Teaching Real-time Beamforming With The C6211 DSK and MATLAB

Michael G. Morrow, Thad B. Welch
Department of Electrical Engineering
U.S. Naval Academy, MD
Cameron H. G. Wright
Department of Electrical Engineering
U.S. Air Force Academy, CO
George York
U.S. Air Force, MD

Abstract

This paper describes our efforts to teach hardware-based beamforming at the undergraduate level. A DSP based beamformer is at the heart of numerous military systems, including active and passive sonars, RF direction finding equipment, and phased-array radars. While the military has needed engineers capable of designing and enhancing these and other systems, civilian applications of these same technologies have recently made beamforming an important topic for many engineers. High interest topics include E911 mobile position location, space division multiple access (SDMA), cellular/PCS base station antenna design, and low Earth orbit (LEO) satellite communications within a densely packed constellation.

Beamforming is routinely discussed in a course on multidimensional signal processing but an actual hardware system to design or enhance is often not available. The use of powerful software, such as MATLAB, to simulate the geometries and techniques necessary to learn about beamforming has become standard. However, proceeding beyond MATLAB simulation to a real-time hardware implementation has been impeded by a very expensive and abrupt transition, in terms of both cost and the learning curve of unfamiliar systems and software. By developing a software bridge between MATLAB and the C6211 DSK, we make it possible to smoothly and incrementally transition from simulation to full hardware implementation, all while retaining the impressive capabilities of the MATLAB display engine.

Using this approach, students can develop and enhance their own sonar system. Initially, a student works with MATLAB generated data files to develop an understanding of various beamforming techniques. Then, they proceed to non-real-time processing of actual data acquired directly into MATLAB from the DSK, then incrementally move the processing into C code running on the DSK, and finally, using MATLAB only for display of the real-time results. The software development cycle of algorithm and display development in MATLAB, followed by real-time implementation with a C6X DSK, can be iterated as many times as the semester will allow. The paper describes in detail the software and inexpensive multichannel ADC daughtercard that facilitate this pedagogical process. The software is freely distributed by the authors for educational use.

I. Introduction

Array signal processing, e.g., [1-3], or beamforming, specifically, is a crucial topic for electrical engineers (EE's) interested in the antenna arrays associated with cellular and PCS systems. Beamforming also remains essential in the fields of seismology, sonar, radar, radio astronomy, and tomographic imaging. Given this diverse topical interest in beamforming, we believe that an understanding of the theories and implementation techniques necessary to construct a beamformer are an essential part of an undergraduate EE education.

The basic block diagram [4] for time domain or frequency domain beamforming, shown for a sonar system, is provided as Fig. 1. The top portion of the figure is common to both the time and frequency domain beamformers. The output of this portion of the figure, labeled "X or Y", then provides the input into the next stage of the beamformer, "X" for the time domain, and "Y" for the frequency domain networks.

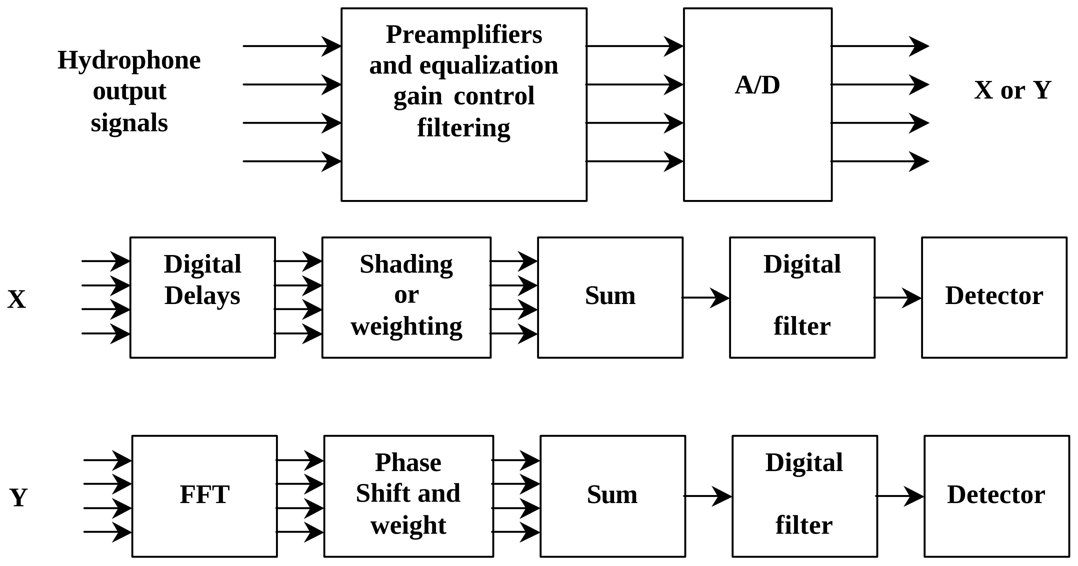

Figure 1. Basic methods of time domain and frequency domain beamforming for passive (and active) systems [4].

Historically, the theoretical aspects of beamforming, as discussed in [1-4] would be covered in a graduate level course. More recently, selected beamforming topics have found their way into the undergraduate curriculum. Additionally, the software package MATLAB (from The MathWorks, Inc.) and its related toolboxes has become a mainstay in most EE programs. Given our students' familiarity with MATLAB, computer exercises that implement the beamforming theory only seem natural. But where does the array data, labeled "hydrophone output signal" in Fig. 1., come from? Do our students generate MATLAB vectors or matrices containing simulated array data? Even if this data is realistic in nature, doesn't the time spent generating this information detract from what we are trying to have our students learn? Do we, as professors, generate the data or provide real data?

We have proposed and constructed a beamforming educational platform based around MATLAB and the Texas Instruments (TI) TMS320C6211 digital signal processing starter kit (DSK). Our students are already very familiar with MATLAB, but we also want our students to learn more about hardware-based digital signal processing (DSP).

The TI TMS320C6211 DSK has the following advantages:

- It comes with a feature-rich development environment (Code Composer Studio),
- it can perform about a billion instructions per second,
- it has plenty of memory, and
- it's relatively inexpensive.

The TI TMS320C6211 DSK's principle disadvantage is that it has only a telephone quality, single channel CODEC.

II. System requirements

Even though the TI TMS320C6211 DSK is a single channel system, we felt the DSK's advantages out-weighed this principle disadvantage[1]. We also wanted to begin teaching linear array processing, so replacing the single channel CODEC was already required. Most of the available audio frequency[2] analog-to-digital (A/D) converters are either one, two, or four channel devices. A four-channel system was selected since a two-channel system did not provide beams that were well defined and an eight-channel system was seen as *overkill* for an educational platform. This four-channel system can easily be expanded to 8, 12, 16,…, channels using the existing four-channel design as a building block.

Although not strictly required, for pedagogical reasons we desire simultaneous sampling of all four input channels. But at what sample frequency? Our initial desire is to work with an acoustic system with source frequencies of at most a few thousand hertz. At this point in the discussion, a drawing of the four-element microphone[3] array as shown in Fig. 2. is helpful. In this figure, each microphone is represented by a circle and is numbered (1, 2, 3, and 4) below the circle.

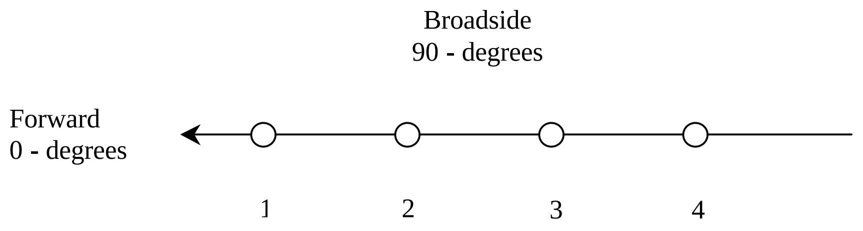

Fig. 2. Four-element array geometry.

[1] For pedagogical reasons, the floating-point version of this DSK, the TI TMS320C6711, will be used when it is available.
[2] The price of the input circuitry of an audio frequency system is much less than for a higher frequency system.
[3] We have now specified an audio system using traditional microphones instead of the hydrophones shown in Fig. 1.

If we assume that a signal source is distant, the arriving wave fronts can be assumed to be linear. Given this assumption and an element separation of $l/2$, where l is the signal source wavelength, a more detailed Fig. 3. can be drawn.

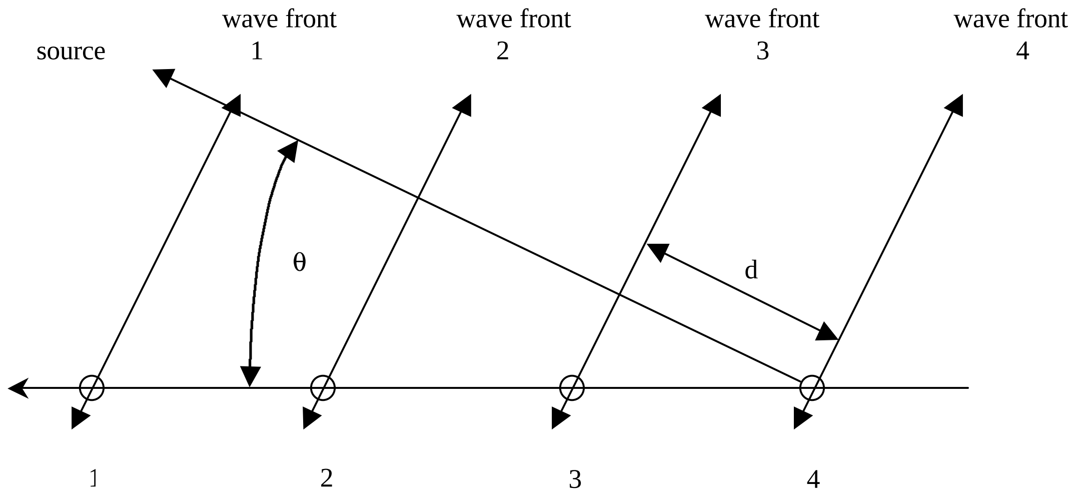

Fig. 3. Detailed four-element array geometry.

Using the notation of Fig. 3., $\cos q = \frac{d}{l/2}$. If we wish to start our students with the most basic beamforming algorithm, delay-and-sum, then knowing the q 's or maximum response axes (MRA) that can be achieved is essential. Let f_s and n denote the sample frequency and an integer (delay) respectively. Remembering that $l\,f = v_p$, where f is the frequency of the waveform and v_p is the velocity of propagation of the waveform, then,

$$\cos q = \frac{d}{l/2} = \frac{n\frac{1}{f_s}v_p}{v_p/2f} = 2n\frac{f}{f_s}.$$

For efficient control of the MRA, using the delay-and-sum algorithm, the ratio $\frac{f}{f_s}$ should be made small. Our microphone separation of $l/2$ was set for a received frequency of 1000 Hz. Given these two constraints a sample frequency of 100 kHz was selected. Now,

$$\cos q = \frac{n}{50} \quad \text{or} \quad q = \arccos\left(\frac{n}{50}\right).$$

This results in the ability to calculate 101 different beams. Our students will discover the usefulness of 101 beams given a four-element array later.

III. Hardware Interface

The beamforming hardware interface was developed as a daughtercard for the DSK, based on the Analog Devices AD7834 four-channel, simultaneous sampling, parallel-bus analog to digital converter (ADC). In addition to the pedagogical reasons discussed previously, a simultaneous-sampling ADC was also chosen to eliminate the need to otherwise keep tight timing control between multiple converters. The converter is capable of operating at speeds of up to 400k samples per second on a single channel, or the same rate divided between up to 4 channels. The converter has a nominal +/- 2.5V input range, so a small, battery-powered preamplifier was constructed to provide the correct signal levels from the microphones to the ADC. The total cost of parts for the entire daughtercard and preamplifier, including circuit board, was under $100 and could easily be assembled by students.

The converter is triggered directly by the DSK Timer1 output, so that it places no burden on the DSK CPU resources. At the end of each conversion set, the CPU is interrupted and reads the four samples from the converter. If additional CPU resources are required for processing, the DSK code could be modified to offload this task to the CPU's direct memory access (DMA) controller. In our case it was unnecessary, as the converter operates with four channels converted and read every 10 microseconds.

To support the ADC, the DSK software was modified as necessary. To allow easy modification, the basic DSK software (supporting the DSK's onboard CODEC) was originally developed so that it was partitioned into generic and device-specific sections. Approximately 80% of the code written for the single channel CODEC version was reused for the new daughtercard without modification. The major areas requiring revision were the hardware set-up and configuration routines (actually greatly simplified due to the simplicity of the converter), and the interrupt service routine that handles the ADC end-of-conversion. The code for command interpretation also required only minor revisions.

IV. Desired Educational Framework

We now have a DSP-based hardware system that can rapidly gather samples from a four-element acoustic array. If our students were proficient at DSP programming, at this point they could develop any sonar system they desired. Our students, however, know MATLAB, not TI's Code Composer Studio. What we need is a tool that will allow for algorithm development in MATLAB and, once the student is comfortable with what they have learned, the ability for the algorithm to migrate, in part or as a whole, onto the DSP hardware.

The desired process would be:

1. Study the traditional DSP theory,
2. work in MATLAB with simulated data,
3. work in MATLAB with real-world data,
4. implement the process (in part or as a whole) in real-time on the TI DSK, and
5. repeat to improve the process or to develop new features.

The third step of this process presents a problem. MATLAB now has a very capable data acquisition (DAQ) toolbox that allows for direct data acquisition and data insertion into the MATLAB workspace. This toolbox works with a number of different DAQ hardware boards, but the toolbox does not support programmable DSP systems. Even if one were to take advantage of the DAQ Toolbox, the result is that between steps 3 and 4, you would unavaoidably need to change platforms in order to transition to a real-time implementation. Since multiple platforms require additional time for our students to learn, a single platform solution is highly desired. To facilitate this single platform development, we developed a direct DSK-to-MATLAB interface.

V. MATLAB to DSK Interface

The interface between MATLAB and the DSK is encapsulated into a generalized interface command set that supports multiple input and output channels, variable sample rates, various triggering configurations, and variable frame sizes. The specific commands available are illustrated in Appendix A. The interface was developed using MATLAB's "mex" facility and Microsoft Visual C++, and is centered around an object that encapsulates the hardware interface between the host PC and the DSK. The application programming interface (API) furnished with the DSK allows operation under Windows 9X/NT. The interface software requires that the DSK tools be installed on the computer, and that the files C6X_DAQ.DLL and DAQ_CODEC.OUT be placed in a MATLAB-accessible directory. At the most basic level, this interface allows a novice user to operate the DSK as a data acquisition board with a simple command sequence, with no requirement to know how to use Code Composer or how to program in C. Initially, all signal processing can be done in the MATLAB environment using the live data acquired from the DSK. As the student progresses, they can move processing functions down to the DSK by altering the DSK code (that was used to create DAQ_CODEC.OUT), and still continue to use MATLAB as a processing and display engine.

To support advanced applications and special situations, the interface can be extended by user-defined read and write commands. These require that the user to write the required support at the DSK level, but do not require any alterations to the MATLAB interface software (C6X_DAQ.DLL).

The interface's ease of use is best illustrated by the sample MATLAB m-file listed below. The m-file is the complete sequence of commands necessary to use MATLAB and the DSK to form a single-channel real-time oscilloscope.

```
% codec_scope.m, 2000
%
% initialize the DSK parameters
c6x_daq('Init', 'daq_codec.out');
c6x_daq('FrameSize', 500);
Fs = c6x_daq('SampleRate', 8000)
numChannels = c6x_daq('NumChannels', 1)
c6x_daq('TriggerMode', 'Auto');
c6x_daq('TriggerSlope', '+');
c6x_daq('TriggerValue', 0.2);
c6x_daq('TriggerChannel', 1);
c6x_daq('LoopbackOn');
c6x_daq('QueueSize', 100);
c6x_daq('FlushQueues');
c6x_daq('GetSettings');


% do double-buffered plotting for speed
data = c6x_daq('GetFrame');
P1=plot(data(:,1),'g');
axis([0 FrameSize -1.1 1.1])
set(gcf,'doublebuffer','on')

while 1 > 0
   data = c6x_daq('GetFrame');
   set(P1,'ydata',data)
   drawnow
end
```

As is evident, the interface allows complete control over the acquisition process with no knowledge of the actual DSK software or hardware operation. Sample m-files, and the source code to support the DSK's single channel CODEC, are available from the authors.

VI. Conclusions

We have developed an educational framework that will allow our students to smoothly transition from multi-channel high-speed data acquisition to real-time system implementation. This process allows real-world data to be gathered and used in the algorithm development and design process while maintaining a link to MATLAB.

The single platform process has now become:

1. Study the traditional DSP theory,
2. develop an algorithm in MATLAB with simulated data,
3. test and update the algorithm in MATLAB with real-world data,
4. implement the process, one part at a time, in real-time, on the same TI DSK that gathers the data, and
5. repeat to improve the process or to develop new features.

During the spring semester of 2000, an EE senior helped us develop a great deal of insight into how to teach beamforming during his research project concerning base station antenna arrays. This project was part of the *Wireless and Cellular Communication Systems II* course taught at the U.S. Naval Academy.

This system will be used in a number of courses and senior design projects during the next year at both the Naval and Air Force Academies. We believe this approach to teaching beamforming will develop better student skills related to MATLAB, C, C++, algorithm development, system interfacing, and integration. Finally, it exposes our EE students to DSP system development in traditional DSP courses, communications courses, and in computer engineering courses. Multidisciplinary exposure to the power of hardware-based DSP can only help to develop the next generation of DSP engineers.

The authors freely distribute this software for educational, non-profit use, and invite user comments and suggestions for improvement. At the time of publication, the final distribution site address was unavailable. Interested parties are invited to contact the authors via e-mail.

References

[1] S.H. Haykin, editor, *Array Signal Processing*, Prentice-Hall, Inc, Englewood Cliffs, New Jersey, 1985.

[2] S.U. Pillai, *Array Signal Processing*, Springer-Verlag, New York, 1989.

[3] D.H. Johson, D.E. Dudgeon, with D.E. Dugeon, *Array Signal Processing: Concepts and Techniques*, Simon and Schuster, Inc., New York, 1992.

[4] A.V. Oppenheim, editor, *Applications of Digital Signal Processing*, Chapter 6 – Sonar Signal Processing, Prentice-Hall, Inc, Englewood Cliffs, New Jersey, 1978.

[5] The MathWorks, Inc., Natick, MA, *MATLAB:The Language of Technical Computin*g, 1996.

MICHAEL G. MORROW, PE, is a Master Instructor in the Department of Electrical Engineering at the U.S. Naval Academy. His research interests include real-time digital systems, embedded systems, and software engineering. He is a member of ASEE and IEEE. Email: morrow@ieee.org

THAD B. WELCH, PhD, PE, is an Assistant Professor in the Department of Electrical Engineering at the U.S. Naval Academy. From 1994–1997 he was an Assistant Professor in the Department of Electrical Engineering at the U.S. Air Force Academy. His research interests include multicarrier communication system design and analysis, RF channel measurements, and real-time signal processing. He is a member of ASEE and Eta Kappa Nu and a senior member of the IEEE. Email: t.b.welch@ieee.org

CAMERON H. G. WRIGHT, PhD, PE, is an Associate Professor in the Department of Electrical Engineering at the U.S. Air Force Academy. His research interests include signal and image processing, biomedical instrumentation, communications systems, and laser/electro-optics applications. He is a member of ASEE, IEEE, SPIE, NSPE, Tau Beta Pi, and Eta Kappa Nu. Email: c.h.g.wright@ieee.org

Appendix A
C6x DSK to Matlab Interface

To use this interface, the file C6X_DAQ.DLL and the desired DSK COFF file (e.g. DAQ_CODEC.OUT) DAQ must be in a directory in the Matlab path, and the commands must be executed within that directory. The 'Init' command must be executed before any other commands are used. The DSK software must also be installed.

All ADC data values received from the DSK are normalized to +/- 1.0 for the maximum ADC range. All DAC data values sent to the DSK must be normalized in the same manner.

Command argument types are denoted as follows:

< >	optional argument
#	numeric argument
'arg'	text argument
X	Matlab variable

Command	Syntax	Description
Init	C6X_DAQ('Init', 'Filename', <#>)	Initializes the C6X DSK and the Matlab interface. The filename of the desired COFF file must be supplied. The optional # argument is used to specify which parallel port to use (1 or 2). The default is 1.
Version	C6X_DAQ('Version')	Displays the version numbers of the C6X_DAQ.dll and DAQ.out files in use.
GetSettings	C6X_DAQ('GetSettings')	Displays the current settings in use.
LoopbackOn	C6X_DAQ('LoopbackOn')	Echoes DSK input data directly to the DSK output. Useful for monitoring.
LoopbackOff	C6X_DAQ('LoopbackOff')	Turns off loopback.
QueueSize	X = C6X_DAQ('QueueSize', #)	Sets the queue size to the value of the second argument, or the maximum queue size, whichever is less. Returns the actual queue size.
FlushQueues	C6X_DAQ('FlushQueues')	Flushes the transmit and receive queues on the DSK.
FrameSize	X = C6X_DAQ('FrameSize', #)	Sets the frame size to the value of the second argument, or the maximum frame size, whichever is less. Returns the actual frame size.
NumChannels	X = C6X_DAQ('NumChannels' #)	Sets the number of active channels to the value of the second argument, or the maximum supported channels, whichever is less. Returns the actual number of active channels.
SampleRate	X = C6X_DAQ('SampleRate', #)	Sets the sample rate to the value of the second argument, or the maximum/minimum sample rate, whichever is less. Return the actual sample rate.
TriggerMode	C6X_DAQ('TriggerMode', 'arg')	Sets the trigger mode to one of three mode values – 'Auto', 'Immediate', or 'Normal'.
TriggerSlope	C6X_DAQ('TriggerSlope', 'arg')	Sets the trigger slope to positive ('+') or negative ('-').
TriggerValue	C6X_DAQ('TriggerValue', #)	Sets the trigger value to the passed value. This should be a number such that $-1.0 < x < +1.0$.
TriggerChannel	C6X_DAQ('TriggerChannel', #)	Sets the trigger channel to the passed value.
GetFrame	X = C6X_DAQ('GetFrame')	Gets a frame of data from the DSK, and returns it in the matrix X. The data is organized on a column per channel basis.
SendFrame	C6X_DAQ('SendFrame', X)	Sends the frame of data in X to the DSK. X must be the correct size for the current frame size and number of channels.
SwapFrame	C6X_DAQ('SwapFrame', X)	Sends the frame of data in X to the DSK, then retrieves a frame of data from the DSK. X must be the correct size for the current frame size and number of channels.
UserRead	C6X_DAQ('UserRead', $\#_1$, $\#_2$, X)	Performs a user-defined read of DSK data by passing first the command ($\#_1$) to the DSK, then reading $\#_2$ elements of 32 bit data from a buffer on the DSK into the MATLAB variable X.
UserWrite	C6X_DAQ('UserWrite', $\#_1$, $\#_2$, X)	Performs a user-defined write to the DSK by passing first the command ($\#_1$) to the DSK, then writing $\#_2$ elements of 32 bit data from MATLAB variable X to a buffer on the DSK.
Close	C6X_DAQ('Close')	Closes the DLL connection with the DSK.