Ilia Kempi

Adaptive Channel Equalization

Multicore Processor Implementation

Helsinki Metropolia University of Applied Sciences

Bachelor of Engineering

Degree Programme in Electronics

Bachelor's Thesis

12.05.2015

Metropolia

Author(s)	Ilia Kempi
Title	Adaptive Channel Equalization
Number of Pages	63 pages + 4 appendices
Date	12th May, 2015
Degree	Bachelor of Engineering
Degree Programme	Degree Programme in Electronics
Specialisation option	
Instructor(s)	Thierry Baills, Senior Lecturer

Advanced telecommunication techniques are more and more relying on the digital processing power that is needed for the signal demodulation and data recovery. In order to assess the practicability of concurrent signal processing in the field of communication, this project was focused on developing the adaptive signal filtering application on a distributed processor system.

To prove and evaluate this concept, different versions of the fundamental and robust Least Mean Squares adaptive filter were implemented on a multi-core digital signal processor. Developed applications were tested by streaming the simulated digital transmission data to the device via Ethernet. Algorithm benchmark results are compared in terms of adaptation rate and execution speed.

The parallelized version of the adaptive algorithm has shown promising results in terms of convergence and the workload distribution while its execution time is still inferior to the single-processor applications. Considering that only the most essential device functionality was used, there is a big room for improvement and optimization. The algorithm evaluation system established in the project can be reused for other concepts and related teaching.

Keywords	Digital Signal Processing, Adaptive filtering, Quadrature Amplitude Modulation, Channel equalization, Least Mean Squares, Multi-core signal processor

Contents

Acronyms

ADC Analog to Digital Converter

ASK Amplitude-Shift Keying

CCS Code Composer Studio

DAC Digital to Analog Converter

DMA Direct Memory Access

DSP Digital Signal Processor

EVM Evaluation Module

FIR Finite Impulse Response

IP Internet Protocol

IPC Inter-Process Communication

ISI Inter-Symbol Interference

LAN Local Area Network

LMS Least Mean Squares

LTI Linear Time-Invariant

LTV Linear Time-Variant

NRZ Non-Return-to-Zero

QAM Quadrature Amplitude Modulation

RF Radio Frequency

RTOS Real-Time Operating System

SNR Signal-to-Noise Ratio

TCP Transmission Control Protocol

UDP User Datagram Protocol

1 Introduction

The demand for bandwidth in telecommunication is always increasing. Modern data transmission techniques are exploiting more and more physical phenomena in order to expand the flow of information. In recent decades, an enormous amount of work has been done in the development of communication. To fit into constraints stated in the international radio regulations, a great deal of acceleration and optimizations has been applied to former models of data encoding and modulation.

With the increase of information density in the channel, adverse transmission effects are becoming more significant in relation to the amount of data transferred. To compensate the distortion, present day radio equipment utilizes special procedures that involve channel analysis and digital signal processing. With vast processing capabilities available nowadays, one of the vital tasks of a digital radio receiver is to adapt to the environment and improve its ability to recover the transmitted information.

This project focuses on adaptive channel equalization techniques, with an intention to reproduce real communication conditions and implement a robust error correction algorithm on the dedicated hardware. The goal is to demonstrate the feasibility and effectiveness of such application based on a distributed processor system.

2 Theoretical Background

2.1 Digital Transmission

2.1.1 Data Mapping

Prior to actual transmission, an outgoing batch of data is organized in a way that is optimal for serial transfer. Especially for large volumes of information, the data is initially *serialized*. This means that digital content is indexed linearly according to a certain convention. For instance a matrix, being a two-dimensional data array, can be presented as a sequence of its concatenated rows. Usually serialization happens on the application level, providing a stream of bytes to the communication system.

Often, serial information is additionally split into small frames named *packets*, in order to ease the transfer and organize the error handling. This practice is common for example in duplex systems, where synchronization and data coherence is very important. In addition to an actual data content or a *payload*, packet should include additional block named *header*, which contains service information [1, 15].

To project the digital signal into a physical medium, sequence of binary values is mapped to the corresponding voltage levels with a Digital to Analog Converter (DAC). Convention of producing a real time logical sequence is called *line coding* and illustrated on figure 1. It can be seen, that each amplitude level means a certain binary value. One stable state of the analog signal is called a *symbol*, and its duration T is what mainly determines the rate of transmitted information. By allowing more signal levels, the amount of bytes per symbol can be increased. The main technical trade-off at this point is to balance the symbol length with allowed precision of the analog signal.

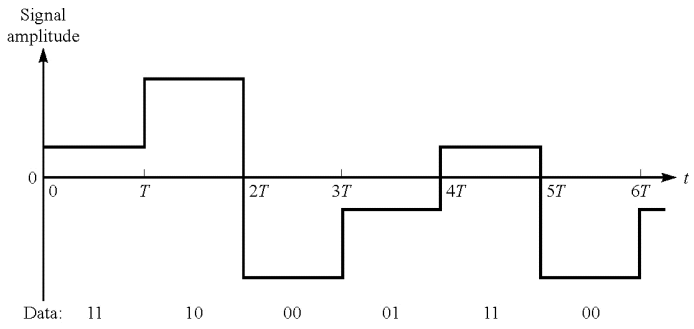


Figure 1: NRZ-encoded logical sequence with 2-bit symbol length (Copied from Proakis (2000) [2]).

This coding rule is known as Non-Return-to-Zero (NRZ) encoding, a convention made with intent to eliminate neutral voltage states in the signal. On the figure 1, bits are mapped according to the *gray code*, such that there is only one bit change between adjacent levels. This technique helps to prevent large bit errors at the receiver [2, 175].

Dynamic logical signal, presented in the figure 1 is labeled as a *baseband* signal, because of its relatively low frequency content. The baseband signal is the closest real world representation of digital data, and often it can be directly transferred to the receiver device simply through the copper wire. The signal B_m then can be expressed ad in equation 1:

$$B_m(t) = A_m g(t), \quad m = 1, 2, ..., M \tag{1}$$

where A_m is a set of M possible amplitudes and $g(t)$ is a reference waveform containing data rate information [2, 176].

2.1.2 Carrier Modulation

To transmit the mapped data by means of electromagnetic waves, the information has to be up-converted to the higher frequencies. This is done by conveying the message into a *carrier* signal. Such procedure is called *modulation*, and Amplitude-Shift Keying (ASK) scheme is of our particular interest. Example of ASK-modulated signal is presented in figure 2.

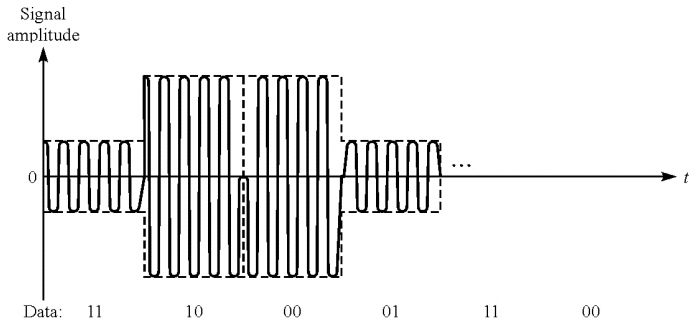


Figure 2: Amplitude-modulated NRZ-coded sequence (Modified from Proakis (2000) [2]).

To produce an amplitude modulated wave, the original baseband signal is multiplied with a carrier inside a *mixer*. Figure 3 shows that because of the modulation process, the spectrum of baseband signal is translated into Radio Frequency (RF) range and becomes centered around the carrier frequency. Together with the negative frequency component, amplitude modulated signal forms a *passband* spectrum, consistent of upper and lower *sidebands* [3].

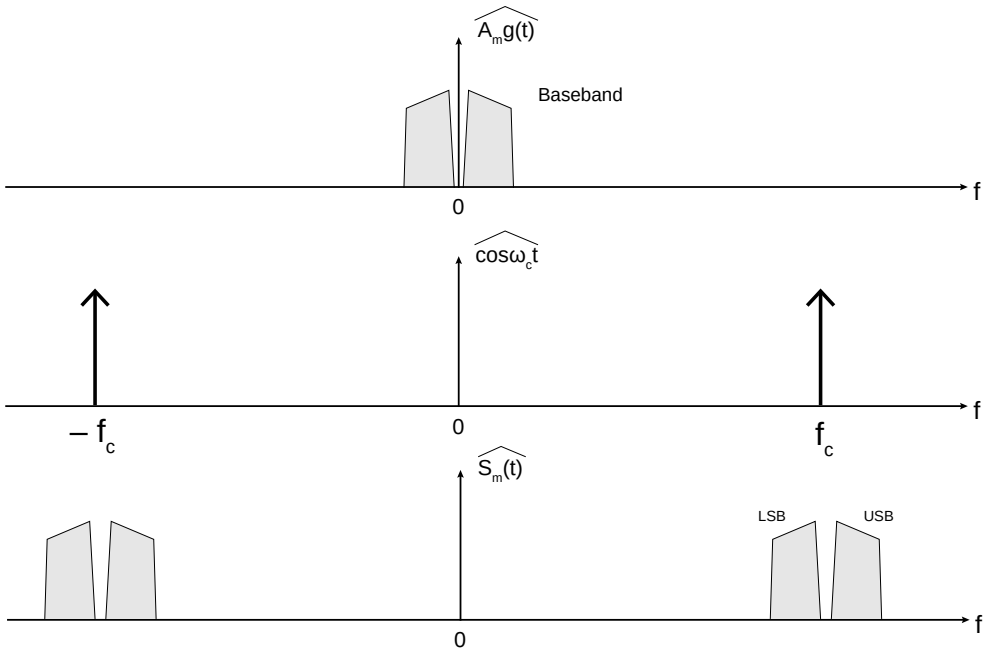


Figure 3: Carrier modulation process, where $\hat{g} = \mathcal{F}[g(t)]$.

The resulting time domain signal is defined simply as shown in equation 2.

$$S_m(t) = A_m g(t) cos(\omega_c t), \quad m = 1, 2, ..., M \tag{2}$$

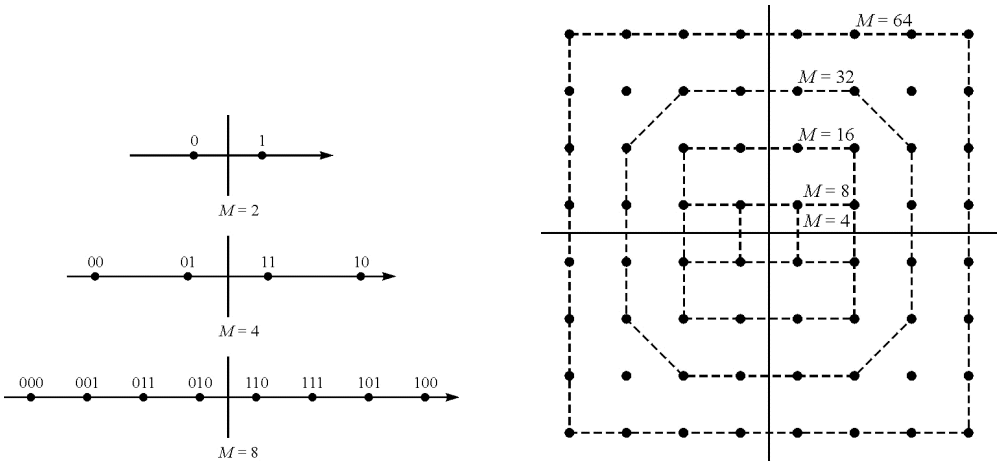
Now $S_m(t)$ contains the initial binary information, yet by taking twice as much bandwidth than initial baseband signal. In order to narrow the frequency band of the message, it is

usually filtered with a low-pass before the modulation. This process is also called *pulse shaping* [2, 545-547].

Amplitude modulation is a fundamental frequency translation technique, and it is widely used in more advanced modulation schemes as a basis for passband signal generation. Main transmission technology of our interest, a very efficient and widespread Quadrature Amplitude Modulation (QAM) scheme is introduced in following sections.

2.1.3 Quadrature Amplitude Modulation (QAM)

Quadrature Amplitude Modulation increases transmission effectiveness by utilizing two-dimensional line coding scheme. In QAM, symbol map is called a *constellation*. Example constellations of various complexity are shown on figure 4.



(a) ASK-modulated signal basis (b) QAM constellation

Figure 4: Complex signal mapping (Copied from Proakis (2000) [2]).

Each point on the symbol map can be defined as a linear combination of two orthogonal unit vectors [2, 163]. In the same way, a time-varying logical sequence is expressed as the combination of signals given by two basis functions: sine and cosine. This is possible due to orthogonality of these functions, verified by their inner product taken between 0 and T to be zero (equation 3).

$$\int_a^b f_1(x) f_2^*(x) dx = 0 \tag{3}$$

$$\int_0^T cos\frac{2\pi mt}{T} sin\frac{2\pi nt}{T} dt = 0 \tag{4}$$

$$\frac{1}{2}\left[-\frac{cos\frac{2\pi(n+m)t}{T}}{\frac{2\pi(n+m)t}{T}} - \frac{cos\frac{2\pi(n-m)t}{T}}{\frac{2\pi(n-m)t}{T}} \right]_0^T = 0 \tag{5}$$

Such property allows to give the symbol its coordinates by varying the amplitudes of two basis functions, as if constellation was a complex plane (equation 6) [4].

$$S_m(t) = (A_{mI} + jA_{mQ})g(t), \quad m = 1, 2, ..., M \tag{6}$$

Imaginary and real symbol components are called as *quadrature-phase* and *in-phase*, denoted by I and Q respectively.

$$S_{sym}(t) = I(t) + jQ(t) \tag{7}$$

From the figure 4 can be noted that capacity of the transmission with QAM is increasing exponentially with the symbol length per basis, compared to the ASK-modulated transfer.

2.1.4 QAM Transmitter

In the QAM transmitter hardware, information goes through a chain of processing tasks, as shown in the schematic diagram (figure 5). First of all, serialized data is mapped to the constellation and its coordinate components are isolated.Quadrature and in-phase components are separately converted to the voltage with the DAC by NRZ scheme. After that, signals are filtered with pulse shaping filter to eliminate quick level transitions and reduce the bandwidth (figure 6).

It should be noted that the latter procedure should be done with special kind of filter, in order to reduce the Inter-Symbol Interference (ISI) in that may be caused by overlapping

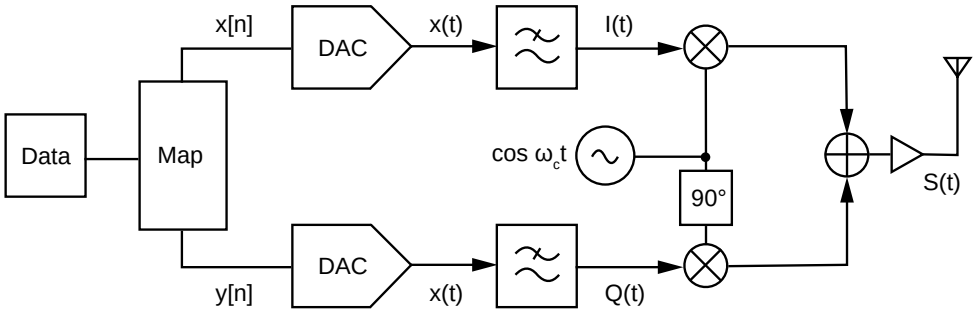


Figure 5: QAM transmitter.

responses of the filter. One example of the non-overlapping impulse response is a *raised cosine* filter [2, 547].

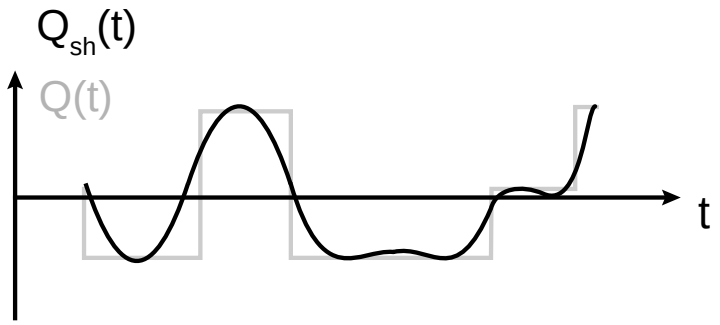


Figure 6: Pulse shaping of the quadrature-phase signal.

Product of the pulse shaping can be modeled as wave function of varying amplitude, namely

$$I(t) = A_I(t)cos(\phi) \tag{8}$$

$$Q(t) = A_Q(t)sin(\phi) \tag{9}$$

Following this notation, I and Q signals are translated to a higher frequencies in two separate mixers with orthogonal carriers (sine and cosine), and resulting signals are added together to yield a passband signal (equation 10) [4].

$$S(t) = I(t)cos(\omega_c t) + Q(t)sin(\omega_c t) \tag{10}$$

$$= A_I(t)cos(\phi)cos(\omega_c t) + A_Q(t)sin(\phi)sin(\omega_c t) \tag{11}$$

$$= A_{IQ}(t)cos\big(\omega_c t + \phi_{IQ}(t)\big) \tag{12}$$

$$= Re\Big[A_{IQ}e^{j(\omega_c t + \phi_{IQ}(t))}\Big] \tag{13}$$

$$= Re\Big[A_{IQ}e^{j\phi_{IQ}(t)}e^{j\omega_c t}\Big] \tag{14}$$

$$A_{IQ} = \sqrt{A_I^2 + A_Q^2}, \quad \phi_{IQ} = tan^{-1}\Big(\frac{A_Q}{A_I}\Big) \tag{15}$$

From the equation 15 we can see that the change of amplitude of I or Q components will cause a change in both amplitude and phase of the real passband signal. This is possible because of the complex factor $A_{IQ}e^{j\phi_{IQ}(t)}$ which is known as a *complex envelope* of the signal (equation 14). [2, 179.]

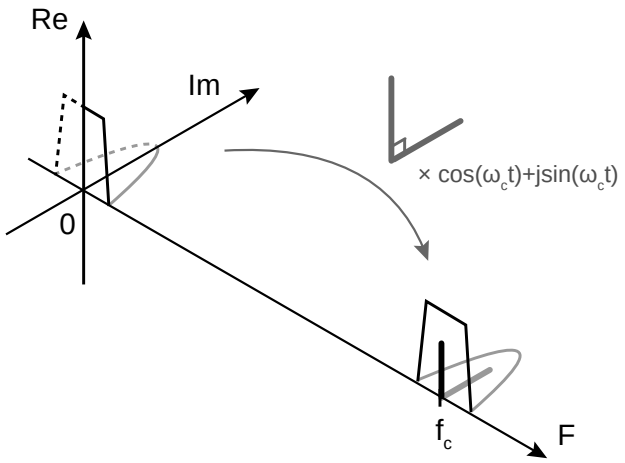


Figure 7: Frequency translation with a complex carrier (Adapted from Lyons (2008) [5, 11]).

Orthogonality of basis functions is the main principle that allows to use quadrature mixing, and because of that it is very important for the transmitter to keep both carriers synchronized on exactly 90 degree phase difference (figure 7). Together these two functions can be viewed as a *complex carrier*, which allows to transmit two real signals with zero correlation, thus preserving the bandwidth [5].

2.1.5 QAM Receiver

The receiver has a structure that is very similar to the transmitter, but the signal is processed in a reversed way. Schematic diagram of the system is shown in the figure 8. From this, we can see that after the preamplifier two identical copies of the RF signal are taken into separate mixers for *demodulation*.

By assuming the ideal case when both transmitter's and receiver's local oscillators are exact in frequency in phase, passband signal is basically mixed with the same complex carrier. This multiplication will cause an extra frequency translation and produce additional components in the signal, some of which are more favorable for further processing.

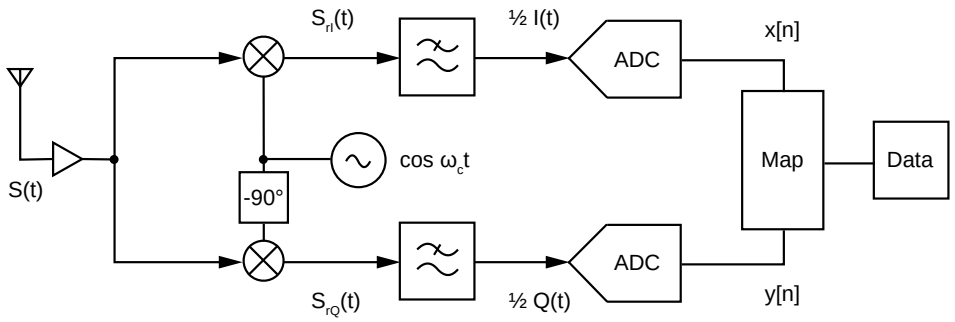


Figure 8: QAM receiver.

To extract the in-phase component from the passband, received signal is mixed with a cosine such that required portion of baseband is shifted to its place, as in equation 16.

$$S_{rI}(t) = S(t)cos(\omega_c t) \tag{16}$$

$$= I(t)cos(\omega_c t)cos(\omega_c t) - Q(t)sin(\omega_c t)cos(\omega_c t) \tag{17}$$

$$= \frac{1}{2}\Big[I(t)\big(cos(0) + cos(2\omega_c t)\big) - Q(t)\big(sin(0) + sin(2\omega_c t)\big)\Big] \tag{18}$$

$$= \frac{1}{2}I(t) + \underbrace{\frac{1}{2}\Big[I(t)cos(2\omega_c t) - Q(t)sin(2\omega_c t)\Big]}_{\text{HF content}} \tag{19}$$

Similarly, the quadrature component is obtained from the RF signal by multiplying it with sine (equation 20). Note the intentionally inverted function, that is necessary for the proper demodulation.

$$S_{rQ}(t) = S(t)(-sin(\omega_c t)) \tag{20}$$

$$= I(t)cos(\omega_c t)sin(\omega_c t) + Q(t)sin(\omega_c t)sin(\omega_c t) \tag{21}$$

$$= \frac{1}{2}\Big[Q(t)\big(cos(0) + cos(2\omega_c t)\big) + I(t)\big(sin(2\omega_c t) - sin(0)\big)\Big] \tag{22}$$

$$= \frac{1}{2}Q(t) + \frac{1}{2}\Big[Q(t)cos(2\omega_c t) + I(t)sin(2\omega_c t)\Big] \tag{23}$$

HF content

High frequency by-products of the mixing are rejected with a low-pass filter. The "clean" passband waveform is preconditioned and sampled with an Analog to Digital Converter (ADC) and original data sequence is reconstructed from the complex symbols.

2.2 Transmission Channel

2.2.1 Channel Environment

Depending on the transport medium, the initial transmitted signal becomes somehow impaired before it arrives at the receiver. To analyze the mechanisms that cause a signal degradation, various transmission channel models have spawned. By fundamental principle, we can categorize all channels as *guided* or *wireless*.

In **guided channels**, such as coaxial cables and twisted pair, wave propagation is very persistent, but not uniform. Due to the properties of the transmission line, the signal sent to long distances usually suffers from *time spreading*. Especially in wire-line connection, a significant interference comes from the adjacent wires, an effect that is also known as *crosstalk*. When the physical parameters of the conductive medium are known, the negative effects become very predictable, and because of that guided channels are commonly modeled as an Linear Time-Invariant (LTI) system [6, 11].

Wireless channels are subjected to harsher conditions. In addition to the path loss due to the transmission distance, various obstacles will also attenuate and reflect the signal, as shown in figure 9. Reduction in the signal power due to the obstruction in the line of sight is called *shadowing*. This effect almost always causes *flat-fading*, meaning that the whole spectrum of the signal is collectively attenuated.

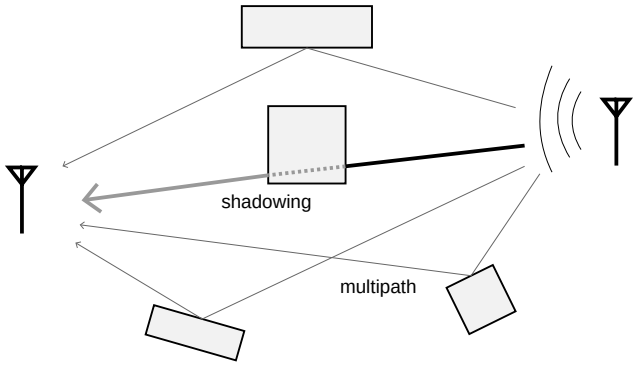


Figure 9: Wireless channel environment.

Those reflected parts of the initial signal that manage to arrive at the receiver are named as *multi-path* components (figure 9). Depending on many factors such as path length and object position, these additional signals can produce destructive or constructive interference. This results in frequency-selective *multi-path fading* [7, 5]. The cumulative effect of factors introduced above is presented in figure 10.

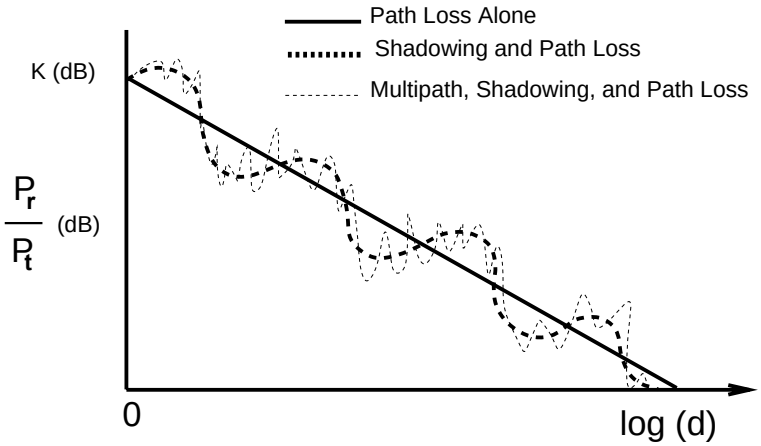


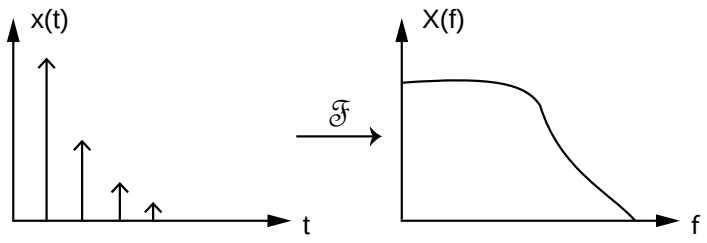
Figure 10: Path Loss, Shadowing and Multipath versus Distance (Copied from Goldsmith (2004) [8]).

From the figure 10, we can see that the impact of the environment is fluctuating a lot depending on numerous factors. It must be noted that very often the setting itself is non-

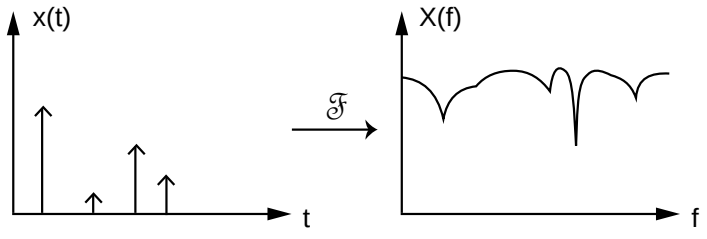
stationary, requiring complex Linear Time-Variant (LTV) modeling schemes to approximate the real environment.

2.2.2 Signal Dispersion

In a guided channel, dispersion of the signal occurs because of the frequency-selective behavior of the physical line. The parasitic inductance of the wire often results in significant attenuation of higher frequency components, causing Inter-Symbol Interference (ISI) due to the time-spreading of the power. The impulse response $x(t)$ of such channel is static; it is modeled as an approximation of a low pass filter (figure 11(a)). Note the equivalent response in frequency domain $X(f)$. In order to compensate signal spreading in cables, passive equalization can be used, such as special line termination.



(a) Dispersion in a cable



(b) Multipath dispersion

Figure 11: Signal spreading models and their frequency response.

Time spreading in wireless channels is a result of various multi-path components, as the time they travel is ultimately longer than that for a direct path. Each multi-path component can be represented by a delayed pulse comprising a time-domain channel response $x(t)$. Numerous arbitrary delays result in significant attenuation of some frequencies, as demonstrated in figure 11(b) by taking Fourier Transform of $x(t)$.

The maximum duration of pulse spreading mainly depends on the location; urban area exhibits many components which arrive relatively fast, open fields with mountains and hills will result in late echo peaks. As all multi-path components are fully dependent on the environment, which is almost never static, the impulse response of the channel is changing with time. The *coherence time* of the channel shows for how long its parameters can be considered constant [7, 7].

Movement of the receiver terminal contributes a lot to signal distortion. Not only that every multi-path component in the model becomes non-static, but also the Doppler effect will influence the transmission. Depending on whether the terminals are approaching or distancing from each other V_{relative} component of terminal velocity V_{term} will be negative or positive. Because of Doppler effect, frequency of the main received signal will drift in either direction by factor called *Doppler frequency* or f_d, as can be seen from the spectrum $X(t)$ of the received signal (figure 12). For multi-path components reflected "behind" the moving terminal, its velocity will appear as reversed, and the frequency will shift in opposite direction.

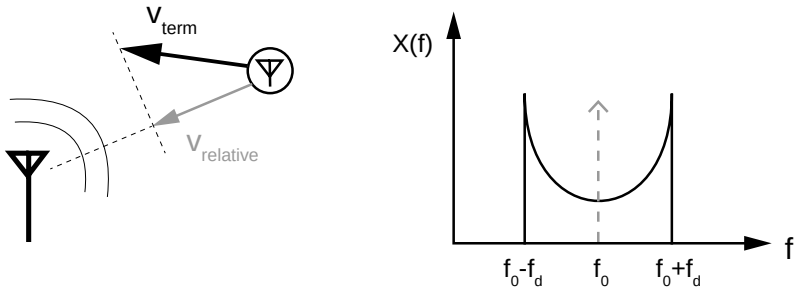


Figure 12: Effect of the Doppler spread due to moving terminal.

From the figure 12 we can observe that the effect of both multi-path propagation and Doppler shift results in spectrum widening of the initial signal with frequency f_0. Maximum displacement of the frequency is then called a *Doppler spread*, a quantity that happens to be inversely related to coherence time (equation 24) [7, 8].

$$\text{Coherence time} \approx \frac{1}{\text{Doppler spread}} \tag{24}$$

Consequences of the wireless channel transmission can be reversed with the adaptive equalization. As already demonstrated, time spreading of the signal can be modeled as a linear filter. This indicates that the received signal can be recovered by applying the opposite filter. Additionally, the time variance of the channel is mitigated by continuously adjusting the input equalization filter.

Most advanced equalization techniques should be implemented in the non-stationary receiver. Carrier recovery is also necessary when frequency and phase of the signal are not constant [2, 336-339].

2.2.3 Channel Model

To make a realistic model of a dynamic wireless channel, its impulse response is approximated and implemented in a form of a digital filter. In order to provide the simulation with realistic parameters, time varying coefficient values $\tilde{a}_n(t)$ of the filter are calculated according to the *scattering function*, which is dependent on a statistical distribution of energy at the receiver. It is known that for the line-of-sight multi-path channel an amplitude of the signal conforms *Rice distribution*, and for channels comprised of only multi-path components *Rayleigh distribution* is used [6, 21].

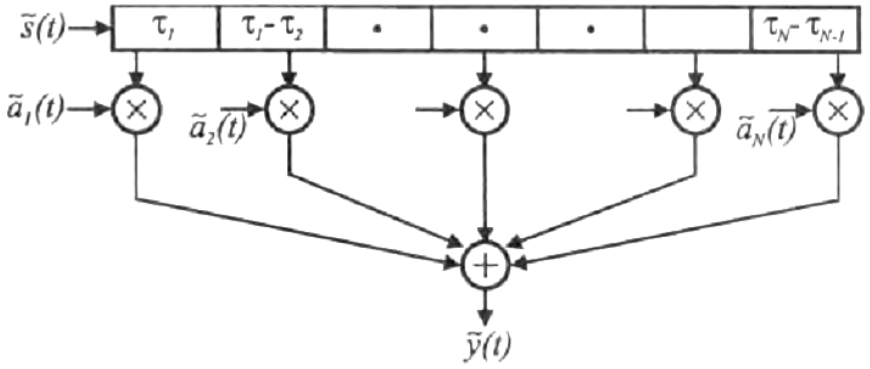


Figure 13: A variable-delay tapped-delay-line channel model (Copied from Jeruchim et al. (2000) [9]).

The time-variant scattering function is derived from the essential channel characteristics: pulse delay profile and Doppler spectral density. Depending on a desired amount of dis-

crete multi-path components, impulse response equivalent to physical is produced. Delay line that realizes this concept is shown in figure 13. Main difficulty with this model is that these real-life variable delays between filter taps are hardly realizable in simulation [9, 573].

One solution to this problem is to approach the bandwidth of the input signal. For that, the channel response $\tilde{c}(\tau, t)$ is additionally convolved with a *sinc* function (figure 14) as in equation 25, where B is a bandwidth of the transmitted signal and T is the period of the delay line. [9, 561-563.]

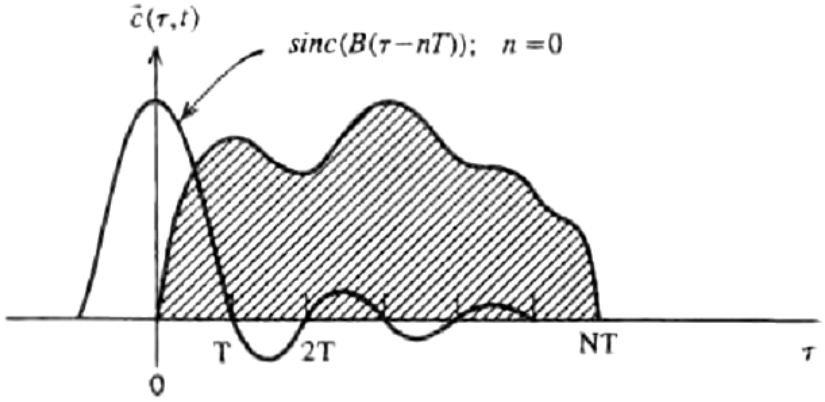


Figure 14: Band-limiting the discrete multi-path model (Copied from Jeruchim et al. (2000) [9]).

$$\tilde{g}_n(t) = \int_{-\infty}^{\infty} \tilde{c}(\tau, t) sinc\big(B(\tau - nT)\big) d\tau \tag{25}$$

The advantage of this technique is to provide us with band-limited channel response which can be effectively sampled and implemented on a uniform delay line.

2.2.4 I and Q Mismatch

In a digital QAM transmission, the receiver device has to deal with an additional set of problems, caused by separate impairments of in-phase and quadrature components of the signal. In the ideal demodulation case, negative frequency elements of the com-

plex transmission should cancel each other. But when the demodulator sine and cosine components are not exact in phase or amplitude, the in-phase signal becomes partially contaminated by the quadrature signal and vice versa [10]. Combined with the frequency-selective properties of the transmission channel, this effect may result in a significant distortion. Figure 15 illustrates the process of signal spectrum overlap with its own mirror image.

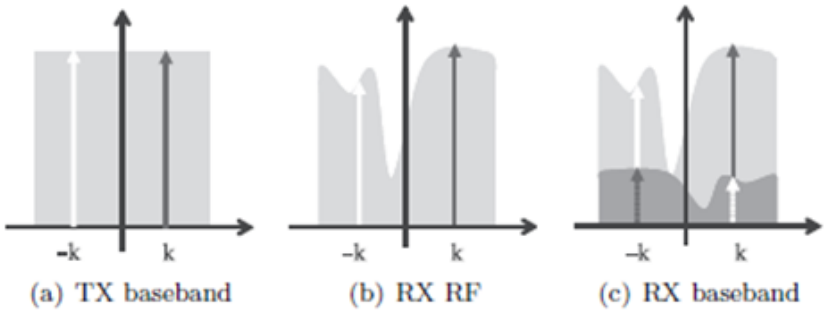


Figure 15: (a) Carrier k and its mirror image –k at the transmitter port. (b) Received signal at RX antenna port. (c) Direct-conversion received signal with IQ mismatch (Copied from Ouameur (2013) [11]).

The mutual interference of real and imaginary components of the complex signal can be modeled by a linear filter with a *complex kernel*. This approach introduces a cross-coupling between the phasor elements, described by the imaginary transfer function of the filter. Likewise, in order to cancel the cross-interference between two signals, a complex equalizer should be used, as described in further sections.

2.3 Adaptive Equalization

2.3.1 Adaptive Filters

In order to reduce the unwanted signal distortion due to a medium that is dynamic, the filtering system has to adapt to a given environment. The idea behind an adaptive filter is basically a continuous readjustment of the filter's processing kernel based on a certain criterion. Usually, the key condition is to fit the output of the system $x(n)$ to match the reference signal $d(n)$. A simplified model of an adaptive filter is shown in figure 16.

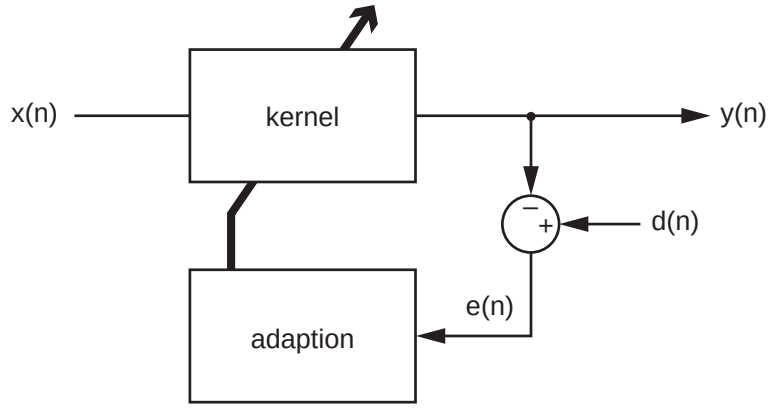


Figure 16: Adaptive filter structure.

In principle, the filter operates in a conventional way: an input signal is processed by the filter kernel and is forwarded to the output. The adaptive part originates from the error value $e(n)$, which is obtained as the difference between output and the reference input. Based on this factor, the adaptive subsystem makes an attempt to improve the filter kernel, forming a kind of feedback loop between output and input via the adaptation mechanism.

Depending on the feedback algorithm, the adjustment process may take time before output results will be satisfying. When the kernel change due to the feedback is settled to a minimal value, the state of the system is said to have *converged* to its maximum efficiency. It may be even so that the system does not converge at all, especially when interference is very random or it is changing very fast. In such cases, a faster adaptive algorithm can be of better use, although the system may become less stable because of that [12, 3].

2.3.2 Linear Equalizer

Since telecommunication signals are almost always facing dynamic interference and distortion, channel equalization is one of the widest application fields for adaptive filters. As the name suggests, a linear equalizer is a type of device that uses a linear coefficient update procedure to counteract the unwanted effects of the environment. [2, 636-637.]

To compensate for the varying characteristics of the channel, an adaptive system should react to the change quickly enough. This property of the equalizer is called *tracking*, and it mainly depends on the type of adaptive algorithm. Knowledge of the initially transmitted data is also quite limited, thus different techniques are used to guide the convergence of the system. Basic transmission scheme with linear equalizer is shown in figure 17.

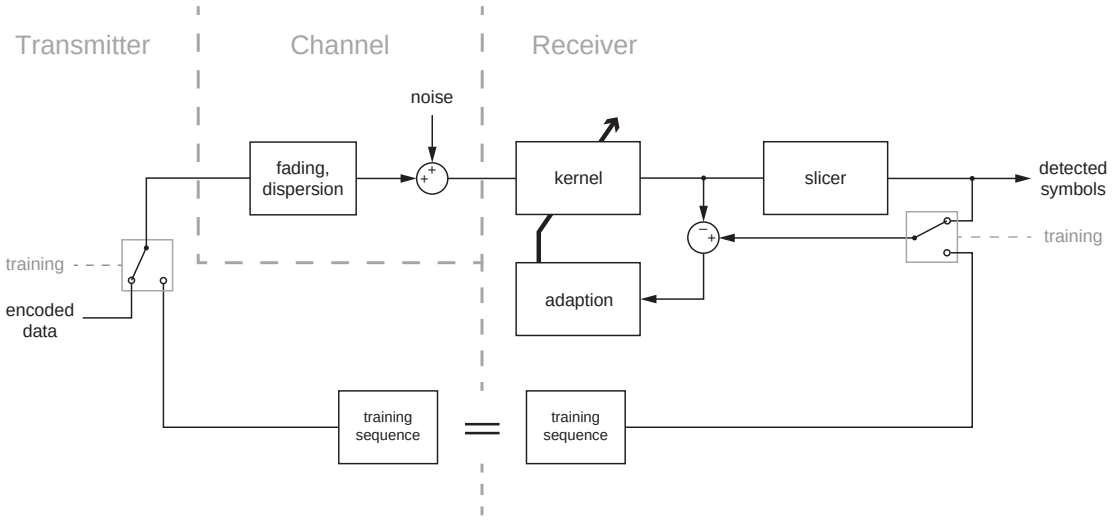


Figure 17: Adaptive linear equalizer.

As can be seen in figure 17, the communication model consists of three basic segments: transmitter, receiver, and channel. As explained in section 2.2, channel parameters such as distortion and fading are modeled as a time-varying transfer function applied to the signal; noise and other interference are additive. In this particular example, modulation and demodulation procedures are omitted by speculating that channel characteristics would ultimately have some effect on the transmitted symbols after demodulation has been done. Equalization of a down-converted signal is known as the *baseband equalization* [2, 648].

On the receiver side, equalizer attempts to improve the obtained baseband signal by compensating the channel imperfections using an appropriate impulse response (kernel). Equalized signal then is fed into the *slicer*, a threshold device used to approximate mapped symbols from the input samples. Adaptation algorithm adjusts the input filter's coefficients according to the error value, provided by the receiver firmware. In general, there are two ways of obtaining the reference signal:

- Training mode
- Decision-directed mode

In training mode, a transmitter sends the sequence of predefined symbols that are already known at the receiver, such that input filter can be "calibrated" with this signal. Example of a training sequence inside a data frame is presented in table 1. After the error has been minimized with training, the equalizer switches to the decision-directed mode and takes a reference signal from the slicer, which will provide a perfectly rounded symbol values.

Table 1: GSM normal burst structure (Adapted from Poole [13]).

Field	Tail	Data	Flag	Training	Flag	Data	Tail	Guard
Length (bits)	3	57	1	26	1	57	3	8.25

In the GSM standard, a transmission burst is structured in such way that the training sequence is placed in the middle of the frame and surrounded by data blocks (table 1). Such arrangement would ease the detection of the exact position of training symbols for the equalizer [13].

2.3.3 Least Mean Squares (LMS) Algorithm

Among various equalizer systems, the LMS algorithm is very common; it is often used in practice because of its simplicity and relative ease of implementation. The LMS filter is one of the fundamental adaptive algorithms and its performance under certain conditions usually serves as a reference for the evaluation of other adaptive filters. [12, 365.]

Block diagram of the adaptive LMS filter is shown in figure 18. Note the vectors represented by bold elements, in contrast to single values shown in regular font. The algorithm repeatedly iterates through three successive phases: signal filtering, error calculation, and filter coefficient update.

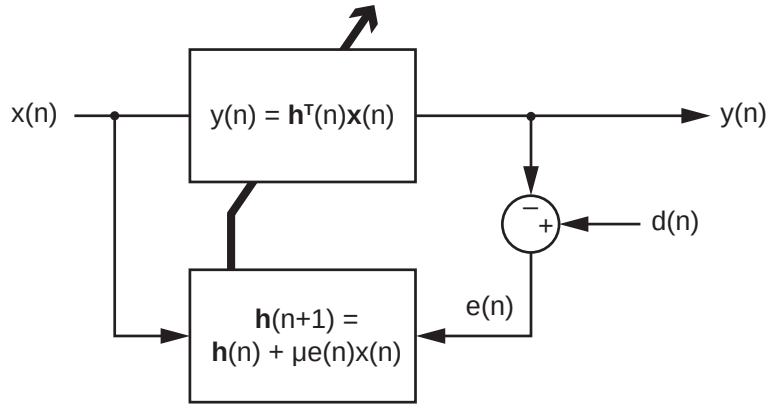


Figure 18: Adaptive LMS filter.

Input signal $\mathbf{x}(n)$ is given as a vector containing present sample followed by $N-1$ samples. Output value of the Finite Impulse Response (FIR) filter is a product between input and a transposed vector of N filter coefficients $\mathbf{h}(n)$ (equation 27).

$$\mathbf{h}(n) = \begin{bmatrix} h_1(n) \\ h_2(n) \\ ... \\ h_N(n) \end{bmatrix}, \quad \mathbf{x}(n) = \begin{bmatrix} x(n) \\ x(n-1) \\ ... \\ x(n-N+1) \end{bmatrix} \tag{26}$$

$$y(n) = \mathbf{h}^{\mathsf{T}}(n)\mathbf{x}(n) \tag{27}$$

$$= \sum_{i=1}^{N} h_i(n)x(n-i+1) \tag{28}$$

$$e(n) = d(n) - y(n) \tag{29}$$

$$\mathbf{h}(n+1) = \mathbf{h}(n) + \mu e(n)\mathbf{x}(n) \tag{30}$$

When the output of the process is known, error $e(n)$ is obtained as a difference between output $y(n)$ and the latest reference signal sample $d(n)$ (equation 29). Following that, the coefficients of the filter are updated to minimize the output mean squared error $E\left[\left(d(n) - y(n)\right)^2\right]$. Coefficient vector $\mathbf{h}(n+1)$ for the next iteration is obtained from the sum of the current coefficient vector $\mathbf{h}(n)$ with the weighted input vector $\mathbf{x}(n)$ (equation 30). The input vector is scaled with the error value $e(n)$ and the adaptation rate μ.

Hardware implementation of the LMS filter is straightforward because its algorithm uses plain math operations: vector addition, and FIR-specific multiply-and-accumulate (dot product). In figure 19, an example diagram of LMS equalizer is presented.

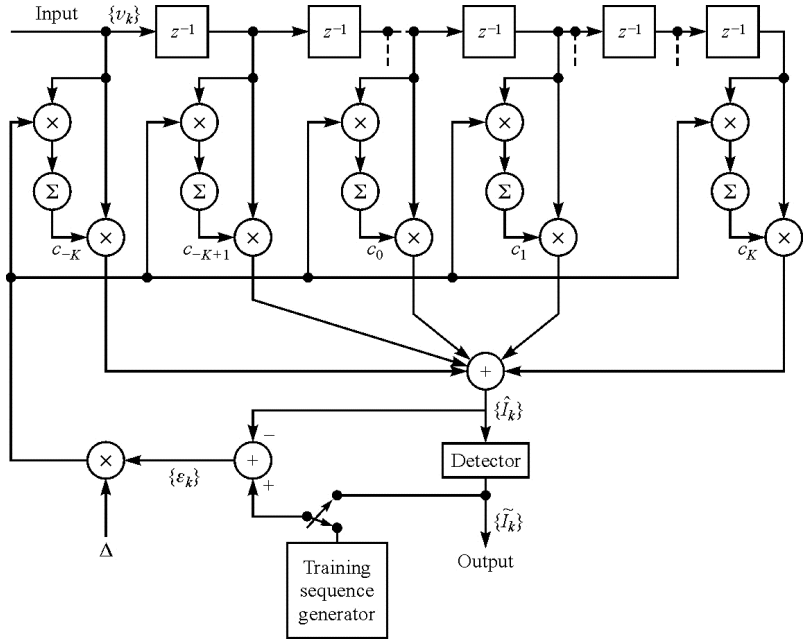


Figure 19: LMS channel equalizer (Copied from Proakis (2000) [2]).

Main trade-off in the LMS filter implementation is to select the proper adaptation rate μ. If a too big value is used, the algorithm can be destabilized by fluctuations in the input signal, and may not converge at all or even cause accumulator overflow. A small μ value will cause the algorithm to converge slowly, which reduces the tracking capability of the equalizer.

To optimize the adaptation rate of the algorithm, a slightly different kernel update procedure is used. Now prior to the filter coefficient modification, the input vector is additionally scaled as shown in the equation 31. This approach normalizes the necessary correction to the $\mathbf{h}(n)$, hence it is known as *normalized* LMS algorithm. [12, 437.]

$$\mathbf{h}(n+1) = \mathbf{h}(n) + \frac{\hat{\mu}e(n)}{\|\mathbf{x}(n)\|^2 + a}\mathbf{x}(n), \quad 0 < \hat{\mu} < 2 \tag{31}$$

An input vector is scaled with the value of squared Euclidean norm. A small constant a is added to prevent an overflow in case of the zero input vector. The adaptation constant $\hat{\mu}$ controls the convergence rate of the algorithm but does not have a dramatic effect on its stability.

2.3.4 Complex LMS Equalizer

The complex interference due to the IQ mismatch introduced in section 2.2.4 can not be simply removed by filtering signal components separately. Isolated real-valued signal processing does not utilize the knowledge of the interference origin, which is mostly the another component of the complex signal. As a means to mitigate this entangled effect, complex-valued equalization algorithm can be used at the receiver. Figure 20 depicts the structure of such mechanism.

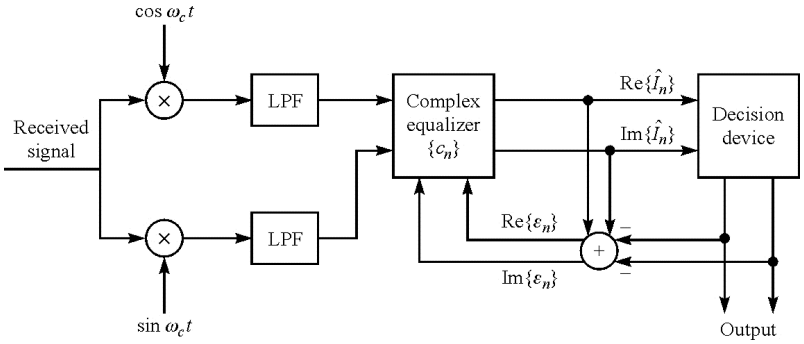


Figure 20: Complex baseband channel equalizer (Copied from Proakis (2000) [2]).

Operation of the complex equalizer follows the general procedure of filtering, symbol detection and adaptation, as previously described in section 2.3.2. The main extension of the LMS algorithm here is that all of its elements are complex, each has real and imaginary part for in-phase and quadrature components respectively, as shown in equations 32-36.

$$\mathbf{h}(n) = \mathbf{h}_I(n) + j\mathbf{h}_Q(n) \tag{32}$$

$$\mathbf{x}(n) = \mathbf{x}_I(n) + j\mathbf{x}_Q(n) \tag{33}$$

$$d(n) = d_I(n) + jd_Q(n) \tag{34}$$

$$y(n) = y_I(n) + jy_Q(n) \tag{35}$$

$$e(n) = e_I(n) + je_Q(n) \tag{36}$$

Using the complex arithmetics, filtering operation can be decomposed to four real-valued vector products, as shown equation 37. Error vector is obtained separately from each element (equation 38). [12, 372-373.]

$$y(n) = \mathbf{h}^{\mathsf{T}}(n)\mathbf{x}(n) \quad \begin{cases} y_I(n) = \mathbf{h}_I^{\mathsf{T}}(n)\mathbf{x}_I(n) - \mathbf{h}_Q^{\mathsf{T}}(n)\mathbf{x}_Q(n) \\ y_Q(n) = \mathbf{h}_Q^{\mathsf{T}}(n)\mathbf{x}_I(n) + \mathbf{h}_I^{\mathsf{T}}(n)\mathbf{x}_Q(n) \end{cases} \tag{37}$$

$$\begin{cases} e_I(n) = d_I(n) - y_I(n) \\ e_Q(n) = d_Q(n) - y_Q(n) \end{cases} \tag{38}$$

$$\mathbf{h}(n+1) = \mathbf{h}(n) + \mu e(n)\mathbf{x}^*(n) \quad \begin{cases} \mathbf{h}_I(n+1) = \mathbf{h}_I(n+1) + \mu\big[e_I(n)\mathbf{x}_I(n) + e_Q(n)\mathbf{x}_Q(n)\big] \\ \mathbf{h}_I(n+1) = \mathbf{h}_I(n+1) + \mu\big[e_Q(n)\mathbf{x}_I(n) - e_I(n)\mathbf{x}_Q(n)\big] \end{cases} \tag{39}$$

Note that the conjugate of the input vector is required to obtain the correct filter coefficients during the adaptation procedure [14]. For this reason, the quadrature component $\mathbf{x}_Q(n)$ is taken as negative as shown in equation 39.

On the hardware, complex filter is usually based on four real-valued filters, as shown on the figure 21. This signal flow diagram includes error calculation for the LMS algorithm, where vector input is emphasized in thick arrows, and single samples are shown as thin lines. Vector $\hat{\mathbf{w}}(n)$ stands for a filter weights estimate and $\mathbf{u}(n)$ is the input sample array. Similarly, the LMS coefficient update is done with four vector multiplications and four additions. This process is illustrated in figure 22, where $z^{-1}I$ is a delay of one sample.

Normalized complex LMS algorithm can be realized in the same way as with the real-valued algorithm, covered in section 2.3.3. During the coefficient update procedure, both components of the complex input vector are normalized by its squared Euclidean norm, as shown in equation 40.

$$\mathbf{h}(n+1) = \mathbf{h}(n) + \frac{\hat{\mu}e(n)}{\|\mathbf{x}(n)\|^2 + a}\mathbf{x}^*(n), \quad 0 < \hat{\mu} < 2 \tag{40}$$

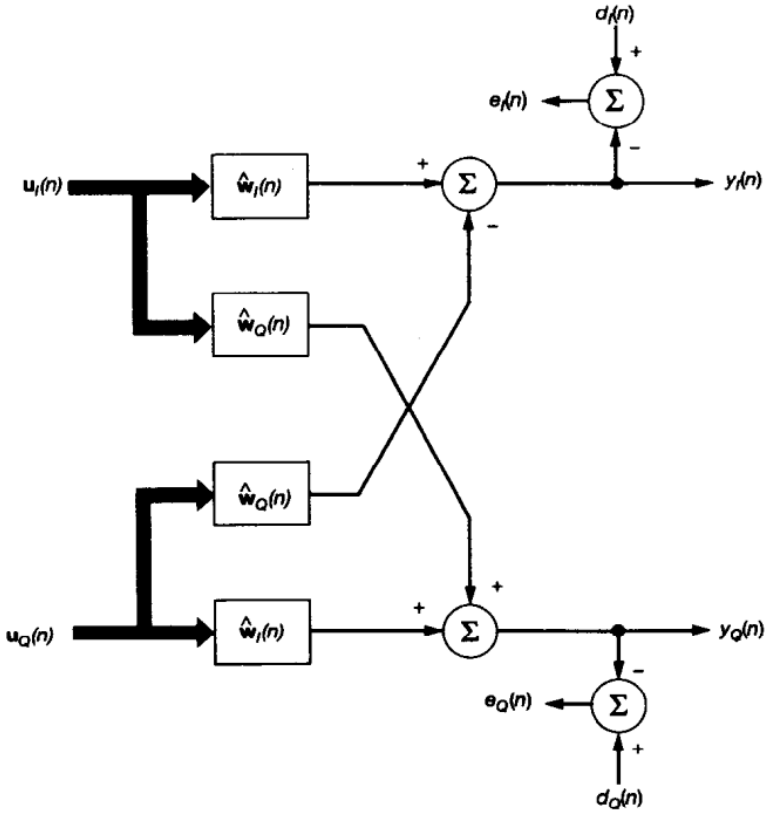


Figure 21: Complex signal filtering with error calculation (Copied from Haykin (1995) [12]).

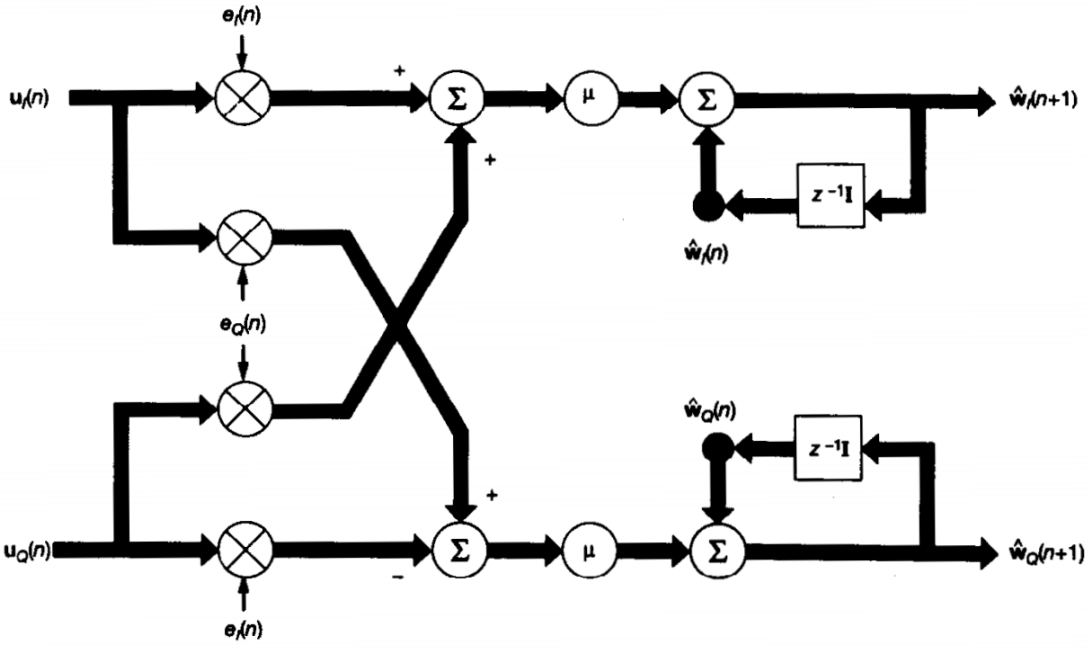


Figure 22: Adaptation of complex LMS filter (Modified from Haykin (1995) [12]).

$$\|\mathbf{x}(n)\|^2 = \sqrt{\|\mathbf{x_I}(n)\|^2 + \|\mathbf{x}_Q(n)\|^2}^2 = \sum_{i=1}^{N} \left(x_I(i)^2 + x_Q(i)^2\right) \tag{41}$$

From the equation 41, we can see that by using the definition of a complex vector norm, the normalization value can be obtained as a sum of every element of an input vector $\mathbf{x}(n)$ squared.

2.4 Parallel Processing Models

The past obstacle of computing such as limited instructions per second is easily overthrown with modern multi-core processors. However, to utilize such hardware efficiently, much more work should be done in the software design. Distribution of workload between processors is not an easy task, and different parallel processing models exist for this purpose. [15, 3.]

The **hierarchical model** that is shown in figure 23, suggests that one processing unit is explicitly controlling others. This implies that the master task is more aware of the whole processing activity than slave tasks, and the system decisions are mainly centralized. This approach is good when operating on large amounts of data e.g. image processing. Given a substantial amount of data, the system can divide the batch efficiently between its subsystems, such that useful work is independently done in parallel. Despite the high efficiency of concurrent processing, it still suffers from computing overhead due to the task scheduling.

The **flow model** is presented in figure 24. Systems in this configuration are processing data in a conveyor-like manner. Its decentralized structure implies that the system is optimized for a particular algorithm without many possible variations. This arrangement of workload is optimal for systems that execute a very complex processing on the streaming data. The design of the flow system is quite challenging, as data throughput depends on nearly every component and bottlenecks are inevitable.

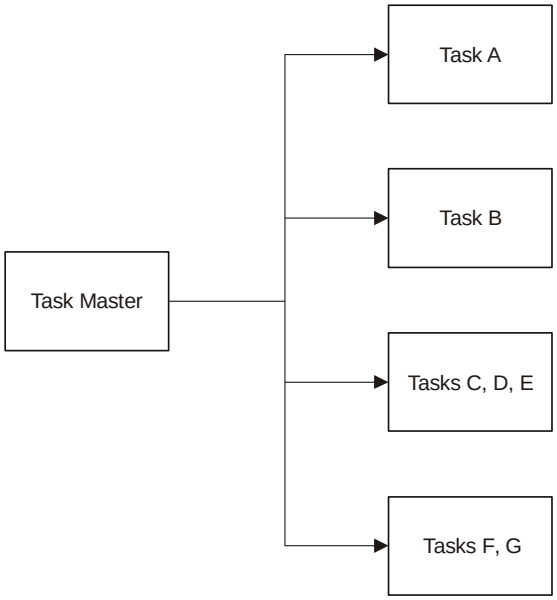


Figure 23: Hierarchical processing model (Copied from TI (2012) [15]).

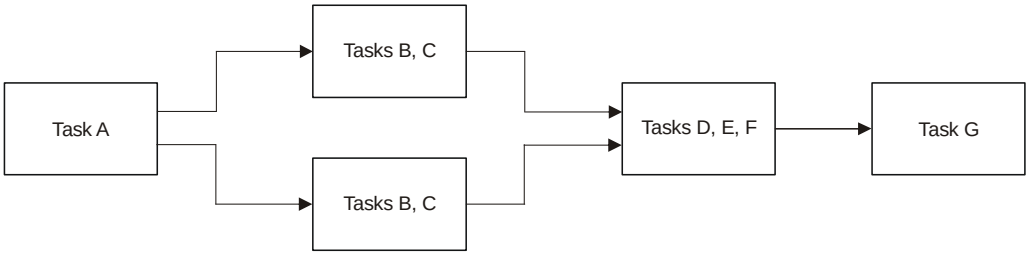


Figure 24: Flow processing model (Copied from TI (2012) [15]).

Before the implementation of digital signal processing on a distributed computing platform, the elements of an algorithm should also be analyzed in terms of *cohesion* and *coupling* [15, 9-10].

Coupling of a separate process indicates how much it depends on the other parts of the program in the general sense of input and output. Parts of the algorithm with low coupling are not very dependent on the other elements in terms of data flow, and therefore they can be safely divided for consecutive execution. Highly coupled parts can not be separated easily, and are usually grouped together.

Cohesion of the system is an abstract measure of the correlation of its subprocesses. Program elements that have a high cohesion factor tend to be easily reusable for multiple actions. Low cohesion systems can be recognized when functions of its sub-elements are barely related. High coupling usually corresponds to low cohesion and vice versa.

3 Methods and Materials

3.1 Implementation Structure

The system for evaluation of the equalizer algorithm is composed of two parts: simulation of the transmission and equalizer software implementation. Because both performance and optimization possibilities of the multi-core software are of main interest here, all of the other components of the transmission model such as modulation, channel, and data analysis are simulated separately. Digital signal is provided to the isolated equalizer application and then recovered after processing is done. Structural diagram of the system is illustrated in figure 25.

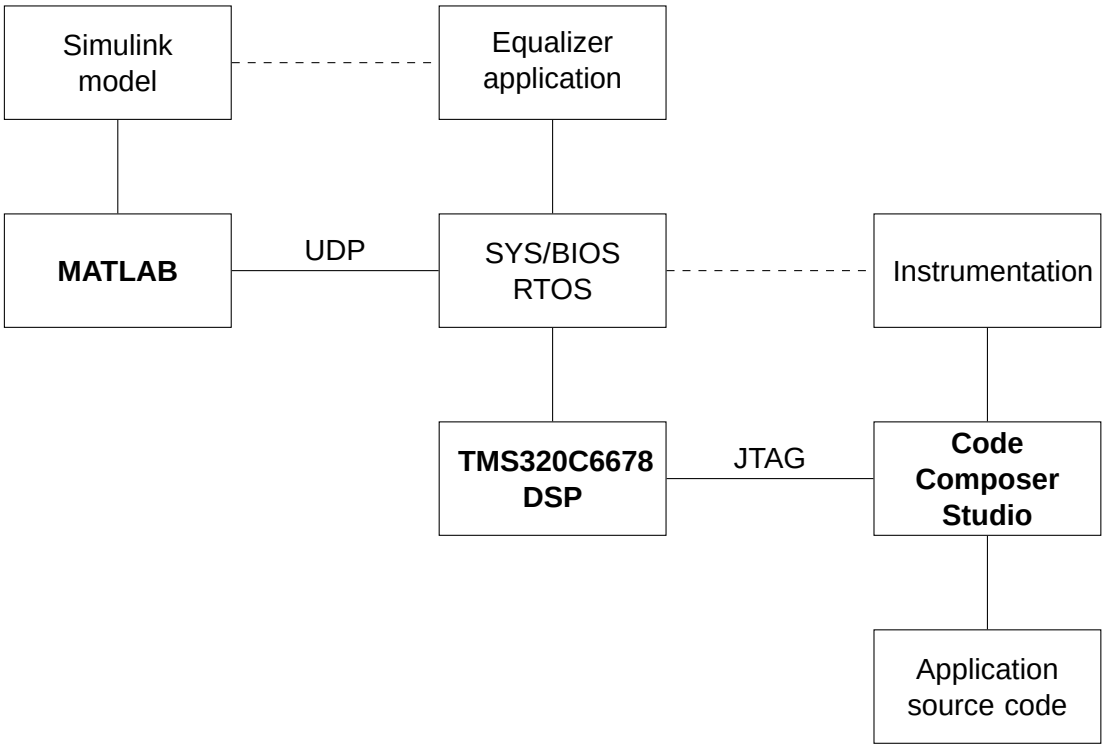


Figure 25: Implementation diagram of the equalization system.

The equalizer software is established on a multi-core TMS320C6678 Digital Signal Processor (DSP) Evaluation Module (EVM). Signal processing algorithm is intentionally integrated into SYS/BIOS Real-Time Operating System (RTOS) to ease the implementation of parallel processing. Transmission simulation is done in Simulink, which is integrated with the MATLAB software. With networking capabilities of both RTOS and MATLAB, interconnection between the model and application is realized with User Datagram Protocol (UDP).

Development of the software is done in Code Composer Studio (CCS). Compiled program image is uploaded into DSP cores using JTAG interface. This access method also allows to debug the firmware execution and collect logged data directly from the hardware. Based on that, the *instrumentation* module enables to perform various RTOS benchmarks such as evaluation of execution time and load of the cores.

3.2 Transmission Simulation

3.2.1 Simulink Model

Simulation of the transmission procedure is carried out in the data flow based environment Simulink. It is a quite versatile software in terms of visual planning, data analysis and model complexity. Simulink models are presented to the user as a combination of functional units in a block diagram. Figure 26 illustrates the digital transmission chain model implemented for this project.

As can be seen from the block diagram, the test data sequence is obtained from the random integer generator block and modulated using the 16-QAM scheme. Transmission channel effects are reproduced by additional processing of the complex baseband signal. Equalizer subsystem block attempts to recover the initial complex signal based on the "clean" reference which is provided separately. In this configuration, the equal-

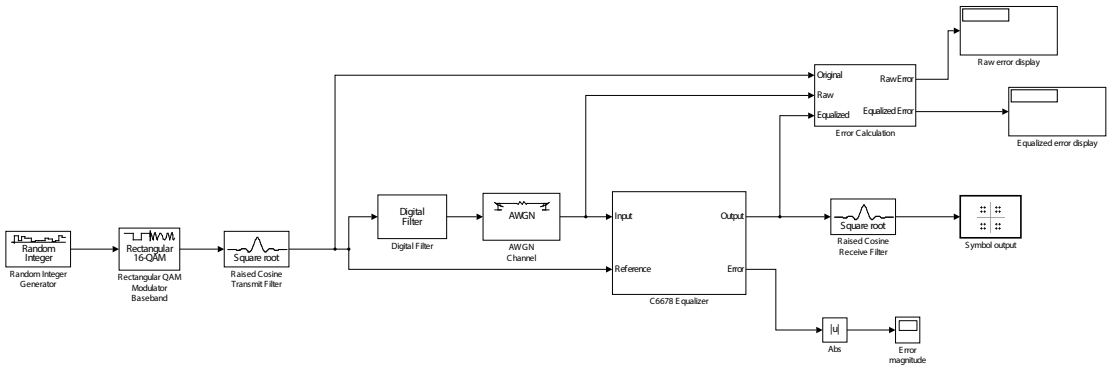


Figure 26: Simulink model of the transmission chain

izer algorithm is assessed in conditions of an constant training sequence, as explained in section 2.3.2.

Baseband output of the equalizer is demodulated and the recovered symbols are given to the complex scatter plot. As a means to evaluate the convergence of the algorithm, the magnitude of obtained error vector is logged with the scope block. Bit error rates for raw and equalized signals are compared by using additional error calculation block that individually demodulates each signal and compares the obtained bits.

In order to replicate continuous processes, Simulink evaluates the mathematical model of each functional block based on a discrete time steps, specified in the simulation settings. This also allows to locally emulate the digital sampling by using decimation, given that the sampling rate of the block is a multiple of the fundamental simulation step. In this particular model, the discrete step is set to 10 microseconds to accommodate the upsampling procedures covered in the section 3.2.2. Neither fundamental step nor modulation sampling frequency do not affect the adaptive algorithm performance.

3.2.2 Conversion to Baseband

Arbitrary experimental data is obtained from the random integer generator in frames of 8 samples. Baseband data is obtained by line encoding and pulse shaping, introduced in

sections 2.1.1 and 2.1.4 respectively. Simulation of this process is performed with block chain presented in figure 27.

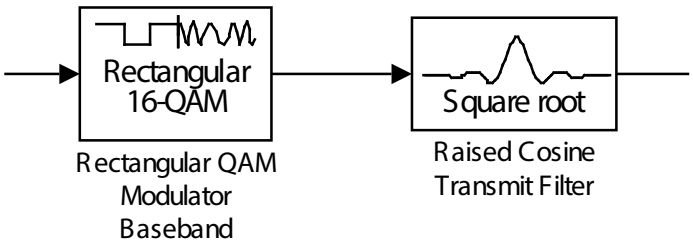


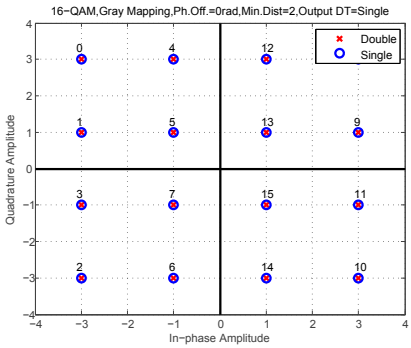
Figure 27: Modulator and pulse shaping block

QAM modulator block converts the digital sequence into a complex signal by mapping the integers in the range $[0, n-1]$ onto constellation with n symbols. The constellation diagram used in this simulation is shown in figure 28(a). Note that symbols are arranged according to the Gray code in order to reduce the bit error rate. The resulting complex signal (figure 28(b)) conveys the instantaneous symbol coordinates, at a rate of one sample for each symbol.

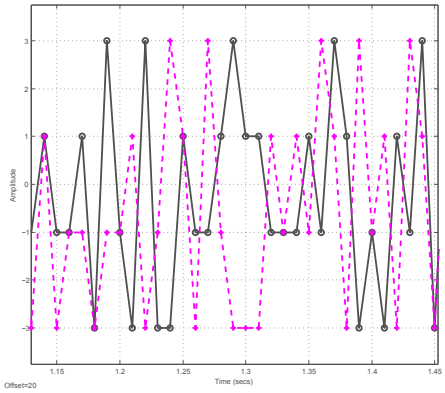
Right after the QAM modulation, the raised-cosine pulse shaping filter is applied to the signal. The impulse response of the filter is presented in the figure 28(c). Figure 28(d) illustrates the removal of quick transitions in the modulated signal. In order to fit multiple baseband data points for one modulated symbol, the resulting signal is *upsampled* by a factor of 8. This means that every 8-symbol frame from the modulator is converted into a smooth waveform consisting of 64 samples. Sampling period for the baseband signal is thus shrinked to keep the constant symbol rate throughout the whole system.

3.2.3 Digital Channel

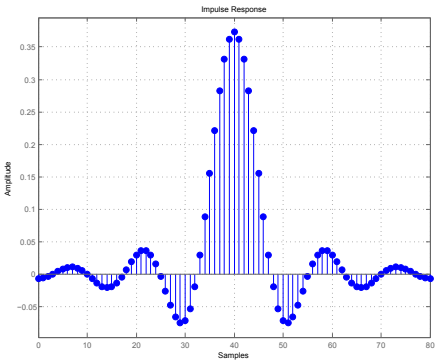
Effects of the wireless transmission channel are imitated with digital filtering and noise addition, as shown in figure 29. In this simulation, a very simple static baseband digital channel is used to evaluate the adaptation of a given equalizer. As an extension to the propagation effect, a small amount of Gaussian-distributed white noise is added to the signal to achieve Signal-to-Noise Ratio (SNR) of 40 dB.



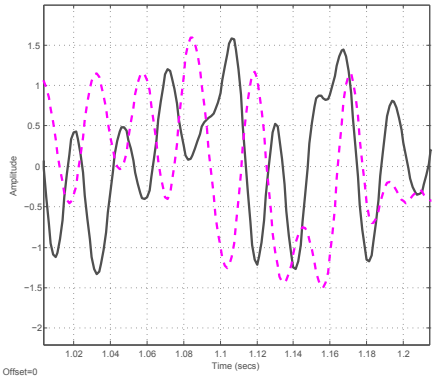
(a) Rectangular 16-QAM constellation



(b) Modulated complex signal



(c) Raised cosine filter kernel



(d) Filtered baseband signal

Figure 28: Modulation and pulse shaping

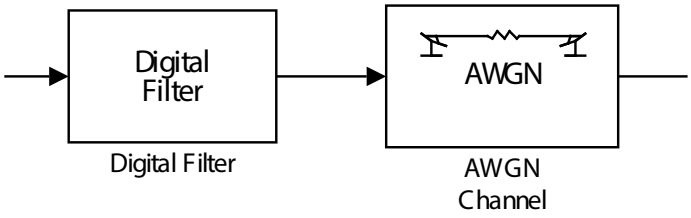
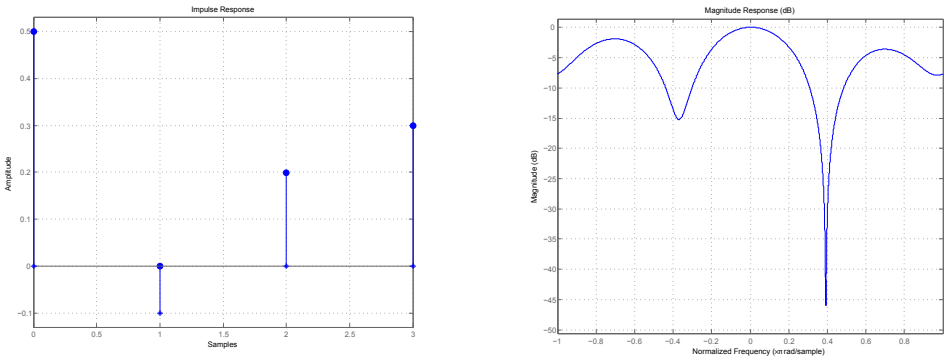


Figure 29: Digital filter and white noise adder blocks

The digital filter block is manually set up with an FIR-type filter that imitates the multi-path propagation. The kernel of this filter is presented in equation 42 and illustrated in figure 30(a). From the impulse response, we can see the power spreading pattern. One element that is indicated with a star marker corresponds to imaginary part of the complex filter and introduces a small leakage of baseband signal components into one another, as previously explained in section 2.2.4.



(a) Impulse response (b) Frequency response

Figure 30: Channel model characteristics

$$H(z) = 0.5z^0 - j0.1z^{-1} + 0.2z^{-2} + 0.3z^{-3} \tag{42}$$

From the figure 30(b) we can see the frequency-selective properties of the filter obtained by taking a Fourier transform of its impulse response. Note that spectrum graph is a function of frequency that is *normalized* to the input sampling rate.

3.2.4 Equalizer Subsystem

To provide a link between the transmission simulation and the equalizer that is implemented on a separate hardware, a special subsystem is created. It operates as a function block, thus giving an interface for communication with the equalizer program via Ethernet using User Datagram Protocol (UDP). Block diagram of the subsystem is shown in figure 31.

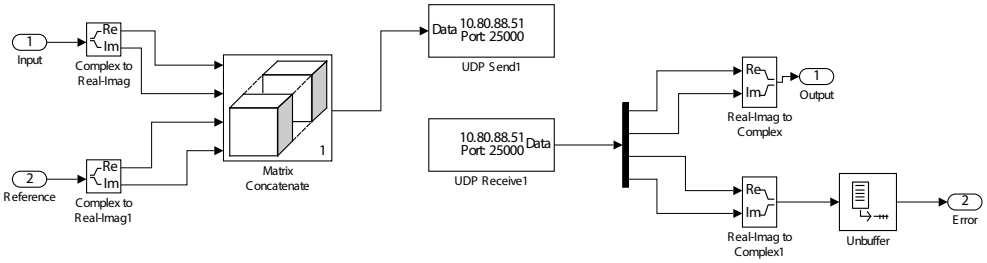


Figure 31: Equalizer subsystem

The subsystem block is provided with two complex input frames: input signal and reference signal, both are 64 samples in length (single precision floating point). During the operation, real and imaginary parts of both input and reference signals are separated and concatenated into one large frame of 256 samples. This cluster is then sent with UDP Send block to the device network address and particular receive port, where equalizer program will fetch the received data. The structure of the data frame is presented in table 2.

Table 2: Structure of the simulation data frame

Equalizer input	I signal	Q signal	I reference	Q reference
Equalizer output	I output	Q output	Error Re	Error Im
Length (SP Float)	64	64	64	64

Equalization results are obtained in a similar way by using UDP Receive block that listens to the output port of the network device. The incoming data frame is split into sections of 64 samples and merged into complex output signal and complex error vector. Execution priority in the subsystem is given to the UDP Send block so that no data is demanded from the equalizer before it is provided with the data packet.

Software implementation of the network interface for equalizer algorithm and the data processing sequence are described in the following sections.

3.3 Equalizer Implementation

3.3.1 Hardware Setup

The equalizer software is implemented on a TMS320C6678 multi-core DSP by Texas Instruments. This hardware solution contains all vital elements for the concurrent software development: eight identical DSP cores, shared memory controller, numerous I/O peripherals, and debugging possibility. Figure 32 depicts the functional block diagram of the processor.

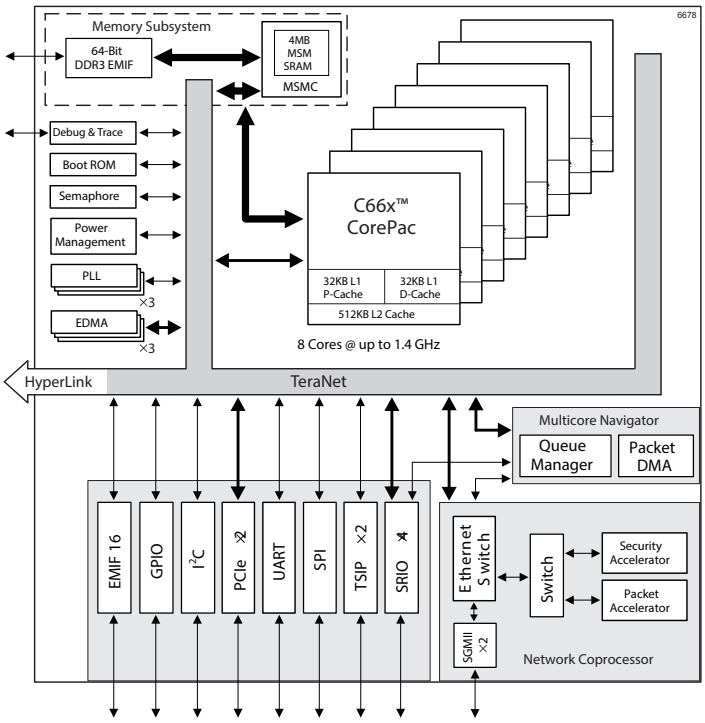


Figure 32: TMS320C6678 functional block diagram (Copied from TI (2010) [16]).

The DSP device is incorporated into an Evaluation Module (EVM) that features necessary hardware extensions such as USB debugger unit, physical network layer module, external DDR memory and many other useful interfaces that allow to utilize the processor in a most efficient way. Figure 33 shows the EVM in operation, connected to the power supply, Local Area Network (LAN) and to the PC workstation via USB cable.

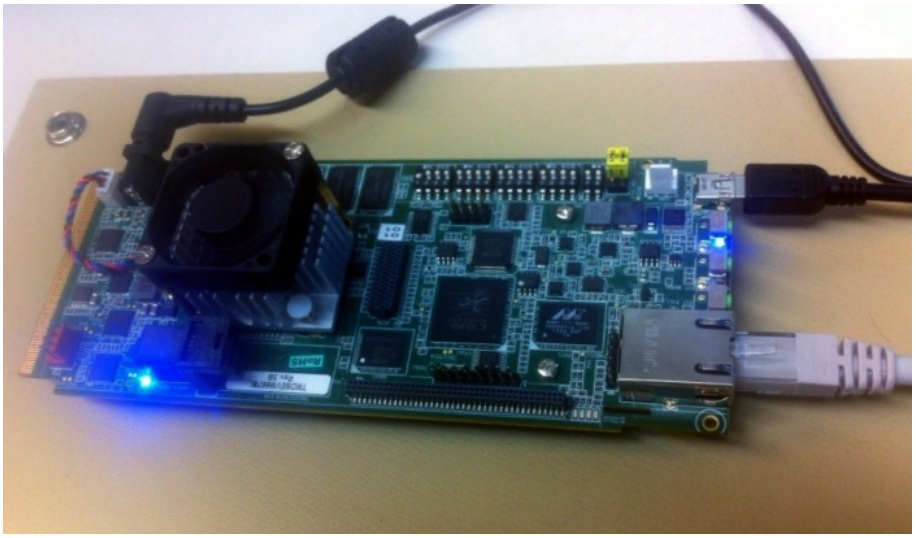


Figure 33: TMS320C6678 EVM in operation

As the multi-core processor development platform, the TMS32C6678 EVM structure is optimized for handling of large amounts of data, such as image and video processing. It does not feature analog interfaces, implying that digitized information should be provided separately, for example from an external ADC. In this sense, the Ethernet interface of the EVM is very advantageous, as it simplifies access to the hardware by sharing processing power via the universal data network.

3.3.2 Application Structure

Adaptive equalizer application utilizes DSP resources to process the pre-formatted data provided via the network interface. A temporary state of the adaptive system then is stored in the local memory, and processed data is returned back to the sender. The system is implemented using three programs running concurrently on separate DSP cores. Schematic diagram of the system is presented on Figure 34.

The main program resides on the 0th or the *master* core and takes care of the network data handling such as DSP packet reception, frame distribution between cores and the collection of processed data. Additionally, this program executes non-vital signal processing tasks during idle time, thus improving the overall algorithm.

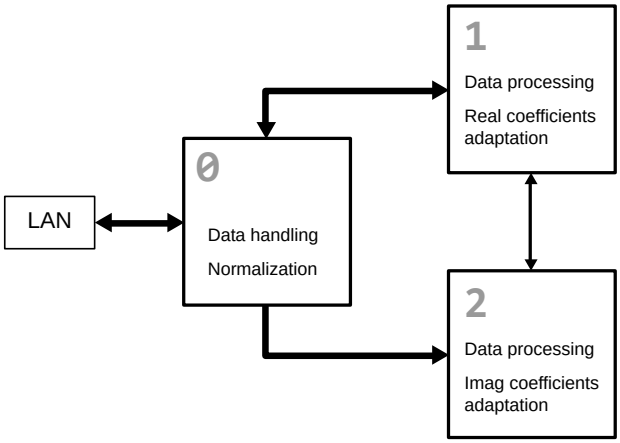


Figure 34: Structure of the equalizer application

Auxiliary cores 1 and 2 are taking full workload in signal processing. Note that between cores there is a cross-coupling that is necessary for the algorithm operation. Together, these sub-processors have to execute mostly identical operations, so both are running the same program image. Minor code additions come in effect when this image starts on the 1st core. Further on this augmented version of the processing program will be referred as the *sub-main*, indicating that it has to collect and return output to the master core.

3.3.3 User Datagram Protocol (UDP) Interface

Because the communication between the Simulink model and equalizer application is quite straightforward, it is implemented by using light-weight User Datagram Protocol (UDP). In comparison to the Transmission Control Protocol (TCP), which has methods of ensuring the data delivery and connection monitoring, the UDP is a connectionless data transfer protocol. As a trade off to very little packet processing overhead, each party in UDP communication has no guarantee that data is arrived at the destination. In a simple case of simulation data streaming via LAN, the UDP connection should be enough.

On the software side, several abstract functions provide the necessary interface for the UDP transfer, such as packet sending and receiving. The logic diagram of communication between server and client is presented in figure 35.

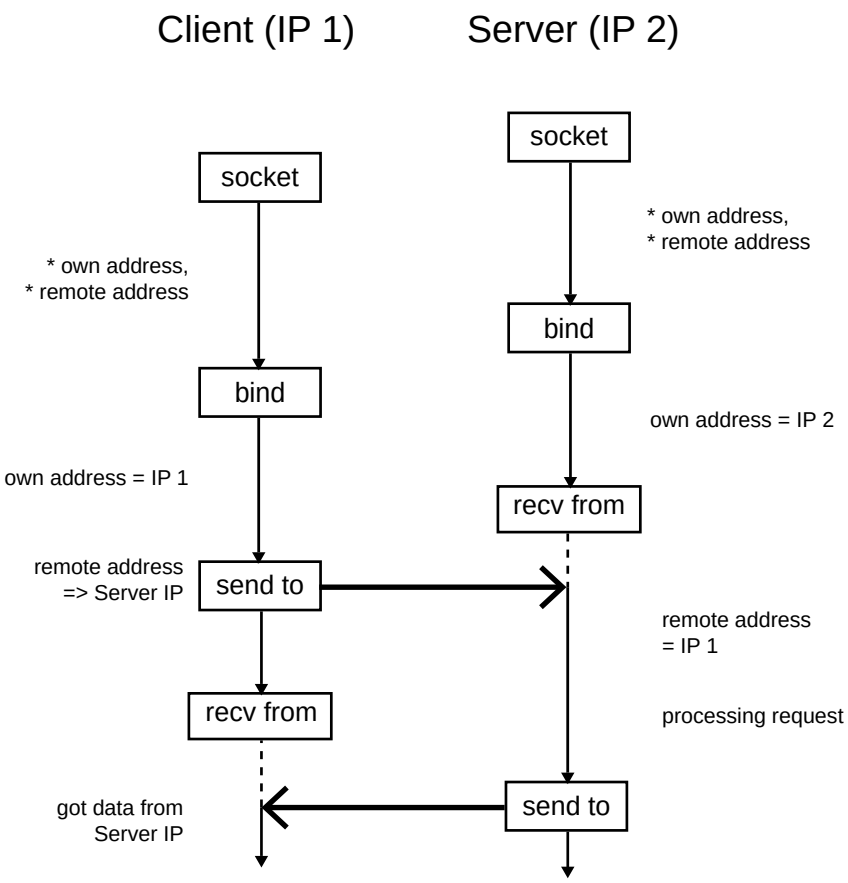


Figure 35: Diagram of the UDP communication

Initially, each side of communication on their own behalf should create a local data structure referred as the socket. The socket is used as a transmission proxy so that the program can listen for the incoming data trough it. To initialize the socket, the program should apply a combination of the device Internet Protocol (IP) address and own communication port to the socket with the bind command. Implementation of the socket on the DSP is presented in listing 1.

```
1  SOCKET  s;
2  struct  sockaddr_in sin1;
3  struct  sockaddr_in sin2;
4
5  // Create test socket
6  s = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
7  if( s == INVALID_SOCKET )
8  {
9      printf("failed socket create (%d)\n",fdError());
10     goto leave;
11 }
12
```

```
13 // Bind the network address to the local socket
14 bzero( &sin1, sizeof(struct sockaddr_in) );
15 sin1.sin_family      = AF_INET;
16 sin1.sin_len         = sizeof( sin1 );
17 sin1.sin_addr.s_addr = inet_addr(LocalIPAddr);
18 sin1.sin_port        = htons(25000);
19
20 if( bind( s, (PSA) &sin1, sizeof(sin1) ) < 0 )
21 {
22     printf("failed bind (%d)\n",fdError());
23     goto leave;
24 }
```

Listing 1: Socket creation and bind

After binding, the program can participate in the data transmission by using commands *sendto* and *recvfrom*. Command **sendto** will transmit the packet to the specified network address, with the own IP address and port attached to the packet header, so the receiver device would be able to tell where the data is from. Command **recvfrom** will block the program execution until some packet will arrive at the socket or the time-out period will expire. If any data is received, the program additionally gets service information such as sender address and the length of the data. In this way, the server would know to whom it should return the processed request (figure 35).

```
1  while(1)
2  {
3    tmp = sizeof( sin2 );
4
5    // Attempt to receive data from the socket
6    i = (int)recvncfrom( s, (void **)&pBuf, 0, (PSA)&sin2, &
       tmp, &hBuffer );
7
8    // If data received then go into processing mode
9    if( i >= 0 )
10   {
11     // Process the data buffer ...
12
13     // Return the data back to the sender
14     if(sendto( s, pBuf, sizeof(float)*FRAME*4, 0, (PSA)&sin2
         , sizeof(sin2) ) < 0)
15       printf("send failed (%d)\n",fdError());
16   recvncfree( hBuffer );
17   }
18 }
```

Listing 2: Data recovery and sending

From the listing 2, we can see the use of the send and receive commands in the main core program. By using an infinite loop, it attempts to recover data from the socket and process any incoming payload if number of received bytes is non-zero. If the timeout period expires, the system simply goes for another waiting round. In this program, we assume that the only incoming data is originated from the Simulink model and that it is correctly pre-formatted as explained in section 3.2.4. Shortly after receiving the simulation data, it will be distributed between auxiliary cores via the Inter-Process Communication (IPC) interface introduced in section 3.3.6.

3.3.4 LMS Filter Programming

A normalized version of the LMS algorithm (figure 36) covered in section 2.3.3 was implemented in a form of subroutine, so that the functionality of the adaptive filter can be called from any part of the program code to operate on the provided signal vectors and filter kernel. For the reasons of logical simplicity, all mathematical operations such as vector multiplication and scaling are implemented using basic repeated iterations, or in other words with loop statements. Listing 3 contains the source code of the normalized LMS filter based on the TI digital signal processing Natural C library [17].

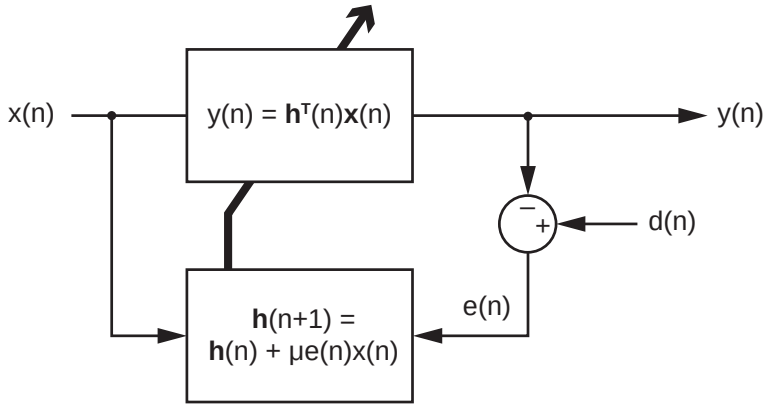


Figure 36: Adaptive LMS filter.

```
1 void lms(const float *x, float *h, const float *y_i,
2     float *y_o, const float ar, float *error, const int nh,
        const int nx)
3 {
4     int i, j;
```

```
5       float sum, norm;
6       const float a = 1.1755e-38;
7
8     // Take the shifted input vector until the end of frame
9       for (i = 0; i < nx; i++)
10      {
11      // Dot product of the kernel and input arrays
12          sum = 0.0f;
13          for (j = 0; j < nh; j++)
14              sum += h[j] * x[i+j];
15
16      // Error calculation
17          y_o[i] = sum;
18          error[i] = y_i[i] - sum;
19
20      // Euclidean norm as a sum of squares
21        norm = 0.0f;
22          for(j = 0; j < nh; j++)
23            norm += x[j+i] * x[j+i];
24
25      // Kernel update with normalized input
26          for (j = 0; j < nh; j++)
27              h[j] = h[j] + (ar * error[i] * x[i+j]) / (norm +
                    a);
28      }
29
30      return;
31  }
```

Listing 3: Normalized LMS filter function

From the lines 22 and 23 of the source code presented in listing 3, we can notice that the input normalization is done by summing the squared input vector values. Small constant a is added to the norm to inhibit the possible division by zero.

The input for the LMS algorithm function is presented by an array of $[n + h - 1]$ samples, where n is the length of the input vector and h is the length of the FIR filter kernel. For the proper continuous filtering, it is necessary to provide the LMS function with the vector that always contains the $h - 1$ past input values preceding the current sample, in the form illustrated in figure 37.

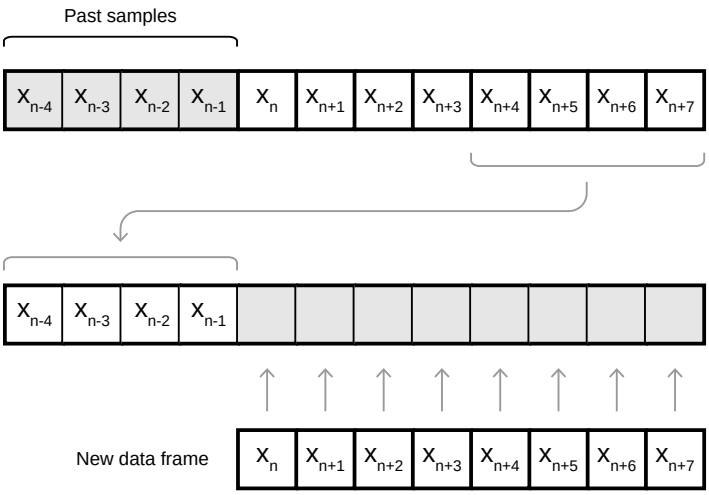


Figure 37: Input data block update procedure

When the data frame processing is finished, the main program shifts the last samples to the beginning of the input vector, as shown in figure 37. New values are copied to the appropriate position, thus realizing a kind of a *circular buffer*.

Similarly, the complex LMS filter function is implemented as an extended version of the basic LMS routine; it is presented in listing 4. As previously described in section 2.3.4, each part of the complex procedure is a combination of several real-valued operations based on the initial algorithm.

```
1  void lms_cplx(const float *x_i1, const float *x_q1,
2      float *h_i1, float *h_q1,
3      const float *y_ii1, const float *y_qi1,
4      float *y_io1, float *y_qo1,
5      const float ar, float *error_i, float *error_q,
6      const int nh, const int nx)
7  {
8      int i, j;
9      float sum_i, sum_q, norm;
10     const float a = 1.1755e-38;
11
12     // Take the shifted input vector until the end of frame
13     for (i = 0; i < nx; i++)
14     {
15         sum_i = 0.0f;
16         sum_q = 0.0f;
17
18     // Two dot products combined for the real part
19         for (j = 0; j < nh; j++)
```

```
20              sum_i += (h_i1[j] * x_i1[i+j]) - (h_q1[j] * x_q1
                    [i+j]);
21
22          y_io1[i] = sum_i;
23
24      // Two dot products combined for the imag part
25          for (j = 0; j < nh; j++)
26              sum_q += (h_q1[j] * x_i1[i+j]) + (h_i1[j] * x_q1
                    [i+j]);
27
28          y_qo1[i] = sum_q;
29
30      // Error calculation is performed individually
31          error_i[i] = y_ii1[i] - y_io1[i];
32          error_q[i] = y_qi1[i] - y_qo1[i];
33
34      // Complex euclidean norm as a sum of squares
35        norm = 0.0f;
36          for(j = 0; j < nh; j++)
37            norm += (x_i1[j+i] * x_i1[j+i]) + (x_q1[j+i] *
                     x_q1[j+i]);
38
39      // Cross-coupled kernel update process
40          for (j = 0; j < nh; j++)
41              h_i1[j] = h_i1[j] + (ar * (error_i[i] * x_i1[i +
                    j] + error_q[i] * x_q1[i+j]))/ (norm + a);
42
43          for (j = 0; j < nh; j++)
44              h_q1[j] = h_q1[j] + (ar * (error_q[i] * x_i1[i +
                    j] - error_i[i] * x_q1[i+j]))/ (norm + a);
45      }
46
47      return;
48  }
```

Listing 4: Normalized complex LMS filter function

From the listing 4, we can see that the complex version of the LMS algorithm is far more computation-intensive. Following sections are analyzing the possibility to divide this processing load between several processors.

3.3.5 Parallel LMS Algorithm

One of the distinctive properties of the complex LMS equalizer described in section 2.3.4 is the interdependency of its sub-processes due to the complex arithmetics; because of that, it can be characterized as a *highly coupled* system. However, there is also a certain degree of symmetry present between real and imaginary components of the algorithm.

If all parts of the algorithm could be provided with the input signal sequence at any moment, the whole system could be separated in two nearly identical sections, as illustrated on the figure 38. The main criterion for the separation is that each of the sections uses and adapts only one filter kernel. As the baseband signal is processed in frames, both parts of the algorithm can utilize the data block simultaneously, provided that each section will initially get a copy of the input and reference signals, as shown on the execution graph presented in figure 39.

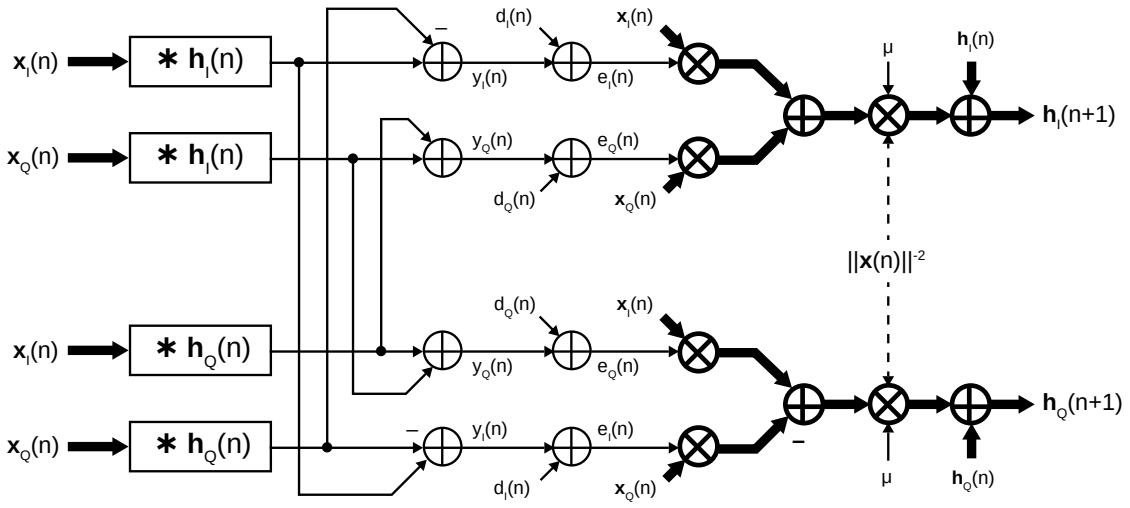


Figure 38: Parallel complex LMS algorithm

While it is impossible to completely eliminate the coupling between the parts the complex LMS algorithm, it can be reduced to an exchange of just four numbers per each iteration, as illustrated in figure 39. In this manner, each section will have the convolution results from the other one; this is necessary to produce both real and imaginary outputs and proceed to the update of the filter kernel. This means that the processing routine now

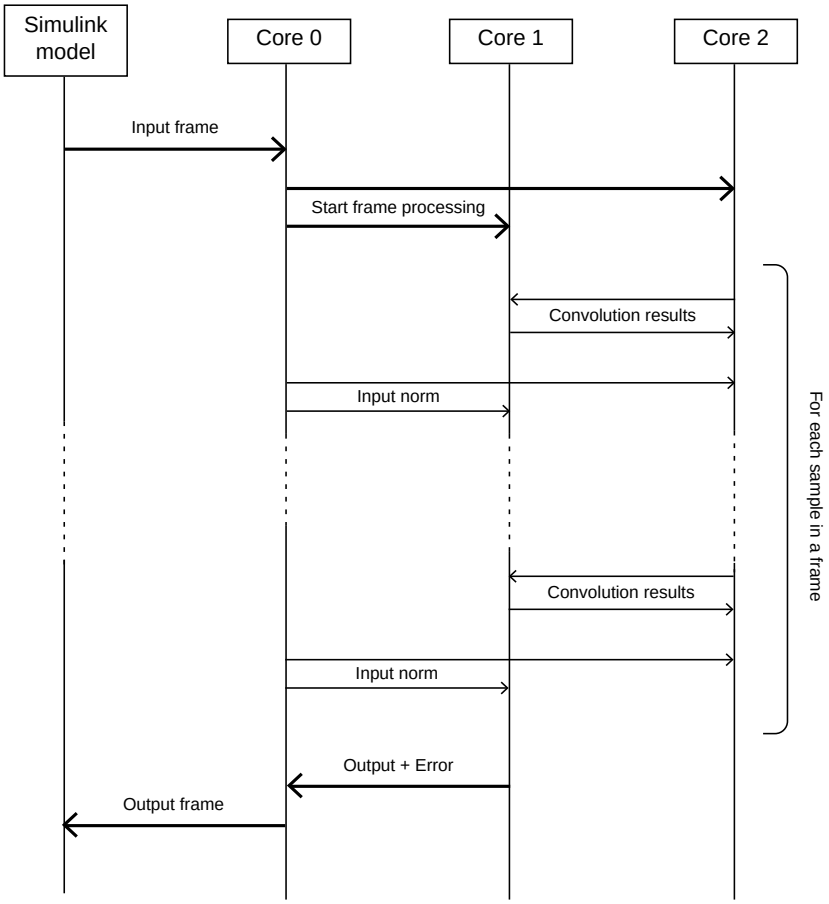


Figure 39: Multicore execution diagram

will additionally incorporate communication elements on the expense of IPC execution overhead caused by sending, waiting and receiving operations.

It also can be noted that two sections presented in the figure 38 have a very similar structure except small logical differences. This allows to reuse the same program logic twice for each processor core and set the section-specific parameters during the runtime.

The upper section of the parallel algorithm from figure 38 will be additionally dedicated to the recording of the results. In previously explained adaptation routine, each processor core will eventually calculate both real and imaginary output and the appropriate error values. During each iteration, the core that executes the upper (real) part of the filter kernel will record the output and error values to the final output vector. When the frame

processing is finished, this array will be returned to the Simulink via the main core, as shown in the execution graph (figure 39).

The normalization element $\|\mathbf{x}(n)\|^2$ is placed in figure 38 between the processing sections and connected to both with the dashed arrow. Notice that its value does not depend on the algorithm outcome but only on the input vector, so it can be calculated separately and then distributed to each core. This functionality is therefore moved onto the main core, such that the squared euclidean norm is calculated after the input frame distribution.

As the input vector for each iteration is shifted by one sample forward, the main core has to provide 64 different normalized values for each iteration. Luckily enough, the specific queuing capabilities of the IPC module allow to store all these values in the shared memory, such that processing sections of the algorithm can access them in the right order when needed. This process will be explained in detail in the following section.

3.3.6 Inter-Process Communication (IPC)

In a concurrent system, data exchange and signaling between its processes is very important. This communication is usually realized by means of a certain middle-man resource. Whether the processes are tasks within one RTOS or a different programs running on separate computers, the IPC shared resource should be accessible to each program and protected from the conflicts, which may occur during simultaneous requests.

Like in many multi-processor systems, the data within the equalizer application is transferred using the shared memory. In order to process the shared information, the processor loads a copy of this data into his own high-speed storage known as cache. Figure 40 depicts the concept of multi-processor shared memory. When one processor attempts to change the shared data, it has to notify the other data users by cache write command. The other processors then should invalidate their own cache to update the state of their local storage.

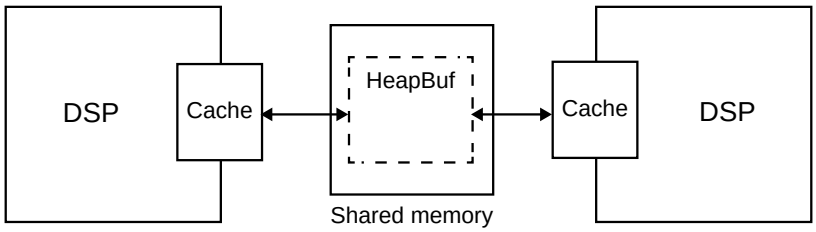


Figure 40: Multiprocessor shared memory

To secure a storage block in the shared region, the DSP should assign the memory area for this purpose by using IPC **HeapBuf** module, as shown in listing 5. Other processors that use the same shared area will have to open the HeapBuf block created by the first DSP. After that, memory allocation and read-write access inside the block are possible. [18, 32-35.]

```
1  #define HEAP_NAME    "myHeapBuf"
2  #define HEAPID       0
3  static HeapBufMP_Handle   heapHandle;
4  static HeapBufMP_Params   heapBufParams;
5
6  // Create the HeapBuf instance
7  HeapBufMP_Params_init(&heapBufParams);
8  heapBufParams.regionId       = 0;
9  heapBufParams.name           = HEAP_NAME;
10 heapBufParams.numBlocks      = 5;
11 heapBufParams.blockSize      = sizeof(datachunk);
12 heapHandle = HeapBufMP_create(&heapBufParams);
13 if (heapHandle == NULL) {
14   System_abort("HeapBufMP_create failed\n" );
15 }
16
17 // The other processor waits until this HeapBuf is open
18 do {
19   status = HeapBufMP_open(HEAP_NAME, &heapHandle);
20   if (status < 0) {
21     Task_sleep(1);
22   }
23 } while (status < 0);
```

Listing 5: HeapBuf creating procedure

Data vectors are transferred between cores using the **Message Queue** mechanism shown in figure 41. This elaborate interface allows to dynamically allocate small data sections

of the shared memory and use them to contain the message payload. The queue itself contains only *pointers* to the shared memory locations, such that the target processor can find the sent message from the shared region.

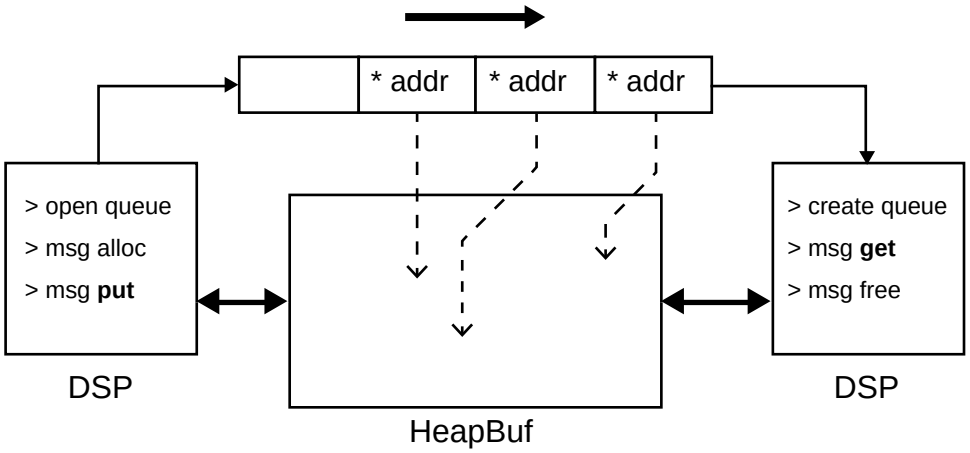


Figure 41: Message queue mechanism

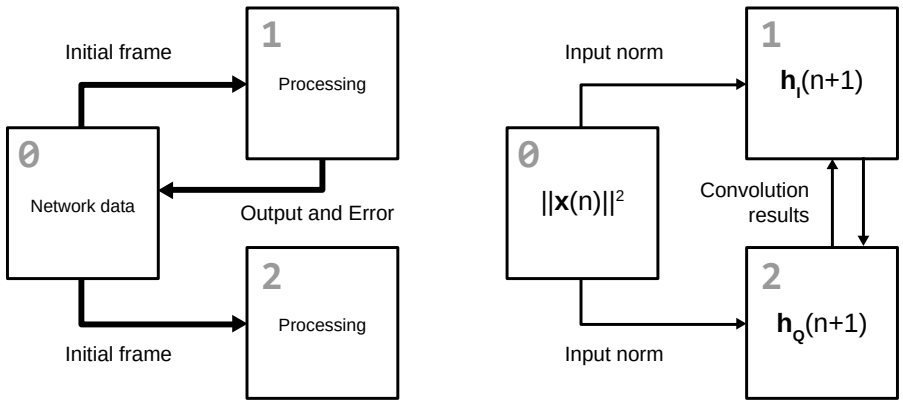
The IPC message queue is a one-directional transport system, and its instance should be created by the processor that receives the data. After that, the sender can open the queue for writing. The whole message passing process is presented in listing 6. At first, the message slot should be allocated from the registered HeapBuf shared memory region. When the message location is populated with data and the cache is written to memory, the message pointer is placed into the queue with the **put** function. To check the queue, the receiver DSP calls the **get** function that blocks the program execution until any message is received or the specified timeout period expires. Upon data acquisition, the receiver process is responsible for cache invalidation and releasing of the reserved memory space. [18, 19-24.]

```
1  struct datachunk * input;
2  struct datachunk * output;
3  Char QueueName[10];
4
5  // Both processors should register the HeapBuf instance
6  MessageQ_registerHeap((IHeap_Handle)heapHandle, HEAPID);
7
8  // Receiver creates the queue under local name
9  localQueue = MessageQ_create(QueueName, NULL);
10 if (data_inQ == NULL) {
11     System_abort("input MessageQ_create failed\n" );
12 }
13
14 // Sender spins until remote queue is opened
```

```
15 do {
16   status = MessageQ_open(QueueName, &remoteQueueId);
17   if (status < 0) {
18     Task_sleep(1);
19   }
20 } while (status < 0);
21
22 // Sender allocates the memory area in HeapBuf
23 output = (datachunk*)MessageQ_alloc(HEAPID, sizeof(datachunk
       ));
24
25 // Populate the message block, write cache and send the
       address
26 Cache_wb(output, sizeof(datachunk),Cache_Type_ALL, FALSE);
27 status = MessageQ_put(remoteQueueId, (MessageQ_Msg)output);
28
29 // Receiever waits forever until message arrives, then
       invalidates the cache
30 status = MessageQ_get(localQueue, (MessageQ_Msg*)&input,
       MessageQ_FOREVER);
31 Cache_inv(input, sizeof(datachunk),Cache_Type_ALL, FALSE);
32
33 // Process the data and release the shared memory space
34 MessageQ_free((MessageQ_Msg)input);
```

Listing 6: Message queue operation

The equalizer application utilizes several message queues to pass sample blocks of different sizes. In general we can separate these into two categories: big blocks to pass the input and output frames, and small blocks that are used for coupling between algorithm sections as explained in section 3.3.5. All these operations are presented in figure 42.



(a) Large buffer operations (b) Small buffer operations

Figure 42: IPC queues in the equalizer

To accommodate these frames in the shared memory, two different HeapBuf instances with distinct block size are created. The main buffer can hold several packets of 256 samples for the frame transmission, and the auxiliary buffer is divided into many 2-sample blocks that are used for the convolution results and the vector norm. The data structures that are mapped to these blocks are presented in listing 7. Each allocated message consists of a payload and the header, which is needed to parse the message correctly.

```
1  #define FRAME      64
2
3  // Full data packet
4  typedef struct datachunk {
5    MessageQ_MsgHeader header;
6    struct payload{
7      float payload_i[FRAME];
8      float payload_q[FRAME];
9      float refload_i[FRAME];
10     float refload_q[FRAME];
11   }payload;
12 } datachunk ;
13
14 // Small data packet
15 typedef struct errorchunk {
16   MessageQ_MsgHeader header;
17   struct small{
18     float i_element;
19     float q_element;
20   } small;
21 } errorchunk ;
```

Listing 7: Message queue data structures

As the messages are retrieved from the queue in the same order as they are sent, the main program can fill the queues with calculated normalization factor without additional synchronization. Both other cores can then recover the appropriate value from their end of the line.

3.3.7 Processor Instrumentation

As a means to evaluate the execution properties of the equalizer application, the dedicated instrumentation module can be used. This system utilizes a specific buffer allocated in the

processor memory to record the informative events produced by the program. Schematic diagram of the program evaluation process is presented in figure 43.

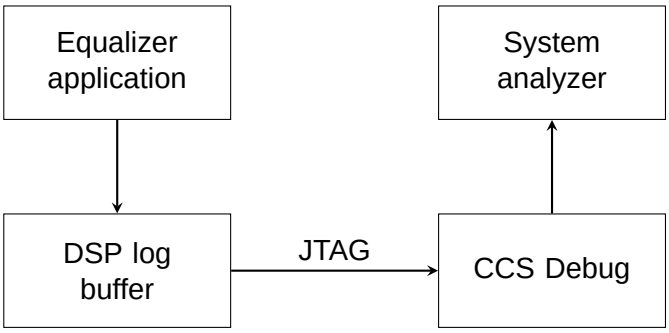


Figure 43: Instrumentation of the DSP application.

Code Composer Studio (CCS) debugging capabilities allow to connect directly to the DSP operating registers and memory, upload the compiled program and step through its execution in a systematic manner. The link between the processor and the software is realized with widely used JTAG debugging interface that is connected to the PC through the USB serial line.

System analyzer module is used as an addition to the CCS debugging mode, and it uses the same debugging JTAG channel to receive and analyze the instrumentation events directly from the processor memory. During the program execution, the events are generated on the specific points of interest, such as CCS task switching or the read-write access. Recorded events are timestamped by the processor and collected in the log buffer. On the specific intervals, the CCS sends the logs to the CCS for evaluation.

To instrument the multi-core equalizer program, special logging commands are added to particular points of the code. An example of these commands is given in listing 8. The start of the LMS algorithm is recorded right after the sub-main core has received the main frame. End of the processing is logged just before sending the processed frame back to the main core.

```
1 // The analyzed process is started
2 Log_writeUC3(UIABenchmark_startInstanceWithAdrs, (IArg)"
      context=0x%x, fnAdrs=0x%x:",(IArg)0, (IArg)&tsk0_func);
3
4 // The analyzed process is finished
```

```
5 Log_writeUC3(UIABenchmark_stopInstanceWithAdrs, (IArg)"
    context=0x%x, fnAdrs=0x%x:",(IArg)0, (IArg)&tsk0_func);
```

Listing 8: Execution log functions

To benchmark the canonical LMS routines presented in section 3.3.4, special hook functions were implemented in the program, as shown in listing 9.

```
1 void functionEntryHook( void (*addr)() ){
2     Log_writeUC3(UIABenchmark_startInstanceWithAdrs, (IArg)"
        context=0x%x, fnAdrs=0x%x:",(IArg)0, (IArg)addr);
3 }
4
5 void functionExitHook( void (*addr)() ){
6     Log_writeUC3(UIABenchmark_stopInstanceWithAdrs,  (IArg)"
        context=0x%x, fnAdrs=0x%x:",(IArg)0, (IArg)addr);
7 }
```

Listing 9: Subroutine execution hooks

When the program enters or exits some subroutine, appropriate hook function will be executed in between, allowing to precisely record the event without much interfering with the program code. [19, 82.]

4 Results

4.1 Algorithm Evaluation System

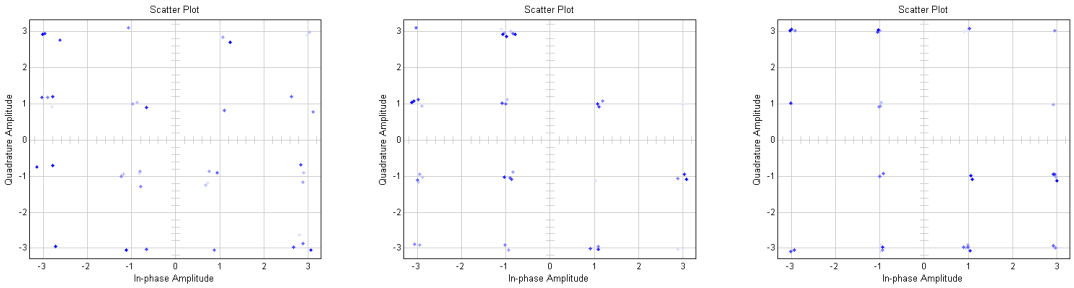
In this project, a highly versatile digital signal processing algorithm evaluation system was established using the state-of-art simulation software together with newest DSP hardware. The user has the advantage of the flexible computing power and the freedom in the program optimization, which allows to freely develop and implement algorithms of various complexity. By using the readily-available simulation toolbox, the developer can construct diverse environments for the algorithm evaluation. High-speed network connection to the simulation model allows to test the developed hardware application in nearly real-life conditions with large quantities of data.

Based on this setup, a distributed complex normalized LMS equalizer was implemented on a multi-core DSP platform in order to evaluate the applicability of the concurrent processing models for the adaptive channel equalization. The performance of the algorithm was evaluated using the QAM baseband signal specially preconditioned in the simulated multipath propagation environment. The effectiveness of the hardware implementation is assessed with the dedicated DSP instrumentation tools. The results of the above tests are presented in the following sections.

4.2 Convergence and Error

Behavior of the non-normalized LMS algorithm is tested under the complex channel conditions described in section 3.2.3. The adaptation rate of the process is intentionally set to the small value in order to ensure stability and observe the convergence tendency. Kernel of the filter is 16 samples in length, initially all zero, with 0th sample set to one

(all-pass). Figure 44 depicts the scatter plots of the detected symbols during the three different stages of the adaptation.



(a) Active adaptation (b) Convergence (c) Error is minimized

Figure 44: Detected symbols during filter adaptation.

From the figure 44, we can see that the detected symbols are gathering on the right constellation positions. The complex error magnitude obtained from the equalizer is presented in figure 45.

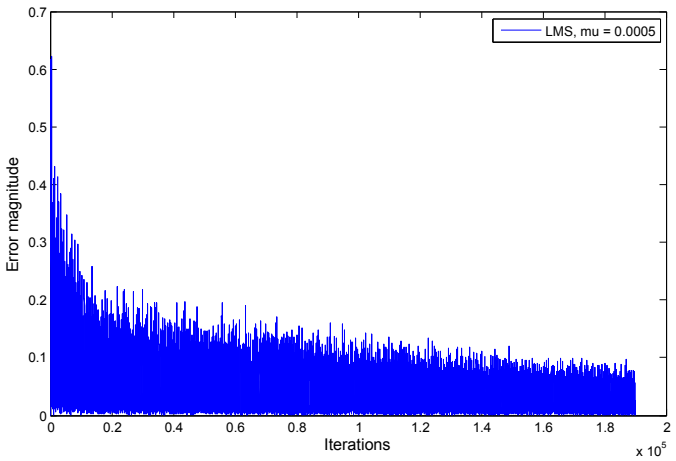


Figure 45: Convergence of the LMS algorithm.

From the error magnitude plot, we can see how the algorithm converges by minimizing the error. With the adaptation step this small, the minimum possible error of 0.1 is achieved only after 160000 iterations. Table 3 shows the bit error rate before and after equalization.

Table 3: Bit error rate comparison.

	Bits received	Errors	Error rate
Raw data	21310	6250	0.2933
Equalized data	21310	100	0.0047

From the table 3, a significant reduction in the amount of errors can be observed after the full convergence. The next section covers the improvement of convergence rate by input normalization.

4.3 Normalization

As previously explained in section 2.3.3, increasing the adaptation coefficient improves the convergence rate but may also destabilize the algorithm. Obviously this solution is far from the optimal, and the special normalization procedure is preferred. Figure 46 compares convergence graph of normalized LMS algorithm with basic LMS which has maximum possible adaptation rate of 0.05, meaning that the larger coefficient, in fact, destabilizes the filter completely. In order to indicate the trending, the fluctuations of the error were removed with the moving average filter of 80 samples in length.

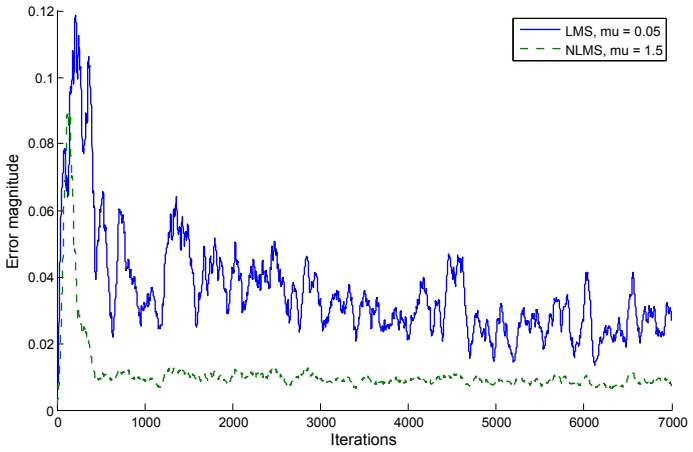
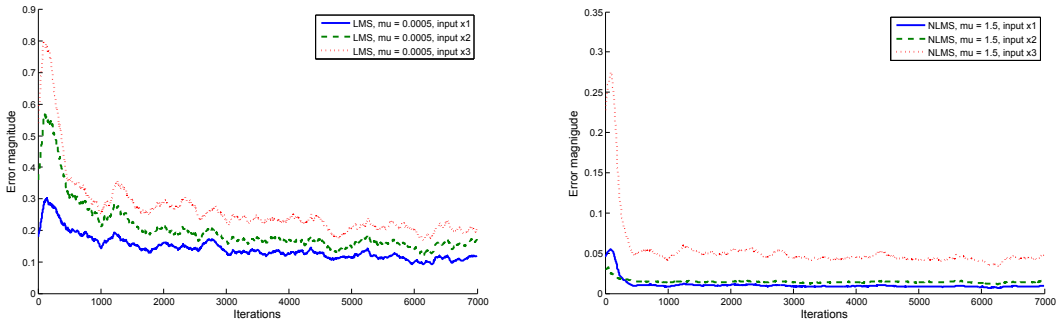


Figure 46: Convergence of normal and normalized LMS algorithm with kernel size of 16 samples.

From the figure 46 the almost immediate convergence of the normalized filter can be observed, reaching the minimum error in about 300 iterations.

The further benefit of the normalized LMS equalizer is demonstrated in figure 47, where two different filters are compared in conditions of an increased input signal amplitude. All presented graphs are additionally smoothened with the moving average filter of 250 samples in length.



(a) Ordinary LMS, 16-sample kernel

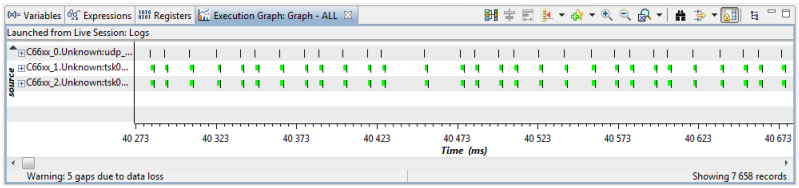
(b) Normalized LMS, 16-sample kernel

Figure 47: Convergence of different algorithms subjected to increased input amplitude.

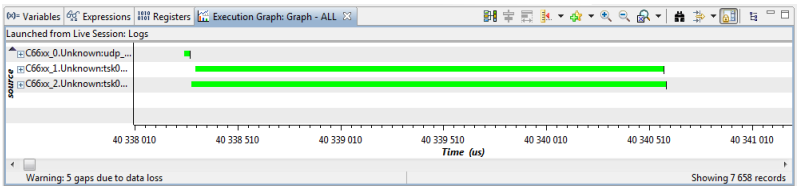
From the figure 47(b) we can clearly see that the absolute input power and the convergence speed of the normalized filter are hardly correlated in comparison to the reaction of the normal LMS algorithm.

4.4 Program Execution and Timing

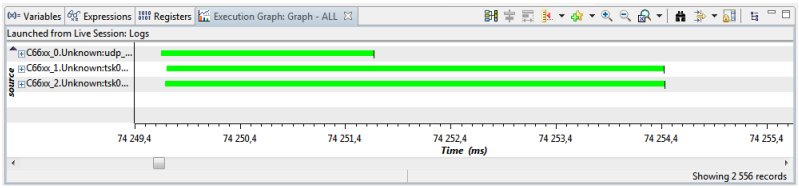
The local execution timestamps of the equalizer application were gathered from each DSP core. Execution graph of the concurrent processes is shown in figure 48.



(a) Large time scale



(b) LMS application



(c) Normalized LMS application

Figure 48: Execution graphs of the multi-core equalizer application.

From the figure 48(a) we can see that the signal processing is executed in short time intervals with the large idle periods, that are most probably caused by waiting for the the PC simulation data to arrive. Figure 48(b) lines up the concurrent activity of cores 0, 1 and 2 in order from top to bottom. We can notice that the main program is being idle almost all the time; this is not the case when the input normalization task is designated to the 0th core, as shown in figure 48(c). In this manner, the processing load is distributed more or less evenly between three programs.

The amount of arithmetical operations needed for the basic LMS algorithm variations is shown in table 4. The variable n stands for the length of the adaptive filter kernel.

Table 4: Operations needed for one iteration of the LMS algorithm.

	Multiplications	Additions
LMS	$3n$	$2n + 1$
NLMS	$5n$	$3n + 1$
Complex LMS	$10n$	$6n + 2$
Complex NLMS	$14n$	$10n + 2$

Table 5 shows the computational cost for the concurrent versions of the complex LMS algorithm. Note that specific amount of LMS operations is necessary each iteration to realize the coupling between different algorithm sections.

Table 5: Operations needed for one iteration of the parallel LMS algorithm.

	Core	Multiplications	Additions	IPC put	IPC get
LMS	1	$5n$	$4n + 4$	1	1
	2	$5n$	$4n + 4$	1	1
Complex LMS	0	$2n$	$2n$	2	0
	1	$6n$	$4n + 4$	1	2
	2	$6n$	$4n + 4$	1	2

The data presented in the above tables is summarized in figure 49 to compare the execution costs for different LMS algorithm implementations. In order to estimate the plot

magnitude, the adaptive filter kernel is assumed to be 3 samples long and the incoming frame is of the size of 5 samples.

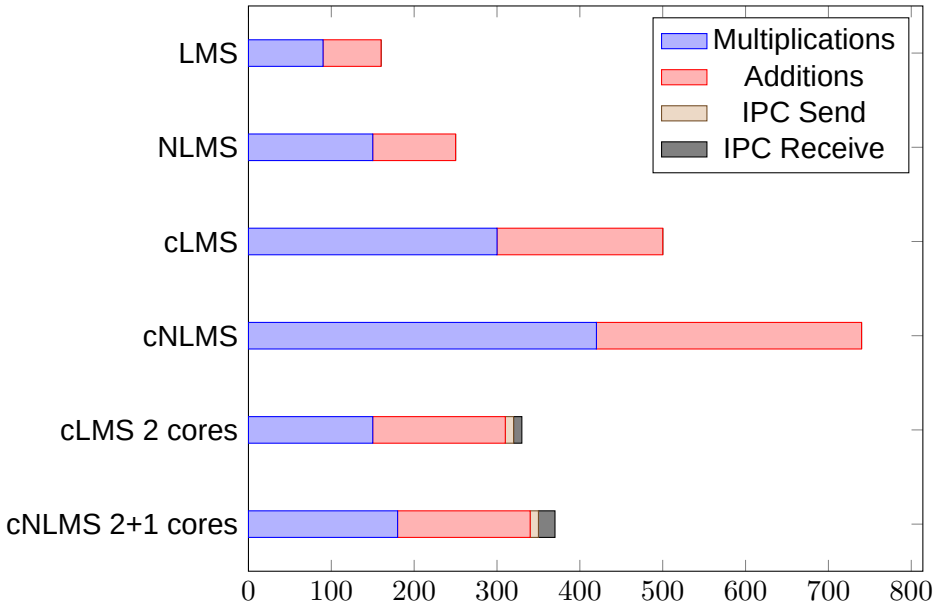


Figure 49: Operations needed for 5-sample block processing with the 3-sample kernel

From the figure 49, we can observe an obvious reduction of the execution cost for the concurrent system. However, a certain IPC overhead is added to each processor. To evaluate this model, execution timing statistic of each version of the algorithm has been taken into account. In order to maximize the algorithm processing workload in relation to the logging period, the 48-sample kernel was used with already familiar frames of 64 values. Table 6 shows the benchmarking results ordered in the same manner by the increase of algorithm complexity. For the multi-core algorithms, only the timing of the sub-main core is presented, as being the longest process in the concurrent system.

Table 6: Algorithm execution benchmarks.

	Total average, ns	Max, ns	Min, ns	Records
LMS	552,296.42	513860	552544	2856
NLMS	679,089.46	641005	679325	1070
Complex LMS	1,174,887.12	1098390	1175245	2208
Complex NLMS	1,298,153.61	1221652	1298738	818
Complex LMS 2 cores	2,289,930.02	2290858	2289170	3532
Complex NLMS 2+1 cores	4,731,336.67	4818911	4616363	2477

For easier comparison with the execution cost model, the total average timing is graphed in figure 50 in the same order.

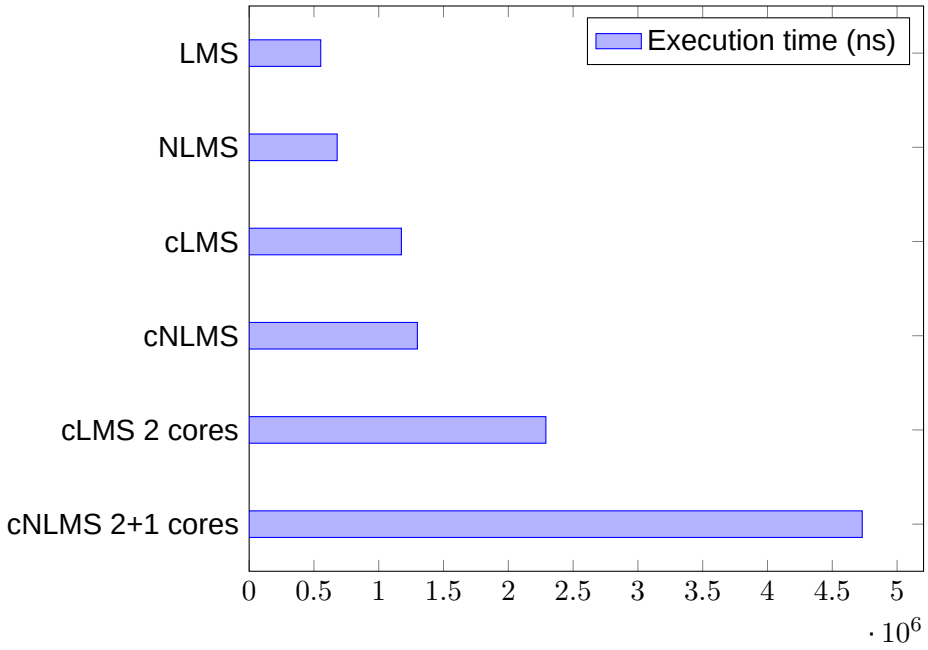


Figure 50: Timing of the algorithm execution.

From the timing graph presented above, the magnitudes of the single-core algorithm implementations can be easily related to the predicted cost values while concurrent versions of the LMS filter are significantly larger than expected, with the normalized version being almost as twice in length as the non-normalized multicore implementation. These enormous values are most likely to be caused by the IPC overhead, including the delays for memory allocation, cache invalidation, and other related processes.

5 Discussion

5.1 Concurrent LMS Equalizer

The implemented LMS equalizer has shown the strong agreement with the theoretical model. The normalized version of the algorithm demonstrated good convergence capabilities that are very important for the tracking of the fast changing LTV systems; this extension can be considered as a vital improvement to the equalizer system. Still, the overall performance of the application has yet to improve in order to match the efficiency of hardware solutions available on the market.

The parallelization of the complex LMS algorithm was partly successful. The re-arranging of the system elements reduced the *coupling* by increasing the data volume to be distributed. This downside can be neglected, as the single memory access overhead is apparently way larger relative to the amount of data being read.

The *cohesion* of each algorithm section was maximized in such way that the same program code could be reused for the both sections on two cores, therefore in the future the equalizer system can even be scaled up if needed. On the other hand, even slight demand for cross-coupling between sections has, in fact, significantly slowed down the overall execution.

The effectiveness of the multi-core approach can be significantly increased by utilizing the low-level IPC interfaces such as Direct Memory Access (DMA). If the communication overhead is minimized, an even more entangled system could be effectively mapped onto multiple processors. However, the in-depth device-specific optimizations are beyond the scope of this project.

5.2 Algorithm Evaluation System

The establishment of the network interface between the simulation model and the DSP has played an important role in the application development and evaluation. Considering the enormous processing power of the DSP platform backed up with the external memory of evaluation module, the advantage of the reusable high-speed interface to the device is evident. When combined with the immense data generation capabilities of the Simulink, the potential of an external multi-core processor can offer support even for the most challenging concepts.

Algorithm execution graphs have shown the tendency of the system to stay idle for most of the time, which means that the network transfer and Simulink processing overhead are obstructing the constant workload of the device. This means that there is still room for the way more complex calculations, given that the EVM local memory is provided with a large set of data beforehand.

Another way to exploit the full potential of the platform is to share the processing power over the network of clients. By utilizing the server *daemon* capability to create a separate process for each network client, the network-enabled module can be used for teaching the concurrent signal processing on the actual hardware. Either of the operating systems supported on this hardware could be configured to create a sandbox-like command environment where the remote users can allocate new processing tasks to different cores and provide them with the streaming data from their own Simulink instance.

6 Conclusions

The concurrent signal processing model was successfully planned, implemented and evaluated. Practical feasibility in nearly real conditions and the potential advantage of the parallel processing are demonstrated. Algorithm properties specific to the adaptive filtering, such as cross-coupling and logical feedback have introduced some practical difficulties, which can be solved by further hardware related optimizations. Established developer environment can be improved and reused for further work in algorithm optimization and evaluation.

There are many adaptive signal processing methods of higher complexity which can be more cooperative in terms of concurrent execution. Further study in the functional structure of the algorithms may shed light on the new ways to parallelize the existing signal processing methods.

The great deal of research should be conducted on the fundamental parallel computing techniques, such as data sharing, task scheduling, process synchronization and others. One of the main advantages of the software signal processing is the fundamental ability to be reconfigured according to the demand of the application.

Bibliography

1 International Organization for Standardization (ISO)/International Electrotechnical Commission (IEC). Standard ISO/IEC 7498-1:1994. Information technology - Open Systems Interconnection - Basic Reference Model [online]; 1994. Available from: http://standards.iso.org/ittf/PubliclyAvailableStandards/s020269_ISO_IEC_7498-1_1994(E).zip [cited April 04, 2015].

2 Proakis JG. Digital Communications. 4th ed. McGraw Hill Higher Education; 2000.

3 National Telecommunications and Infomation Administration (NTIA). Federal Standard 1037C. Telecommunications: Glossary of Telecommunication Terms [online]; 1996. Available from: http://www.its.bldrdoc.gov/fs-1037/fs-1037c.htm [cited April 11, 2015].

4 National Instruments. Quadrature Amplitude Modulation (QAM) [online]; 2014. Available from: http://www.ni.com/white-paper/3896/en/ [cited April 12, 2015].

5 Lyons R. Quadrature Signals [online]; 2008. Available from: www.ieee.li/pdf/essay/quadrature_signals.pdf [cited April 12, 2015].

6 Magesacher T. Channels and Channel Models [online]; 2008. Available from: www.eit.lth.se/fileadmin/eit/courses/eit140/ofdm_channels.pdf [cited April 20, 2015].

7 Jain R. Channel Models: A Tutorial [online]; 2007. Available from: www.eit.lth.se/fileadmin/eit/courses/eit140/ofdm_channels.pdf [cited April 20, 2015].

8 Goldsmith A. Wireless Communications. Cambridge University Press; 2004.

9 Jeruchim MC, Balaban P, Shanmugan SK. Simulation of Communication Systems. 2nd ed. Springer US; 2000.

10 National Instruments. Sources of Error in IQ Based RF Signal Generation [online]; 2007. Available from: http://www.ni.com/tutorial/5657/en/ [cited May 2, 2015].

11 Ouameur A. RF Imperfection and Compensation [online]; 2013. Available from: http://nutaq.com/en/blog/rf-imperfection-and-compensation-part-1-effects-iq-imbalance-and-compensation-receiver [cited May 2, 2015].

12 Haykin S. Adaptive Filter Theory. 3rd ed. Prentice Hall; 1995.

13 Poole I. GSM Radio Air Interface, GSM Slot & Burst [online];. Available from: http://www.radio-electronics.com/info/cellulartelecomms/gsm_technical/gsm-radio-air-interface-slot-burst.php [cited April 17, 2015].

14 Widrow B, McCool J, Ball M. The Complex LMS Algorithm. Proceedings of the IEEE. 1975;63:719--720.

15 Texas Instruments. Multicore Programming Guide [online]; 2012. Available from: http://www.ti.com/lit/an/sprab27b/sprab27b.pdf [cited April 18, 2015].

16 Texas Instruments. TMS320C6678 Multicore Fixed and Floating-Point Digital Signal Processor (Rev. E) [online]; 2010. Available from: http://www.ti.com/lit/ds/symlink/tms320c6678.pdf [cited May 7, 2015].

17 Texas Instruments. DSP Library (DSPLIB) for C64x+/C66x/C674x processors [online]; 2014. Available from: http://software-dl.ti.com/sdoemb/sdoemb_public_sw/dsplib/latest/index_FDS.html [cited May 8, 2015].

18 Texas Instruments. SYS/BIOS Inter-Processor Communication (IPC) 1.25 User's Guide [online]; 2012. Available from: http://www.ti.com.cn/cn/lit/ug/sprugo6e/sprugo6e.pdf [cited May 9, 2015].

19 Texas Instruments. System Analyzer User's Guide [online]; 2014. Available from: http://www.ti.com.cn/cn/lit/ug/spruh43f/spruh43f.pdf [cited May 9, 2015].

1 Main program source code

The main core image is based on the modified **TCP/IP Stack 'Hello World!'** example by Texas Instruments

```
/*
 *
 * TCP/IP Stack 'Hello World!' Example ported to use BIOS6 OS.
 *
 * Copyright (C) 2007, 2011 Texas Instruments Incorporated - http://www.ti.com/
 *
 *
 *  Redistribution and use in source and binary forms, with or without
 *  modification, are permitted provided that the following conditions
 *  are met:
 *
 *    Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *
 *    Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the
 *    distribution.
 *
 *    Neither the name of Texas Instruments Incorporated nor the names of
 *    its contributors may be used to endorse or promote products derived
 *    from this software without specific prior written permission.
 *
 *  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 *  "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 *  LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
 *  A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
 *  OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 *  SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
 *  LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
 *  DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
 *  THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 *  (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
 *  OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
```

```
36  */
37
38  //-----------------------------------------------------------------------
39  // IP Stack 'Hello World!' Example
40  //
41  // This is a skeleton application, intended to provide application
42  // programmers with a basic Stack setup, to which they can start
43  // adding their code.
44  //
45  // To test it as is, use with helloWorld.exe from \winapps directory
46
47  #include <stdio.h>
48  #include <string.h>
49  #include <ti/ndk/inc/netmain.h>
50
51  /* BIOS6 include */
52  #include <ti/sysbios/BIOS.h>
53  /* XDC&SYSBIOS includes */
54  #include <xdc/std.h>
55  #include <xdc/cfg/global.h>
56  #include <xdc/runtime/IHeap.h>
57  #include <xdc/runtime/System.h>
58  #include <xdc/runtime/Memory.h>
59  #include <xdc/runtime/Log.h>
60
61  #include <ti/sysbios/heaps/HeapBuf.h>
62  #include <ti/sysbios/heaps/HeapMem.h>
63
64  #include <ti/ipc/GateMP.h>
65  #include <ti/ipc/Ipc.h>
66  #include <ti/ipc/MessageQ.h>
67  #include <ti/ipc/HeapBufMP.h>
68  #include <ti/ipc/MultiProc.h>
69
70  #include <ti/sysbios/knl/Task.h>
71  #include <ti/sysbios/knl/Semaphore.h>
72  #include <ti/sysbios/knl/Clock.h>
73  #include <ti/sysbios/knl/Mailbox.h>
74  #include <ti/sysbios/hal/Cache.h>
75
76  #include <ti/dsplib/dsplib.h>
77
78  /* Platform utilities include */
79  #include "ti/platform/platform.h"
80  #include "ti/platform/resource_mgr.h"
81
82  // For the multicore instrumentation
83  #include <xdc/runtime/Log.h>
```

```
84  #include <ti/uia/runtime/LogSync.h>
85
86  #include <xdc/runtime/Timestamp.h>
87  #include <xdc/runtime/Types.h>
88  #include <xdc/runtime/Diags.h>
89
90  #include <ti/uia/runtime/LogUC.h>
91  #include <ti/uia/events/UIABenchmark.h>
92
93  #include <c6x.h>
94
95  /* Platform Information - we will read it form the Platform Library */
96  platform_info  gPlatformInfo;
97
98  #define INVALID_SOCKET (HANDLE)0xFFFFFFFF   // Used by socket() and accept()
99
100
101 //-----------------------------------------------------------------------
102 // Title String
103 //
104 char *VerStr = "\nTCP/IP Stack 'Hello World!' Application\n\n";
105
106 // Our NETCTRL callback functions
107 static void    NetworkOpen();
108 static void    NetworkClose();
109 static void    NetworkIPAddr( IPN IPAddr, uint IfIdx, uint fAdd );
110
111 // Fun reporting function
112 static void    ServiceReport( uint Item, uint Status, uint Report, HANDLE hCfgEntry
        );
113
114 // RTOS tasks
115
116 void udp_echo_test();
117 void StackTest();
118
119 //-----------------------------------------------------------------------
120 // Configuration
121 //
122 char *HostName    = "tidsp";
123 char *LocalIPAddr = "10.80.88.51";
124 char *LocalIPMask = "255.255.255.0";     // Not used when using DHCP
125 char *GatewayIP   = "192.168.2.101";     // Not used when using DHCP
126 char *DomainName  = "demo.net";          // Not used when using DHCP
127 char *DNSServer   = "0.0.0.0";           // Used when set to anything but zero
128
129 Uint8 clientMACAddress [6] = {0x5C, 0x26, 0x0A, 0x69, 0x44, 0x0B};
130
```

```
131 Task_Params        tskParams1;
132
133 // IPC queue names
134 Char localQueueName[10];
135 Char realQueueName[10];
136 Char imagQueueName[10];
137 Char rnormQueueName[10];
138 Char inormQueueName[10];
139
140 // Definitions for the shared memory
141 #define HEAP_NAME   "myHeapBuf"
142 #define HEAP_AUX_NAME   "auxHeapBuf"
143 #define HEAPID      0
144 #define HEAPID_AUX      1
145
146 // Data block length and filter length
147 #define FRAME      64
148 #define FIR_LEN   16
149
150 // Full data packet
151 typedef struct datachunk {
152   MessageQ_MsgHeader header;
153   struct payload{
154     float payload_i[FRAME];
155     float payload_q[FRAME];
156     float refload_i[FRAME];
157     float refload_q[FRAME];
158   }payload;
159 } datachunk ;
160
161 // Small data packet
162 typedef struct errorchunk {
163   MessageQ_MsgHeader header;
164   struct small{
165     float i_element;
166     float q_element;
167   } small;
168 } errorchunk ;
169
170 // Data structure for local processing
171 typedef struct localchunk{
172   float inphase[FRAME];
173   float quadrat[FRAME];
174 } localchunk;
175
176 // Shared memory variables
177 static HeapBufMP_Handle              heapHandle;
178 static HeapBufMP_Params              heapBufParams;
```

```
179 static HeapBufMP_Handle              heapHandle_aux;
180 static HeapBufMP_Params              heapBufParams_aux;
181
182
183 /*************************************************************************
184  *  @b EVM_init()
185  *
186  *  @n
187  *
188  *  Initializes the platform hardware. This routine is configured to start in
189  *   the evm.cfg configuration file. It is the first routine that BIOS
190  *   calls and is executed before Main is called. If you are debugging within
191  *  CCS the default option in your target configuration file may be to execute
192  *  all code up until Main as the image loads. To debug this you should disable
193  *  that option.
194  *
195  *  @param[in]  None
196  *
197  *  @retval
198  *      None
199  *************************************************************************/
200 void EVM_init()
201 {
202  platform_init_flags   sFlags;
203  platform_init_config  sConfig;
204  /* Status of the call to initialize the platform */
205  int32_t pform_status;
206
207  /*
208   * You can choose what to initialize on the platform by setting the following
209   * flags. Things like the DDR, PLL, etc should have been set by the boot loader.
210  */
211  memset( (void *) &sFlags,  0, sizeof(platform_init_flags));
212  memset( (void *) &sConfig, 0, sizeof(platform_init_config));
213
214     sFlags.pll  = 0; /* PLLs for clocking    */
215     sFlags.ddr  = 0;     /* External memory    */
216     sFlags.tcsl = 1; /* Time stamp counter   */
217 #ifdef _SCBP6618X_
218     sFlags.phy  = 0; /* Ethernet       */
219 #else
220     sFlags.phy  = 1; /* Ethernet       */
221 #endif
222     sFlags.ecc  = 0; /* Memory ECC       */
223
224     sConfig.pllm = 0;  /* Use libraries default clock divisor */
225
226  pform_status = platform_init(&sFlags, &sConfig);
```

```
227
228   /* If we initialized the platform okay */
229   if (pform_status != Platform_EOK) {
230     /* Initialization of the platform failed... die */
231     while (1) {
232       (void) platform_led(1, PLATFORM_LED_ON, PLATFORM_USER_LED_CLASS);
233       (void) platform_delay(50000);
234       (void) platform_led(1, PLATFORM_LED_OFF, PLATFORM_USER_LED_CLASS);
235       (void) platform_delay(50000);
236     }
237   }
238
239 }
240
241 //------------------------------------------------------------------------
242 // Main Entry Point
243 //------------------------------------------------------------------------
244 int main()
245 {
246     Int status;
247
248   //Write the sync point for the execution logger
249     LogSync_writeSyncPoint();
250
251   //Initialize the UDP application task
252     Task_Params_init(&tskParams1);
253     tskParams1.priority = 3;
254     tskParams1.stackSize = 5120;
255     Task_create(udp_echo_test, &tskParams1, NULL);
256
257   //Define the names of the data queues
258     System_sprintf(localQueueName, "%s", "netw\0");
259     System_sprintf(realQueueName, "%s",  "real\0");
260     System_sprintf(imagQueueName, "%s",  "imag\0");
261     System_sprintf(rnormQueueName, "%s",  "rnorm\0");
262     System_sprintf(inormQueueName, "%s",  "inorm\0");
263
264   //Synchronize all cores and start the IPC
265     status = Ipc_start();
266     if (status < 0) {
267         System_abort("Ipc_start failed\n");
268     }
269
270     /* Start the BIOS 6 Scheduler */
271     BIOS_start ();
272 }
273
274 //
```

```
275 // Main Thread
276 //
277 void StackTest()
278 {
279     int             rc;
280     int         i;
281     HANDLE          hCfg;
282     QMSS_CFG_T      qmss_cfg;
283     CPPI_CFG_T      cppi_cfg;
284 
285  /* Get information about the platform so we can use it in various places */
286  memset( (void *) &gPlatformInfo, 0, sizeof(platform_info));
287  (void) platform_get_info(&gPlatformInfo);
288 
289  (void) platform_uart_init();
290  (void) platform_uart_set_baudrate(115200);
291  (void) platform_write_configure(PLATFORM_WRITE_ALL);
292 
293  /* Clear the state of the User LEDs to OFF */
294  for (i=0; i < gPlatformInfo.led[PLATFORM_USER_LED_CLASS].count; i++) {
295    (void) platform_led(i, PLATFORM_LED_OFF, PLATFORM_USER_LED_CLASS);
296  }
297 
298     /* Initialize the components required to run this application:
299      *  (1) QMSS
300      *  (2) CPPI
301      *  (3) Packet Accelerator
302      */
303     /* Initialize QMSS */
304     if (platform_get_coreid() == 0)
305     {
306         qmss_cfg.master_core        = 1;
307     }
308     else
309     {
310         qmss_cfg.master_core        = 0;
311     }
312     qmss_cfg.max_num_desc       = MAX_NUM_DESC;
313     qmss_cfg.desc_size          = MAX_DESC_SIZE;
314     qmss_cfg.mem_region         = Qmss_MemRegion_MEMORY_REGION0;
315     if (res_mgr_init_qmss (&qmss_cfg) != 0)
316     {
317         platform_write ("Failed to initialize the QMSS subsystem \n");
318         goto main_exit;
319     }
320     else
321     {
322      platform_write ("QMSS successfully initialized \n");
```

```
323     }
324
325     /* Initialize CPPI */
326     if (platform_get_coreid() == 0)
327     {
328         cppi_cfg.master_core        = 1;
329     }
330     else
331     {
332         cppi_cfg.master_core        = 0;
333     }
334     cppi_cfg.dma_num            = Cppi_CpDma_PASS_CPDMA;
335     cppi_cfg.num_tx_queues      = NUM_PA_TX_QUEUES;
336     cppi_cfg.num_rx_channels    = NUM_PA_RX_CHANNELS;
337     if (res_mgr_init_cppi (&cppi_cfg) != 0)
338     {
339         platform_write ("Failed to initialize CPPI subsystem \n");
340         goto main_exit;
341     }
342     else
343     {
344      platform_write ("CPPI successfully initialized \n");
345     }
346
347
348     if (res_mgr_init_pass()!= 0) {
349         platform_write ("Failed to initialize the Packet Accelerator \n");
350         goto main_exit;
351     }
352     else
353     {
354      platform_write ("PA successfully initialized \n");
355     }
356
357     rc = NC_SystemOpen( NC_PRIORITY_HIGH, NC_OPMODE_INTERRUPT );//NC_PRIORITY_HIGH
           now is 8
358     if( rc )
359     {
360         platform_write("NC_SystemOpen Failed (%d)\n",rc);
361         for(;;);
362     }
363
364     // Print out our banner
365     platform_write(VerStr);
366
367     //
368     // Create and build the system configuration from scratch.
369     //
```

```
370
371     // Create a new configuration
372     hCfg = CfgNew();
373     if( !hCfg )
374     {
375         platform_write("Unable to create configuration\n");
376         goto main_exit;
377     }
378
379     // We better validate the length of the supplied names
380     if( strlen( DomainName ) >= CFG_DOMAIN_MAX ||
381         strlen( HostName ) >= CFG_HOSTNAME_MAX )
382     {
383         printf("Names too long\n");
384         goto main_exit;
385     }
386
387     // Add our global hostname to hCfg (to be claimed in all connected domains)
388     CfgAddEntry( hCfg, CFGTAG_SYSINFO, CFGITEM_DHCP_HOSTNAME, 0,
389                  strlen(HostName), (UINT8 *)HostName, 0 );
390
391     // Shared memory for the big data blocks
392          HeapBufMP_Params_init(&heapBufParams);
393          heapBufParams.regionId       = 0;
394          heapBufParams.name           = HEAP_NAME;
395          heapBufParams.numBlocks      = 5;
396          heapBufParams.blockSize      = sizeof(datachunk);
397          heapHandle = HeapBufMP_create(&heapBufParams);
398          if (heapHandle == NULL) {
399              System_abort("HeapBufMP_create failed\n" );
400          }
401
402     // Shared memory for small data blocks
403          HeapBufMP_Params_init(&heapBufParams_aux);
404          heapBufParams_aux.regionId       = 0;
405          heapBufParams_aux.name           = HEAP_AUX_NAME;
406          heapBufParams_aux.numBlocks      = 132;
407          heapBufParams_aux.blockSize      = sizeof(errorchunk);
408          heapHandle_aux = HeapBufMP_create(&heapBufParams_aux);
409          if (heapHandle == NULL) {
410              System_abort("HeapBufMP_create failed\n" );
411          }
412
413     // If the IP address is specified, manually configure IP and Gateway
414 #if defined(_SCBP6618X_) || defined(_EVMTCI6614_) || defined(DEVICE_K2H) ||
      defined(DEVICE_K2K)
415     /* SCBP6618x, EVMTCI6614, EVMK2H, EVMK2K always uses DHCP */
416     if (0)
```

```
417 #else
418     if (!platform_get_switch_state(1))
419 #endif
420     {
421         CI_IPNET NA;
422         CI_ROUTE RT;
423         IPN      IPTmp;
424 
425         // Setup manual IP address
426         bzero( &NA, sizeof(NA) );
427         NA.IPAddr  = inet_addr(LocalIPAddr);
428         NA.IPMask  = inet_addr(LocalIPMask);
429         strcpy( NA.Domain, DomainName );
430         NA.NetType = 0;
431 
432         // Add the address to interface 1
433         CfgAddEntry( hCfg, CFGTAG_IPNET, 1, 0,
434                             sizeof(CI_IPNET), (UINT8 *)&NA, 0 );
435 
436         // Add the default gateway. Since it is the default, the
437         // destination address and mask are both zero (we go ahead
438         // and show the assignment for clarity).
439         bzero( &RT, sizeof(RT) );
440         RT.IPDestAddr = 0;
441         RT.IPDestMask = 0;
442         RT.IPGateAddr = inet_addr(GatewayIP);
443 
444         // Add the route
445         CfgAddEntry( hCfg, CFGTAG_ROUTE, 0, 0,
446                             sizeof(CI_ROUTE), (UINT8 *)&RT, 0 );
447 
448         // Manually add the DNS server when specified
449         IPTmp = inet_addr(DNSServer);
450         if( IPTmp )
451             CfgAddEntry( hCfg, CFGTAG_SYSINFO, CFGITEM_DHCP_DOMAINNAMESERVER,
452                         0, sizeof(IPTmp), (UINT8 *)&IPTmp, 0 );
453     }
454     // Else we specify DHCP
455     else
456     {
457         CI_SERVICE_DHCPC dhcpc;
458 
459         // Specify DHCP Service on IF-1
460         bzero( &dhcpc, sizeof(dhcpc) );
461         dhcpc.cisargs.Mode   = CIS_FLG_IFIDXVALID;
462         dhcpc.cisargs.IfIdx  = 1;
463         dhcpc.cisargs.pCbSrv = &ServiceReport;
464         CfgAddEntry( hCfg, CFGTAG_SERVICE, CFGITEM_SERVICE_DHCPCLIENT, 0,
```

```
465                         sizeof(dhcpc), (UINT8 *)&dhcpc, 0 );
466     }
467
468     //
469     // Configure IPStack/OS Options
470     //
471
472     // We don't want to see debug messages less than WARNINGS
473     rc = DBG_WARN;
474     CfgAddEntry( hCfg, CFGTAG_OS, CFGITEM_OS_DBGPRINTLEVEL,
475                  CFG_ADDMODE_UNIQUE, sizeof(uint), (UINT8 *)&rc, 0 );
476
477     //
478     // This code sets up the TCP and UDP buffer sizes
479     // (Note 8192 is actually the default. This code is here to
480     // illustrate how the buffer and limit sizes are configured.)
481     //
482
483     // UDP Receive limit
484     rc = 8192;
485     CfgAddEntry( hCfg, CFGTAG_IP, CFGITEM_IP_SOCKUDPRXLIMIT,
486                  CFG_ADDMODE_UNIQUE, sizeof(uint), (UINT8 *)&rc, 0 );
487
488     //
489     // Boot the system using this configuration
490     //
491     // We keep booting until the function returns 0. This allows
492     // us to have a "reboot" command.
493     //
494
495     do
496     {
497             rc = NC_NetStart( hCfg, NetworkOpen, NetworkClose, NetworkIPAddr );
498     } while( rc > 0 );
499
500     // Delete Configuration
501     CfgFree( hCfg );
502
503     // Close the OS
504  main_exit:
505       NC_SystemClose();
506       return;
507 }
508
509 // Equalizer data distribution task
510
511 void udp_echo_test()
512 {
```

```
513
514     SOCKET  s;
515     struct  sockaddr_in sin1;
516     struct  sockaddr_in sin2;
517     int     i,tmp,j,n;
518     float    *pBuf;
519       struct  timeval timeout;
520       HANDLE  hBuffer;
521
522     // Data structures for local processing
523
524       struct localchunk local;
525
526       float x_i[FRAME+FIR_LEN];
527       float x_q[FRAME+FIR_LEN];
528       float temp[FIR_LEN];
529       float sqnorm;
530
531     // Data transport pointers and queue handles
532
533       struct datachunk * input;
534       struct datachunk * output;
535       struct errorchunk * inorm;
536       struct errorchunk * rnorm;
537
538       MessageQ_Handle  messageQ;
539       MessageQ_QueueId realQueueId;
540       MessageQ_QueueId imagQueueId;
541       MessageQ_QueueId rnormQueueId;
542       MessageQ_QueueId inormQueueId;
543
544       Int              status;
545
546     // Register the shared memory heaps allocated previously
547
548       MessageQ_registerHeap((IHeap_Handle)heapHandle, HEAPID);
549       MessageQ_registerHeap((IHeap_Handle)heapHandle_aux, HEAPID_AUX);
550
551       printf("Heap registered\n");
552
553       // Create the local message queue for the incoming packets
554       messageQ = MessageQ_create(localQueueName, NULL);
555       if (messageQ == NULL) {
556           System_abort("MessageQ_create failed\n" );
557       }
558
559     // Open the remote processor queues
560
```

```
561       do {
562           status = MessageQ_open(imagQueueName, &imagQueueId);
563           if (status < 0) {
564               Task_sleep(1);
565           }
566       } while (status < 0);
567
568       do {
569           status = MessageQ_open(realQueueName, &realQueueId);
570           if (status < 0) {
571               Task_sleep(1);
572           }
573       } while (status < 0);
574
575       do {
576           status = MessageQ_open(rnormQueueName, &rnormQueueId);
577           if (status < 0) {
578               Task_sleep(1);
579           }
580       } while (status < 0);
581
582       do {
583           status = MessageQ_open(inormQueueName, &inormQueueId);
584           if (status < 0) {
585               Task_sleep(1);
586           }
587       } while (status < 0);
588
589       printf("Opened the queues\n");
590
591       // Allocate the file environment for this task
592       fdOpenSession( TaskSelf() );
593
594       printf("\n== Start UDP Echo Client Test ==\n");
595
596     // Create test socket
597     s = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
598     if( s == INVALID_SOCKET )
599     {
600          printf("failed socket create (%d)\n",fdError());
601          goto leave;
602     }
603
604     // Bind own address to the socket
605     bzero( &sin1, sizeof(struct sockaddr_in) );
606     sin1.sin_family      = AF_INET;
607     sin1.sin_len         = sizeof( sin1 );
608     sin1.sin_addr.s_addr = inet_addr(LocalIPAddr);
```

```
609     sin1.sin_port        = htons(25000);
610
611     if( bind( s, (PSA) &sin1, sizeof(sin1) ) < 0 )
612     {
613         printf("failed bind (%d)\n",fdError());
614         goto leave;
615     }
616
617     // Setting the data processing vectors
618     memset(x_i, 0, sizeof(x_i));
619     memset(x_q, 0, sizeof(x_q));
620
621       printf("binded the address\n");
622
623     // Configure our timeout
624     timeout.tv_sec  = 1;
625     timeout.tv_usec = 0;
626
627     setsockopt( s, SOL_SOCKET, SO_SNDTIMEO, &timeout, sizeof( timeout ) );
628     setsockopt( s, SOL_SOCKET, SO_RCVTIMEO, &timeout, sizeof( timeout ) );
629
630     // Allocate a buffer for the data
631     if( !(pBuf = mmBulkAlloc( sizeof(float)*FRAME*4 )) )
632     {
633       printf("failed temp buffer allocation\n");
634         goto leave;
635     }
636
637       while(1)
638       {
639           tmp = sizeof( sin2 );
640
641       // Attempt to receive data from the socket
642           i = (int)recvncfrom( s, (void **)&pBuf, 0, (PSA)&sin2, &tmp, &hBuffer );
643
644           // If data received then go into processing mode
645           if( i >= 0 )
646           {
647        // A record to the execution logger
648             Log_writeUC3(UIABenchmark_startInstanceWithAdrs,(IArg)"context=0x%x,
                   fnAdrs=0x%x:",(IArg)0, (IArg)&udp_echo_test);
649
650         // Allocate and send the data frame to the 2nd core
651               output = (datachunk*)MessageQ_alloc(HEAPID, sizeof(datachunk));
652             memcpy(output->payload.payload_i, pBuf, sizeof(float)*FRAME*4+1);
653             Cache_wb(output, sizeof(datachunk),Cache_Type_ALL, FALSE);
654               MessageQ_put(imagQueueId, (MessageQ_Msg)output);
655
```

```
656         // Allocate and send the data frame to the 1st core
657               output = (datachunk*)MessageQ_alloc(HEAPID, sizeof(datachunk));
658             memcpy(output->payload.payload_i, pBuf, sizeof(float)*FRAME*4+1);
659             Cache_wb(output, sizeof(datachunk),Cache_Type_ALL, FALSE);
660               MessageQ_put(realQueueId, (MessageQ_Msg)output);
661
662         // Parse data for the local processing (Euclidean norm)
663               memcpy(local.inphase, pBuf, sizeof(float)*FRAME*2+1);
664               memcpy(x_i+FIR_LEN-1, local.inphase, sizeof(float)*FRAME);
665               memcpy(x_q+FIR_LEN-1, local.quadrat, sizeof(float)*FRAME);
666
667         // Calculate the norm for each sample and
668         // distribute it to other processors
669
670               for(j=0;j<FRAME;j++){
671
672                 sqnorm = 0.0f;
673                   for(n=0;n<FIR_LEN;n++)
674                     sqnorm += (x_i[j+n]*x_i[j+n]) + (x_q[j+n]*x_q[j+n]);
675
676                   inorm = (errorchunk*)MessageQ_alloc(HEAPID_AUX, sizeof(errorchunk
                          ));
677                   inorm->small.i_element = sqnorm;
678                   Cache_wb(inorm, sizeof(errorchunk),Cache_Type_ALL, FALSE);
679                   MessageQ_put(inormQueueId, (MessageQ_Msg)inorm);
680
681                   rnorm = (errorchunk*)MessageQ_alloc(HEAPID_AUX, sizeof(errorchunk
                          ));
682                   rnorm->small.i_element = sqnorm;
683                   Cache_wb(rnorm, sizeof(errorchunk),Cache_Type_ALL, FALSE);
684                   MessageQ_put(rnormQueueId, (MessageQ_Msg)rnorm);
685
686               }
687
688         // Shift back the remaining samples
689               memcpy(temp, x_q+FRAME, sizeof(float)*FIR_LEN-1);
690               memcpy(x_q, temp, sizeof(float)*FIR_LEN-1);
691               memcpy(temp, x_i+FRAME, sizeof(float)*FIR_LEN-1);
692               memcpy(x_i, temp, sizeof(float)*FIR_LEN-1);
693
694         // A record to the execution logger
695               Log_writeUC3(UIABenchmark_stopInstanceWithAdrs,  (IArg)"context=0x%x,
                        fnAdrs=0x%x:",(IArg)0, (IArg)&udp_echo_test);
696
697         // Erase the UDP buffer, just in case
698               memset(pBuf, 0, sizeof(float)*FRAME*4+1);
699
700         // Get the processed data from the submain core
```

```
701                 MessageQ_get(messageQ, (MessageQ_Msg*)&input, MessageQ_FOREVER);
702                 Cache_inv(input, sizeof(datachunk),Cache_Type_ALL, FALSE);
703                 memcpy(pBuf, input->payload.payload_i, sizeof(float)*FRAME*4+1);
704                 MessageQ_free((MessageQ_Msg)input);
705
706           // Return the data back to the sender
707                 if(sendto( s, pBuf, sizeof(float)*FRAME*4, 0, (PSA)&sin2, sizeof(sin2
                        ) ) < 0){
708                   printf("send failed (%d)\n",fdError());
709                 }
710                 recvncfree( hBuffer );
711             }
712
713         }
714
715     leave:
716       if( pBuf )
717         mmBulkFree( pBuf );
718         if( s != INVALID_SOCKET )
719             fdClose( s );
720         printf("== End UDP Echo Client Test ==\n\n");
721
722 }
723
724 //
725 // NetworkOpen
726 //
727 // This function is called after the configuration has booted
728 //
729 static void NetworkOpen(){}
730
731 //
732 // NetworkClose
733 //
734 // This function is called when the network is shutting down,
735 // or when it no longer has any IP addresses assigned to it.
736 //
737 static void NetworkClose() {}
738
739
740 //
741 // NetworkIPAddr
742 //
743 // This function is called whenever an IP address binding is
744 // added or removed from the system.
745 //
746 static void NetworkIPAddr( IPN IPAddr, uint IfIdx, uint fAdd )
747 {
```

```
748     IPN IPTmp;
749
750     if( fAdd )
751         printf("Network Added: ");
752     else
753         printf("Network Removed: ");
754
755     // Print a message
756     IPTmp = ntohl( IPAddr );
757     printf("If-%d:%d.%d.%d.%d\n", IfIdx,
758             (UINT8)(IPTmp>>24)&0xFF, (UINT8)(IPTmp>>16)&0xFF,
759             (UINT8)(IPTmp>>8)&0xFF, (UINT8)IPTmp&0xFF );
760 }
761
762 //
763 // Service Status Reports
764 //
765 // Here's a quick example of using service status updates
766 //
767 static char *TaskName[]   = { "Telnet","HTTP","NAT","DHCPS","DHCPC","DNS" };
768 static char *ReportStr[] = { "","Running","Updated","Complete","Fault" };
769 static char *StatusStr[] = { "Disabled","Waiting","IPTerm","Failed","Enabled" };
770 static void ServiceReport( uint Item, uint Status, uint Report, HANDLE h )
771 {
772     printf( "Service Status: %-9s: %-9s: %-9s: %03d\n",
773             TaskName[Item-1], StatusStr[Status],
774             ReportStr[Report/256], Report&0xFF );
775
776     //
777     // Example of adding to the DHCP configuration space
778     //
779     // When using the DHCP client, the client has full control over access
780     // to the first 256 entries in the CFGTAG_SYSINFO space.
781     //
782     // Note that the DHCP client will erase all CFGTAG_SYSINFO tags except
783     // CFGITEM_DHCP_HOSTNAME. If the application needs to keep manual
784     // entries in the DHCP tag range, then the code to maintain them should
785     // be placed here.
786     //
787     // Here, we want to manually add a DNS server to the configuration, but
788     // we can only do it once DHCP has finished its programming.
789     //
790     if( Item == CFGITEM_SERVICE_DHCPCLIENT &&
791         Status == CIS_SRV_STATUS_ENABLED &&
792         (Report == (NETTOOLS_STAT_RUNNING|DHCPCODE_IPADD) ||
793          Report == (NETTOOLS_STAT_RUNNING|DHCPCODE_IPRENEW)) )
794     {
795         IPN IPTmp;
```

```
796
797         // Manually add the DNS server when specified
798         IPTmp = inet_addr(DNSServer);
799         if( IPTmp )
800             CfgAddEntry( 0, CFGTAG_SYSINFO, CFGITEM_DHCP_DOMAINNAMESERVER,
801                          0, sizeof(IPTmp), (UINT8 *)&IPTmp, 0 );
802     }
803 }
```

Listing 10: Main core program code

2 Main processor configuration

The main core image is based on the modified **TCP/IP Stack 'Hello World!'** example by Texas Instruments

```
/*
 * helloWorld.cfg
 *
 * Memory Map and Program intiializations for the helloWorld example Utility
 *
 * Copyright (C) 2010-2011 Texas Instruments Incorporated - http://www.ti.com/
 *
 *  Redistribution and use in source and binary forms, with or without
 *  modification, are permitted provided that the following conditions
 *  are met:
 *
 *    Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *
 *    Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the
 *    distribution.
 *
 *    Neither the name of Texas Instruments Incorporated nor the names of
 *    its contributors may be used to endorse or promote products derived
 *    from this software without specific prior written permission.
 *
 *  THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
 *  "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
 *  LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
 *  A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
 *  OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 *  SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
 *  LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
 *  DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
 *  THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
 *  (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
 *  OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
```

```
36 */
37
38 /*
39  *   @file  helloWorld.cfg
40  *
41  *   @brief
42  *      Memory Map and Program intiializations for the HPDSP Utility.
43  *
44  */
45
46 /*********************************************************
47 *  Specify all needed RTSC MOudles and configure them.  *
48 *********************************************************/
49
50 var Memory  =   xdc.useModule('xdc.runtime.Memory');
51 var BIOS    =   xdc.useModule('ti.sysbios.BIOS');
52 var Task    =   xdc.useModule('ti.sysbios.knl.Task');
53 var HeapBuf =   xdc.useModule('ti.sysbios.heaps.HeapBuf');
54 var Log     =   xdc.useModule('xdc.runtime.Log');
55 var Mailbox =  xdc.useModule('ti.sysbios.knl.Mailbox');
56 var Semaphore = xdc.useModule('ti.sysbios.knl.Semaphore');
57
58 /*
59 ** Allow storing of task names. By default if you name a task with a friendly
       display name it will not be saved
60 ** to conserve RAM. This must be set to true to allow it. We use friendly names on
       the Task List display.
61 */
62 Task.common$.namedInstance = true;
63
64 var Clock   =   xdc.useModule ('ti.sysbios.knl.Clock');
65 var Hwi = xdc.useModule ('ti.sysbios.hal.Hwi');
66 var Ecm = xdc.useModule ('ti.sysbios.family.c64p.EventCombiner');
67
68 /*
69 ** Configure this to turn on the CPU Load Module for BIOS.
70 **
71 */
72
73
74 var Diags       = xdc.useModule('xdc.runtime.Diags');
75
76 /* Load the CSL package */
77 var Csl               =   xdc.useModule('ti.csl.Settings');
78 /* Load the CPPI package */
79 var Cppi                        =   xdc.loadPackage('ti.drv.cppi');
80 /* Load the QMSS package */
81 var Qmss                        =   xdc.loadPackage('ti.drv.qmss');
```

```
82  /* Load the PA package */
83  var Pa              =   xdc.useModule('ti.drv.pa.Settings');
84
85  /* Load the Platform/NDK Transport packages */
86  var PlatformLib  = xdc.loadPackage('ti.platform.evmc6678l');
87  var NdkTransport = xdc.loadPackage('ti.transport.ndk');
88
89  var System   = xdc.useModule('xdc.runtime.System');
90  var SysStd   = xdc.useModule('xdc.runtime.SysStd');
91  System.SupportProxy = SysStd;
92
93  var MultiProc = xdc.useModule('ti.sdo.utils.MultiProc');
94
95  var UIASync = xdc.useModule('ti.uia.events.UIASync');
96  var UIABenchmark = xdc.useModule('ti.uia.events.UIABenchmark');
97  var Timestamp = xdc.useModule('xdc.runtime.Timestamp');
98
99  var LoggingSetup = xdc.useModule('ti.uia.sysbios.LoggingSetup');
100
101 var LogSync = xdc.useModule('ti.uia.runtime.LogSync');
102 LogSync.enableEventCorrelationForJTAG = true;
103
104 var Idle = xdc.useModule('ti.sysbios.knl.Idle');
105 Idle.addFunc('&ti_uia_runtime_LogSync_idleHook');
106
107 LoggingSetup.sysbiosTaskLogging = false;
108
109 LoggingSetup.eventUploadMode = LoggingSetup.UploadMode_JTAGRUNMODE;
110 LoggingSetup.disableMulticoreEventCorrelation = false;
111
112 LoggingSetup.loadLoggerSize = 2048;
113 LoggingSetup.mainLoggerSize = 32768;
114 LogSync.defaultSyncLoggerSize = 2048;
115 LoggingSetup.sysbiosLoggerSize = 32768;
116
117 BIOS.libType = BIOS.LibType_Custom;
118
119 /*
120  *  Since this is a single-image example, we don't (at build-time) which
121  *  processor we're building for.  We therefore supply 'null'
122  *  as the local procName and allow IPC to set the local procId at runtime.
123  */
124 MultiProc.setConfig(null, ["CORE0", "CORE1", "CORE2"]);
125
126 var MessageQ    = xdc.useModule('ti.sdo.ipc.MessageQ');
127 var Ipc         = xdc.useModule('ti.sdo.ipc.Ipc');
128 var HeapBufMP   = xdc.useModule('ti.sdo.ipc.heaps.HeapBufMP');
129 var MultiProc   = xdc.useModule('ti.sdo.utils.MultiProc');
```

```
130
131 /*
132 ** Sets up the exception log so you can read it with ROV in CCS
133 */
134 var LoggerBuf = xdc.useModule('xdc.runtime.LoggerBuf');
135 var Exc = xdc.useModule('ti.sysbios.family.c64p.Exception');
136 Exc.common$.logger = LoggerBuf.create();
137 Exc.enablePrint = true; /* prints exception details to the CCS console */
138
139 /*
140 **  Give the Load module it's own LoggerBuf to make sure the
141 **  events are not overwritten.
142 */
143 /* var loggerBufParams = new LoggerBuf.Params();
144 loggerBufParams.exitFlush = true;
145 loggerBufParams.numEntries = 64;
146 Load.common$.logger = LoggerBuf.create(loggerBufParams);
147 */
148 var Global       = xdc.useModule('ti.ndk.config.Global');
149
150 /*
151 ** Use this load to configure NDK 2.2 and above using RTSC. In previous versions
    of
152 ** the NDK RTSC configuration was not supported and you should comment this out.
153 */
154
155 /*
156 ** This allows the heart beat (poll function) to be created but does not generate
    the stack threads
157 **
158 ** Look in the cdoc (help files) to see what CfgAddEntry items can be configured.
    We tell it NOT
159 ** to create any stack threads (services) as we configure those ourselves in our
    Main Task
160 ** thread hpdspuaStart.
161 */
162 Global.enableCodeGeneration = false;
163
164 /* Define a variable to set the MAR mode for MSMCSRAM as all cacheable */
165 var Cache       =   xdc.useModule('ti.sysbios.family.c66.Cache');
166 var Startup     =   xdc.useModule('xdc.runtime.Startup');
167 var System      =   xdc.useModule('xdc.runtime.System');
168
169 /*
170 ** Create a Heap.
171 */
172 var HeapMem = xdc.useModule('ti.sysbios.heaps.HeapMem');
173 var heapMemParams = new HeapMem.Params();
```

```
174 heapMemParams.size = 0x300000;
175 heapMemParams.sectionName = "systemHeap";
176 Program.global.heap0 = HeapMem.create(heapMemParams);
177
178 /* This is the default memory heap. */
179 Memory.defaultHeapInstance  =  Program.global.heap0;
180 Program.sectMap["sharedL2"] = "DDR3";
181 Program.sectMap["systemHeap"] = "DDR3";
182 Program.sectMap[".sysmem"]  = "DDR3";
183 Program.sectMap[".args"]    = "DDR3";
184 Program.sectMap[".cio"]     = "DDR3";
185 Program.sectMap[".far"]   =   "DDR3";
186 Program.sectMap[".rodata"]  =   "DDR3";
187 Program.sectMap[".neardata"]  =   "DDR3";
188 Program.sectMap[".cppi"]  =   "DDR3";
189 Program.sectMap[".init_array"]  =   "DDR3";
190 Program.sectMap[".qmss"]  =   "DDR3";
191 Program.sectMap[".cinit"]   =   "DDR3";
192 Program.sectMap[".bss"]   = "DDR3";
193 Program.sectMap[".const"] = "DDR3";
194 Program.sectMap[".text"]  = "DDR3";
195 Program.sectMap[".code"]  = "DDR3";
196 Program.sectMap[".switch"]  = "DDR3";
197 Program.sectMap[".data"]  = "DDR3";
198 Program.sectMap[".fardata"] =   "DDR3";
199 Program.sectMap[".args"]  =   "DDR3";
200 Program.sectMap[".cio"]   =   "DDR3";
201 Program.sectMap[".vecs"]  =   "DDR3";
202 Program.sectMap["platform_lib"]   =   "DDR3";
203 Program.sectMap[".far:taskStackSection"] = "L2SRAM";
204 Program.sectMap[".stack"] = "L2SRAM";
205 Program.sectMap[".nimu_eth_ll2"] = "L2SRAM";
206 Program.sectMap[".resmgr_memregion"] = {loadSegment: "L2SRAM", loadAlign:128};
207 /* QMSS descriptors region  */
208 Program.sectMap[".resmgr_handles"] = {loadSegment: "L2SRAM", loadAlign:16};
209 /* CPPI/QMSS/PA Handles     */
210 Program.sectMap[".resmgr_pa"] = {loadSegment: "L2SRAM", loadAlign:8};
211 /* PA Memory          */
212 Program.sectMap[".far:IMAGEDATA"] = {loadSegment: "L2SRAM", loadAlign: 8};
213 Program.sectMap[".far:NDK_OBJMEM"] = {loadSegment: "L2SRAM", loadAlign: 8};
214 Program.sectMap[".far:NDK_PACKETMEM"] = {loadSegment: "L2SRAM", loadAlign: 128};
215
216 /* Required if using System_printf to output on the console */
217 SysStd              =   xdc.useModule('xdc.runtime.SysStd');
218 System.SupportProxy     =   SysStd;
219
220 /*************************************************************
221 * Define hooks and static tasks  that will always be running.*
```

```
222  ************************************************************/
223
224 /*
225 ** Register an EVM Init handler with BIOS. This will initialize the hardware. BIOS
        calls before it starts.
226 **
227 ** If yuo are debugging with CCS, then this function will execute as CCS loads it
        if the option in your
228 ** Target Configuraiton file (.ccxml) has the option set to execute all code
        before Main. That is the
229 ** default.
230 */
231 Startup.lastFxns.$add('&EVM_init');
232
233 /*
234 ** Create the stack Thread Task for our application.
235 */
236 var tskNdkStackTest     =   Task.create("&StackTest");
237 tskNdkStackTest.stackSize   =   0x1400;
238 tskNdkStackTest.priority    =   0x5;
239
240 Ipc.procSync = Ipc.ProcSync_ALL;
241
242 /* Shared Memory base address and length */
243 var SHAREDMEM           = 0x0C000000;
244 var SHAREDMEMSIZE       = 0x00200000;
245
246 Program.global.HEAP_MSGSIZE = 2048;
247
248 /*
249  *  Need to define the shared region. The IPC modules use this
250  *  to make portable pointers. All processors need to add this
251  *  call with their base address of the shared memory region.
252  *  If the processor cannot access the memory, do not add it.
253  */
254 var SharedRegion = xdc.useModule('ti.sdo.ipc.SharedRegion');
255 SharedRegion.setEntryMeta(0,
256     { base: SHAREDMEM,
257       len:  SHAREDMEMSIZE,
258       ownerProcId: 0,
259       isValid: true,
260       name: "DDR2 RAM",
261     });
262
263 /*
264 ** If you are using RTSC configuration with NDK 2.2 and above, this is done by
        default, else
265 ** register hooks so that the stack can track all Task creation
```

```
266 Task.common$.namedInstance  =   true;
267 Task.addHookSet ({ registerFxn: '&NDK_hookInit', createFxn: '&NDK_hookCreate', });
268
269 /* Enable BIOS Task Scheduler */
270 BIOS.taskEnabled      =   true;
271
272 /*
273  * Enable Event Groups here and registering of ISR for specific GEM INTC is done
274  * using EventCombiner_dispatchPlug() and Hwi_eventMap() APIs
275  */
276
277 Ecm.eventGroupHwiNum[0] = 7;
278 Ecm.eventGroupHwiNum[1] = 8;
279 Ecm.eventGroupHwiNum[2] = 9;
280 Ecm.eventGroupHwiNum[3] = 10;
281
282 var Timer = xdc.useModule('ti.sysbios.hal.Timer');
283  var timerParams = new Timer.Params();
284  timerParams.startMode = Timer.StartMode_AUTO;
285  timerParams.period = 100000;        // 100,000 uSecs = 100ms
286  var timer0 = Timer.create(Timer.ANY, '&ti_uia_runtime_LogSync_timerHook',
       timerParams);
```

Listing 11: Main processor configuration script

3 Subprogram source code

The auxiliary core image is based on the modified **Multiprocessor MessageQ** example by Texas Instruments

```
/*
 * Copyright (c) 2012, Texas Instruments Incorporated
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * *  Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *
 * *  Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * *  Neither the name of Texas Instruments Incorporated nor the names of
 *    its contributors may be used to endorse or promote products derived
 *    from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
 * THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
 * PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS;
 * OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
 * WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
 * OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE,
 * EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 * */
/*
 *  ======== message_multicore.c ========
 *  Multiprocessor MessageQ example
 *
```

```
36   *  This is an example program that uses MessageQ to pass a message
37   *  from one processor to another.
38   *
39   *  Each processor creates its own MessageQ first and then will try to open
40   *  a remote processor's MessageQ.
41   *
42   *  See message_multicore.k file for expected output.
43   */
44
45  #include <xdc/std.h>
46  #include <string.h>
47
48  /*  ----------------------------------XDC.RUNTIME module Headers    */
49  #include <xdc/runtime/System.h>
50  #include <xdc/runtime/IHeap.h>
51
52  /*  ----------------------------------- IPC module Headers           */
53  #include <ti/ipc/Ipc.h>
54  #include <ti/ipc/MessageQ.h>
55  #include <ti/ipc/HeapBufMP.h>
56  #include <ti/ipc/MultiProc.h>
57
58  #include <ti/dsplib/dsplib.h>
59
60  /*  ----------------------------------- BIOS6 module Headers         */
61  #include <ti/sysbios/BIOS.h>
62  #include <ti/sysbios/knl/Task.h>
63  #include <ti/sysbios/hal/Cache.h>
64  #include <ti/sysbios/knl/Swi.h>
65
66  /*  ----------------------------------- To get globals from .cfg Header */
67  #include <xdc/cfg/global.h>
68
69  #include <xdc/runtime/Log.h>
70  #include <ti/uia/runtime/LogSync.h>
71  #include <c6x.h>
72  #include <xdc/runtime/Timestamp.h>
73  #include <xdc/runtime/Types.h>
74  #include <xdc/runtime/Diags.h>
75  #include <ti/uia/runtime/LogUC.h>
76  #include <ti/uia/events/UIABenchmark.h>
77
78  // Definitions for the shared memory
79  #define HEAP_NAME   "myHeapBuf"
80  #define HEAP_AUX_NAME   "auxHeapBuf"
81  #define HEAPID      0
82  #define HEAPID_AUX      1
83
```

```
84  // Data block length and filter length
85  #define BLOCK_LEN   64
86  #define FIR_LEN   16
87
88  Char localQueueName[10];
89  Char auxQueueName[10];
90  Char ownQueueName[10];
91  Char backQueueName[10];
92  Char normQueueName[10];
93
94  int submain = 0;
95
96  #define FRAME      64
97  #define MU_STEP    0.05 // Adaptation rate
98
99  // Full data packet
100 typedef struct datachunk {
101   MessageQ_MsgHeader header;
102   struct payload{
103     float payload_i[FRAME];
104     float payload_q[FRAME];
105     float refload_i[FRAME];
106     float refload_q[FRAME];
107   } payload;
108 } datachunk ;
109
110 // Small data packet
111 typedef struct errorchunk {
112   MessageQ_MsgHeader header;
113   struct small{
114     float i_element;
115     float q_element;
116   } small;
117 } errorchunk ;
118
119 // Signal processing task
120 Void tsk0_func(UArg arg0, UArg arg1)
121 {
122   // Data transport pointers and queue handles
123
124     MessageQ_Handle  data_inQ;
125     MessageQ_Handle  aux_inQ;
126     MessageQ_Handle  norm_inQ;
127
128     MessageQ_QueueId auxQueueId;
129     MessageQ_QueueId backQueueId;
130
131     Int              status;
```

```
132     HeapBufMP_Handle     heapHandle;
133     HeapBufMP_Handle     heapHandle_aux;
134
135     struct datachunk * input;
136     struct datachunk * output;
137     struct errorchunk * auxin;
138     struct errorchunk * auxout;
139     struct errorchunk * normin;
140
141     int i, j;
142     float ep_q;
143
144   // 2nd core has to invert one coefficient for the adaptation
145     if(submain)
146       ep_q = 1.0f;
147     else
148       ep_q = -1.0f;
149
150     float y_i, y_q;
151     float err_i, err_q, norm;
152     const float a = 1.1755e-38;
153
154   // Local variables for processing
155     float kernel[FIR_LEN];
156     float x_i[BLOCK_LEN + FIR_LEN-1];
157     float x_q[BLOCK_LEN + FIR_LEN-1];
158     float y_ii[BLOCK_LEN];
159     float y_qi[BLOCK_LEN];
160     float temp[FIR_LEN];
161
162   // Initialize the arrays
163     memset(x_i, 0, sizeof(x_i));
164     memset(x_q, 0, sizeof(x_q));
165     memset(kernel, 0, sizeof(kernel));
166     memset(y_ii, 0, sizeof(err_i));
167     memset(y_qi, 0, sizeof(err_q));
168
169   // Initial kernel setting is all pass
170   // Real 0th coefficient = 1
171     if(submain)
172       kernel[FIR_LEN-1] = 1.0f;
173
174   // Open the heaps created by main processor
175     do {
176         status = HeapBufMP_open(HEAP_NAME, &heapHandle);
177         if (status < 0) {
178             Task_sleep(1);
179         }
```

```
180     } while (status < 0);
181
182     do {
183         status = HeapBufMP_open(HEAP_AUX_NAME, &heapHandle_aux);
184         if (status < 0) {
185             Task_sleep(1);
186         }
187     } while (status < 0);
188
189     System_printf("opened the heap\n");
190
191   // Register the shared memory heaps
192
193     MessageQ_registerHeap((IHeap_Handle)heapHandle, HEAPID);
194     MessageQ_registerHeap((IHeap_Handle)heapHandle_aux, HEAPID_AUX);
195
196   // Create the local message queue for the incoming packets
197
198     data_inQ = MessageQ_create(localQueueName, NULL);
199     if (data_inQ == NULL) {
200         System_abort("input MessageQ_create failed\n" );
201     }
202
203     aux_inQ = MessageQ_create(ownQueueName, NULL);
204     if (aux_inQ == NULL) {
205         System_abort("input MessageQ_create failed\n" );
206     }
207
208     norm_inQ = MessageQ_create(normQueueName, NULL);
209     if (norm_inQ == NULL) {
210         System_abort("input MessageQ_create failed\n" );
211     }
212
213   // Open the remote processor queues
214
215     do {
216         status = MessageQ_open(auxQueueName, &auxQueueId);
217         if (status < 0) {
218             Task_sleep(1);
219         }
220     } while (status < 0);
221
222     do {
223         status = MessageQ_open(backQueueName, &backQueueId);
224         if (status < 0) {
225             Task_sleep(1);
226         }
227     } while (status < 0);
```

```
228
229     while(1){
230
231     // Retrieve the data frame from the main core
232     status = MessageQ_get(data_inQ, (MessageQ_Msg*)&input, MessageQ_FOREVER);
233     Cache_inv(input, sizeof(datachunk),Cache_Type_ALL, FALSE);
234
235     // Copy the input samples in front of the past samples
236         memcpy(x_i+FIR_LEN-1, input->payload.payload_i, sizeof(float)*BLOCK_LEN);
237         memcpy(y_ii, input->payload.refload_i, sizeof(float)*BLOCK_LEN);
238         memcpy(x_q+FIR_LEN-1, input->payload.payload_q, sizeof(float)*BLOCK_LEN);
239         memcpy(y_qi, input->payload.refload_q, sizeof(float)*BLOCK_LEN);
240     MessageQ_free((MessageQ_Msg)input);
241
242     // Submain allocates the output frame
243     if(submain)
244       output = (datachunk*)MessageQ_alloc(HEAPID, sizeof(datachunk));
245
246     // A record to the execution logger
247     Log_writeUC3(UIABenchmark_startInstanceWithAdrs, (IArg)"context=0x%x, fnAdrs=0
          x%x:",(IArg)0, (IArg)&tsk0_func);
248
249
250     for(i = 0; i < BLOCK_LEN; i++){
251
252       y_i = 0.0f;
253       y_q = 0.0f;
254
255       // Calculate the convolution
256           for (j = 0; j < FIR_LEN; j++){
257               y_i += kernel[j] * x_i[i + j];
258             y_q += kernel[j] * x_q[i + j];
259           }
260
261       // 2nd core has to invert the imaginary product
262           if(!submain)
263             y_q = -y_q;
264
265       // Prepare the data for exchange with other processor
266       auxout = (errorchunk*)MessageQ_alloc(HEAPID_AUX, sizeof(errorchunk));
267
268       auxout->small.i_element = y_i;
269       auxout->small.q_element = y_q;
270
271       Cache_wb(auxout, sizeof(errorchunk),Cache_Type_ALL, FALSE);
272
273       // Exchange the data
274       // Processors wait for the data in a reciprocal way to avoid deadlocking
```

```
275         if(submain){
276
277           status = MessageQ_get(aux_inQ, (MessageQ_Msg*)&auxin, MessageQ_FOREVER);
278
279           status = MessageQ_put(auxQueueId, (MessageQ_Msg)auxout);
280
281         }else{
282
283           status = MessageQ_put(auxQueueId, (MessageQ_Msg)auxout);
284
285           status = MessageQ_get(aux_inQ, (MessageQ_Msg*)&auxin, MessageQ_FOREVER);
286         }
287
288         Cache_inv(auxin, sizeof(errorchunk),Cache_Type_ALL, FALSE);
289
290         // Inject the received elements to calculate the complex output
291         y_i += auxin->small.q_element;
292         y_q += auxin->small.i_element;
293
294         MessageQ_free((MessageQ_Msg)auxin);
295
296         // Submain program has to record the data
297         // Other processor error calculation is other way round
298         if(submain){
299           err_i = y_ii[i] - y_i;
300           err_q = y_qi[i] - y_q;
301           output->payload.payload_i[i] = y_i;
302           output->payload.payload_q[i] = y_q;
303           output->payload.refload_i[i] = err_i;
304           output->payload.refload_q[i] = err_q;
305         }else{
306
307           err_i = y_qi[i] - y_i;
308           err_q = y_ii[i] - y_q;
309         }
310
311         // Get the normalization factor from the queue
312         status = MessageQ_get(norm_inQ, (MessageQ_Msg*)&normin, MessageQ_FOREVER);
313         Cache_inv(normin, sizeof(errorchunk),Cache_Type_ALL, FALSE);
314         norm = normin->small.i_element;
315         MessageQ_free((MessageQ_Msg)normin);
316
317         // Complex adaptation process
318             for (j = 0; j < FIR_LEN; j++)
319               kernel[j] = kernel[j] + (MU_STEP/(norm+a))*( x_i[i+j] * err_i + x_q[i+
                      j] * err_q * ep_q );
320       }
321
```

```
322     // A record to the execution logger
323     Log_writeUC3(UIABenchmark_stopInstanceWithAdrs,  (IArg)"context=0x%x, fnAdrs=0
            x%x:",(IArg)0, (IArg)&tsk0_func);
324
325     // Submain program should send the recorded vector back to main
326     if(submain){
327       Cache_wb(output, sizeof(datachunk),Cache_Type_ALL, FALSE);
328       status = MessageQ_put(backQueueId, (MessageQ_Msg)output);
329     }
330
331     // Shift back the remaining samples
332         memcpy(temp, x_q+BLOCK_LEN, sizeof(float)*FIR_LEN-1);
333         memcpy(x_q, temp, sizeof(float)*FIR_LEN-1);
334         memcpy(temp, x_i+BLOCK_LEN, sizeof(float)*FIR_LEN-1);
335         memcpy(x_i, temp, sizeof(float)*FIR_LEN-1);
336     }
337 }
338
339 /*
340  *  ======== main ========
341  *  Synchronizes all processors (in Ipc_start) and calls BIOS_start
342  */
343 Int main()
344 {
345     Int status;
346
347   //Write the sync point for the execution logger
348     LogSync_writeSyncPoint();
349
350     /*
351      *  Ipc_start() calls Ipc_attach() to synchronize all remote processors
352      *  because 'Ipc.procSync' is set to 'Ipc.ProcSync_ALL' in *.cfg
353      */
354     status = Ipc_start();
355     if (status < 0) {
356         System_abort("Ipc_start failed\n");
357     }
358
359   // Determine the ID of the processor and set the appropriate queue names
360     if(MultiProc_self() == 1){
361         System_sprintf(localQueueName, "%s", "real\0");
362         System_sprintf(auxQueueName, "%s", "toim\0");
363         System_sprintf(ownQueueName, "%s", "tore\0");
364         System_sprintf(normQueueName, "%s", "rnorm\0");
365         submain = 1; // Processor 1 is the submain!
366     }else{
367         System_sprintf(localQueueName, "%s", "imag\0");
368         System_sprintf(auxQueueName, "%s", "tore\0");
```

```
369           System_sprintf(ownQueueName, "%s", "toim\0");
370           System_sprintf(normQueueName, "%s", "inorm\0");
371       }
372
373     System_sprintf(backQueueName, "%s", "netw\0");
374
375       BIOS_start();
376
377       return (0);
378   }
```

Listing 12: Auxiliary core program code

4 Subprocessor configuration

The auxiliary core image is based on the modified **Multiprocessor MessageQ** example by Texas Instruments

```
/*
 * Copyright (c) 2012, Texas Instruments Incorporated
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 * *  Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 *
 * *  Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 *
 * *  Neither the name of Texas Instruments Incorporated nor the names of
 *    its contributors may be used to endorse or promote products derived
 *    from this software without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
 * THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
 * PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR
 * CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
 * EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
 * PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS;
 * OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
 * WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
 * OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE,
 * EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 * */

var MultiProc = xdc.useModule('ti.sdo.utils.MultiProc');

var Log = xdc.useModule('xdc.runtime.Log');
```

```
36 var UIASync = xdc.useModule('ti.uia.events.UIASync');
37 var UIABenchmark = xdc.useModule('ti.uia.events.UIABenchmark');
38 var Timestamp = xdc.useModule('xdc.runtime.Timestamp');
39
40 var LoggingSetup = xdc.useModule('ti.uia.sysbios.LoggingSetup');
41
42 var LogSync = xdc.useModule('ti.uia.runtime.LogSync');
43 LogSync.enableEventCorrelationForJTAG = true;
44
45 var Idle = xdc.useModule('ti.sysbios.knl.Idle');
46 Idle.addFunc('&ti_uia_runtime_LogSync_idleHook');
47
48 LoggingSetup.sysbiosTaskLogging = false;
49
50 LoggingSetup.eventUploadMode = LoggingSetup.UploadMode_JTAGRUNMODE;
51 LoggingSetup.disableMulticoreEventCorrelation = false;
52
53 LoggingSetup.loadLoggerSize = 2048;
54 LoggingSetup.mainLoggerSize = 32768;
55 LoggingSetup.sysbiosLoggerSize = 32768;
56 LogSync.defaultSyncLoggerSize = 2048;
57
58 /*
59  *  Since this is a single-image example, we don't (at build-time) which
60  *  processor we're building for.  We therefore supply 'null'
61  *  as the local procName and allow IPC to set the local procId at runtime.
62  */
63 MultiProc.setConfig(null, ["CORE0", "CORE1", "CORE2"]);
64
65 /*
66  *  The SysStd System provider is a good one to use for debugging
67  *  but does not have the best performance. Use xdc.runtime.SysMin
68  *  for better performance.
69  */
70 var System    = xdc.useModule('xdc.runtime.System');
71 var SysStd    = xdc.useModule('xdc.runtime.SysStd');
72 System.SupportProxy = SysStd;
73
74 /* Modules explicitly used in the application */
75 var MessageQ     = xdc.useModule('ti.sdo.ipc.MessageQ');
76 var Ipc          = xdc.useModule('ti.sdo.ipc.Ipc');
77 var HeapBufMP    = xdc.useModule('ti.sdo.ipc.heaps.HeapBufMP');
78 var MultiProc    = xdc.useModule('ti.sdo.utils.MultiProc');
79
80 /* BIOS/XDC modules */
81 var BIOS         = xdc.useModule('ti.sysbios.BIOS');
82 BIOS.heapSize    = 0x8000;
83 var Task         = xdc.useModule('ti.sysbios.knl.Task');
```

```
84
85  var tsk0 = Task.create('&tsk0_func');
86  tsk0.instance.name = "tsk0";
87
88  /* Synchronize all processors (this will be done in Ipc_start) */
89  Ipc.procSync = Ipc.ProcSync_ALL;
90
91  /* Shared Memory base address and length */
92  var SHAREDMEM           = 0x0C000000;
93  var SHAREDMEMSIZE       = 0x00200000;
94
95  Program.global.HEAP_MSGSIZE = 2048;
96
97  /*
98   *  Need to define the shared region. The IPC modules use this
99   *  to make portable pointers. All processors need to add this
100  *  call with their base address of the shared memory region.
101  *  If the processor cannot access the memory, do not add it.
102  */
103
104 var SharedRegion = xdc.useModule('ti.sdo.ipc.SharedRegion');
105 SharedRegion.setEntryMeta(0,
106     { base: SHAREDMEM,
107       len:  SHAREDMEMSIZE,
108       ownerProcId: 0,
109       isValid: true,
110       name: "DDR2 RAM",
111     });
112
113 tsk0.stackSize = 4096;
114
115 var Timer = xdc.useModule('ti.sysbios.hal.Timer');
116  var timerParams = new Timer.Params();
117  timerParams.startMode = Timer.StartMode_AUTO;
118  timerParams.period = 100000;        // 100,000 uSecs = 100ms
119  var timer0 = Timer.create(Timer.ANY, '&ti_uia_runtime_LogSync_timerHook',
         timerParams);
```

Listing 13: Sub processor configuration script