target\periph.h" #include "c:\fltc\include\math.h" #include "c:\OCMDSP\DSPErrorCodes.h" #define PI 3.14159265359

// to convert the AIX A/D integer result to an rms voltage. Note that // The AIX module converts the maximum +- 2 Volt signal (4 Volt // peak-to-peak) to the maximum 16 bit signed integer (2^15 with the // appropriate sign in the sixteenth bit). The sqrt(2) converts to an // rms value. #define __INLINE static inline #define PIEZO_FREQUENCY 122300 // This is the piezo driving frequency in Hz. #define VOXELS_PER_PLANE 10998 // A plane is 100 by 100 voxels, with 10 extra // y rows on the top to make sure that the // galvoscanners are functioning appropriately. #define MAX_SAMPLES_PER_VOXEL 1024 // This is the maximum number of samples per // voxel the program is designed to handle. // These next two constants set g_samples_per_piezo_period and // g_piezo_periods_per_voxel (declared below). Eventually, LabView will be // passing the values for those variables instead of having them set by these // constants. #define SAMPLES_PER_PIEZO_PERIOD 8 // This is the number of samples we take per piezo period. #define PIEZO_PERIODS_PER_VOXEL 4 // This is how many piezo periods we sample over. // The next four constants are involved in the DMA transfer of // g_plane_of_voxel_intensities[] to the PC. // Out mailbox of M44. #define OUTBOX 0 // In mailbox of M44. #define INBOX 1 #define DMA_CHANNEL 0 // Channel for DMA transfer. #define FLAG 0x01 // This code is sent by the LabVIEW VI. #define DONE 0x11 // This code is sent by the M44 on completion // of the DMA transfer. float g_plane_of_voxel_intensities[VOXELS_PER_PLANE]; // This is an array in which to store the voxel intensities // for the current scan plane. It will be of size VOXELS_PER_PLANE. unsigned int *g_pc_shared_memory_buffer; // This is the address in PC RAM into which the M44 will // transfer the plane of voxel intensities. unsigned int g_external_interrupt_bit; $\ensuremath{{\prime}}\xspace$ // This will be used to store the bit number which corresponds // to the external interrupt we're using. unsigned int g_host_interrupt_bit; // This will be used to store the bit number which corresponds // to the host-generated interrupt we're using. // These next three global variables are initialized by calling // InitializeSineTables(). Currently, these values are set to the same values // as the #define constants above. Eventually, we want these values to be // parameters passed from LabView to the DSP. unsigned int g_samples_per_piezo_period; // This is the number of samples we take per piezo period. unsigned int g_piezo_periods_per_voxel; // This is how many piezo periods we sample over. unsigned int g_samples_per_voxel = SAMPLES_PER_PIEZO_PERIOD * PIEZO_PERIODS_PER_VOXEL; // This is the total number of samples per voxel. float g_voltage_scalar; // We set this value equal to VOLTAGE_SCALAR / q_samples_per_voxel // in InitializeSineTables. float g_sine_fundamental_table[MAX_SAMPLES_PER_VOXEL]; // These four arrays float g_cosine_fundamental_table[MAX_SAMPLES_PER_VOXEL]; // will be sine and float g_sine_first_harmonic_table[MAX_SAMPLES_PER_VOXEL]; // cosine tables for float g_cosine_first_harmonic_table[MAX_SAMPLES_PER_VOXEL]; // use in the discrete

// Fourier transform.

void StatisticalAnalysis(void); // This function performs statistical analysis of the data produced // by the board for testing purposes. void ReportError(int in_error_code); // This function reports a fatal error to the user. It currently // calls printf to write the error number to the console. void Initialize(void); // This function is called at the beginning of the program to perfom // initialization. void InitializeSineTables(unsigned int in_samples_per_piezo_period, unsigned int in_piezo_periods_per_voxel); // This function builds the sine and cosine tables for both the // fundamental frequency and the first harmonic. // in_samples_per_piezo_period and in_piezo_periods_per_voxel have // the same values as their global variable counterparts above. The // sine tables have the size // in_samples_per_piezo_period * in_piezo_periods_per_sample; // this value should not be larger than MAX_SINE_TABLE_LENGTH. int ReadExternalInterruptFlag(void); // This function reads the current state of the external interrupt // flag. int ReadHostMailboxFlag(void); // This function reads the current state of the host mailbox flag. void ResetExternalInterruptFlag(void); // This function resets the external interrupt flag to its // untriggered state. void ResetHostMailboxFlag(void); // This function resets the host mailbox flag to its untriggered // state. void ConvertAIXBufferToFloatArray(unsigned int in_g_samples_per_voxel, unsigned int* in_aix_buffer, float* out_converted_aix_data); // This function takes the first 16 bits of each value in // in_aix_buffer and converts it into a floating point value. // in_samples_per_voxel -- the number of samples we want to read from the // aix buffer // in_aix_buffer -- the raw output from AIX_bleed_fifo() // out_converted_aix_data -- outputs the float values of the AIX // data void SendPlaneToLabview(int in_error); $\ensuremath{{\prime}}\xspace$ // This function uses DMA to send a finished plane scan to LabView. // If inError reports an error, the program should indicate to // LabView that something is wrong. float ReadAndProcessVoxel(void); $\ensuremath{{\prime}}\xspace$ // This function processes a single voxel by reading buffered data // from the AIX module, restarting the AIX's sampling, and performing // a partial discrete Fourier transform on the data to find the // power in the fundamental piezo frequency component and the first // harmonic frequency component. It returns the voxel's intensity // in IEEE floating point format?????? _INLINE void StatisticalAnalysis(void) // Declare variables: float sort_switcher; int sort_checker; float sig_fig_checker; float sorted_plane[VOXELS_PER_PLANE]; float mean_value = 0; int array_index; int count; int min_voxel_index, max_voxel_index; float min_voxel_value = 100, max_voxel_value = -100;

```
// Find and print the mean value of the data set, and prepare
// the data set to be sorted:
for (count = 0; count < VOXELS_PER_PLANE; count++)
{</pre>
```

```
mean_value += g_plane_of_voxel_intensities[count];
       sorted_plane[count] = g_plane_of_voxel_intensities[count];
}
mean_value = mean_value / VOXELS_PER_PLANE;
printf("mean voxel value is: %f\n", mean_value);
printf("second voxel contains: %f\n\n", g_plane_of_voxel_intensities[1]);
// Find min_voxel_index and max_voxel_index by stepping through the voxel array.
for (count = 0; count < VOXELS_PER_PLANE; count++)</pre>
{
       if (g_plane_of_voxel_intensities[count] < min_voxel_value)
       {
               min_voxel_value = g_plane_of_voxel_intensities[count];
               min_voxel_index = count;
       if (g_plane_of_voxel_intensities[count] > max_voxel_value)
       {
               max_voxel_value = g_plane_of_voxel_intensities[count];
               max_voxel_index = count;
       }
printf("min_voxel_index is: %d\t min_voxel_value is: %f\n", min_voxel_index,
               g_plane_of_voxel_intensities[min_voxel_index]);
printf("max_voxel_index is: %d\t max_voxel_value is: %f\n\n", max_voxel_index,
               g_plane_of_voxel_intensities[max_voxel_index]);
// Sort the data set from least to greatest:
sort_checker = 1;
while (sort_checker != 0)
{
       sort_checker = 0;
       for (count = VOXELS_PER_PLANE - 1; count > 0; count--)
       ł
               if (sorted_plane[count] < sorted_plane[count - 1])</pre>
               {
                       sort_switcher = sorted_plane[count];
                       sorted_plane[count] = sorted_plane[count - 1];
                       sorted_plane[count - 1] = sort_switcher;
                       sort_checker++;
               }
       }
}
// Display the results of the sort:
array_index = 0;
sig_fig_checker = mean_value - (mean_value / 100);
while (sig_fig_checker > sorted_plane[array_index])
{
       array_index++;
}
printf("%d voxels were more than 1 percent below the mean.\n", array_index);
array_index = VOXELS_PER_PLANE - 1;
sig_fig_checker = mean_value + (mean_value / 100);
while (sig_fig_checker < sorted_plane[array_index])</pre>
{
       array_index--;
}
printf("%d voxels were more than 1 percent above the mean.\n\n",
                                       VOXELS_PER_PLANE - 1 - array_index);
array_index = 0;
sig_fig_checker = mean_value - (mean_value / 1000);
while (sig_fig_checker > sorted_plane[array_index])
{
       array_index++;
}
printf("%d voxels were more than 0.1 percent below the mean.\n", array_index);
```

```
array_index = VOXELS_PER_PLANE - 1;
    sig_fig_checker = mean_value + (mean_value / 1000);
    while (sig_fig_checker < sorted_plane[array_index])
    {
            array_index--;
    }
    printf("%d voxels were more than 0.1 percent above the mean.\n\n",
                                                            VOXELS_PER_PLANE - 1 - array_index);
    array_index = VOXELS_PER_PLANE * 0.5;
    printf("median value: %f\n\n",
                     (sorted_plane[array_index] + sorted_plane[array_index - 1]) * .5);
    printf("min value: %f\n", sorted_plane[0]);
    array_index = VOXELS_PER_PLANE * .25;
    printf("25 percent: %f\n", sorted_plane[array_index]);
    array_index = VOXELS_PER_PLANE * .4;
    printf("40 percent: %f\n", sorted_plane[array_index]);
    array_index = VOXELS_PER_PLANE * .6;
    printf("60 percent: %f\n", sorted_plane[array_index]);
    array_index = VOXELS_PER_PLANE * .75;
    printf("75 percent: %f\n", sorted_plane[array_index]);
    printf("max value: %f\n\n", sorted_plane[VOXELS_PER_PLANE - 1]);
}
 _INLINE void ReportError(int in_error_code)
    printf("Error %d occurred.\n",in_error_code);
    while (TRUE)
    ł
            \ensuremath{{\prime}}\xspace ) Do nothing so the user can see the error message.
}
  INLINE void Initialize(void)
    float dds_clock_frequency = PIEZO_FREQUENCY * SAMPLES_PER_PIEZO_PERIOD * 8;
                    // This variable is used below to set the timebase of the DDS clock,
                    // the timer for the AIX.
    //disable_interrupts(); Do we need this command?????? I don't know!
                    // This function causes the m44 to cease responding to processor
                    \ensuremath{{\prime}}\xspace in terrupts. It is called from chip.h, which is included in
                    // misc.h, which is included in periph.h, which is included above.
    ResetHostMailboxFlag();
    InitializeSineTables(SAMPLES_PER_PIEZO_PERIOD, PIEZO_PERIODS_PER_VOXEL);
            //g_plane_of_voxel_intensities = (void *)(&Periph->GlobalRam[0]);
            // Point voxel array to base of Global RAM.
    timebase(DDS_TIMER, dds_clock_frequency, DDS_TIMEBASE);
            // This sets up the AIX's timer.
                                           // This gates OFF all FIFO buffers,
    AIX_gate(0, OFF);
                                                           // preventing the storage of samples.
    AIX_reset_fifo(0);
                                           // This resets the FIFO buffers on the AIX
                                                           // module installed on site 0.
    AIX_enable_fifo(0, 0, ON); // This starts sampling on AIX channel pair 0.
                                                           // The AIX module is on site 0 on the \rm M44
```

```
// board.
                                          // This gates ON all FIFO buffers,
    AIX_gate(0, ON);
                                                          // enabling the storage of samples.
}
__INLINE void InitializeSineTables(unsigned int in_samples_per_piezo_period,
                                                                  unsigned int
in_piezo_periods_per_voxel)
{
    int count;
    g_samples_per_piezo_period = in_samples_per_piezo_period;
    g_piezo_periods_per_voxel = in_piezo_periods_per_voxel;
    g_samples_per_voxel = g_samples_per_piezo_period * g_piezo_periods_per_voxel;
    g_voltage_scalar = VOLTAGE_SCALAR / g_samples_per_voxel;
                    // g_voltage_scalar will be used in ReadAndProcessVoxel!
    if (g_samples_per_voxel > MAX_SAMPLES_PER_VOXEL)
    {
            ReportError(ERR_maxSampleSequenceLengthExceeded);
    }
    // Fill the sine tables:
    for (count = 0; count < g_samples_per_voxel; count++)</pre>
    {
            g_cosine_fundamental_table[count] =
                           cos(2.0 * PI * count / g_samples_per_piezo_period);
            g_sine_fundamental_table[count] =
                           sin(2.0 * PI * count / g_samples_per_piezo_period);
            g_cosine_first_harmonic_table[count] =
                           cos(4.0 * PI * count / g_samples_per_piezo_period);
            g_sine_first_harmonic_table[count] =
                           sin(4.0 * PI * count / g_samples_per_piezo_period);
    }
}
 _INLINE int ReadExternalInterruptFlag(void)
{
   return PollInterrupt(EINT2_INTERRUPT);
}
  INLINE int ReadHostMailboxFlag(void)
{
    int host_message;
    return(read_mb_terminate(&host_message, 0, INBOX) == FLAG);
}
 _INLINE void ResetExternalInterruptFlag(void)
{
    ClearInterrupt(EINT2_INTERRUPT);
}
  _INLINE void ResetHostMailboxFlag(void)
{
    clear_mailboxes();
}
__INLINE void ConvertAIXBufferToFloatArray(unsigned int in_samples_per_voxel,
                            unsigned int* in_aix_buffer, float* out_converted_aix_data)
{
    AIO_PAIR *typecast_aix_buffer_pointer = (AIO_PAIR*)in_aix_buffer;
            // Change the type of in_aix_buffer so we can access it as two
            // signed 16 bit integers. When optimized, this should not generate
```

```
35
```

```
// an extra instruction.
    float *converted_data_pointer = out_converted_aix_data;
    int count;
    for(count=in_samples_per_voxel - 1; count >= 0; count--)
    {
            *out_converted_aix_data = typecast_aix_buffer_pointer->half.low;
                    // Convert the low 16 bits of in_aix_buffer[count] into a
                    // signed integer, then into a floating point value.
                    // AIO_PAIR is declared in A4D4.h
            out_converted_aix_data++;
            typecast_aix_buffer_pointer++;
    }
}
 _INLINE void SendPlaneToLabview(int in_error)
    ResetHostMailboxFlag();
    write_mailbox(VOXELS_PER_PLANE, OUTBOX);// Send array size to PC via mailbox.
    g_pc_shared_memory_buffer = (unsigned int *)read_mailbox(INBOX);
                                                                   // Receive address of PC shared
RAM.
    // Set up busmaster transfer and send data to host PC shared memory via DMA.
    bm_init(g_pc_shared_memory_buffer, NULL, DMA_CHANNEL);
    bm_transfer((void *)g_plane_of_voxel_intensities, VOXELS_PER_PLANE,
                                                    (void *)0, 0, OUTPUT, VOXELS_PER_PLANE);
    transfer_complete();
                                           // Wait for busmaster transfer to finish.
    write_mailbox(DONE, OUTBOX); // Send done signal to LabVIEW CIN.
}
 _INLINE float ReadAndProcessVoxel(void)
    unsigned int aix_buffer[MAX_SAMPLES_PER_VOXEL + 1];
            // This is the array in which we first store the raw data from the AIX
            // board. Allocate one extra point in order to correct for the
            // AIX_bleed_fifo problem documented below.
    int count;
    float converted_aix_data[MAX_SAMPLES_PER_VOXEL];
            // This is the array in which we will store converted voltage data from
            // the AIX board.
    float dsp_result;
            // This is where we store the result of the DSP's calculations.
    float cosine_of_fundamental = 0;
    float sine_of_fundamental = 0;
    float cosine_of_first_harmonic = 0;
    float sine_of_first_harmonic = 0;
    float square_accumulator;
    float *pointer_to_cosine_fundamental_table = g_cosine_fundamental_table;
    float *pointer_to_sine_fundamental_table = g_sine_fundamental_table;
    float *pointer_to_cosine_first_harmonic_table = g_cosine_first_harmonic_table;
    float *pointer_to_sine_first_harmonic_table = g_sine_first_harmonic_table;
    float *pointer_to_converted_aix_data = converted_aix_data;
    AIX_gate(0, OFF);
            \ensuremath{{\prime\prime}}\xspace // This disables FIFO gating on the AIX module installed on site 0.
            // We must stop the AIX from sampling to ensure that we have accurate
            \ensuremath{\prime\prime}\xspace )/ timing whether or not the AIX board has run out of FIFO buffer space.
    AIX_bleed_fifo(0, 0, aix_buffer, g_samples_per_voxel + 1);
```

```
// This puts the samples from the AIX FIFO buffer into aix_buffer[].
```

```
// We noticed that AIX_bleed_fifo puts 0 into the first element of
        // aix_buffer. Therefore, we read g_samples_per_voxel + 1 points and fix
        // this in the call to ConvertAIXBufferToFloatArray.
AIX_reset_fifo(0);
        // This erases the contents of the AIX FIFO buffer.
AIX_gate(0, ON);
        // This starts the AIX gating data to the FIFO buffer again.
ConvertAIXBufferToFloatArray(g_samples_per_voxel, aix_buffer + 1,
                                                                        converted aix data);
        // We begin converting samples at aix_buffer + 1 in order to correct for
        // the problem with AIX_bleed_fifo documented above.
// Perform a partial Discrete Fourier Transform:
for (count = g_samples_per_voxel; count > 0; count --)
{
       cosine_of_fundamental += *pointer_to_cosine_fundamental_table
                                               * *pointer_to_converted_aix_data;
       pointer_to_cosine_fundamental_table++;
       pointer_to_converted_aix_data++;
}
pointer_to_converted_aix_data = converted_aix_data;
for (count = g_samples_per_voxel; count > 0; count --)
{
       sine_of_fundamental += *pointer_to_sine_fundamental_table
                                                * *pointer_to_converted_aix_data;
       pointer_to_sine_fundamental_table++;
       pointer_to_converted_aix_data++;
}
pointer_to_converted_aix_data = converted_aix_data;
for (count = g_samples_per_voxel; count > 0; count --)
ł
       cosine_of_first_harmonic += *pointer_to_cosine_first_harmonic_table
                                                       * *pointer_to_converted_aix_data;
       pointer_to_cosine_first_harmonic_table++;
       pointer_to_converted_aix_data++;
}
pointer_to_converted_aix_data = converted_aix_data;
for (count = g_samples_per_voxel; count > 0; count --)
ł
       sine_of_first_harmonic += *pointer_to_sine_first_harmonic_table
                                                       * *pointer_to_converted_aix_data;
       pointer_to_sine_first_harmonic_table++;
       pointer_to_converted_aix_data++;
}
square_accumulator = cosine_of_fundamental * cosine_of_fundamental;
square_accumulator += sine_of_fundamental * sine_of_fundamental;
square_accumulator += cosine_of_first_harmonic * cosine_of_first_harmonic;
square_accumulator += sine_of_first_harmonic * sine_of_first_harmonic;
dsp_result = sqrt(square_accumulator);
dsp_result *= g_voltage_scalar;
        // This scales dsp_result to represent an RMS voltage value.
return dsp_result; // to_ieee(dsp_result);
                                       // This converts the floating point value dsp result
                                       \ensuremath{{\prime}}\xspace // to an IEEE standard float.
```

```
void main(void)
```

}

```
{
    unsigned int current_voxel_index = 0;
            // When a new voxel intensity is computed, this index will tell us where
            // in g_plane_of_voxel_intensities to put the new value.
    unsigned int first_pulse_flag = FALSE;
            // This flag records whether we have received the first pulse from the
            // AT-MIO board yet.
    float dsp_result; // the result of the DSP algorithm
    int failure_counter = 0;
    Initialize();
    ResetExternalInterruptFlag();
    while (TRUE)
    {
            if (ReadExternalInterruptFlag())
            {
                ResetExternalInterruptFlag();
                    dsp_result = ReadAndProcessVoxel();
                    if (first_pulse_flag)
                    {
                           if (current_voxel_index < VOXELS_PER_PLANE)
                                   // During testing, this "if" loop provides an
                                   // exit for the program upon completion of a plane.
                           {
                                   *(g_plane_of_voxel_intensities + current_voxel_index) =
            dsp_result;
                                           // This plugs the DSP result directly
                                           // into the voxel plane array.
                                   current_voxel_index++;
                           else
                           {
                                   printf("current voxel index: %d\t voxels per plane: %d\n",
                                                                 current_voxel_index,
VOXELS_PER_PLANE);
                                   printf("%d voxels took too long to process.\n\n",
                                                                                 failure_counter);
                                   StatisticalAnalysis();
                                   ReportError(ERR_bufferOverflow);
                           }
                    }
                    else
                    {
                           first_pulse_flag = TRUE;
                           // This assures that data acquired before the scan began is not
                           // stored. The flag is set true now, since an external
                           // interrupt must have been received for this line to execute.
                    }
                    if (ReadExternalInterruptFlag())
                    {
                           failure_counter++;
                                   //ReportError(ERR_outOfTime);
                                   // If another interrupt has already been triggered,
                                   // then the calculation took too long.
                    }
            }
            if (ReadHostMailboxFlag())
                           // This checks to see if LabView has sent a mailbox request to
                           // mailbox 0 signaling the end of a scan.
```

```
38
```

```
{
                     if (current_voxel_index == (VOXELS_PER_PLANE - 1))
                     {
                             dsp_result = ReadAndProcessVoxel();
                             *(g_plane_of_voxel_intensities + current_voxel_index) =
                              dsp_result;
                             SendPlaneToLabview(ERR_noErr);
                     }
                     else if (current_voxel_index > (VOXELS_PER_PLANE -1))
                     {
                             ReportError(ERR_bufferOverflow);
SendPlaneToLabview(ERR_bufferOverflow);
                     }
                     else
                     {
                             ReportError(ERR_notEnoughPointsInPlane);
                             SendPlaneToLabview(ERR_notEnoughPointsInPlane);
                     }
                     ResetHostMailboxFlag();
           }
    }
}
```

References:

1. Fujimoto, J.G. "Biomedical imaging using optical coherence tomography", <u>Technical Digest Series</u>, <u>Vol.7</u>. 1999 p. 256, 25

2. Rudolph, W.; Kempe, M. "Trends in optical biomedical imaging", <u>Journal of Modern Optics</u>, vol.44, <u>no.9 p. 1617-42</u>, 1997

3. Tuchin, V.V. "Tissue optics: tomography and topography" Proceedings of the SPIE - The International Society for Optical Engineering. 2000 vol.3726 p. 168-98

4 J.A. Izatt, M.D. Kulkarni, H.-W. Wang, K. Kobayashi, and M.V. Sivak, Jr., "Optical coherence tomography and microscopy in gastrointestinal tissues," IEEE J. Sel. Topics Quant. Electron. 2, 1017-1028 (1996).

5. S.A. Boppart, M.E. Brezinski, B.E.Bouma, G.J. Tearney, and J.G.Fujimoto, "Imaging developing neural morphology using optical coherence tomography," J. Neurosci. Methods **70**, 65-72 (1996)

6. S.A. Boppart, M.E. Brezinski, B.E.Bouma, G.J. Tearney, and J.G.Fujimoto, "Noninvasive assessment of the developing *Xenopus* cardiovascular system using optical coherence tomography," Proc. Natl. Acad. Sci **94**, 4256-4261 (1997)

7. S.R. Chinn and E.A. Swanson, "Blindness limitations in optical coherence domain reflectometry," Electronics Letters **29**, 2025-2027 (1993)

8. Barbara M. Hoeling, Andrew D. Fernandez, Richard C. Haskell, Eric Huang, Whittier R. Myers, Daniel C. Petersen, Sharon E. Ungersma, Ruye Want, and Mary E. Williams. "An optical coherence microscope for 3-dimensional imaging in developmental biology" Optics Express 6: 136-146 March 27, 2000

9. Whittier R. Myers Senior Thesis. Harvey Mudd College, physics, class of 1999.

10. Barbara M. Hoeling, Andrew D. Fernandez, Richard C. Haskell, and Daniel C. Petersen. "Phase Modulation at 125 kHz in a Michelson Interferometer Using an Inexpensive Piezoelectric Stack Driven at Resonance". Review of Scientific Instruments – to be published in may 2001.

11. Optics, third ed. Eugene Hecht. Adelphi University. Addison Wesley longman, inc. 1998. Chapter 7.4 pg 306-311