Application Report
SWRA621–June 2018

Introduction to the DSP Subsystem in the IWR6843

Anil Mani, Sandeep Rao,Jasbir Nayyar, Mingjian Yan, and Brian Johnson

ABSTRACT

This application report introduces the digital signal processing subsystem (called the DSS) of the IWR6843, discusses the key blocks of the DSS, provides benchmarks for the digital signal processor (DSP), the Radar Hardware Accelerator (HWA) and enhanced direct memory access (EDMA), and presents typical radar processing chains. It is intended for engineers who wish to implement signal processing algorithms on the IWR6843.

Contents

List of Figures

List of Tables

Trademarks

1 Introduction

The IWR6843 device is the industry's first fully-integrated 57 GHz to 64 GHz, radar-on-a-chip solution for industrial radar applications. [1] The device comprises of the entire mmWave RF and analog baseband signal chain for three transmitters (TX) and four receivers (RX), two customer-programmable processor cores in the form of a C674x DSP and an ARM® Cortex®-R4F MCU, as well as a hardware accelerator (HWA) meant for offloading common radar algorithms.

This application report provides an introduction to the computational capabilities of the device with a focus on the DSP and the HWA. It is targeted towards engineers looking to implement radar signal processing algorithms on IWR6843, or port existing implementations from IWR16xx [2] to IWR6843.

The IWR6843 device is organized into different subsystems, each with a specific role. The mmWave RF and analog baseband signal chain are part of the 'Radar Subsystem'. The 'Radar Sub-system' is not customer programmable, and can be controlled only through pre-defined APIs (defined in [3]). The remaining two subsystems are customer programmable: 'Master Sub-System' (or MSS) and 'DSP Sub-System' (or DSS).

The MSS contains an R4F processor (clocked @ 200 Mhz) and is expected to be used to control communication between the device and external equipments. Spare MIPs on the MSS can be used to run late-stage algorithms like trackers (Kalman filters, Fusion (from different sub-frames), fencing, and so forth.). However, since these algorithms are not 'core signal processing' (MIPS intensive) algorithms, the MSS is not considered in this document.

The DSS contains the C674x DSP, the HWA and some peripherals and is expected to do the heavy lifting as far as FMCW processing is concerned, and it alone is the focus of this user's guide.

The guide is organized as follows - Section 2 provides an overview of the DSS including the capabilities of the DSP, the HWA and the EDMA. Section 3 describes a few typical processing chains that can be implemented using the DSS, as well as benchmarks for common algorithms. Section 4 discusses the data transfer associated with radar processing. Section 5 provides some hints on organizing the cache on the DSP. Finally Section 6 provides links and references to further information.

1.1 Abbreviations

Table 1 shows the abbreviations used this document:

Table 1. Abbreviations

No.	Abbreviation	Expansion
1	DSP	Digital signal processor
2	HWA	Hardware accelerator
3	Fencing	Counting the number of detected objects in a given volume
4	EDMA	Enhanced – DMA.

2 Overview of the DSP Subsystem (DSS)

Figure 1 shows a programmer's view of the DSS with callouts showing the suggested division of labor between the different modules of the DSS. The heart of the DSS is the 600 MHz C674x DSP core, which customers can program to run radar signal-processing algorithms. Two levels of memory (L1 and L2) are local to the DSP core. The L1 (32KB L1D, 32KB L1P) runs at the DSP clock rate of 600 Mhz. The L2 (256KB) runs at 300 Mhz (or half the DSP clock rate).

Assisting the DSP is a hardware accelerator (or HWA) that can take on common FMCW algorithms (FFT, log computation, CFAR-CA, and so forth), thereby, lightening the load on the DSP. The HWA has four local buffers (ACCEL_MEM0 … MEM3) each of 16KB each and is clocked at 200 Mhz.

Both the DSP and the HWA have access to external memories that include the ADC buffer, which presents the digitized radar IF signal to the DSS). The L3 memory (clocked at 200 Mhz and primarily used to store radar data) and a handshake RAM (that can be used to share data between the MSS and the DSS).

Finally, there are Enhanced-DMA (or EDMA [4]) engines that can be programmed by the DSP (or R4F) to efficiently transport data between these external memories, the DSP's L1 or L2, and the HWA's local memories.

Each of these components is described in more detail in the sub-sections below.

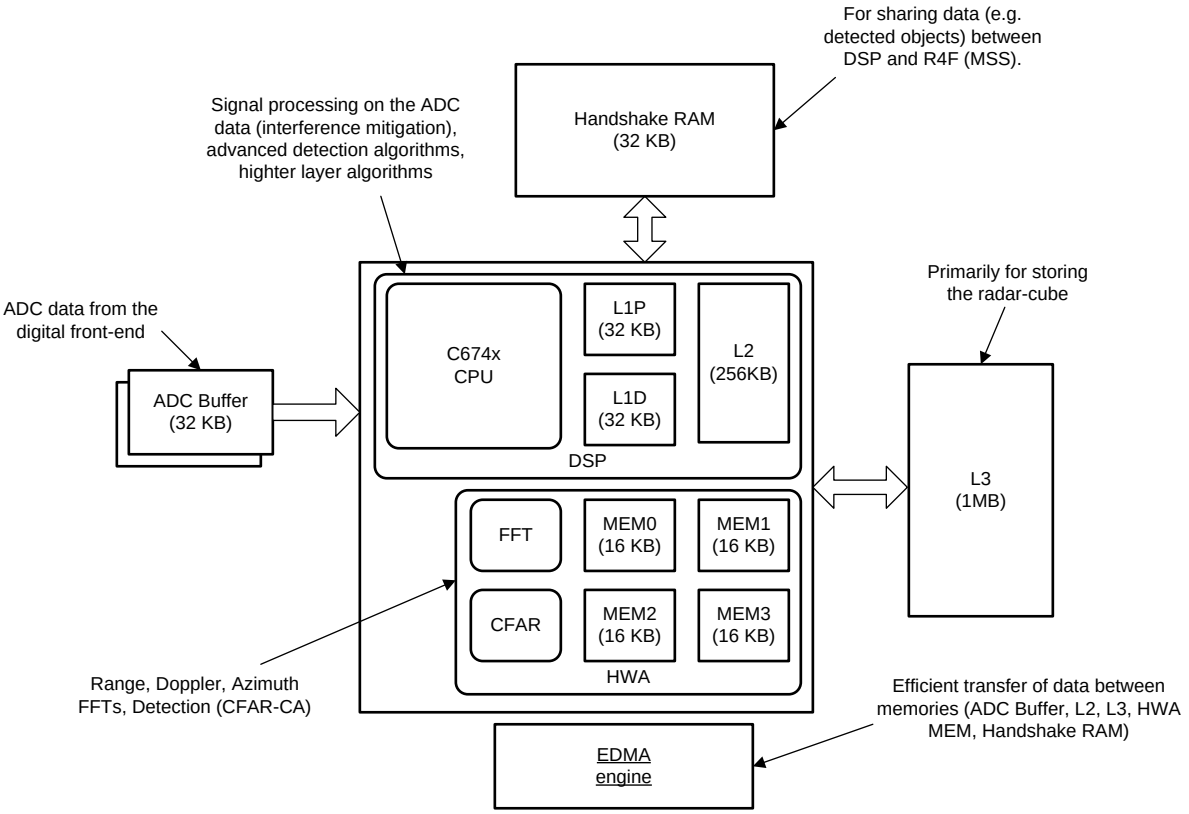

Figure 1. DSP Subsystem Overview

2.1 C674x DSP

The C674x DSP ([4] through [9]) belongs to the C6000™ family and combines the instruction sets of two prior DSPs: C64x+ (a fixed-point DSP) and C67x (a floating-point DSP). The DSP in the IWR6843 runs at 600 MHz. The DSP has the following key features:

- **Instruction parallelism**: The CPU has eight functional units that can operate in parallel, which allows for a maximum of eight instructions to be executed in a single cycle [4].
- **Rich instruction set**: In addition to instruction parallelism, the DSP supports a rich instruction set [4], which further speeds up signal processing computations. These instructions include single instruction multiple data (SIMD) instructions, which can operate on multiple sets of input data (for example, ADD2 can perform two 16-bit additions in one cycle and MAX2 can perform two 16-bit comparisons in one cycle). There are also other specialized instructions for commonly used signal-processing math operations (such as CMPRY, an instruction which is a single-cycle complex multiplication, specialized instructions for dot products, and so on).
- **Optimizing C-compiler**: The C6000 family of DSPs comes with a very good optimizing C compiler [7] [8], which lets engineers develop highly efficient signal-processing routines using linear C code. The compiler ensures optimal scheduling of instructions, to ensure the best possible use of the DSP parallelism. The compiler also supports the use of intrinsics [7], which lets the programmer directly access specialized CPU instructions from within the C code.

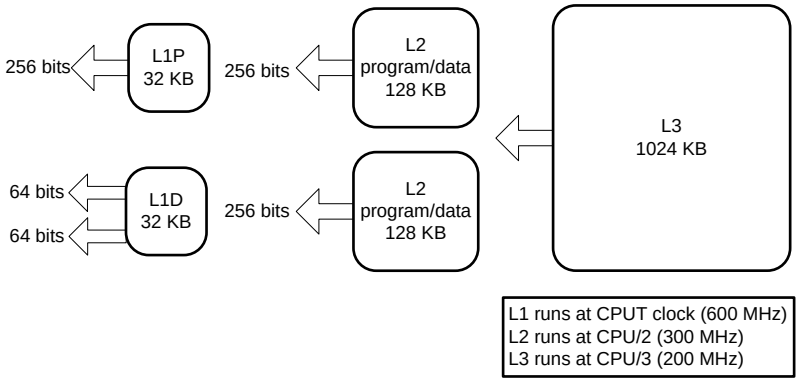

Figure 2. Memory Hierarchy in the DSS

- **Memory architecture**: The DSP uses a 3-level memory architecture (L1, L2, and L3) (see Figure 2) [6].
  - L1 memory is the closest to the CPU and runs at the CPU speed (600 MHz). L1 memory consists of 32KB each of program and data memory commonly referred to as L1P and L1D, respectively.
  - L2 memory consists of two 128KB-banks, which can store both program and data. Each bank is accessible through its own memory port, which improves efficiency by allowing access to both L2 banks simultaneously (for example, data and program simultaneously, if kept on separate banks). L2 runs at a speed of CPU/2 (300 MHz). L1 and L2 memories reside within the DSP core.
  - In addition, the DSP can also access an external memory (L3). In the IWR6843device, L3 memory runs at 200 MHz and is accessible through a 128-bit interconnect. Both L1 and L2 memories can be configured (partly or wholly) as cache for the next higher level memory [6]. A typical radar application might configure L1P and L1D as cache, place the code and data in L2, and use L3 primarily for storing the radar-data (as noted in Section 5 other variations are possible as well).
- **Collateral**: Besides the documents referenced above ([4] through [8]) , there is other collateral available in the form of optimized libraries. The TI C6000 DSPLIB [9] is an optimized DSP function library for C programmers. The library includes many C-callable, optimized, general-purpose, signal-processing routines and contains routines for fast Fourier transforms (FFTs), matrix and vector manipulation, filtering, and more. TI also provides an optimized library called the mmwaveLib (as part of the mmWaveSDK [10]), that complements the DSPLIB and includes additional signal processing routines that are commonly used in radar signal processing. Finally, signal processing chains demonstrating functionality are available as part of the mmWaveSDK.

2.2 Hardware Accelerator (HWA)

Radar processing flows mainly consists of repeated FFTs performed in order to resolve 'ADC data' in range (the range-FFT or 1D-FFT), velocity (velocity-FFT, or 2D-FFT) and angle (azimuth FFT or 3D-FFT). As such, if there was custom hardware to perform the FFT repeatedly, a significant proportion of the compute-load on the DSP could be relieved. Standardized algorithms like detection algorithms, log-computation, and so forth, also constitute a significant proportion of the DSP's load.

The HWA, after being programmed, can independently complete all of these procedures without needing processor supervision. It is clocked at 200 Mhz (or 1/3rd the speed of the DSP) but, for the algorithms it implements, its speed is comparable (or better) than the DSP. The following sub-sections provide a brief overview of the HWA. Additional information can be found in the HWA user's guide [11]. Relevant subsections of the user's guide are referred to in the next few sub-sections.

As shown in Figure 3, the HWA consists of four local memories (called Accelerator Local Memories), and an 'Accelerator Engine'. These sub-modules are discussed in detail below.

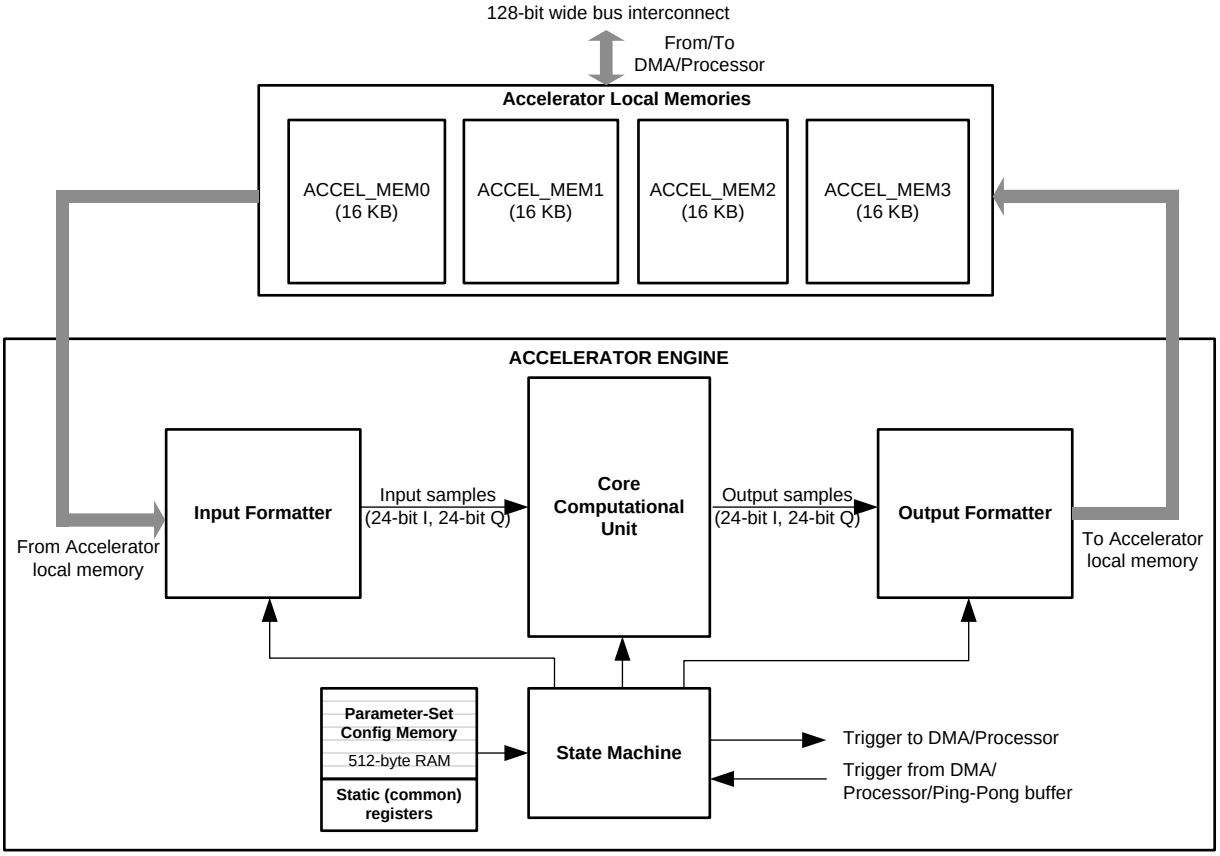

Figure 3. Hardware Accelerator Block Diagram

2.2.1 Accelerator Local Memories

These local memories (ACCEL_MEM0…3) are essentially 'scratch-pads' for the HWA. Data can be bought in from external memories (like the L3, or the DSP L2/L1, Handshake RAM, and so forth) into any one of these local memories, and taken out after processing is complete. Each 'local memory' is of size 16 KB, and can therefore store at most 4096 complex 16 bit-samples (or 2048 complex 32 bit samples).

Having four such local memories allows for the Ping-Pong operation of the HWA. That is, one pair of memories can serve as the input and output buffers for the HWA while the other two can be used to bring in new data and take out processed data. For example, while data is being filled into ACCEL_MEM1 (from L3 using EDMA), the HWA can read data from ACCEL_MEM0 and place the computed output in ACCEL_MEM2. Processed data on ACCEL_MEM3 can (in the same period) be taken out to L3 (again using an EDMA). Note that the same local memory cannot be used for bringing in data and for processing (or for taking the data out) at the same time.

Another point to note is that the HWA can (optionally) access the 'ADC Buffer' directly. In such a case, two of the local memories are disabled and in their place, the first half (16KB) of the ADC Buffer (both Ping and Pong) is directly readable. In other words, using this mode, removes the need to bring the data from the ADC Buffer to the Accelerator memories for performing the FFT. This comes in handy in range-FFT processing (see Section 4.2.1 for an example).

2.2.2 Accelerator Engine

The Accelerator Engine consists of a 'Core Computational Unit', an 'Input Formatter', an 'Output Formatter', and a State Machine. The State machine controls the other modules based on user-programmed registers called as 'Parameter-Sets' and 'static registers'.

The 'Core Computational Unit' is the computing heart of the HWA and can perform the following operations:

- FFT Computation with programmable FFT size (powers of 2, radix-2 stages) up to 1024-pt complex FFT. The internal bitwidth of each butterfly stage is 24bits (24-bit I and 24-it Q) guaranteeing good SQNR performance. Each stage also has programmable scaling by 2 after the butterfly computation. ('Core Computational Unit - FFT' Section in [11])
  - It also has built-in capabilities for pre-FFT processing – specifically, programmable windowing, basic interference zeroing-out based on a pre-programmed threshold and BPM removal.
  - Constant false alarm rate - cell averaging (CFAR-CA) including variants like CFAR-CASO (SO – smaller of) and CFAR-CAGO (GO – greater of). The detection inputs can be linear or logarithmic.
- Complex vector multiplication and Dot product capability for vectors up to 512 in size. (*Core Computational Unit - CFAR Engine* section in [11]).
- Magnitude (absolute value) and log-magnitude computation capability (*Core Computational Unit – Magnitude and Log-Magnitude Post-Processing* section in [11]).

The Core Computational Unit is designed as a streaming accelerator meaning that in steady state it can spit out one output sample per cycle [@200 MHz].

The Input Formatter and the output Formatter have two different functions:

- They exist to interface between the input (or output) data format and the internal HWA data format.

  For example, the input can be real (not complex), and either 16 or 32 bit, whereas, the internal HWA format is fixed to be 24-bit complex. In such cases, the 'Input Formatter' can be programmed to scale the real 16 bit (or 32 bit) input to the 24-bit complex data format. Similarly, the output can be either scaled up to 32 bit or down to 16 bit in the 'Output Formatter'.
- They can read data from (and write data to) the 'local memories' in either linear or transpose fashion. Unlike the DSP, or the EDMA, the transpose operation is essentially free (in the sense of cycles). This feature is discussed in Section 4, and is important for efficiently using the HWA.

The State Machine allows a programmer to chain and loop a sequence of accelerator operations one after another with no intervention from the main processor (the main processor being the R4F or the DSP). It is programmed by writing to two sets of registers: the parameter-set memories and static (common) registers.

- Parameters corresponding to a sequence of computations (say Doppler-processing, or Range-Processing) need to be programmed into a ‘parameter-set memory’ which is itself segmented into 16 'parameter-sets’. Each parameter-set is simply a collection of registers. The state machine reads a parameter-set and (from its contents) controls ‘the accelerator engine’. A single parameter set can be configured to do the following operations:
  - The selection of a core operation (say, a windowed-FFT with interference mitigation, or CFAR-CASO, or complex vector multiplication, and so forth).
  - The control of of the input and output formatters, including the scaling and access patterns. For example, Input/Output formatters can be used to extract (or place) constituent 1D-vectors (one after another) of an arbitrary 2D matrix existing in (or copied to) the local memories. Each subsequent call to a parameter set, can automatically update the indices of the vector that is extracted from (or emplaced to) the local memory.
  - The number of times a parameter set is to be used before moving to the next parameter set (also called the ‘BCNT’ parameter). For example, if a 256-pt FFT has to be performed 512 times, then the BCNT can be set to 511, and the same parameter-set will run 512 times.
  - Each parameter-set can also be stalled until or triggered by a certain condition (say a trigger from the DSP) is met.
- Static registers (or common registers) instruct the State machine on which is the ‘starting’ parameter set (out of the 16 available), the number of parameter sets to run and how many times (the number of loops) that subset of the parameter-sets is to be called.

The state machine then, essentially reads each Parameter-set, programs the accelerator, triggers the computation, waits until the computation specified by the parameter-set is complete (including the BCNTs), and then moves on to the next parameter-set as per the static registers’ value, until it finishes the final parameter-set.

NOTE: Both the EDMA and the HWA are controlled by what is called parameter-set registers (in the case of EDMA, they are also called PaRAMSet). However, the EDMA and the HWA are separate IPs, and there is no relation between the PaRAMSet for EDMA and the parameter-set for HWA. The registers names for certain parameters are similar between the HWA and EDMA, however, their use internally in their respective modules will be subtly different. For instance certain registers (SRCBCNT) in the EDMA are zero based and the HWA are 1-based.

2.2.3 Collatorals

The Hardware Accelerator User guide [11] covers the high-level architecture, key features such as windowing, FFT, and log-magnitude and also the CFAR-CA detector, complex multiplication. Additional collateral exists as part of the mmwaveSDK, which abstracts out the programming of the HWA with APIs. The use of these APIs are demonstrated in the out-of-box demo as part of the mmWaveSDK ([10]).

2.3 Enhanced-DMA (EDMA)

The EDMA engine handles direct memory transfers between various memories in IWR6843. In the context of radar signal processing, the amount of data to be stored per frame is typically much larger than what can be accommodated in the higher-level memories (such as L1 , L2 or the HWA local memories). Consequently, the bulk of the data is usually stored in a lower-level memory (L3). The EDMA enables data to be brought in from the lower-level memory to the higher-level memories for processing and subsequently shipped back to the lower-level memory. Proper use of the EDMA features (such as multidimensional transfers, chaining, and linking, which are explained next) can ensure that data transfers across memories occur with minimal overhead. This subsection provides a concise overview of the EDMA and its capabilities. For a more detailed description of the EDMA, see [4].

An EDMA engine consists of a channel controller (CC) and one or more transfer controllers (TCs). The CC schedules various data transfers, while the TC performs transfers dictated by requests submitted by the CC. The IWR6843 device has two CCs, each with two TCs. Because multiple TCs can operate in parallel, up to four transfers can be performed in parallel.

Parameters corresponding to memory transfers must be programmed into a parameter RAM (PaRAM) table. The PaRAM table is segmented into PaRAM sets. Each PaRAM set includes fields for parameters that define a DMA transfer, such as source and destination address, transfer counts, and configurations for triggering, linking, chaining, and indexing. Each EDMA engine (each CC) supports 64 logical channels, and the first 64 PaRAM sets are directly mapped to the 64 logical channels.

In the context of radar, the EDMA is used to extract (or emplace) matrices from/or to memories on the device. For example:

- From the ADC Buffer into L3
- From (or to) the L1/L2 DSP to (or from) the L3 or the HWA local memories.
- From (or to) the HWA local memories to (or from) the L3 or the DSP L1/L2 memories.

In most cases, these matrices correspond to ADC input or FFT outputs. They are typically from either one chirp or one range-gate and their dimensions are typically ‘ADC data’ x ‘number of receivers’, or ‘range-gate’ x ‘number of receivers’.

NOTE: Many matrices need to be read (or written) in ‘transpose’. In such cases, the samples of the matrices are not contiguous in memory. While ‘transpose read’ using the EDMA is 4x as slow as a ‘linear read/write’, ‘transpose write’ can be substantially slower (by approximately 16x). For the maximum throughput and efficient use of the device, care should be taken to minimize the overhead of EDMA reads and writes. For more details, see Section 4.

2.3.1 EDMA Features

The EDMA engine has a rich feature set, all of which can be programmed using the PaRAM sets. These features include the following:

- Multidimensional transfers: An EDMA data transfer is defined in terms of three dimensions.
  - The first dimension (A transfer) transfers a contiguous array of ACNT bytes from source to destination.
  - The second dimension (B transfer) transfers BCNT arrays (each of size ACNT bytes). Subsequent arrays in the B transfer are separated by SRCBIDX (DSTBIDX) bytes at the source (destination).
  - The third dimension transfers CCNT frames, with each frame consisting of BCNT arrays. Subsequent arrays are separated by SRCCIDX (DSTCIDX) bytes at the source (destination). The transfer counts (ACNT, BCNT, and CCNT) and indices (SRCBIDX and so on) are programmed in the PaRAM set.

NOTE: As stated in Section 2.2.2 the input/output formatter of the HWA are also capable of 2-dimensional transfers from and to the local memories of the HWA. The registers in the HWA’s parameter-sets reuse some of the register names of the EDMA, including, SRCACNT, BCNT, SRCBIDX, DSTBIDX, and so forth. They should not be mistaken for the registers (from the EDMA).

These multidimensional transfers allow considerable flexibility in defining the data streaming from the input to the output. For example, a matrix can be picked up from a source location and stored in a transpose fashion at the destination. This is shown in Figure 4, where a 4 × 5 matrix (each element 4 bytes in size) is transposed at the destination.

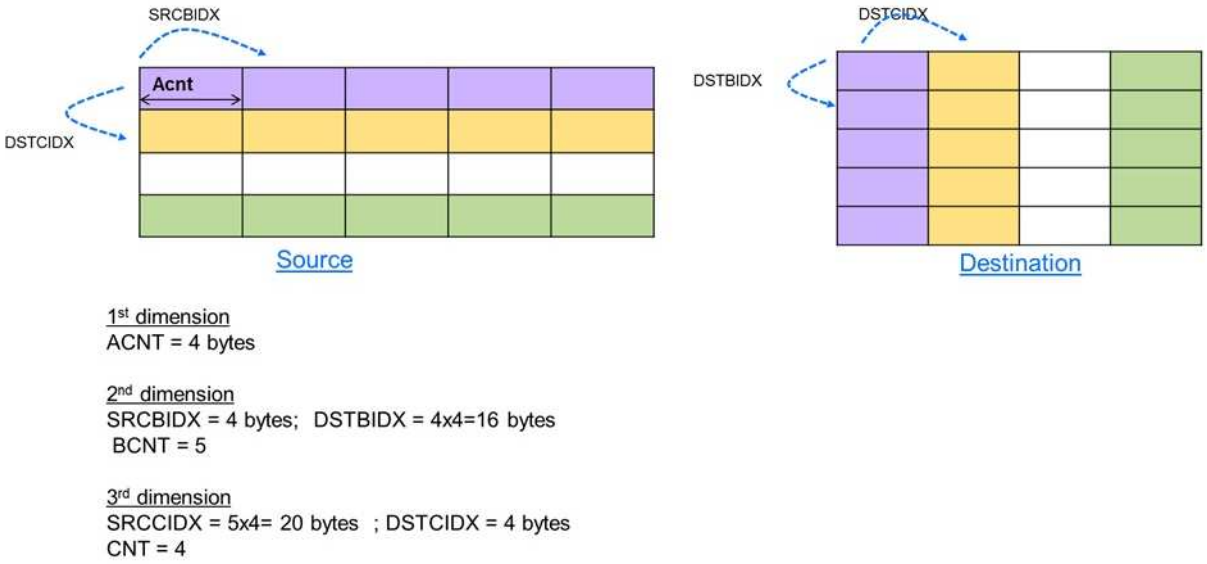

Figure 4. Multidimensional Transfer Capabilities of the EDMA

- Chaining: Chaining is a mechanism by which the completion of one EDMA transfer (corresponding to one EDMA channel) automatically triggers another EDMA channel. The chaining mechanism helps to orchestrate a sequence of DMA transfers.
- Linking: Once the transfers corresponding to a ParRAM set have been completed (and its counter fields such as ACNT, BCNT, and CCNT have decremented to 0), the EDMA provides a mechanism to load the next PaRAM set. The 16-bit parameter LINK (in each PaRAM set) specifies the byte address offset from which the next PaRAM set should be reloaded. Once reloaded, the PaRAM set is ready to be triggered again. The linking feature eliminates the need to reprogram the PaRAM set for subsequent transfers. It is also useful in programming the EDMA so that subsequent transfers alternate between ping-pong buffers. The first 64 PaRAM sets are directly mapped to the corresponding EDMA channels. The byte address offset programmed in LINK, is however, free to point to a PaRAM set beyond these first 64 (the two CCs in the support 128 and 256 entries, respectively, in their PaRAM tables).
- Triggering: Once an EDMA channel is programmed (using the PaRAM set), it can be triggered in multiple ways. These ways include:
  - Event-based triggering (for example, a *chirp available* interrupt from the ADC buffer, see Section 2.4.1)
  - Software triggering by the DSP CPU
  - Chain triggering by a prior completed EDMA

The previously listed features of the EDMA are ideally suited for performing data transfers in the context of radar signal processing. For example, radar processing involves multidimensional FFT processing, which requires data to be rearranged along various dimensions (range bins, Doppler bins, and antennas). The multidimensional transfers supported by the EDMA are very useful in this context. The PaRAM set-based programming model allows multiple transfers to be programmed up front. For example, EDMA transfers from the ADC buffer to the DSP core/HWA local memories, and between the DSP core/HWA local memories and the L3 memory can all be programmed up front in various logical channels. Subsequently, these channels are triggered during the data flow. Further, the linking feature allows automatic reloading of PaRAM sets (for example, for each successive frame), thus eliminating the need for the CPU to periodically reprogram the EDMA channels.

2.4 External Interfaces

2.4.1 ADC Buffer

Digitized samples from the digital front end (DFE) are stored in the ADC buffer. The ADC buffer exists as a ping-pong pair, each 32KB in size. So when the DFE is streaming data to the ping (respectively, pong) buffer, the DSP/HWA has access to the pong (respectively, ping) buffer. The number of chirps that can be stored in a buffer is programmable. When the DFE has streamed the programmed number of chirps into a buffer, a *chirp available* interrupt is raised and the DFE switches buffers. This interrupt can serve as a cue to the DSP/HWA to begin processing the stored data. Alternatively, the interrupt can also be used to trigger an EDMA transfer of the ADC samples to the local memory of the DSP/HWA (such as L1, L2, HWA local memories), with the EDMA subsequently interrupting the DSP/HWA on completion of the transfer.

In most cases the first 'signal processing' algorithm that is run on the ADC data is an FFT. The HWA can be configured to read the ADC buffer directly, and perform FFTs on its contents. In such a mode, two of the HWA's local memories are replaced by the ADC Buffer (ping and pong). The advantage of this configuration is that the latency associated with moving memory from the ADC buffer to the local memory is removed. There is an associated disadvantage though, the size of the ADC Buffer is halved from 32 kB to 16 kB.

The HWA can perform a host of pre-processing algorithms prior to the FFT, including single threshold 'interference mitigation' algorithm. This algorithm will zero-out any sample in the ADC data, that exceeds pre-programmed threshold value. However, this simple algorithm may not be sufficient.

Figure 5 (left side) shows an ADC buffer that has been programmed to store one chirp per ping-pong. The four colored rows refer to the ADC samples corresponding to the four antennas (the four RX channels). In Figure 5 (right side), the ADC buffer has been programmed to store four chirps per ping-pong. The data for each antenna (across the four chirps) is stored contiguously.

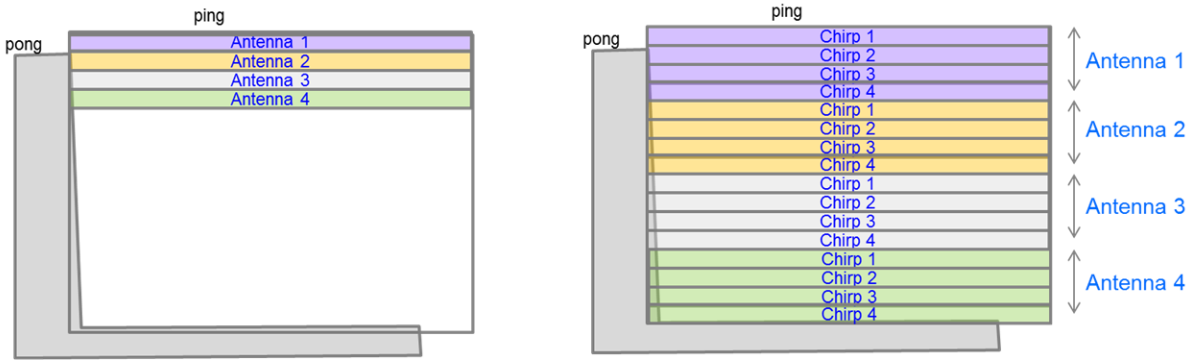

Figure 5. ADC Buffers

2.4.2 L3 Memory

In the IWR6843 device, up to 1MB of the L3 memory is available for use by the DSP or HWA. The primary use of the L3 memory is to store the radar-data arising from the ADC samples corresponding to a frame (*radar-cube*). The L3 Memory can be accessed via a 128-bit interconnect at speeds of up to 200 Mhz (or 16 contiguous bytes @ 200 Mhz in 'steady state').

2.4.3 Handshake Memory

An additional 32KB of memory is available at the same hierarchy level as that of L3 memory. The primary intent of a separate memory is to have another memory for sharing the data with other masters, such as master R4F or DMA in the MSS, without interfering with the L3 memory access efficiency. One prime use can be to share the final object list with the master R4F through this RAM.

3 Algorithm Chain and Benchmarks

This section provides benchmarks for some common radar signal-processing routines for both DSP and HWA. This section also examines the DSP loading in the context of some typical processing chains.

3.1 DSP Benchmarks

These benchmarks assume the data and program to be in L1.

Table 2. C674x Benchmarks

	Cycles	Timing (µs) at 600 MHz	Source and Function Name
128-point FFT (16 bit)	516	0.86 µs	DSPLIB (DSP_fft16x16)
256-point FFT (16 bit)	932	1.55 µs	DSPLIB (DSP_fft16x16)
128-point FFT (32 bit)	956	1.59 µs	DSPLIB (DSP_fft32x32)
Windowing (16 bit)	0.595 N + 70	0.37 µs (for N = 256)	mmwavelib (mmwavelib_windowing16x16)
Windowing (32 bit)	N + 67	0.32 µs (for N = 128)	mmwavelib (mmwavelib_windowing16x32)
Log2abs (16 bit)	1.8 N + 75	0.89 µs (for N = 256)	mmwavelib (mmwavelib_log2Abs16)
Log2abs (32 bit)	3.5 N + 68	0.86 µs (for N = 128)	mmwavelib (mmwavelib_log2Abs32)
CFAR-CA detection	3 N + 161	0.91 µs	mmwavelib (mmwavelib_cfarCadB)
Maximum of a vector of length 256	70	0.12 µs	DSPLIB (DSP_maxval)
Sum of complex vector of length 256 (16-bit I, Q)	169	0.28 µs	–
Multiply two complex vectors of length 256 (16-bit)	265	0.44 µs	–

3.2 HWA Benchmarks

The HWA is a streaming accelerator clocked at 200 Mhz. So, in steady state (once the initialization delays are accounted for), it can compute one output sample per cycle (per 5 ns) for any algorithm that it runs. Since the algorithm initialization delays are algorithm specific, a selection of benchmarks for common algorithms is provided.

Table 3. HWA Benchmarks

	Approximate Cycles	Timing (µs) (@200 MHz)	Notes
N point windowed FFT repeated k times	N*(k+1) + 1	3.2 µs (for N = 128, k = 4) 6.4 µs (for N = 256, k = 4)	An N-pt FFT in steady state, takes N cycles. The addition of windowing takes 1 extra cycle.
N sample Log-Magnitude	3 + N	1.25 µs (N = 256)	
N sample CFAR-CA (window size – w, guard length - g)	(w + g)*2 + 1 + N	1.4 µs (N = 256, w = 8, g = 4) .8 µs (N = 128, w = 8, g = 4)	

In the above benchmarks, it is important to note the time it takes for the data to be bought into the local memories of the HWA has not been taken into account. However, since the HWA has four local memories, it is possible to configure two such memories as ping memories (as an input buffer and an output buffer) and the remaining two as pong memories (similarly organized as input and output buffer). So, while the HWA processes data in the ping input buffer, and stores the result into the ping output buffer, data can be bought in via EDMA to the pong input buffer and processed data can be taken out through the pong output buffer. In such a case, the EDMA delays would only occur on the first run of each algorithm.

3.3 Radar Signal Processing Chain

The fundamental transmission unit of an FMCW radar is a frame, which consists of a number (for example, N_{chirp}) of equispaced chirps (see Figure 6). Central to FMCW radar signal processing is a series of three FFTs commonly called the range-FFT, Doppler-FFT, and angle-FFT, which respectively resolve objects in range, Doppler (relative velocity with regard to the radar), and angle of arrival. The range-FFT is performed across ADC samples for each chirp (one range-FFT for each RX antenna). The range-FFT is usually performed inline (during the intraframe period as the samples corresponding to each chirp become available). The Doppler-FFT operates across chirps and can only be performed when all the range-FFTs corresponding to all the chirps in a frame have become available. Lastly, the angle-FFTs are performed on the range-Doppler processed data across RX antennas. For more in-depth information on FMCW signal processing [12].

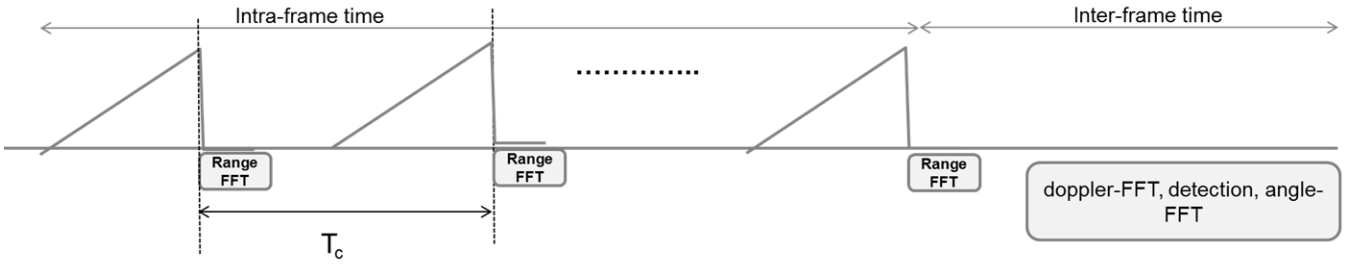

Figure 6. FMCW Frame Structure

Three similar signal processing chains are described that can be implemented on the IWR6843. The first maintains (by appropriately scaling the FFT) the bit width of the output at 16-bits even after the Doppler-FFT, the second allows the Doppler-FFT to grow to more than 16-bits (32-bit output) and one that uses a 32-bit floating point chain from the Doppler-FFT onwards to allow for increased dynamic range.

3.3.1 16-Bit Processing Chain

Figure 7 is a representative signal processing chain for FMCW radar. The processing is performed on each frame of received data. The ADC samples corresponding to each chirp (across the 4 RX antennas) is received in the ADC buffer. Colors indicate whether the processing should be performed on the DSP or the HWA (or either).

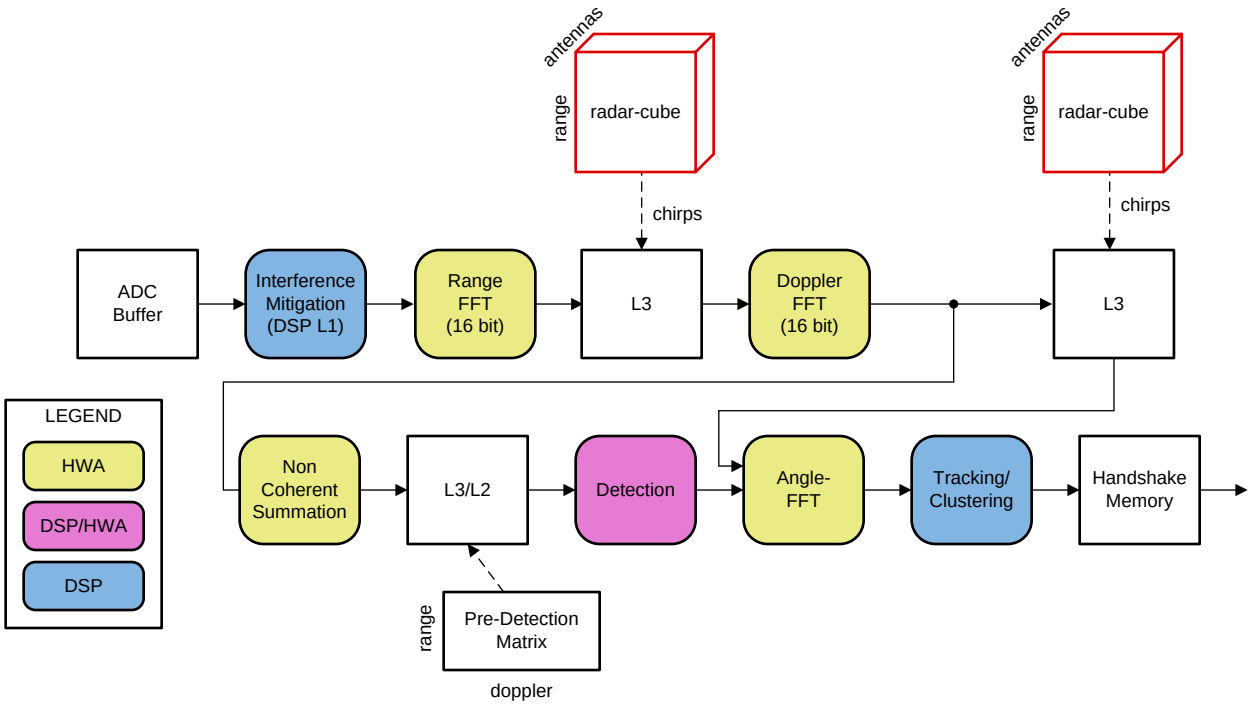

Figure 7. 16-Bit Processing Chain Data Flow

The first step on the ADC data of a chirp is typically an 'interference mitigation' algorithm. A simple version (zeroing out based on a fixed threshold) can be performed on the HWA. However, more complicated algorithms may be necessary and these need to be done on the DSP. Data is piped in (via an EDMA) from the ADC Buffer into the DSP's L1/L2, where this algorithm is performed. The output of the algorithm is then sent (via another EDMA) to the HWA. The HWA performs a sequence of FFTs (one for each RX antenna) and the range-FFT output is then placed in L3 memory. The data in L3 memory is best visualized as a 3 dimensional matrix indexed along range, chirps and antennas (the 'radar-cube').

Once all the range-FFTs have been computed for all the chirps in a frame, Doppler-FFT's are performed. For each range-bin (and for each antenna) a Doppler-FFT is performed across the N_{chirp} chirps. In the example of Figure 7 both the range and Doppler-FFT's are 16-bit FFTs. Hence the results of the Doppler-FFT can be stored in-place (overwriting the corresponding range-FFT samples) in the radar-cube in L3 memory. At the end of the Doppler-FFT processing, the radar-cube is now indexed along range, Doppler, and antennas. Note that FFT's processing gain can result in the output of the Doppler-FFT being more than 16 bit (leading to saturation).

Care should be taken either by configuring the HWA FFT hardware's internal scaling, or by scaling after the FFT (in the output formatter), to keep the output within 16 bit. There is no automatic scaling in the HWA FFT.

Immediately after the doppler-FFT for a particular range-gate (while the output is still in the local memory), the HWA does additional processing. It creates a 'pre-detection array' by non-coherently summing the doppler-FFT output along the antenna dimension. This is then stored back in L3 in an array that is indexed along range and doppler. The pre-detection matrix is much smaller than the radar-cube. In the example above, the pre-detection matrix would be 1/8th the size of the radar-cube: the reduction due to the collapsing along the antenna dimension contributing a factor of 1/4th and the fact that the pre-detection matrix is real, while the radar-cube is complex contributing a factor of 1/2. Consequently, pre-detection matrix can be stored either in L3 or L2.

One more algorithm can be run on the 'pre-detection array' (again, while it is still in the HWA local memory) – a detection algorithm along the doppler direction. This algorithm identifies bins that correspond to the valid objects. If the detection algorithm is one of the supported variants of CFAR-CA then it can be performed by the HWA. The output of the algorithm is a list of detected objects per range-gate.

If the DSP has to perform the detection algorithm, it is better to let the HWA complete the doppler-processing of all the range-gates and then, interrupt the DSP on completion. The DSP can then pipe in the pre-detection matrix and perform detection on it.

Finally, once the doppler-processing is complete, a number of other algorithms need to be run. A final detection algorithm (in the range-dimension) performed on the 'pre-detection matrix' at certain indices to create a final list of detected objects. These indices correspond to the detection output in the doppler-processing. Such an algorithm is better performed on the DSP, since it only has to be done on a few indices, and not on the entire pre-detection matrix.

For each such object, an angle-FFT (on the HWA) is then performed on the corresponding range-Doppler bins in the radar-cube to identify the angle of arrival of that object.

Finally the estimated parameters for the detected objects such as range, doppler and angle of arrival can be shared with the MSS via the hand-shake memory. There is also an option to perform further processing such as clustering,tracking and classification in the DSP itself.

Since the DSP is not likely to be used during the intraframe period and during much of the interframe processing period, it should have sufficient MIPs to perform these algorithms. In fact, it can continue to do tracking/classification in a lower priority thread, while a higher priority thread is used to control and finish the intraframe processing.

Alternatively these higher layer algorithms can be performed in the MSS (the R4F MCU).

3.3.1.1 Memory and Computational Requirements

The bulk of the memory and computational requirements need to be computer for a representative FMCW frame structure using the processing flow indicated in Figure 7. A frame with 128 chirps (N_{chirp}=128) was considered, with each chirp having 256 samples (N_{adc} = 256). With 4 RX antennas, the radar-cube memory requirements is 256 samples x 128 chirps x 4 antennas x 4 bytes/sample= 512KB. There are processing requirements during two distinct phases: intra-frame processing and inter-frame-processing.

The intraframe period (or the active-period) is the period during which chirps of one frame are being generated, transmitted and received. Once all chirps of a frame have been transmitted, the inter-frame period is entered during which the RF is idling. The recommended ratio of the active period to the inter-frame period is 50%.

- Intraframe processing: Each chirp, once completed, will consist of up to 4 Rx's worth of ADC data. From Table 3, a sequence of 4 256-pt range-FFTs with windowing takes 6.4 µs. Note that if the FFT were performed on the DSP, the same computation would take ~ 7.7 µs (Because each windowing and FFT operation takes (1.55+0.37) = 1.92 µs (from Table 2). Thus for 4 RX antennas it would take 1.92 x 4 = 7.7 µs). Since, typical chirp periodicities (T_c in Figure 6) in are in the order of 10 µs, range-FFT processing can be comfortably performed in the time (T_c) between the arrival of ADC samples from consecutive chirps. For example, with a sampling rate of 10 MHz, and N_{adc} = 256, T_c would be at least 256/10 = 25.6 µs (excluding the interchirp idle time of a few us).

  The range-FFT output is then stored in L3 to be fetched back for performing the Doppler-FFT. Timing considerations for this processing is covered in Section 4.
- Interframe processing: The first step in interframe processing is the Doppler-FFT. A 128-pt Doppler-FFT needs to be performed for each of the 256 range-gates and across the 4 RX antennas. Again from Table 3, 4 128-pt Doppler-FFTs takes 3.2 µs. Each doppler-FFT can be immediately followed by the log-magnitude computation (of the 4 FFT outputs) which would take approx. 2.6 µs , this can be followed by a computation of the log-magnitude sum (again 2.6 µs). Since the sum would be present in the local memory, CFAR-CA in the doppler dimension can also be done. This would take ~ 0.8 us. The cumulative time taken for these operations is 9.2 µs.These four operations have to be repeated 256 times, taking approximately 2.4 ms (256 x 9.2 ms) for the full inter-frame processing.

  The operation would result in a pre-detection matrix of 256 x 128 samples, each sample being a 16-bit real (positive) number. The first dimension corresponds to range, and the 2nd dimension to Doppler.

NOTE: Note that this computation ignores further downstream processing that is performed on the detected objects such as additional detection steps to reconfirm initial detections, angle-estimation, SNR computation, clustering, tracking, and so forth

Also the timing computation assumes that there is no delay in the memory access. Concerns relating to memory access and data flow are discussed in Section 4.

3.3.2 32-Bit Processing Chain

In the chain described in Figure 7, the Doppler-FFT processing did not increase the size of the radar-cube (allowing the Doppler-FFT output to be written over the range-FFT). There could be cases where performance requirements might dictate a higher dynamic range at the output of the Doppler-FFT. For more information, see the discussion in Section 3.4. Figure 8 depicts a possible processing chain in such a scenario. While the range-FFT is still 16-bit, the Doppler-FFT has an increased dynamic range and outputs 32-bit complex samples.

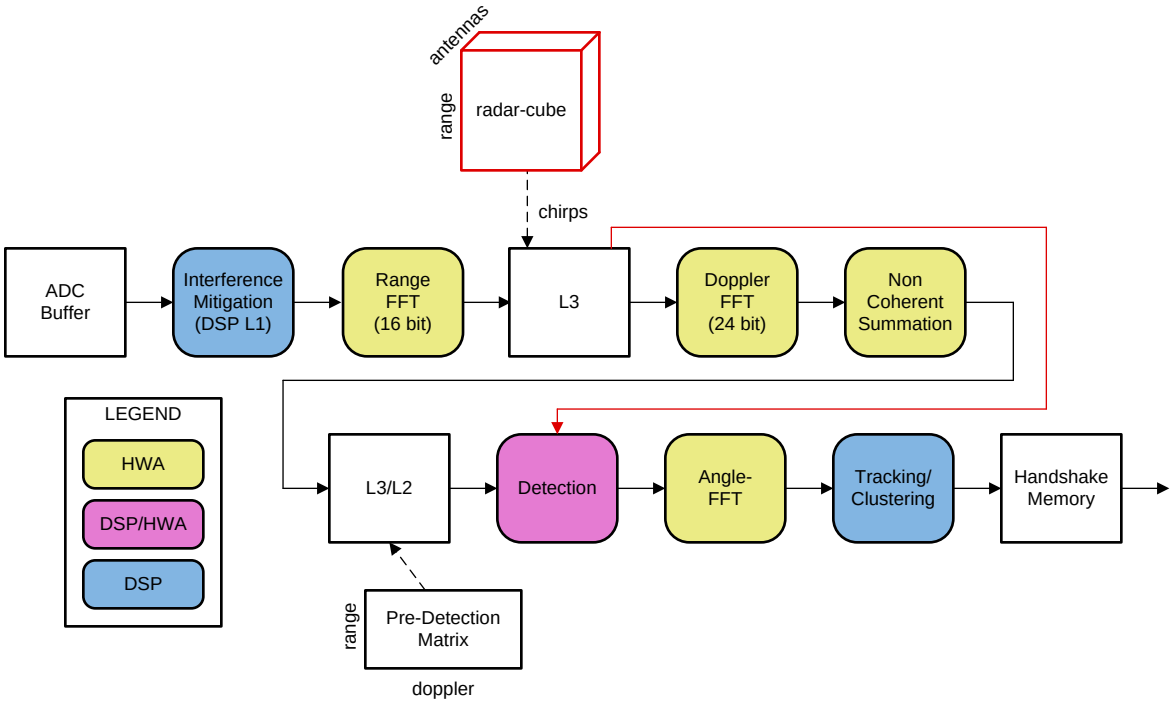

Figure 8. 32-Bit Processing Chain Data Flow

While these samples could be stored back in the radar-cube it would mean a doubling of the radar-cube size in L3. Hence, the chain in Figure 8 follows a different approach. Once the Doppler-FFT is computed, it is non-coherently summed up across antennas to create a pre-detection matrix. This result is not stored back in L3. Once the detection algorithm is run, and objects have been identified, a Doppler-DFT (followed by the angle-FFT) is selectively recomputed only for the detected objects.

In most cases, using 32-bit FFT is overkill, and the 24-bit HWA FFT (whose performance is shown in Section 3.4.1) would suffice. Especially since the HWA FFT allows programmable scaling at the output of each butterfly, preventing overflow. However, if a 32-bit FFT is necessary then it has to be performed on the DSP. 32-bit windowed FFT on the DSP is more than 2x slower than the 24-pt FFT on the HWA. More specifically, for four 128-pt FFTs it would take (1.59 + 0.32)*4 or 7.6 µs, whereas it would take only 3.2 µs on the HWA.

3.3.2.1 Memory and Computational Requirements

The requirements for memory are exactly the same as the 16-bit processing chain, since only the 16-bit range-FFT output is stored. Hence for a frame with 128 chirps (N_{chirp} = 128), with each chirp having 256 samples (N_{adc} = 256) and with 4 RX antennas, the radar-cube memory requirements is 256 samples x 128 chirps x 4 antennas x 4 bytes/sample= 512KB.

- Intraframe processing:Intra-chirp processing would also take the same number of cycles as 16-bit processing chain (6.4us), since the HWA would be used for the 1D-FFT.
- Interframe processing:If the HWA is used, the cycle counts would remain unchanged from the 16-bit processing. If the DSP is solely used for the inter-frame processing, then you need to use the DSP benchmarks to compute the time taken. For a 128-pt 32-bit Doppler-FFT (performed for each of the 256 range-bins and across the 4 RX antennas), the time taken would be (1.59 µs + 0.32 µs + 0.86 +0.28) x 256 x 4 or 3.2 ms. (The time taken for a 32-bit FFT and 32-bit windowing and 32-bit log magnitude are taken from Table 1). The output of this collective set of functions would be the pre-detection matrix of 256 x 128 samples, each sample being a 16-bit real (positive) number. The first dimension corresponds to range, and the 2nd dimension to Doppler. Subsequently, a detection algorithm (say CFAR-CA) is run on the matrix. Running this detector along each of the 256 x 4 Doppler lines would take 0.9 µs x 256 x 4 ≈ 1 ms

Hence the core computation blocks in 32-bit inter-frame processing would collectively take about 4.2 ms.

3.3.3 Floating-Point Processing Chain

In the processing chains, the low-level processing chain (up to the angle-FFT) uses fixed-point arithmetic. (Subsequent higher layer processing, such as clustering and tracking, is typically done in floating point.) However, it is also possible to construct a MIPS-efficient chain where all the processing starting from the Doppler-FFT is in floating point as explained below. In such a case, the HWA cannot be used as it has no floating point capability.

C674x DSP provides a set of floating-point instructions that can accomplish addition, subtraction, multiplication and conversion between 32-bit fixed point and floating point, in single cycle for single-precision floating point, and in one to two cycles for double precision floating-point. There are also fast instructions to calculate reciprocal and reciprocal square root in single cycle with 8-bit precision. With one a more iterations of Newton-Raphson interpolation, one can achieve higher precision in 10 to couple of 10s of cycles.

Another advantage of using floating-point arithmetic is that users can maintain both precision and dynamic range of the input and output signal without spending cycles rescaling intermediate computation results, and enable them to skip or do less re-qualification of the fixed-point implementation of an algorithm, which makes algorithm porting easier and faster.

3.3.3.1 Memory and Computational Requirements

Floating point FFTs in the DSP are almost as efficient as a 32-bit fixed point FFT (see Table 2, again more than twice as slow as the HWA FFT). Hence, the times computed in the 32-bit processing chain (Inter-frame processing taking approximately 4.3 ms) remain roughly valid.

3.4 FFT Performance

FFT processing forms a significant part of the lower level radar signal processing in FMCW radar.

3.4.1 HWA FFT Performance

Details of the HWA's FFT can be found in [11]. Summarized, the HWA's FFT takes a 24-bit complex input (24-bit real, 24-bit imag) and produces a 24-bit complex output. It is a pipelined FFT consisting of up to 10 radix 2 stages and is capable of a sustained throughput of 1 complex sample per cycle (5 ns).

Since the FFT is fixed point, and the pipeline bitwidth is a fixed 24-bits, the processing gain of each butterfly stage should be taken into account. At the output of each radix-2 stage of the FFT, bit-growth by one bit can occur. To prevent overflow, the output of each butterfly can be divided by 2 (round off one LSB) or saturated to 24 bits. There is no auto-scaling, and the stages at which division/saturation is to occur have to be decided up front.

Applying a window to the input prior to the FFT happens in line, meaning that the cycle cost of windowing is only 5 ns (no matter the number of FFTs being performed per parameter set).

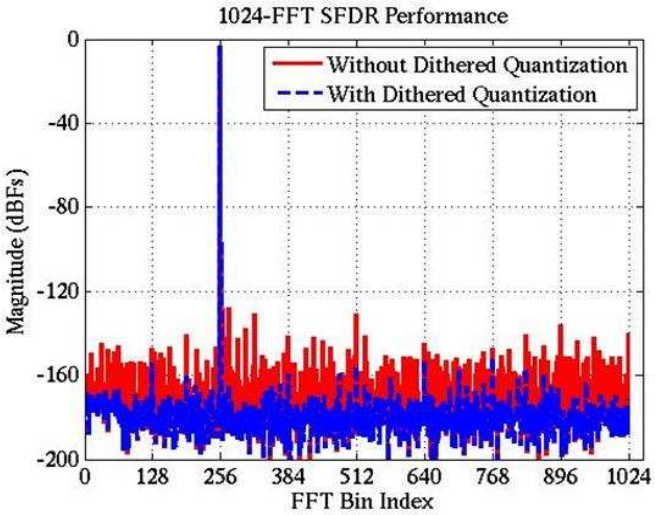

Figure 9. FFT SFDR Performance With and Without Dithering

The SFDR performance of the HWA's FFT can be better than approximately 150 dBc.

3.4.2 DSP FFT performance

It is recommended that all FFTs be performed using the HWA. However, in certain cases, it may be necessary for the DSP to perform FFTs. For example, it can happen that the HWA is fully loaded (performing the basic processing for the current frame), and the DSP is idle and can perform angle-FFTs for the previous frame. Efficient DSP-FFT routines for such cases are provided in the DSPLIB . The following section describes the different DSP-FFT implementations (both fixed and floating point variants) that have been specifically optimized for the C674x DSP.

The fastest routine is the DSP_fft16 x 16(), which takes 0.86 µs (128 point FFT) and 1.55 µs (256 point FFT) on a C674x running at 600 MHz. It operates on a 16 bit complex input to produce a 16 bit complex output. It accesses a precomputed twiddle factor table which is also stored as 16 bit complex numbers. This FFT uses multiple radix-4 butterfly stages, with the last stage being either a radix-2 or radix-4 depending on whether the FFT length is an odd or even power of 2. Note that every radix-4 stage (except the last stage) has a scaling by 2. There is no scaling in the last stage. So there is a bit-growth (worst case) of 1 in every (radix-4) stage preceding the last stage. The last stage has a bit-growth of 1 or 2 depending on whether it is a radix-2 or radix-4. Thus for a 128 point FFT this routine has a bit growth of 4 (=1+1+1+1) and can handle a 12 bit signed input with no clipping; resulting (for tone at the input) in a full scale peak at the FFT output.

In most radar applications, the 16 × 16 FFT routines previously described suffices for the first dimension (range) processing. However, subsequent FFT processing stages could involve bit growth that extends beyond 16 bits. The use of 32-bit FFT routines available in DSPLIB allows these subsequent stages to realize their full processing gain without being limited by either bit overflow or the quantization noise of the FFT routine. As an example, DSP_fft32x32() works on 32-bit complex inputs producing 32-bit complex outputs, and uses a precomputed twiddle factor table also stored as 32-bit complex numbers.

A list of some key FFT routines available with DSPLIB are provided in Table 4. Additionally, the mmwave library [10] also has some useful associated routines such as windowing.

Table 4. List of FFT Routines in DSPLIB

Function Name	Description
DSP_fft16x16	Fixed-point FFT using 16-bit complex numbers for input and output (16-bit I and 16-bit Q). The twiddle factor table is also stored as 16-bit complex. There is a scaling by 2 after every radix-4 stage (except the last stage which can be either a radix 4 or a radix 2). The minimum length of the FFT is 16. All buffers are assumed to be 8-byte aligned. The FFT works for input lengths which are powers of 2 or 4.
DSP_fft16x32	Fixed-point FFT using 16-bit complex numbers for input and output (16-bit I and 16-bit Q). The twiddle factor table is stored as 32-bit complex. There is a scaling by 2 after every radix-4 stage (except the last stage which can be either a radix 4 or a radix 2). The minimum length of the FFT is 16. All buffers are assumed to be 8-byte aligned. The FFT works for input lengths which are powers of 2 or 4.
DSP_fft32x32	Fixed-point FFT using 32-bit complex numbers for input and output (32-bit I and 32-bit Q). The twiddle factor table is also stored as 32-bit complex. There is no scaling in this FFT. The minimum length of the FFT is 16. All buffers are assumed to be 8-byte aligned. The FFT works for input lengths which are powers of 2 or 4.
DSPF_sp_fftSPxSP	This FFT uses complex floating-point input and output. The twiddle factors are also in floating point. There is no scaling in this FFT. The minimum length of the FFT is 16. All buffers are assumed to be 8-byte aligned. The FFT works for input lengths which are powers of 2 or 4.

Table 5. MIPS (Cycles) Performance of FFTs

Function Name	N = 32	N = 64	N = 128	N = 256	N = 512	N = 1024	N = 2048
DSP_fft16x16	160	240	516	932	2168	4216	9868
DSP_fft16x32	241	369	816	1488	3503	6831	16078
DSP_fft32x32	261	413	956	1772	4267	8363	19914
DSPF_sp_fftSPxSP	305	473	1066	1962	4683	9163	21740

Ensure that the quantization noise coming from the FFT processing does not limit the output SNR. The SQNR performance metrics (in dBFS) for various FFT routines has been listed in Table 5. For input data with effective bit width of B bits, the SNR at the input is Bx6+1.76dB. The ideal SNR after an N point 1st dimension FFT will be Bx6+1.76+10log10(N). If B=10 bits and N=128 the ideal output SNR is less than 83dB. Consequently the use of DSP_fft16x16 (which, from Table 6 has an SQNR of 85 dB) will not limit the output SNR. Similarly, depending on the input SNR and the FFT length, other options such as DSP_fft32x32() can be considered. For optimal performance the input back off should be appropriately chosen to prevent clipping at the output. For example when using DSP_fft16x16Scaled an input back-off of -12dBFS is ideal.

Table 6. SQNR (dB) Performance of FFTs

Function Name	N = 32	N = 64	N = 128	N = 256	N = 512	N = 1024	N = 2048
DSP_fft16x16	89	87.5	88.3	85.3	87	84.2	86.3
DSP_fft16x32	101.3	105.4	104.4	108.7	108.7	112.7	113.4
DSP_fft32x32	175.7	173	169.9	167	164.1	161.2	158.2

Note that the 32-bit FFT routine provide plenty of room for bit-width growth. A useful technique, when using this routine is to left shift/scale the input by a few bits. The “free” lower bits thus obtained ensure that the quantization noise of subsequent FFT stages does not limit the processing gain.

3.5 Recommended Division of Labor Between the DSP and HWA.

The DSP, being a general purpose processor, can implement the algorithms of the HWA. It can do efficient FFTs, CFAR-CA detectors, complex vector multiplications, and so forth. This brings us to a common question, what should be the division of labor between the DSP and the HWA?

The answer would depend on factors like the MIPS necessary for the application (Is the DSP sufficient? If so, use it) and the specifications of the algorithm (Does it need floating point operations? Can it be implemented on the HWA?) and so forth.

Some other factors to keep in mind when dividing algorithms between the HWA and DSP are:

- Repetitive standardized algorithms like FFTs are best performed on the HWA, since the entire processing can be completed without any interruption from the processor. The DSP is then better used for other higher level algorithms during this period – like tracking, classification, and so forth.
- Specialized (and proprietary) algorithms used for detection (other than CFAR-CA), histogram computation, MUSIC, interference mitigation using multiple thresholds, and so forth can only be performed on the DSP.
- The HWA is a fixed-point accelerator with a fixed 24-bit internal bitwidth. Computations that need 32-bit precision need to use the DSP. Likewise all floating point operations need the DSP.
- Certain arithmetic operations (like complex vector multiplication) may actually be faster on the DSP, since it is running at 3x the clock of the HWA, and can perform one complex multiplication per cyclen.

4 Data Flow

While Section 3 focused on algorithmic benchmarks, this section focuses on data flows. A typical radar signal-processing chain requires data transfer from the ADC buffer to the DSP local memory (L1 or L2), and to and fro between the DSP local memory and L3. These data transfers are primarily accomplished using the EDMA. It is important to understand the various types of EDMA transfers and their associated latencies. This will enable the stitching together of a data flow that has minimal overhead and is as nonintrusive as possible.

4.1 Types of EDMA Transfers

Three kinds of EDMA transfers are relevant in the context of FMCW radar signal processing.

- Contiguous-Read Contiguous-Write (or *contiguous*): This is the simplest and fastest kind of data transfer, which involves moving a portion of memory from a source to a destination with no data rearrangement involved during the transfer. This transfer is accomplished as a single-dimensional transfer (with ACNT specifying the number of bytes to be transferred). The speed of this transfer is 128 bits per cycle (at 200 MHz). In a typical radar signal-processing chain where each data element is a 32-bit complex number (16-bit I and 16-bit Q), this amounts to four samples being transferred in every cycle. Thus a transfer of 256 complex samples would take about 0.32 µs. The transfer of data from the ADC buffer to the L2 memory of the DSPs before range-FFT processing would be an example of a contiguous transfer. Another example would be the transfer of the output of interference-mitigation to one of the HWA local memories.

- Contiguous-Read Transpose-Write (or *transpose-write*): A vector that is contiguously spaced at the source must be placed in a transpose fashion at the destination. This is accomplished using a 2-dimensional transfer with ACNT representing the size of each sample and BCNT representing the number of samples. Assuming that the sample size ACNT <128 bits, the transfer takes four cycles (at 200 MHz) per sample. As a result, it would take about 5.12 µs to transfer 256-contiguous complex samples using a transpose-write. Figure 10 is an example of such a transfer where the range-FFT output (corresponding to the four RX channels of a chirp) is placed in a transpose fashion in L3 memory. Such a transpose-write ensures that the data on which the Doppler-FFT operates is available in a contiguous fashion at the end of a frame (as shown in the blue inset).

  Techniques exist which can increase the effective speed of the transpose-write transfer. One such technique is to parallelize the transfer by employing multiple TCs. In the example of Figure 10, range-FFTs for channels 1 and 2 could be transferred using one TC, while range-FFTs for channels 3 and 4 could be transferred using another TC. This would speed up the transfer by a factor 2. This technique works because the latencies for such transfers arise primarily within the TC and not at the memory interfaces of the source or destination.

NOTE: For all depictions of memory in this document, the memory address is assumed to be contiguous along the rows of the array that represents the memory.

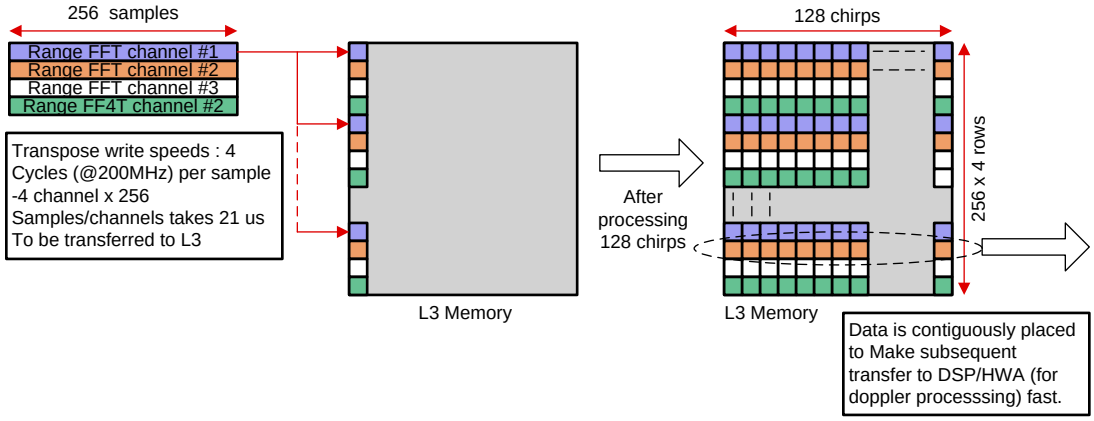

Figure 10. Contiguous-Read Transpose-Write EDMA Access

- Transpose-Read Contiguous-Write (or *transpose-read*): Data that is noncontiguously placed at the source must be contiguously placed at the destination. As in the earlier case, this is accomplished using a 2-dimensional transfer with ACNT representing the size of each sample and BCNT representing the number of samples. Assuming that the sample size ACNT <128 bits, the transfer takes 1 cycle (at 200 MHz) per sample. As a result, it would take about 1.28 µs to transfer 256 complex samples. In the context of radar signal processing, such a transfer would be needed if range-FFT data across chirps in a frame had been contiguously stored in L3, as shown in Figure 11. A transpose-read access would be needed to transfer the data corresponding to a specific range bin (and a specific antenna) to the local memory of the DSP to perform the Doppler-FFT.

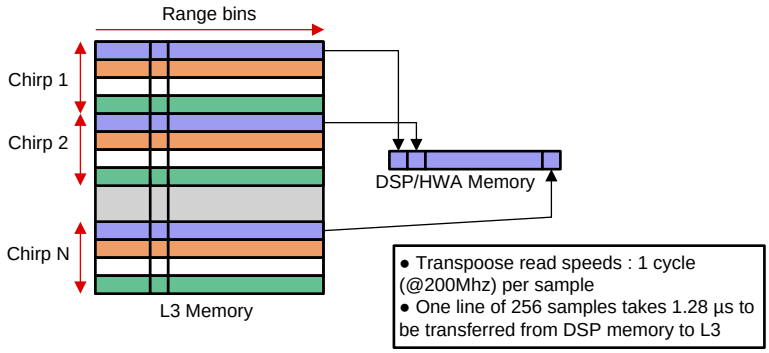

Figure 11. Transpose-Read Contiguous-Write EDMA Access

Table 7. 'steady state' EDMA Transfer Costs (in cycles)

Read Mode	Write Mode	ACNT Bytes	Bits/Cycles	Time for 1024 complex samples (us)
Contiguous	Contiguous	x	128 bits/cycle	1.28 µs
Contiguous	Transpose	4	32 bits/4 cycles	20.48 µs
		16	128 bits/4 cycle	5.12 µs
Transpose	Contiguous	4	32 bits/ cycle	5.12 µs
		16	128 bits/cycle	1.28 µs

Finally, Table 7 summarizes the transfer costs when using EDMA. Note that these are best case numbers, and in general, the. When designing the EDMA transfer sequences be aware that arbitration (when multiple TPTC vie for the same bus) can introduces a significant increase in the transfer costs. This application report will be updated with more accurate numbers at a later date.

4.2 Example Use Cases: Intraframe Processing (Range-FFT Processing)

The use of the EDMA during range-FFT processing is demonstrated using a few illustrative examples (by no means is this an exhaustive set of examples).

4.2.1 Single-Chirp Use Case

Single chirp refers to the use case where the ADC buffer has been programmed to issue an interrupt after data for each chirp (across four RX channels) becomes available. In this diagram no pre-processing is performed in the DSP prior to the range-FFT, and so the HWA can be configured to allow direct access to the ADC Buffer. There is no need to pipe-in data to the accelerator via an EDMA.

NOTE: If some pre-processing needs to be performed on the DSP, then an EDMA would have to be configured to transfer the ADC buffer contents to the DSP. The transfer to the DSP's L2 would be contiguous and hence fast. For example, transferring 256 ADC samples across 4 RX channels would take 256 x 4/4/200 ≈ 1.28 µs. Once the pre-processing is complete, another EDMA can be triggered to transfer the ADC data to the local memories of the HWA for FFT, which would take another 1.28 µs

The FFT output is then transferred to the radar-cube memory in L3 via an EDMA. In this example transfer from HWA's 'local memory' to L3 is a transpose-write and hence is slower (by a factor of 16 compared to a contiguous transfer). The transfer of the 256 x 4 range-FFT samples would thus take about 21 µs.

NOTE: The total latency of this DMA transfer is in many cases less than the chirp period T_c and hence shouldn't be a bottle-neck (for a typical 5 MHz sampling rate with N_{adc} = 256, T_c will be 51.2 µs). However, since the AWR1843 allows sampling rates as high as 12.5 Mhz (which implies that for N_{adc} = 256, T_c = 20 µs), there will not be enough time to do a transpose write (which would take 21 µs). Care should be taken so that the bottle-neck is accounted for.

The advantage of this configuration of the EDMA is that it eases the data transfer latencies during the Doppler-processing. Since the range-FFT output is written out in a transpose manner, the data required for a Doppler-FFT would be contiguously placed. Therefore, when the Doppler-FFT outputs need to be written back (see Section 3.3.1), the write is contiguous and fast.

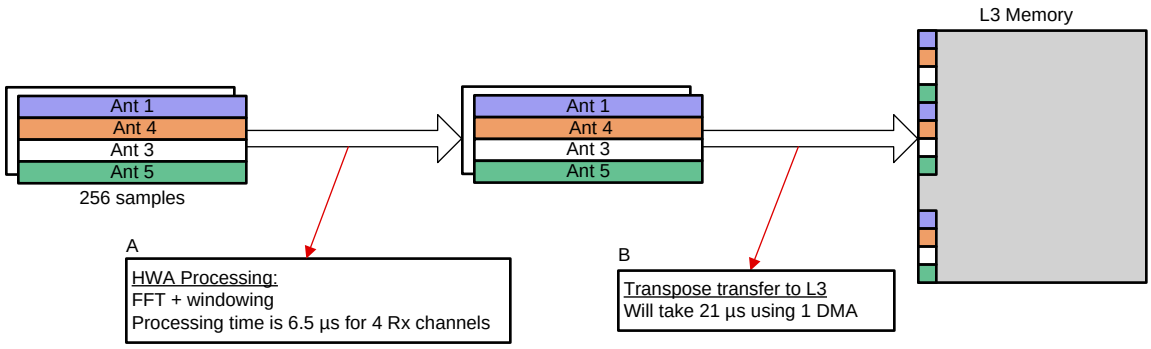

Figure 12. Single-Chirp Use Case

In Figure 13, the buffer management scheme is shown for the typical use-case. While the ADC Buffer's pong buffer is being filled with ADC data, the ping buffers contents are being processed by the HWA and stored in one of the local memories (MEM2). During the same period previous chirp's output (stored in MEM3) is being transferred out to L3 via an EDMA. Since the HWA was configured such that the ADC buffers (ping and pong) can be directly accessed, there is no need to move ADC data to HWA's internal buffers.

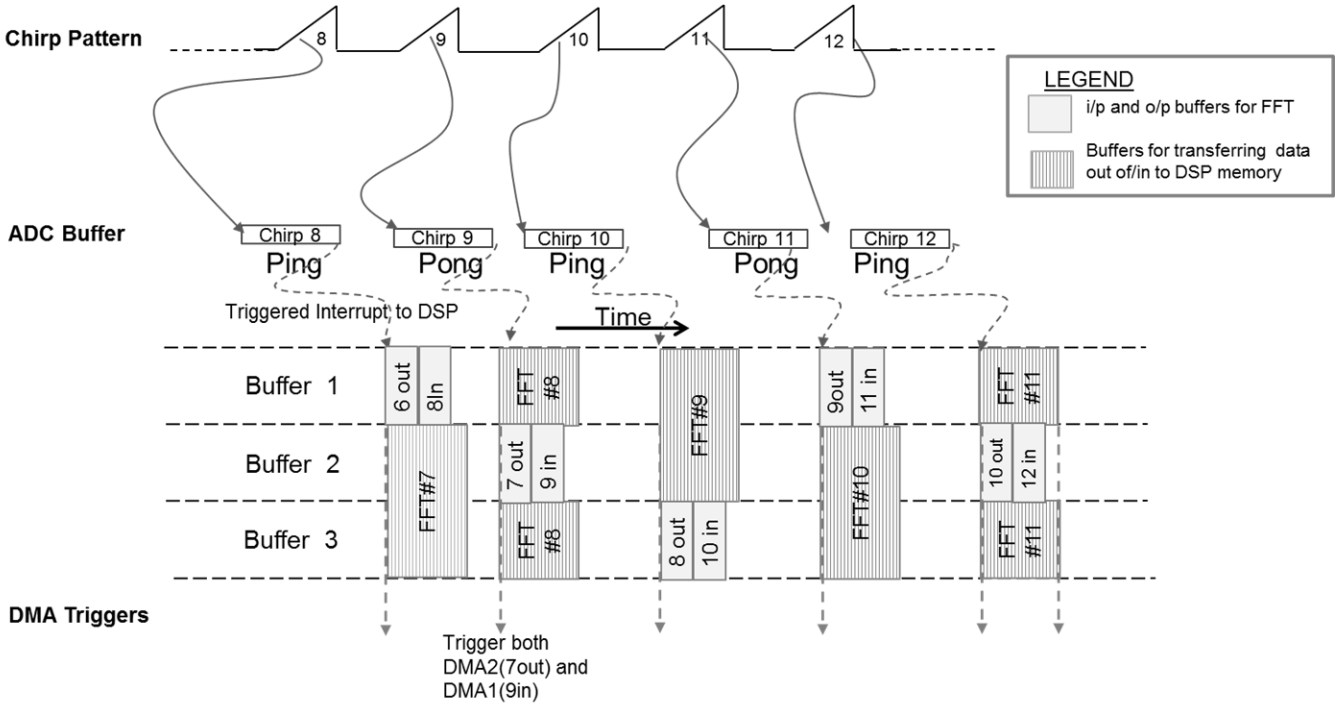

Figure 13. Buffer Management for Data Flow

4.2.1.1 *Speeding Up Transpose Transfer.*

A transpose-write-based EDMA transfer is 16 times slower than a contiguous EDMA transfer. Nevertheless, as was demonstrated in the previous use cases, it is possible to come up with data flows that ensure that the transpose-transfer does not become a bottleneck. However, should there be a need; there are options available to speed up this transfer. One option, discussed in Section 4.1, involves the use of two EDMA transfers operating in parallel using different TCs. Another option is described in the following discussion.

A transpose-write EDMA transfer takes four cycles to move one sample. This latency remains the same for any sample up to 128 bits in size (because the width of the bus that accesses L3 memory is 128 bits long). Therefore any sample size that is less than 128 bits (16 bytes) is making inefficient use of available bandwidth. Thus a complex sample of size 32 bits (16-bit I and 16-bit Q) is using only 1/4 the bandwidth. This can be remedied by performing a transpose on the range-FFT results prior to initiating a transpose transfer to L3. As shown in Figure 14 (matrix A), the range-FFT output for each antenna is contiguously placed. The DSP then performs a transpose operation on this matrix such that data for a specific range-bin is interleaved across antennas (matrix B). A transpose-write EDMA is now initiated with an ACNT of 128 bits. With this enhancement, the EDMA transfer that originally took 20.48 µs (in Figure 12), now takes 1/4 the time (5.12 µs). At the end of the intrachirp period, each row in L3 memory consists of data for a specific range-bin, interleaved across antennas. Subsequently, a contiguous transfer of each row can be used to transfer the data to DSP memory for Doppler-FFT processing. There will be some overhead during the Doppler processing to deinterleave this data. This method results in a 4× improvement in the latency of the transpose-write EDMA transfer. The additional overhead on the DSP (for interleaving the data after each range-FFT and deinterleaving prior to Doppler-FFT) is small and usually acceptable.

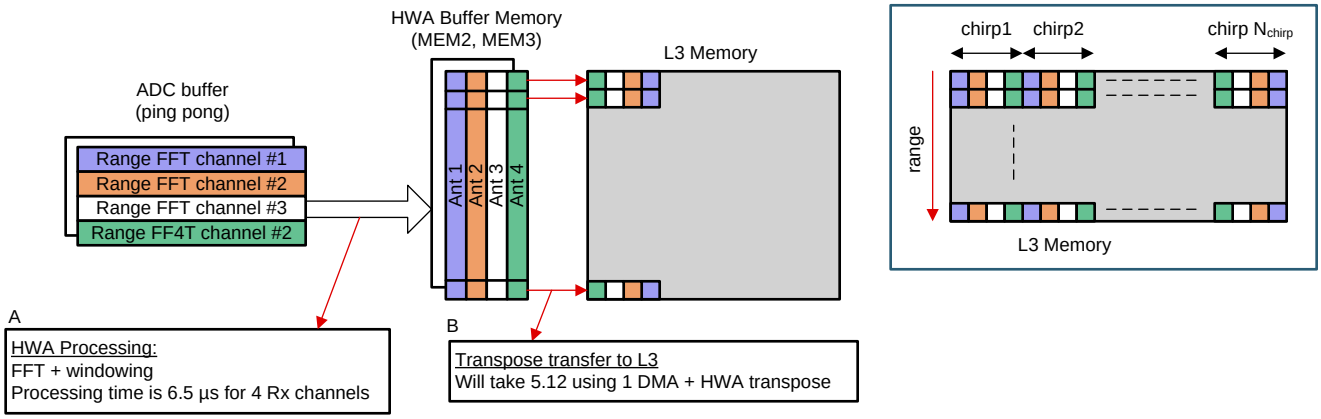

Figure 14. Improving Efficiency of the Transpose Transfer

NOTE: The DSP can transpose a 4 × 256 matrix in 576 cycles (approximately 0.96 µs).

4.2.2 Multichirp Use Case

Multichirp refers to the use case where the ADC buffer has been configured to interrupt the DSP after a fixed number of chirps have been collected. Figure 15 shows the timing diagram of a multichirp use case where the ADC buffer interrupts the HWA/DSP after every four chirps have been collected. Such a configuration results in fewer interrupts to the HWA/DSP and also longer gaps between processing associated with successive interrupts (contrast this with Figure 6, which depicts a single-chirp use-case). The DSP/HWA could enter a sleep mode during these gaps in processing, making it a useful power saving feature [debatable]. Alternatively, the DSP could perform higher level processing (such as clustering and tracking for prior frames) as a background process in these gaps.

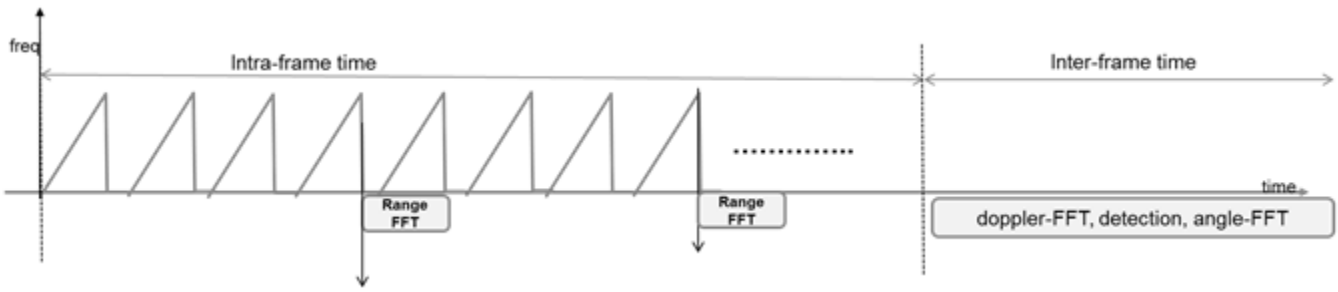

Figure 15. Timing for Multichirp Use Case

While it would be easy to adapt the techniques discussed in Section 4.2.1 to the multichirp use case, a slightly different approach is highlighted here. It is assumed that pre-processing is necessary, and that output of the HWA's FFT is going to be written to L3 contiguously, meaning that, the HWA must perform the transpose (prior to transfer to L3). Since the HWA's local memories are half the size of the ADC Buffers, the entire content of the ADC buffer cannot be bought into one of the HWA's 'local memory'. So, half of the processed chirps are brought into MEM0 and the other half into MEM1.

In the implementation, shown in Figure 16 and Figure 17, each row in the ADC buffer (a single chirp for a single antenna) is transferred to the DSP's L2 at a time that keeps the L2 buffering requirements to a minimum. The processed data from L2 is then transferred to the HWA's MEM0/MEM1 (ping or pong) in a contiguous transfer. There are two ping-pong buffer pairs that respectively manage the data that is input to the DSP (from the ADC buffer) and output to the HWA. Since both the EDMA transfers are contiguous, it ensures that the transfer latencies do not become a bottle-neck. The latencies for both these EDMA transfers (represented by A and C in Figure 16) is 0.32 µs, while the corresponding DSP pre-processing latency should be higher (say 2 µs). Since B > A+C, it ensures that the DSP never has to wait for an EDMA completion.

Once one HWA memory is filled, the HWA is triggered to perform the range-FFT processing with an additional transpose step. Following which, the range-FFT data can be transferred to L3 so that the number of contiguous bytes is 128 bits (Figure 17). Consequently the access for Doppler-FFT will now involve a linear-read EDMA access.

Once this process is complete, the DSP will trigger the HWA a second time to process the contents of HWA MEM1, which will now hold the processed ADC samples of the remaining Rx Antennas.

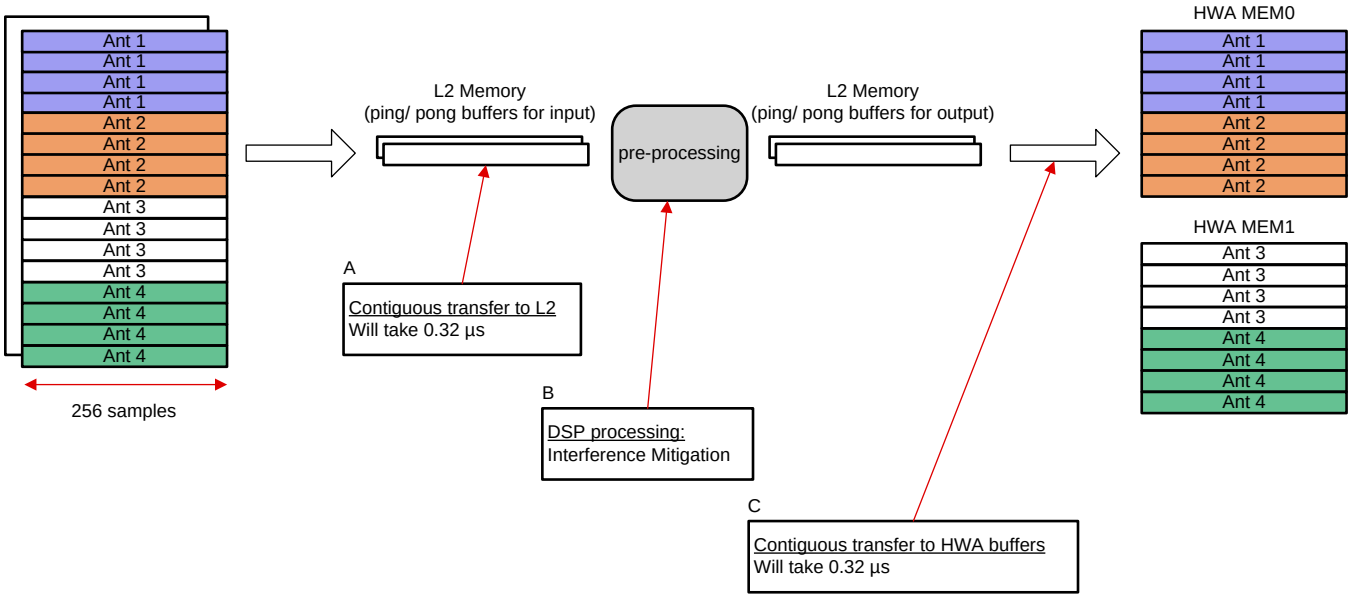

Figure 16. Multichirp Use Case (DSP to HWA)

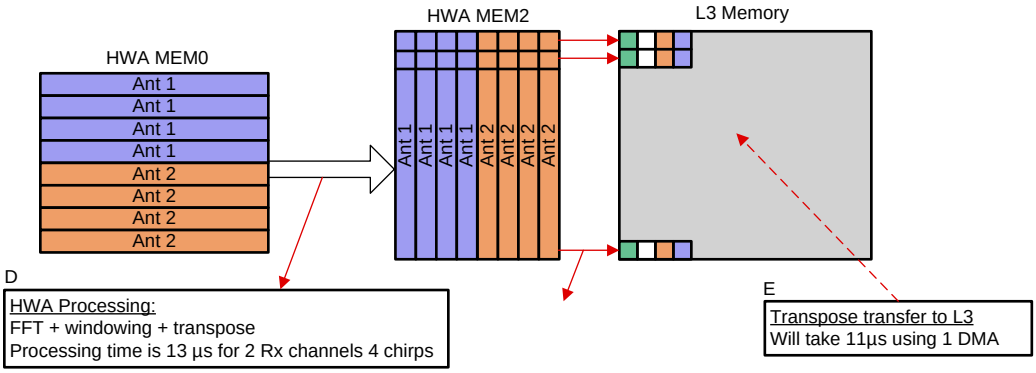

Figure 17. Multichirp Use Case (HWA to L3)

4.3 Example Use Cases: Interframe Processing

Interframe processing primarily involves Doppler-FFT computation, detection, and angle estimation. The data flow is largely determined by the way the range-FFT output has been stored in L3 memory during intraframe processing, and three such cases are discussed.

- Case 1

  If the range-FFT output has been placed in L3 using a transpose-write (as in for example, Figure 12) then the data required for performing a Doppler-FFT is contiguously available. This represents the simplest scenario for interframe processing, because all L3 accesses are now through contiguous EDMA transfers. As shown in Figure 18, 4 lines from L3 memory is transferred to the HWA local memory (say MEM0/MEM1) using a ping-pong scheme. After the Doppler-FFT has been computed, the result can be optionally transferred back to L3 (not shown in the figure).

  During interframe processing all the required input data is available in L3. Hence, to prevent any stalling in the processing, it is important that the EDMA transfer latencies keep up with the DSP/HWA processing latencies. In the case depicted in Figure 16 (with 128 chirps, 4 Rx channels), an EDMA transfer would take 0.64 µs, while a 128-pt FFT (for 4 Rx channels) takes 3.2 µs. Consequently the EDMA transfer should not be a bottle-neck even in the case where the Doppler-FFT output is written back to the radar-cube

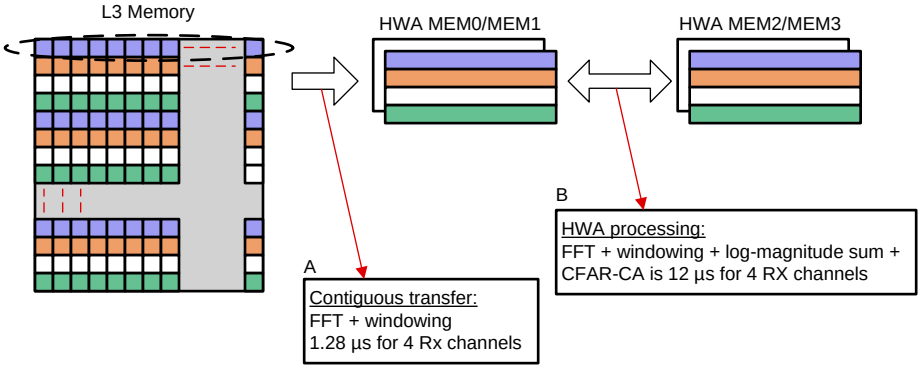

Figure 18. Interframe Processing Case 1

- Case 2

  Figure 19 depicts the interframe processing for the case where the range-FFT has been placed in L3 using the scheme suggested in Figure 13. Each line in L3 consists of data for a specific range-bin interleaved across antennas. The data can be transferred from L3 to the HWA. Since the FFT routines expect input data to be contiguously placed, the HWA is programmed to access data in transpose (de-interleave). There is no penalty in this case.

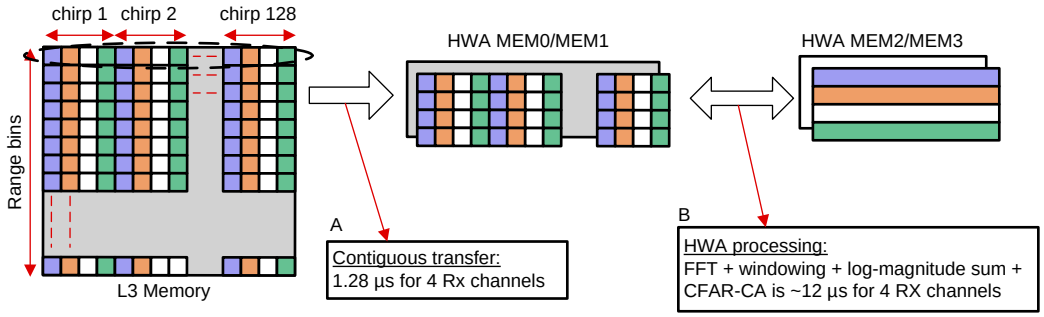

Figure 19. Interframe Processing Case 2

- Case 3

  If the range-FFT output is stored contiguously in L3, then Doppler-FFT processing will entail an EDMA transfer with transpose-read access as illustrated in Figure 11. Taking the example of 4 consecutive 128-pt FFTs (performed on the HWA), the FFTs would take 3.2 µs, while the corresponding EDMA transfer (transpose-read access for 128 x 4 samples) would take only 2.56 µs.

  In addition to the Doppler-FFT computation, the intra-frame processing will include other computations (such as computing the magnitude or 'log-magnitude' of the FFT result). Hence it is likely that the EDMA transfer will not be a bottle-neck and will be able to keep up with DSP processing. However, performing a write back of the Doppler-FFT results in the radar-cube would involve a larger penalty (since it would require a EDMA transpose-write, which in this example (4x128 samples) takes 11 µs). Hence this data flow architecture might not be optimal if such a write back is required

5 Discussion on Cache Strategy

Recall from Section 2.1 that L1 and L2 can partly or wholly be configured as cache (for both program and data). L1P is a direct-mapped cache, L1D is 2-way set associative, and L2 is a 4-way set associative cache [10]. Choice of an appropriate cache strategy can help achieve the desired trade-off between performance, code size, and development time.

The simplest strategy follows:

- Configure both the L1P and the L1D as cache.
- Place scratch pad data and program in L2, and use L3 only to store the radar-cube.

You should be able to achieve the desired performance using this approach (with the overhead due to caching being of the order of 15%). In data flow discussions in Section 4, it was implicitly assumed that various ping-pong buffers and other temporary buffers resided in L2 with L1D cache enabled. The C674x DSP supports a snoop-read and snoop-write [10] feature, which ensures cache coherency between L1D and L2 in the presence of EDMA reads and writes to L2.

However, given the highly deterministic nature of the radar signal-processing chain, additional improvements can be achievable by configuring a portion of L1P and L1D as RAM. Key routines which constitute the bulk of the radar signal processing, such as FFT and windowing routines, can be placed in L1P, while the associated scratch pad buffers (such as some of the ping-pong buffers depicted in the earlier data-flows) can be placed in L1D. In one experiment, all real-time frame-work and algorithm code (up to clustering) is put into 24KB L1P SRAM (leaving only 8KB for L1P cache), and saw modest improvement of cycle performance of the chain. 16KB of L1D is also configured as data SRAM (leaving 16KB for L1D cache) to hold temporary processing buffers (reused between range processing and Doppler processing). By doing so the cycle performance of range processing was improved by 5~10%. Also, there is much less cycle fluctuation in range processing due to a decrease in L1 cache activity. The reduced size of L1 cache did not seem to degrade the cycle performance of other algorithms that worked out of buffers in L2. This placement of code and data in L1P and L1D also saved 40KB of L2 memory, without negative impact of cycle performance of the whole chain

Alternatives to EDMA-based L3 access: The data flows discussed in Section 4 relied on the EDMA engine to access data in L3. However, in some scenarios direct access of L3 memory by the CPU might also be considered. Each EDMA transfer has an overhead in terms of triggering the EDMA and subsequently verifying its completion (either through an interrupt or polling). These overheads could swamp out the benefits of using an EDMA for small sized transfers. Such a scenario could occur, for example, in the context of an FMCW frame with a small number of chirps (16 or 32) resulting in small sized Doppler-FFTs. One option is to fetch data for multiple Doppler-FFTs in every EDMA transfer. This reduces the EDMA overhead, though at the cost of proportionally larger buffer sizes. In such cases direct access to L3 (with caching in L1D) could be a viable alternative. In one of the experiments with size 32 Doppler-FFT's, 4 Rx antennas, and 1024 range bins, direct L3 access (with L1 cache on) performed slightly faster compared to an EDMA based approach. This also reduced the code complexity (with the elimination of EDMA triggering/polling). We also experimented on having the pre-detection matrix inside L3 memory with direct CPU accesses (via cache). While it was observed that the detection algorithm (CFAR) had cycle degradation of about 2x (vs. the pre-detection matrix residing in L2), the freeing up of L2 memory might make it a worthwhile trade-off in certain cases.

Code in L3: Customers also have the option of storing program code in L3. This option is typically suited for placing code which would be used once per frame (in one go) rather than being executed multiple times with the other code in an interleaved manner (the intent is to reduce the loss due to repeated code eviction). One example is Kalman filtering or detection. While actual results will vary depending on the nature of the code, our experience running a Kalman filter from L3 shows a hit of about 2× for the first invocation and almost no overhead for subsequent invocations (due to the program being cached in L1P).

Whenever L3 is being directly accessed by the CPU (either for code or data), multiple caching options are possible. One option is to cache L3 directly to L1 (L1P, L1D). Another option is to create a hierarchical cache structure, with a small part of L2 (about 32KB) also configured as cache.

6 References

1. *Industrial Radar Device Family Technical Reference Manual*
2. *IWR1642 Single-Chip 76- to 81-GHz mmWave Sensor*
3. *mmWave Device Firmware Package (DFP)*
4. *TMS320C6748 DSP Technical Reference Manual*
5. *TMS320C674x DSP CPU and Instruction Set Reference Guide*
6. *TMS320C674x DSP Megamodule Reference Guide*
7. *TMS320C600 Optimizing Compiler User's Guide*
8. *TMS320C6000 Programmer's Guide*
9. *TMS320C6000 DSP Library (DSPLIB)*: Required DSPLIB libraries that must be installed for the C674x:
   a. The fixed-point library for the C64x+
   b. The floating-point library for C674x (documentation included with the installation provides APIs for all the routines and a cycle benchmarking report)
10. *mmWave Software Development Kit (SDK)*: Install the mmwave-SDK and then navigate to \packages\ti\alg\mmwavelib to view both the documentation and code for mmwavelib.
11. *Radar Hardware Accelerator User's Guide*
12. *Introduction to mm-wave sensing: FMCW Radars*
13. *TMS320C6000 DSP Library (DSPLIB)*

IMPORTANT NOTICE FOR TI DESIGN INFORMATION AND RESOURCES

Texas Instruments Incorporated ('TI") technical, application or other design advice, services or information, including, but not limited to, reference designs and materials relating to evaluation modules, (collectively, "TI Resources") are intended to assist designers who are developing applications that incorporate TI products; by downloading, accessing or using any particular TI Resource in any way, you (individually or, if you are acting on behalf of a company, your company) agree to use it solely for this purpose and subject to the terms of this Notice.

TI's provision of TI Resources does not expand or otherwise alter TI's applicable published warranties or warranty disclaimers for TI products, and no additional obligations or liabilities arise from TI providing such TI Resources. TI reserves the right to make corrections, enhancements, improvements and other changes to its TI Resources.

You understand and agree that you remain responsible for using your independent analysis, evaluation and judgment in designing your applications and that you have full and exclusive responsibility to assure the safety of your applications and compliance of your applications (and of all TI products used in or for your applications) with all applicable regulations, laws and other applicable requirements. You represent that, with respect to your applications, you have all the necessary expertise to create and implement safeguards that (1) anticipate dangerous consequences of failures, (2) monitor failures and their consequences, and (3) lessen the likelihood of failures that might cause harm and take appropriate actions. You agree that prior to using or distributing any applications that include TI products, you will thoroughly test such applications and the functionality of such TI products as used in such applications. TI has not conducted any testing other than that specifically described in the published documentation for a particular TI Resource.

You are authorized to use, copy and modify any individual TI Resource only in connection with the development of applications that include the TI product(s) identified in such TI Resource. NO OTHER LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE TO ANY OTHER TI INTELLECTUAL PROPERTY RIGHT, AND NO LICENSE TO ANY TECHNOLOGY OR INTELLECTUAL PROPERTY RIGHT OF TI OR ANY THIRD PARTY IS GRANTED HEREIN, including but not limited to any patent right, copyright, mask work right, or other intellectual property right relating to any combination, machine, or process in which TI products or services are used. Information regarding or referencing third-party products or services does not constitute a license to use such products or services, or a warranty or endorsement thereof. Use of TI Resources may require a license from a third party under the patents or other intellectual property of the third party, or a license from TI under the patents or other intellectual property of TI.

TI RESOURCES ARE PROVIDED "AS IS" AND WITH ALL FAULTS. TI DISCLAIMS ALL OTHER WARRANTIES OR REPRESENTATIONS, EXPRESS OR IMPLIED, REGARDING TI RESOURCES OR USE THEREOF, INCLUDING BUT NOT LIMITED TO ACCURACY OR COMPLETENESS, TITLE, ANY EPIDEMIC FAILURE WARRANTY AND ANY IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT OF ANY THIRD PARTY INTELLECTUAL PROPERTY RIGHTS.

TI SHALL NOT BE LIABLE FOR AND SHALL NOT DEFEND OR INDEMNIFY YOU AGAINST ANY CLAIM, INCLUDING BUT NOT LIMITED TO ANY INFRINGEMENT CLAIM THAT RELATES TO OR IS BASED ON ANY COMBINATION OF PRODUCTS EVEN IF DESCRIBED IN TI RESOURCES OR OTHERWISE. IN NO EVENT SHALL TI BE LIABLE FOR ANY ACTUAL, DIRECT, SPECIAL, COLLATERAL, INDIRECT, PUNITIVE, INCIDENTAL, CONSEQUENTIAL OR EXEMPLARY DAMAGES IN CONNECTION WITH OR ARISING OUT OF TI RESOURCES OR USE THEREOF, AND REGARDLESS OF WHETHER TI HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

You agree to fully indemnify TI and its representatives against any damages, costs, losses, and/or liabilities arising out of your non-compliance with the terms and provisions of this Notice.

This Notice applies to TI Resources. Additional terms apply to the use and purchase of certain types of materials, TI products and services. These include; without limitation, TI's standard terms for semiconductor products http://www.ti.com/sc/docs/stdterms.htm), evaluation modules, and samples (http://www.ti.com/sc/docs/sampterms.htm).

Mailing Address: Texas Instruments, Post Office Box 655303, Dallas, Texas 75265