DSP-Based Audio Encryption and Decryption on the TMS320C6713 Platform

1. Introduction

This report details the design and implementation considerations for a simple Digital Signal Processor (DSP)-based audio encryption and decryption system utilizing the Texas Instruments (TI) TMS320C6713 device. The primary objective is to explore basic techniques suitable for real-time execution on this specific DSP platform, focusing on methods like pseudo-random key stream generation with XOR operations and data permutation, rather than computationally intensive, standardized cryptographic algorithms (e.g., AES, RSA). The scope encompasses an analysis of the TMS320C6713 architecture, its associated Development Starter Kit (DSK), the development environment (Code Composer Studio), selection and implementation of simple encryption algorithms, integration into a real-time audio processing framework, and methods for testing and verification.

It is crucial to emphasize that the encryption techniques discussed herein are intended for basic confidentiality or deterring casual eavesdropping, not for providing robust security against determined adversaries.[1] The focus remains on the practical aspects of implementing computationally lightweight algorithms within the real-time constraints imposed by audio signal processing on the target hardware.

The report is structured as follows: Section 2 provides an overview of the TMS320C6713 DSP and its DSK platform. Section 3 details the development environment and tools. Section 4 presents a basic framework for real-time audio input/output. Section 5 explores simple encryption algorithms suitable for DSP implementation. Section 6 discusses the integration of these algorithms into the real-time framework and analyzes performance constraints. Section 7 outlines testing and verification strategies. Finally, Section 8 concludes the report with a summary and recommendations.

2. The TMS320C6713 DSP Platform

The Texas Instruments TMS320C6713 is a high-performance, floating-point DSP specifically chosen for this project due to its processing power and peripheral set optimized for audio applications.[4] Understanding its architecture and the features of its associated Development Starter Kit (DSK) is fundamental to implementing real-time audio processing tasks.

2.1 TMS320C6713 Architecture and Performance

The core of the TMS320C6713 is the advanced VelociTI™ Very Long Instruction Word (VLIW) TMS320C67x™ DSP architecture.[7] This architecture is designed for high parallelism, enabling the execution of multiple instructions per clock cycle. Key architectural features include:

- **Eight Independent Functional Units:** The CPU incorporates eight functional units operating in parallel: two fixed-point Arithmetic Logic Units (ALUs), four ALUs capable of both fixed-point and floating-point operations, and two multipliers also capable of fixed-point and floating-point operations.[7] This high degree of parallelism allows the C6713 to execute up to eight 32-bit instructions per clock cycle.[4]
- **Floating-Point Capabilities:** The C67x core natively supports IEEE 754 standard 32-bit single-precision and 64-bit double-precision floating-point arithmetic, making it well-suited for algorithms requiring high dynamic range and precision.[7]
- **Load-Store Architecture:** The processor employs a load-store architecture, meaning that operations are performed primarily on data held in registers, with dedicated instructions for moving data between registers and memory. It features 32 general-purpose 32-bit registers.[7]
- **Instruction Set Features:** The instruction set includes features like instruction packing to reduce code size, conditional execution for all instructions to minimize branching overhead, byte-addressability (8-, 16-, 32-bit data), saturation arithmetic, and bit manipulation capabilities.[7]
- **Performance Metrics:** The TMS320C6713 is available in various clock speeds, including 167, 200, 225, and 300 MHz.[4] At 225 MHz, it delivers up to 1800 Million Instructions Per Second (MIPS) and 1350 Million Floating-Point Operations Per Second (MFLOPS).[7] At 300 MHz, performance increases to 2400 MIPS and 1800 MFLOPS.[7] The dual multipliers enable up to 450 Million Multiply-Accumulate operations per Second (MMACS) at 225 MHz and 600 MMACS at 300 MHz.[7] This computational power provides significant headroom for real-time audio processing, including the execution of encryption algorithms alongside other potential signal processing tasks.

2.2 Memory System

The C6713 utilizes a two-level cache-based memory architecture to bridge the speed gap between the fast CPU core and potentially slower external memory [7]:

- **Level 1 Cache (L1):** Consists of a 4K-byte direct-mapped program cache (L1P) and a 4K-byte 2-way set-associative data cache (L1D).[7] These small, fast caches hold recently used instructions and data, respectively, allowing for rapid access

by the CPU.

- **Level 2 Memory/Cache (L2):** Provides 256K bytes of on-chip memory, shared between program and data space.[7] This L2 memory is configured as follows:
  - 64K bytes can be flexibly configured as mapped RAM (directly addressable static RAM), cache (acting as a larger, slower cache backing up L1), or a combination of both.[7]
  - The remaining 192K bytes function as dedicated mapped RAM.[7] Placing time-critical code and frequently accessed data in L1/L2 memory is crucial for achieving maximum real-time performance.
- **External Memory Interface (EMIF):** A 32-bit EMIF provides a glueless interface to a variety of external memory types, including Synchronous DRAM (SDRAM), Static RAM (SRAM), EPROM, and Flash memory.[7] It supports a total addressable external memory space of up to 512M bytes.[7] Accessing external memory incurs higher latency compared to on-chip L1/L2 access.

2.3 Key Peripherals for Audio Processing

The C6713 integrates a rich set of peripherals optimized for audio and communication tasks [4]:

- **Multichannel Audio Serial Ports (McASPs):** The device features two McASPs, which are highly flexible serial ports designed specifically for audio applications.[7] Key McASP features include:
  - Independent transmit (TX) and receive (RX) sections, each with potentially independent clocking and frame synchronization.[7]
  - Support for Time Division Multiplexing (TDM) streams with 2 to 32 time slots, enabling multi-channel audio interfacing.[7]
  - Programmable slot sizes (8 to 32 bits) and data formatters for bit manipulation.[7]
  - Support for various standard serial audio formats like I^2S and similar bit streams.[7]
  - An integrated Digital Audio Interface Transmitter (DIT) supporting formats like S/PDIF, AES-3, and IEC60958-1.[7]
  - Extensive error checking capabilities.[7]
  - Up to eight serial data pins per port, assignable to different clock zones.[7] The McASPs provide the necessary hardware support for interfacing with multiple audio codecs or other digital audio devices simultaneously.[14]
- **Multichannel Buffered Serial Ports (McBSPs):** Two McBSPs are included, offering general-purpose synchronous serial communication.[7] They support:
  - Serial Peripheral Interface (SPI) mode.[7]

    - High-speed TDM interfacing.[7]
    - AC97 interface for legacy audio codecs.[7] On the C6713 DSK, the McBSPs are crucial for interfacing with the onboard AIC23 audio codec.[15]
- **Enhanced Direct Memory Access (EDMA) Controller:** The EDMA controller features 16 independent channels and facilitates high-speed data transfers between memory locations and peripherals (like McASPs/McBSPs) without CPU intervention.[7] Utilizing EDMA is essential for achieving high throughput and freeing up the CPU for core processing tasks, particularly at higher audio sample rates.[17]
- **Other Peripherals:** The C6713 also includes two Inter-Integrated Circuit (I^2C) interfaces (often used for controlling peripherals like codecs), two 32-bit general-purpose timers, a dedicated General-Purpose Input/Output (GPIO) module, and a Host Port Interface (HPI) for communication with a host processor.[7]

2.4 TMS320C6713 DSK Hardware Context

The TMS320C6713 DSK is a widely used, low-cost development platform that provides the necessary hardware to evaluate and develop applications for the C6713 DSP.[11] Key components of the DSK include:

- **TMS320C6713 DSP:** Typically operating at 225 MHz.[10]
- **AIC23 Stereo Audio Codec:** A TLV320AIC23B codec provides analog-to-digital (ADC) and digital-to-analog (DAC) conversion capabilities.[10] It supports sample rates from 8 kHz up to 96 kHz [10] (or potentially higher, up to 192kHz according to [23], though DSK implementations often use up to 96kHz) and various data bit widths (16-32 bits).[10]
- **Memory:** The DSK includes 16 MBytes (MB) of external SDRAM [15] (note: [16] clarifies this is 8MB based on the 64Mbit chip and 32-bit interface) and 512 KBytes (KB) of external Flash ROM.[15] By default, due to the C6713's boot mode configuration, only 256 KB of the Flash memory is typically usable without software modifications.[15]
- **Audio Jacks:** Four standard 3.5mm audio jacks provide connections for Microphone Input (MIC IN), Line Input (LINE IN), Line Output (LINE OUT), and Headphone Output (HP OUT).[10] The codec can select between MIC IN and LINE IN as the active input source.[11]
- **User Interface:** Four user-definable LEDs and a 4-position DIP switch allow for simple status indication and user input.[10]
- **Connectivity:** An embedded USB JTAG controller provides the interface for programming and debugging from a host PC via a USB cable.[20]
- **Codec Interface:** On the DSK, the AIC23 codec communicates with the C6713

DSP using the McBSP peripherals. McBSP0 is typically used as a unidirectional SPI-based control channel to configure the codec registers, while McBSP1 serves as the bidirectional data channel for transferring the actual audio samples.[15]

3. Development Environment

Developing applications for the TMS320C6713 DSK relies heavily on Texas Instruments' Code Composer Studio (CCS) Integrated Development Environment (IDE) and associated support libraries.

3.1 Code Composer Studio (CCS)

CCS is the primary toolsuite for developing and debugging embedded applications on TI DSPs.[18] It integrates various essential tools into a unified environment:

- **Source Code Editor:** For writing C, C++, and assembly code.
- **C/C++ Compiler and Assembler:** Tools specific to the C6000 architecture, translating high-level code and assembly into machine code executable by the DSP.[18]
- **Linker:** Combines compiled object files and libraries into a single executable .out file, managing memory allocation according to a linker command file (.cmd).
- **Debugger:** Allows developers to connect to the target DSK (via the USB JTAG interface), load the executable program, control execution (run, halt, step), set breakpoints, and inspect memory, registers, and variables.[18]
- **Profiler:** Helps analyze code performance by measuring execution time and cycle counts for different functions (use with caution in real-time systems as profiling can be intrusive).
- **Project Management:** Organizes source files, libraries, and build settings into projects.[25]

Various versions of CCS have been used with the C6713 DSK over time, including CCS v3.1 [20], v5.1 [26], and v6.1.[25] While newer versions are based on the Eclipse framework, older versions had a different interface. The specific version may influence the exact menu options and setup procedures, but the core concepts remain similar. Installation typically requires selecting support for the C6000 family and the specific Spectrum Digital DSK drivers.[25]

3.2 Project Workflow in CCS

A typical workflow for creating and running a project on the C6713 DSK using CCS (versions 5.x/6.x) involves the following steps [25]:

1. **Launch CCS and Select Workspace:** Start CCS and choose a directory to store

project files.[25]

2. **Create New CCS Project:** Use the File -> New -> CCS Project menu option.[25]
3. **Configure Project:**
    - Provide a project name (e.g., AudioEncrypt).
    - Select the target device: TMS320C6713.[25] Ensure a compatible compiler version (e.g., v7.x for C6713 in CCS v6.1, as v8+ may not be supported) is installed and selected.[25]
    - Choose a project template, typically Empty Project to start from scratch.[25]
    - **Crucially, configure Advanced Settings:**
        - Specify the **Linker command file** (.cmd). This file defines the memory map for the specific hardware (DSK) and tells the linker where to place code and data sections (e.g., in internal L2 RAM or external SDRAM). A standard link6713.cmd file is usually provided with DSK support files or examples.[25]
        - Set the **Output format** to legacy COFF (Common Object File Format), which is the format expected by the C6713 tools and debugger.[25]
4. **Add Files to Project:** Right-click the project in the Project Explorer and use "Add Files..." to include [25]:
    - Your C source code files (e.g., main.c, encrypt.c).
    - Any necessary assembly files (.asm).
    - Required libraries (e.g., rts6700.lib, dsk6713bsl.lib, csl6713.lib). These are typically found in the CCS installation directories or DSK support packages.[25] Select the option to copy the files into the project directory.
5. **Configure Build Options:** Access Project -> Properties -> Build. Key settings include:
    - **Compiler Options:**
        - Set the Target Version to C671x or similar.[10]
        - Define preprocessor symbols, often CHIP_6713 is required by BSL/CSL headers.[10]
        - Specify Include Search Paths to tell the compiler where to find header files (.h) for libraries like BSL and CSL (e.g., C:\ti\c6000\dsk6713\include).[10]
    - **Linker Options:** Ensure the correct Runtime Support Library is specified and that the linker command file is correctly referenced.[22]
6. **Build Project:** Select Project -> Build Project. CCS compiles the source files and links them with libraries to create an executable .out file.[25] Check the Console window for errors or warnings.
7. **Debug/Load/Run:**
    - **Connect DSK:** Ensure the DSK is powered on and connected to the PC via

USB.[25]
    - **Target Configuration:** Create or add a Target Configuration File (.ccxml) specific to the DSK6713 with its USB emulator. This file tells CCS how to connect to the hardware. Ensure the configuration uses the correct emulator (Spectrum Digital XDS510 USB Emulator or similar) and references the appropriate GEL (General Extension Language) file (e.g., dsk6713.gel) for board initialization.[25]
    - **Launch Debug Session:** Select the project and click the Debug icon (often a green bug) or use Run -> Debug.[25] CCS will connect to the DSK, load the .out file into the DSP's memory, and halt execution at the entry point of the main function.[25]
    - **Run/Resume:** Use the Run -> Resume command (or F8 key) to start program execution.[25]
    - **Halt/Suspend:** Pause execution using Run -> Suspend or the pause button.[25]
    - **Breakpoints:** Set breakpoints by double-clicking in the margin of the source editor to halt execution at specific lines.[27]
    - **Inspect:** While halted, use debugger windows (Expressions, Memory Browser, Registers) to examine the program state.[27]
    - **Terminate:** End the debug session using Run -> Terminate.[25]

3.3 Support Libraries (RTS, CSL, BSL)

Efficient development on the C6713 DSK leverages several key libraries:

- **Runtime Support Library (RTS):** Provides standard ANSI C library functions (like printf, memcpy, basic math functions) compiled for the C67x architecture. The specific library file (e.g., rts6700.lib, rts67pluse.lib) depends on the target and desired floating-point support.[28] It's essential for almost any C program.
- **Chip Support Library (CSL):** Offers a low-level C language interface for configuring and controlling the on-chip peripherals of the TMS320C6713 (e.g., McBSPs, McASPs, EDMA, Timers) by providing functions to access peripheral registers.[10] It provides finer control than the BSL but requires more detailed knowledge of the peripheral hardware. The library file is typically csl6713.lib.[31]
- **Board Support Library (BSL):** Specifically designed for the DSK board, the BSL provides higher-level functions to initialize and control the DSK-specific hardware components, most notably the AIC23 audio codec, user LEDs, and DIP switches.[10] It abstracts away many of the complexities of direct CSL and register manipulation for common DSK tasks. The library file is typically dsk6713bsl.lib.[20] Key BSL functions for audio I/O via the AIC23 include [10]:
    - DSK6713_init(): Initializes the DSK board (PLL, EMIF, CPLD). Must be called

first.
- DSK6713_AIC23_openCodec(instance, config): Opens and configures the AIC23 codec. Returns a handle used by other codec functions.
- DSK6713_AIC23_closeCodec(handle): Closes the codec.
- DSK6713_AIC23_read(handle, &sample_pair): Reads a 32-bit stereo sample (left channel in upper 16 bits, right in lower 16 bits) from the codec. Often used in a polling loop (while(!DSK6713_AIC23_read(...));).
- DSK6713_AIC23_write(handle, sample_pair): Writes a 32-bit stereo sample to the codec. Often used in a polling loop (while(!DSK6713_AIC23_write(...));).
- DSK6713_AIC23_setFreq(handle, freq_id): Sets the codec's sampling frequency (e.g., DSK6713_AIC23_FREQ_8KHZ, DSK6713_AIC23_FREQ_48KHZ).
- DSK6713_AIC23_rset(handle, regnum, regval): Allows direct writing to specific AIC23 configuration registers for advanced control (e.g., changing input source, gain).[36]

Documentation for the BSL functions is typically found in the c6713dsk.hlp help file provided with the DSK software installation.[20]

3.4 Basic Audio Input/Output Framework (using BSL)

The DSK BSL significantly simplifies the setup for basic audio input and output compared to direct manipulation of the McBSP and AIC23 registers. The following C code provides a fundamental polling-based framework for reading audio from the codec, processing it (placeholder), and writing it back out. This framework serves as the foundation for integrating the encryption/decryption algorithms.

C

```c
#include "dsk6713.h"       // DSK Board Support Library main header
#include "dsk6713_aic23.h"  // AIC23 Codec specific BSL functions

// AIC23 Codec configuration settings
// Uses default configuration defined in dsk6713_aic23.h
// Alternatively, define custom settings as shown in [34] or [10]
DSK6713_AIC23_Config config = DSK6713_AIC23_DEFAULTCONFIG;
DSK6713_AIC23_CodecHandle hCodec; // Handle to the codec

void main() {
    // Union allows accessing the 32-bit stereo pair as a single Uint32
```

```
    // or as two individual 16-bit short integers (left/right channels)
    // Assumes Little Endian: channel = Right, channel[1] = Left
    // Check documentation or test if using Big Endian.
    union {
        Uint32 uint32;
        short channel[2]; // channel typically right, channel[1] typically left
    } data;

    // Initialize the DSK board hardware (PLL, CPLD, etc.)
    DSK6713_init();

    // Open the codec instance 0 with the default configuration
    hCodec = DSK6713_AIC23_openCodec(0, &config);

    // Set the codec sampling frequency (e.g., 8kHz)
    // Other options: DSK6713_AIC23_FREQ_16KHZ, _32KHZ, _44_1KHZ, _48KHZ, _96KHZ
    DSK6713_AIC23_setFreq(hCodec, DSK6713_AIC23_FREQ_8KHZ);

    // --- Optional: Modify Codec Settings (Example: Select Mic Input) ---
    // Consult AIC23 datasheet (TLV320AIC23B) and dsk6713_aic23.h for register details
    // Example: Select Mic input (Analog Audio Path Control Register 4, MICIN bit)
    // DSK6713_AIC23_rset(hCodec, 4, config.regs[3] | 0x0004); // Set MICSEL bit
    // Example: Increase Mic gain (Left Line Input Channel Volume Control Reg 0, LRS=1, LINMUTE=0, LIV
bits)
    // DSK6713_AIC23_rset(hCodec, 0, 0x0097); // Example: +12dB gain, not muted (Verify value!)

    // Infinite loop for continuous audio processing
    for(;;) {
        // Wait until the codec has a new stereo sample ready, then read it
        // DSK6713_AIC23_read polls the McBSP receive ready flag
        while(!DSK6713_AIC23_read(hCodec, &data.uint32));

        // *** PLACEHOLDER: Insert Encryption/Decryption Algorithm Here ***
        // Process the audio data stored in 'data.uint32' or 'data.channel' (Right)
        // and 'data.channel[1]' (Left).
        // Example: Simple pass-through (no processing)
        // processed_data.uint32 = data.uint32;

        // Wait until the codec is ready to accept a new stereo sample, then write it
```

```
        // DSK6713_AIC23_write polls the McBSP transmit ready flag
        // Replace 'data.uint32' with 'processed_data.uint32' if processing is done
        while(!DSK6713_AIC23_write(hCodec, data.uint32));
    }

    // --- Codec Closing (Not reached in this infinite loop, but good practice) ---
    // DSK6713_AIC23_closeCodec(hCodec);
}
```

Code based on examples and BSL function descriptions from.[10]

This BSL-based framework significantly accelerates development by handling the low-level details of McBSP configuration and codec communication.[10] While polling (while(!DSK6713_AIC23_read...)) is simple, it consumes CPU cycles waiting for the codec. For more demanding applications, alternative I/O methods are necessary. Interrupt-driven I/O frees the CPU between samples but requires writing Interrupt Service Routines (ISRs) and managing the interrupt controller.[20] EDMA offers the highest efficiency by transferring data blocks automatically, minimizing CPU load, but involves complex configuration of the EDMA controller and buffer management (e.g., ping-pong buffering).[17] The optimal choice depends on the computational load of the encryption algorithm relative to the processing time available per sample, which is inversely proportional to the sample rate.[10] At 8 kHz, there is 125 µs per sample; at 48 kHz, this drops to approximately 20.8 µs, placing much tighter constraints on processing time.[40]

4. Simple Audio Encryption Techniques for DSP

The goal is to implement simple audio encryption/decryption suitable for the real-time constraints of the TMS320C6713. These techniques aim for basic confidentiality or scrambling, not high-level cryptographic security.[1] Core methods include substitution (like XOR) and permutation (reordering).

4.1 Conceptual Overview

- **Objective:** Mask audio intelligibility for unauthorized listeners using computationally inexpensive methods.[1]
- **Scrambling:** Altering the signal structure, often via permutations in the time or frequency domain, to reduce intelligibility.[1]
- **Permutation:** Reordering data units. This can occur at the level of audio samples within a block [2], or by shuffling the bits within individual samples.[43] FFT coefficient permutation scrambles frequency components but adds significant overhead.[42]

- **Substitution:** Replacing data values with others. A common and simple method is the bitwise XOR operation with a pseudo-random key stream.[1]
- **Layering:** Combining permutation and substitution can potentially increase the scrambling effect.[43]

4.2 Algorithm 1: XOR Encryption with Pseudo-Random Key Stream

This approach uses the bitwise XOR operation, a fundamental building block in many ciphers.

- **Principle:** The core operation is Ciphertext = Plaintext XOR Key. Since XORing twice with the same key restores the original value (Plaintext = Ciphertext XOR Key), the same process works for both encryption and decryption.[1] The operation is computationally very fast, involving only basic bitwise logic.
- **Key Stream Generation (LFSR):** The security of XOR encryption depends entirely on the key stream's properties: it should be unpredictable, non-repeating (over a long period), and kept secret.[1] A Linear Feedback Shift Register (LFSR) provides a computationally efficient method for generating long Pseudo-Random Binary Sequences (PRBS) on embedded systems like DSPs.[45]
  - An LFSR consists of a shift register (representing the state) and a feedback path based on XORing specific bits (taps) of the state, determined by a feedback polynomial.[47]
  - Choosing a "primitive" polynomial ensures the LFSR generates a maximal-length sequence (2^N-1 states for an N-bit register) before repeating.[47]
  - The output bit (often the LSB or MSB) forms the pseudo-random sequence.[48]
  - **Example C Code (16-bit Fibonacci LFSR):** This demonstrates generating pseudo-random bytes. The taps (0xB400u) correspond to the primitive polynomial x¹⁶ + x¹⁴ + x¹³ + x¹¹ + 1 for maximal length.[47] The seed (0xACE1u) must be non-zero.

```
C
// State variable for the LFSR (must be non-zero)
static unsigned short lfsr_state = 0xACE1u;

// Function to generate one pseudo-random byte using the LFSR
unsigned char generate_lfsr_byte() {
    unsigned char output_byte = 0;
    for (int i = 0; i < 8; i++) { // Generate 8 bits for the byte
        // Taps for polynomial x^16 + x^14 + x^13 + x^11 + 1 (primitive)
        // Feedback is XOR sum of bits 16, 14, 13, 11 (using 1-based indexing for powers)
        // Corresponding bits in 16-bit register (0-based index): 15, 13, 12, 10
```

```
        unsigned short bit = ((lfsr_state >> 0) ^  // Bit 0 corresponds to x^11 term influence after shifts
                              (lfsr_state >> 2) ^  // Bit 2 corresponds to x^13 term influence
                              (lfsr_state >> 3) ^  // Bit 3 corresponds to x^14 term influence
                              (lfsr_state >> 5)) & 1u; // Bit 5 corresponds to x^16 term influence

        // Build the output byte (LSB first)
        output_byte |= (lfsr_state & 1u) << i;

        // Shift the register and XOR in the feedback bit at the MSB position
        lfsr_state = (lfsr_state >> 1) | (bit << 15);
    }
    return output_byte;
}

// Alternative: Galois LFSR (often more efficient in hardware, potentially software)
// Example for x^16 + x^12 + x^3 + x + 1 (Taps at 16, 12, 3, 1)
// static unsigned short lfsr_galois_state = 0xACE1u;
// unsigned char generate_galois_lfsr_byte() {
//     unsigned char output_byte = 0;
//     for (int i = 0; i < 8; i++) {
//         output_byte |= (lfsr_galois_state & 1u) << i; // Output LSB
//         unsigned short lsb = lfsr_galois_state & 1u; // Get LSB
//         lfsr_galois_state >>= 1; // Shift state right
//         if (lsb) { // If LSB was 1, XOR state with polynomial taps (shifted)
//             lfsr_galois_state ^= 0xB001; // Polynomial 1011000000000001 (adjust based on representation)
//         }
//     }
//     return output_byte;
// }
```

(LFSR concepts based on [45])

- **Implementation:** The generated pseudo-random byte (or word, if generating 16/32 bits at a time) is XORed with each corresponding audio sample byte (or word) from the codec.[1] This operation is applied sample-by-sample within the real-time audio processing loop.
- **Security:** Simple XOR with a single, unkeyed, short-cycle LFSR provides very weak security. The sequence is deterministic and easily reproduced if the polynomial and seed are known or guessed. It is highly vulnerable to known-plaintext attacks (if an attacker knows some original audio and its encrypted version, they can recover the key stream). For slightly better security, one might combine outputs from multiple LFSRs or use a more complex PRNG, but this increases computational cost.

4.3 Algorithm 2: Permutation Scrambling

Permutation involves rearranging data elements rather than changing their values directly.

- **Sample-Level Permutation:** This approach divides the audio stream into blocks (frames) and reorders the samples *within* each block according to a permutation sequence.[2] The permutation sequence can be fixed or generated pseudo-randomly (e.g., using an LFSR to determine swaps). This primarily affects the temporal order and can reduce intelligibility, but may introduce audible artifacts at block boundaries if not handled carefully.
- **Bit-Level Permutation:** This method operates *within* each audio sample, shuffling the positions of the bits.[44] For example, the upper and lower bytes of a 16-bit sample could be swapped, or a more complex, key-dependent bit-shuffling pattern could be applied. This directly distorts the value of each sample, potentially making the output sound more like noise or heavy distortion.

C

```
// Conceptual example: Simple fixed bit permutation within a 16-bit sample
unsigned short permute_bits_fixed(unsigned short sample) {
    unsigned short output = 0;
    // Example 1: Swap upper and lower bytes
    // output = (sample << 8) | (sample >> 8);

    // Example 2: Simple rotation
    // output = (sample << 3) | (sample >> (16 - 3));

    // Example 3: More complex fixed shuffle (illustrative)
    // output |= (sample & 0x0001) << 15; // Move bit 0 to 15
    // output |= (sample & 0x8000) >> 15; // Move bit 15 to 0
    // output |= (sample & 0x0010) << 7;  // Move bit 4 to 11
    // output |= (sample & 0x0800) >> 7;  // Move bit 11 to 4
    //... (define permutation for all bits)...

    // For simplicity, let's just swap bytes
    output = (sample << 8) | (sample >> 8);
    return output;
}

// Conceptual example: Key-dependent bit shift [44]
unsigned short permute_bits_keyed(unsigned short sample, unsigned short key_value) {
    // Use key_value (e.g., from LFSR) to determine shift amount
    int shift_amount = key_value % 16; // Ensure shift is within 0-15
```

```
    unsigned short output = (sample << shift_amount) | (sample >> (16 -
shift_amount));
    return output;
}
```

- **FFT Coefficient Permutation:** This technique transforms the audio into the frequency domain using an FFT, permutes the frequency coefficients based on a key, and then transforms back using an IFFT.[42] While potentially effective at scrambling, the computational cost of FFT/IFFT operations is significantly higher than simple time-domain methods, making real-time sample-by-sample implementation challenging on the C6713. It might be feasible for frame-based processing but pushes the definition of "simple."
- **Security:** Permutation primarily obscures the signal structure. Simple, fixed permutations are easily reversed. Key-dependent permutations offer more security but are still vulnerable to statistical analysis, especially sample-level permutations. Bit-level permutation can be more effective at distortion but might still retain some characteristics of the original signal if the permutation is simple.

4.4 Considerations for DSP Implementation

- **Fixed-Point Arithmetic:** Audio data from the AIC23 codec is typically 16-bit signed integers.[39] Since XOR and bit permutations are inherently integer/bitwise operations, performing them directly on the fixed-point samples is the most natural and efficient approach on the C6713. Converting to floating-point for these specific operations would add unnecessary overhead, despite the C6713's strong floating-point capabilities.[4]
- **Endianness:** The C6713 supports both Little Endian and Big Endian modes.[7] The DSK environment usually defaults to one (often Little Endian). Consistency is vital when performing byte-level operations (like XORing byte-by-byte) or bit manipulations across byte boundaries within a sample. Ensure the algorithm implementation correctly interprets the byte order of samples and keys.

The fundamental challenge lies in the trade-off between the simplicity required for real-time DSP implementation and the level of security achieved.[1] Computationally cheap algorithms like basic XOR with an LFSR or fixed bit permutations offer minimal protection against anything beyond casual listening. The effectiveness of XOR relies entirely on the unpredictability and secrecy of the key stream; a simple LFSR provides a predictable stream.[1] Permutation methods obscure the data differently: sample-level permutation affects temporal structure, while bit-level permutation distorts individual sample values.[2] FFT permutation offers frequency-domain scrambling but at a much

higher computational cost.[42]

4.5 Comparison of Simple Encryption Methods

The following table summarizes the characteristics of the discussed simple methods in the context of implementation on the TMS320C6713:

Table 4.1: Comparison of Simple Audio Encryption Methods for TMS320C6713

Feature	XOR w/ LFSR Key Stream	Bit-Level Permutation	Sample-Level Permutation	FFT Coefficient Permutation
Core Operation	Bitwise XOR	Bit shuffling within sample	Sample reordering within frame	FFT -> Permute Coeffs -> IFFT
Key Requirement	LFSR Seed & Polynomial	Permutation pattern (fixed or keyed)	Permutation sequence (fixed or keyed)	Permutation sequence (fixed or keyed)
Est. Complexity	Low	Low	Low (addressing), Med (if complex key)	High (FFT/IFFT)
Memory Needs	Low (LFSR state)	Low	Med (Frame buffer, key sequence)	High (FFT buffers, coeff buffer)
Pros	Very Fast, Simple	Fast, Simple, Distorts sample value	Can reduce intelligibility	Frequency domain scrambling
Cons	Very Low Security (key stream crucial)	Low Security, Reversible	Low Security, Block artifacts possible	High CPU load, Latency
Real-Time Suitability	High (Sample-by-sample)	High (Sample-by-sample)	Medium (Frame-based likely needed)	Low (Frame-based essential)

5. Integrating Encryption into the Real-Time Audio Framework

Successfully implementing audio encryption requires integrating the chosen algorithm (from Section 4) into the real-time audio input/output framework (established in Section 3.4) while respecting the processing constraints of the TMS320C6713.

5.1 Implementation Strategies: Sample vs. Frame

Two primary strategies exist for processing the audio stream in real-time:

- **Sample-by-Sample Processing:** In this approach, each incoming audio sample (or stereo pair) is processed individually immediately after being read from the codec and before being written back.
    - **Integration:** The encryption/decryption function call is placed directly within the main audio loop (if using polling) or inside the Interrupt Service Routine (ISR) that handles the audio data ready interrupt.
    - **Pros:** Lowest possible latency, simple control flow. Suitable for very simple algorithms like XOR or fixed bit permutation where the processing per sample is minimal.
    - **Cons:** Can be inefficient if the algorithm has per-sample overhead or if polling is used (wasting CPU cycles). May not leave enough time for complex algorithms between samples, especially at higher sample rates.
- **Frame-Based Processing:** This strategy involves collecting a block (frame) of multiple audio samples into a buffer before processing the entire buffer at once.
    - **Integration:** Requires buffer management, often using techniques like ping-pong buffering where one buffer is filled (e.g., by EDMA) while the CPU processes the other previously filled buffer. The encryption/decryption function operates on the entire buffer. EDMA is highly recommended for filling/emptying buffers efficiently without CPU intervention.[17]
    - **Pros:** Can be more efficient for algorithms with initialization overhead or those optimized for block processing (e.g., FFT-based methods, though likely too complex here). Amortizes function call overhead. Maximizes CPU availability for the algorithm by using EDMA for I/O. Essential for higher sample rates or more complex algorithms.
    - **Cons:** Introduces latency equivalent to the frame size (e.g., a 256-sample frame at 48kHz introduces 5.33ms latency). Requires more complex buffer management and EDMA configuration.

The choice between these strategies is critical. Sample-by-sample is easier to start with for basic XOR or permutation, but frame-based processing using EDMA becomes necessary as algorithm complexity increases or higher sample rates (like 44.1kHz or 48kHz) are used, to ensure the processing completes within the available time

budget.[23]

5.2 Locating the Encryption/Decryption Code

The encryption or decryption function call must be inserted at the appropriate point in the audio data flow:

- **Encryption (e.g., transmitting or recording):** Apply the encryption algorithm *after* reading the plain audio sample/frame from the input (codec ADC or memory) and *before* writing the resulting ciphertext sample/frame to the output (codec DAC or memory/network).
- **Decryption (e.g., receiving or playback):** Apply the decryption algorithm *after* reading the ciphertext sample/frame from the input (codec ADC, memory, or network) and *before* writing the resulting plain audio sample/frame to the output (codec DAC or memory).

A mechanism (e.g., a global flag, DIP switch input) should control whether the system operates in encryption or decryption mode, ensuring the correct function is called.

5.3 Illustrative Code Snippets (Integration)

These snippets show where hypothetical encryption/decryption functions would fit into the frameworks:

C

```c
// --- Sample-by-sample XOR Integration (Polling BSL framework from Sec 3.4) ---
#include "dsk6713.h"
#include "dsk6713_aic23.h"
#include "encrypt_decrypt.h" // Assume this header defines functions below

//... (LFSR state, config, handle declarations as before)...

void main() {
    union { Uint32 uint32; short channel[2]; } data;
    union { Uint32 uint32; short channel[2]; } processed_data;
    unsigned short key_word; // Assuming 16-bit key for stereo pair XOR

    //... (DSK_init, AIC23_openCodec, AIC23_setFreq as before)...
```

```
    // Initialize encryption module (e.g., seed LFSR)
    initialize_encrypt_decrypt();

    for(;;) {
        while(!DSK6713_AIC23_read(hCodec, &data.uint32));

        // Generate the next key element (e.g., 16 or 32 bits)
        key_word = generate_key_word(); // From encrypt_decrypt module

        // Apply XOR encryption/decryption to the 32-bit stereo pair
        // Assumes encrypt_sample and decrypt_sample perform the XOR
        // A mode switch (e.g., based on DIP switch) would select which function to call
        if (/* encryption_mode */) {
            processed_data.uint32 = encrypt_sample_xor(data.uint32, key_word);
        } else { // decryption_mode
            processed_data.uint32 = decrypt_sample_xor(data.uint32, key_word);
        }

        while(!DSK6713_AIC23_write(hCodec, processed_data.uint32));
    }
}
```

C

```
// --- Frame-based Integration (Conceptual EDMA callback/task) ---
#include "encrypt_decrypt.h" // Assume this header defines functions below

#define FRAME_SIZE 256 // Example frame size

// Assume EDMA ping-pong buffers: ping_buffer, pong_buffer
short ping_buffer; // *2 for stereo
short pong_buffer;
short key_buffer; // Buffer for key stream if needed

volatile int active_buffer = 0; // Buffer currently being processed by CPU

// Function called by framework when a buffer is filled by EDMA and ready for processing
```

```
void process_audio_buffer() {
    short* buffer_to_process;

    if (active_buffer == 0) {
        buffer_to_process = ping_buffer;
    } else {
        buffer_to_process = pong_buffer;
    }

    // Generate key stream for the whole buffer if needed by algorithm
    // generate_key_stream(key_buffer, FRAME_SIZE * 2);

    // Apply encryption/decryption to the entire buffer
    // Again, a mode switch would select encrypt vs. decrypt
    if (/* encryption_mode */) {
        encrypt_buffer(buffer_to_process, FRAME_SIZE * 2 /*, key_buffer */);
    } else {
        decrypt_buffer(buffer_to_process, FRAME_SIZE * 2 /*, key_buffer */);
    }

    // Signal EDMA to start transmitting the processed buffer
    // setup_edma_transmit(buffer_to_process, FRAME_SIZE * 2);

    // Switch active buffer for next iteration
    active_buffer = 1 - active_buffer;
}

// EDMA completion ISR would typically trigger process_audio_buffer (e.g., via SWI or semaphore)
// void edma_completion_isr() {
//     // Signal that a buffer is ready for processing
//     // Post semaphore or SWI to run process_audio_buffer
// }
```

5.4 Performance Analysis and Real-Time Feasibility

Ensuring the chosen encryption algorithm can run in real-time is paramount. This requires analyzing its computational load against the available processing time.

- **Cycle Counting:** The number of CPU clock cycles required by the encryption/decryption function per sample or per frame must be determined. This can be estimated by analyzing the C code and generated assembly, or measured

more accurately using the CCS profiler (though profiling itself adds overhead) or by toggling a GPIO pin before and after the function call and measuring the duration with an oscilloscope. Instruction latencies on the C67x vary; most fixed-point operations take 1 cycle, but loads, branches, and floating-point operations can take longer (e.g., 1-10 cycles for FP ops).[8]

- **Time Budget:** The maximum time allowed for processing each sample is determined by the sample rate: Time per sample = 1 / SampleRate. For 8 kHz, this is 125 µs; for 48 kHz, it's ~20.8 µs.[23]
- **Available Cycles:** The number of CPU cycles available per sample period is: Cycles per sample = ClockSpeed / SampleRate. For a 225 MHz C6713 at 48 kHz: 225,000,000 / 48,000 ≈ 4687 cycles/sample.
- **Feasibility Check:** The total cycles consumed per sample period (including audio I/O handling *and* the encryption/decryption algorithm) must be less than the Available Cycles per sample. If AlgorithmCycles + I/O_OverheadCycles > AvailableCycles, the system cannot operate in real-time and will experience buffer overflows, glitches, or incorrect output.[39] The high MIPS/MFLOPS rating of the C6713 provides a substantial cycle budget [4], but this budget shrinks rapidly at higher sample rates.
- **Memory Usage:** The algorithm's memory footprint must also be considered. This includes:
    - **Code Size:** Does the compiled algorithm fit comfortably within internal memory (L2 SRAM/Cache) for optimal performance?
    - **Data Size:** How much memory is needed for keys, LFSR states, lookup tables, or temporary variables?
    - **Buffer Size:** Frame-based processing requires buffer memory (e.g., two frames for ping-pong buffering). For 256 stereo samples (16-bit) per frame, each buffer needs 256 * 2 * 2 = 1024 bytes. The C6713 has 256KB of L2 memory/cache and the DSK has external SDRAM.[7] Keeping critical code and data on-chip is crucial for performance.[8]

For the simple algorithms discussed (XOR, bit permutation), the cycle counts per sample are expected to be very low. An XOR operation is typically a single-cycle instruction. Bit manipulation might take a few cycles depending on the complexity. LFSR updates also require only a few shifts and XORs per bit generated. Therefore, these simple algorithms are highly likely to be feasible even at high audio sample rates (e.g., 48kHz or 96kHz) on the 225MHz C6713 when implemented efficiently, especially if using integer operations directly on the 16-bit fixed-point samples from the codec.[39] Converting to floating-point for these bitwise operations would add significant,

unnecessary overhead and should be avoided.

5.5 Optimization

While simple algorithms may not require extensive optimization, good practices include:

- **Compiler Optimization:** Utilize the optimization levels provided by the TI C compiler (-O2 or -O3), balancing code size and speed.
- **Data Types:** Use the smallest appropriate integer types (e.g., unsigned short for 16-bit samples, unsigned char for bytes) for bitwise operations.
- **Efficient C Code:** Write clear C code, avoiding excessive function calls within tight loops, using loop unrolling where appropriate, and leveraging pointers effectively.
- **Memory Placement:** Use the linker command file (.cmd) and #pragma directives to place time-critical functions and data in internal L2 memory.
- **Assembly/Intrinsics:** For highly critical sections, consider writing optimized linear assembly or using C6000 compiler intrinsics (special functions mapping directly to efficient DSP instructions), although this may not be necessary for the simplest algorithms.

6. System Testing and Verification

Thorough testing is essential to ensure the audio encryption/decryption system functions correctly, meets performance requirements, and achieves the desired level of basic confidentiality.

6.1 Functional Testing

- **Loopback Integrity Test:** The most fundamental test involves feeding an audio signal (e.g., sine wave, music clip) into the DSK, running the system with encryption immediately followed by decryption (or in a loopback configuration where the encrypted output is fed back as input for decryption), and comparing the final decrypted output to the original input signal.
- **Bit-Level Comparison:** For lossless algorithms like XOR and permutation, the decrypted output must be *identical* to the original input. This can be verified by:
    - Capturing blocks of original input samples and decrypted output samples in memory buffers within the DSP program.
    - Performing a bitwise comparison of the buffers.
    - Alternatively, saving the original and decrypted data to files (e.g., via CCS file I/O or by transferring buffers to the host PC) and comparing them offline using file comparison tools. Any difference indicates an error in the encryption

or decryption implementation.

- **Separate Encryption/Decryption Test:** To isolate problems, test the encryption process independently by inputting audio, encrypting it, and saving/analyzing the encrypted output. Then, load known encrypted data and test the decryption process separately, comparing the result to the known original plaintext.

6.2 Subjective Testing

Since the goal is often to reduce intelligibility rather than achieve cryptographic strength, subjective listening tests are crucial:

- **Encrypted Audio Assessment:** Listen carefully to the *encrypted* audio output. Does it sound sufficiently scrambled or masked? Is the original speech or music unintelligible? The level of residual intelligibility determines the effectiveness of the scrambling.[1] Test with various input types (speech, different music genres).
- **Decrypted Audio Quality:** Listen carefully to the *decrypted* audio output (after the loopback test). Are there any audible artifacts like clicks, pops, noise, distortion, or glitches introduced by the encryption/decryption process? The decrypted audio should ideally sound identical to the original input.

Objective tests confirm reversibility, while subjective tests evaluate the practical effectiveness (scrambling quality) and the impact on audio fidelity. Both are necessary for simple audio encryption systems where perceptual masking is the primary goal.

6.3 Performance Verification

- **Real-Time Operation Check:** Monitor the system during operation at the target sample rate. Check for signs of failure to keep up, such as audio glitches, dropouts, or buffer overflow indicators (if using frame-based processing). Toggling an LED or GPIO pin within the main processing loop and observing it with an oscilloscope can provide a visual indication of whether processing completes within the sample period. A stable toggle rate indicates real-time operation, while irregular timing suggests the system is struggling.
- **CPU Load Measurement:** Estimate or measure the CPU load. While the CCS profiler can provide cycle counts, its intrusiveness might disrupt real-time behavior. A less intrusive method is to use a hardware timer (one of the C6713's general-purpose timers) to measure the execution time of the core processing section (encryption/decryption) and compare it to the total time available per sample (1 / SampleRate). CPU Load ≈ (Measured Execution Time) / (Time per Sample). This verifies that sufficient headroom exists.

6.4 Debugging Strategies

- **Isolate and Conquer:** Test the audio I/O framework (simple pass-through loopback) thoroughly before introducing any encryption/decryption code.[34] Test the encryption/decryption algorithm logic separately using known test vectors in a non-real-time environment (e.g., a separate CCS project or even on the host PC) before integrating it into the real-time framework.
- **Simplify Inputs:** Begin testing with simple, predictable audio inputs like pure sine waves generated digitally or from a function generator. This makes it easier to spot distortions or errors.
- **Instrumentation (Use Sparingly):**
  - **LEDs/GPIO:** Use the DSK's user LEDs or GPIO pins connected to an oscilloscope to signal specific events (e.g., entering/exiting a function, buffer full/empty) or indicate status. This provides low-overhead visibility into program flow.
  - **printf:** Use printf statements directed to the CCS console output window for displaying variable values or status messages during debugging. However, be aware that printf is computationally expensive and can significantly disrupt real-time timing; use it sparingly and remove it from the final real-time code.
- **Memory Inspection:** Utilize the CCS debugger's Memory Browser window to examine the contents of input buffers, output buffers, key streams, and LFSR state variables while the program is halted. This helps verify data integrity at different stages.
- **Step-by-Step Execution:** Use the debugger to step through the code line by line, especially around the encryption/decryption functions and I/O calls, to observe the program state changes.

Effective verification requires a combination of functional checks, perceptual evaluation, and performance measurement to ensure the system works correctly, achieves its basic confidentiality goal, and operates reliably in real-time.

7. Conclusion and Recommendations

7.1 Summary of Process

This report has outlined the key considerations for developing a simple DSP-based audio encryption and decryption system on the TI TMS320C6713 platform. The process involves understanding the C6713's VLIW architecture, memory hierarchy, and audio-centric peripherals (McASP, McBSP, EDMA), particularly within the context of the DSK hardware which utilizes McBSPs and the AIC23 codec. Development leverages Code Composer Studio for project management, compilation, and

debugging, along with essential support libraries (RTS, CSL, BSL). The BSL simplifies initial audio I/O setup. Simple encryption techniques like XOR with LFSR-generated key streams and bit/sample permutation were explored, focusing on their low computational cost suitable for real-time implementation. Integrating these algorithms requires choosing between sample-by-sample or frame-based (often with EDMA) processing strategies, carefully analyzing the cycle budget dictated by the sample rate and DSP clock speed, and performing thorough functional, subjective, and performance testing.

7.2 Security Limitations

It must be reiterated that the simple encryption techniques discussed (basic XOR, simple permutations) offer **minimal cryptographic security**.[1] They are easily broken by standard cryptanalysis techniques (e.g., known-plaintext attacks for XOR, statistical analysis for permutation) and should **not** be used for protecting sensitive or high-value information. Their applicability is limited to scenarios requiring only basic confidentiality against casual listeners or simple access control mechanisms where robust security is not a primary requirement.

7.3 Performance Recap

The TMS320C6713 provides substantial processing power (up to 1800 MFLOPS / 2400 MIPS) [4], making the implementation of simple, computationally inexpensive algorithms like XOR or bit permutation feasible in real-time, even at standard audio sample rates (e.g., 48 kHz). However, real-time operation is constrained by the fixed number of CPU cycles available per sample period (ClockSpeed / SampleRate).[39] Careful implementation, efficient I/O handling (potentially using EDMA for higher rates), and adherence to the cycle budget are crucial. Operating directly on fixed-point integer samples for bitwise encryption operations is recommended for efficiency.

7.4 Recommendations and Future Work

- **Incremental Development:** Start with the simplest implementation: use the BSL polling framework (Section 3.4) and integrate a basic XOR encryption with a simple LFSR. Verify functionality and real-time operation at a lower sample rate (e.g., 8 kHz). Incrementally increase complexity by moving to higher sample rates, implementing interrupt-driven or EDMA-based I/O if needed, or exploring slightly more complex algorithms.
- **Improved PRNG for XOR:** If slightly better (but still not cryptographically strong) security is desired for XOR, investigate more robust pseudo-random number generation techniques. This could involve using longer LFSRs, non-linear combining functions for multiple LFSR outputs, or potentially seeding LFSRs using

values derived from cryptographic hash functions (though hashing itself adds computational load).

- **Consider Standard Cryptography (with caveats):** For applications requiring genuine security, standard algorithms like AES are necessary.[3] However, implementing AES efficiently in software on the C6713 to achieve real-time performance for audio sample rates can be very challenging due to its computational complexity. It would likely require significant optimization efforts (assembly language, intrinsics), frame-based processing, and potentially accepting lower sample rates or higher latency. Hardware acceleration for such algorithms is not available on the C6713.
- **Explore McASP:** For applications requiring more advanced audio interfacing than provided by the DSK's AIC23/McBSP setup (e.g., higher channel counts, different TDM formats, S/PDIF output), investigate utilizing the C6713's onboard McASP peripherals directly.[7] This requires using the CSL or direct register programming instead of the DSK BSL audio functions.

By following a structured approach, leveraging the capabilities of the TMS320C6713 and its development tools, and understanding the inherent trade-offs between simplicity, security, and real-time performance, a functional basic audio encryption/decryption system can be successfully implemented on this platform.

Works cited

1. www.comp.nus.edu.sg, accessed April 8, 2025, https://www.comp.nus.edu.sg/~mohan/papers/mp3_scram.pdf
2. Permutation based speech scrambling for next generation mobile communication - Engg Journals Publications, accessed April 8, 2025, https://www.enggjournals.com/ijet/docs/IJET16-08-02-023.pdf
3. A Study on Current Scenario of Audio Encryption - ResearchGate, accessed April 8, 2025, https://www.researchgate.net/publication/276129131_A_Study_on_Current_Scenario_of_Audio_Encryption
4. TMS320C6713B data sheet, product information and support | TI.com, accessed April 8, 2025, https://www.ti.com/product/TMS320C6713B
5. SM320C6713B-EP data sheet, product information and support | TI.com - Texas Instruments, accessed April 8, 2025, https://www.ti.com/product/SM320C6713B-EP
6. TMS320C6713 Digital Signal Processor Optimized for High Performance Multichannel Audio Systems - Digchip, accessed April 8, 2025, http://application-notes.digchip.com/001/1-1920.pdf
7. TMS320C6713B | Buy TI Parts | TI.com - Texas Instruments, accessed April 8, 2025, https://www.ti.com/product/TMS320C6713B/part-details/TMS320C6713BGDP300

8. A BDTI Analysis of the Texas Instruments TMS320C67x, accessed April 8, 2025, https://www.ti.com/lit/pdf/sprt280
9. TMS320C6713 Floating-Point Digital Signal Processor (Rev. L - uri=media.digikey, accessed April 8, 2025, https://media.digikey.com/pdf/Data%20Sheets/Texas%20Instruments%20PDFs/TMS320C6713.pdf
10. Lab 1 Introduction to TI's TMS320C6713 DSK Digital Signal Processing Board ∗, accessed April 8, 2025, http://www.ee.nmt.edu/~rene/fall_12/ee451/lab01_fall_12.pdf
11. Tms320c6713 User Manual | PDF | Digital Signal Processor - Scribd, accessed April 8, 2025, https://www.scribd.com/document/323605799/Tms320c6713-User-Manual
12. TMS320C6713BZDP300 Texas Instruments - Mouser Electronics, accessed April 8, 2025, https://www.mouser.com/ProductDetail/Texas-Instruments/TMS320C6713BZDP300?qs=64toYSwBvjzVCH2WiwIt3g%3D%3D
13. TMS320C6713 Floating-Point Digital Signal Processor, accessed April 8, 2025, https://www.zaikostore.com/jsp/pdf/PDFFILE/TI/FIL0036343.PDF
14. How I can get the digital/analog output from TMS320C6713? - ResearchGate, accessed April 8, 2025, https://www.researchgate.net/post/How_I_can_get_the_digital_analog_output_from_TMS320C67132
15. Spectrum Digital TMS320C6713 DSK Module - Price, Specs - ArtisanTG, accessed April 8, 2025, https://www.artisantg.com/TestMeasurement/97522-1/Spectrum-Digital-TMS320C6713-DSK-Module
16. spinlab.wpi.edu, accessed April 8, 2025, https://spinlab.wpi.edu/courses/ece4703_2010/tms320c6713dsk_technical_reference.pdf
17. www.ti.com.cn, accessed April 8, 2025, https://www.ti.com.cn/cn/lit/an/spra677/spra677.pdf
18. Spectrum Digital: C6713 DSP Kit (DSK) for TI TMS320C6713 - element14 Community, accessed April 8, 2025, https://community.element14.com/products/devtools/technicallibrary/w/documents/9302/spectrum-digital-c6713-dsp-kit-dsk-for-ti-tms320c6713
19. On the Use of Real-Time DSP Framework Code for Laboratory Instruction on the TI TMS320C6713 DSK - MSU College of Engineering, accessed April 8, 2025, https://www.egr.msu.edu/~gunn/ASEE%20North%20Central%202007/Dunne(D2-2).pdf
20. Digital Signal Processing and Applications with the TMS320C6713 and the TMS3206416 DSK - spinlab, accessed April 8, 2025, https://spinlab.wpi.edu/courses/dspworkshop/dspworkshop_part1_2006.pdf
21. Spectrum Digital TMS320C6713 DSK Datasheet | ArtisanTG, accessed April 8, 2025, https://www.artisantg.com/info/Spectrum_Digital_TMS320C6713_DSK_Datasheet

_201928121318.pdf
22. Speech Processing using DSP TMS320C6713 - WordPress.com, accessed April 8, 2025, https://2ec402bdf.files.wordpress.com/2015/09/speech-processing-using-dsp6713.pdf
23. Real-time Digital Signal Processing Demonstration Platform - ASEE PEER, accessed April 8, 2025, https://peer.asee.org/real-time-digital-signal-processing-demonstration-platform.pdf
24. DSP Development System - UOP eClass, accessed April 8, 2025, http://old-eclass.uop.gr/modules/document/file.php/TST141/various/ChasChapter01.pdf
25. rt-dsp.com, accessed April 8, 2025, https://rt-dsp.com/3rd_ed/app_a/App_CCS_6_1_DSK6713.pdf
26. APPENDIX A. CODE COMPOSER STUDIO (CCS) v5.1: A BRIEF TUTORIAL FOR THE DSK6713 - RT-DSP, accessed April 8, 2025, https://rt-dsp.com/3rd_ed/app_a/App_CCS_5_1_dsk6713.pdf
27. TMS320C6713 DSP Starter Kit (DSK) Tutorial - YouTube, accessed April 8, 2025, https://www.youtube.com/watch?v=wcp2TzxCHHA
28. How to Install Code Composer Studio | CCS Tutorial for Beginners - YouTube, accessed April 8, 2025, https://www.youtube.com/watch?v=eCHhHlOujA0
29. Digital Signal Processing and Applications with the TMS320C6713 DSK - spinlab, accessed April 8, 2025, https://spinlab.wpi.edu/courses/dspworkshop/dspworkshop_part1_2007.pdf
30. Quick Start Installation Guide - Control Systems Laboratory, accessed April 8, 2025, https://coecsl.ece.illinois.edu/se423/datasheets/dsk6713/6713_dsk_quickstartguide.pdf
31. ECE4703 B Term 2006 Project 1 - spinlab, accessed April 8, 2025, https://spinlab.wpi.edu/courses/ece4703_2006/lab1.pdf
32. Labprogramming The TMS320C6713 DSP Starter Kit (DSK) Module6 - Scribd, accessed April 8, 2025, https://www.scribd.com/document/231103490/labprogramming-the-TMS320C6713-DSP-Starter-Kit-DSK-module6
33. Contents, accessed April 8, 2025, https://user.eng.umd.edu/~tretter/commlab/c6713slides/ch2.pdf
34. C6713 DSK audio loopback - Processors forum - TI E2E, accessed April 8, 2025, https://e2e.ti.com/support/processors-group/processors/f/processors-forum/281253/c6713-dsk-audio-loopback
35. AIC23 Codec - spinlab - Worcester Polytechnic Institute, accessed April 8, 2025, https://spinlab.wpi.edu/courses/ece4703_2009/lecture1.pdf
36. how to take input from mic in channel in DSK6713? - Processors forum - TI E2E, accessed April 8, 2025, https://e2e.ti.com/support/processors-group/processors/f/processors-forum/256268/how-to-take-input-from-mic-in-channel-in-dsk6713

37. ECE4703 Laboratory Assignment 1 - spinlab, accessed April 8, 2025, https://spinlab.wpi.edu/courses/ece4703/lab1.pdf
38. Using the ADS8380 with the TMS320C6713 DSP - Texas Instruments, accessed April 8, 2025, https://www.ti.com/lit/pdf/slaa240
39. Implementation of Real Time Programs on the TMSC6713DSK Processor - ResearchGate, accessed April 8, 2025, https://www.researchgate.net/publication/232237591_Implementation_of_Real_Time_Programs_on_the_TMSC6713DSK_Processor
40. Implementation of Audio Signal processing Application using TMS320C6713 - ijirset, accessed April 8, 2025, http://www.ijirset.com/upload/2016/may/139_Implementation.pdf
41. Progressive Audio Scrambling in Compressed Domain - NUS Computing, accessed April 8, 2025, https://www.comp.nus.edu.sg/~mohan/papers/audscram.pdf
42. (PDF) Speech encryption based on fast Fourier transform permutation, accessed April 8, 2025, https://www.researchgate.net/publication/3890894_Speech_encryption_based_on_fast_Fourier_transform_permutation
43. A Multilayered Audio Signal Encryption Approach for Secure Voice Communication - MDPI, accessed April 8, 2025, https://www.mdpi.com/2079-9292/12/1/2
44. A Novel Audio Encryption Algorithm with Permutation-Substitution ..., accessed April 8, 2025, https://www.mdpi.com/2079-9292/8/5/530
45. adsp | how to implement LFSR in C? - DSPRelated.com, accessed April 8, 2025, https://www.dsprelated.com/showthread/adsp/2785-1.php
46. Efficient CRC calculation with minimal memory footprint - Embedded, accessed April 8, 2025, https://www.embedded.com/efficient-crc-calculation-with-minimal-memory-footprint/
47. Tutorial: Linear Feedback Shift Registers (LFSRs) - Part 1 - EE Times, accessed April 8, 2025, https://www.eetimes.com/tutorial-linear-feedback-shift-registers-lfsrs-part-1/
48. An example LFSR - ZipCPU, accessed April 8, 2025, https://zipcpu.com/dsp/2017/11/11/lfsr-example.html
49. Generating Maximum Length Sequence using Galois LFSR| page 2 - DSPRelated.com, accessed April 8, 2025, https://www.dsprelated.com/showthread/comp.dsp/114924-2.php
50. Linear-feedback shift register implementation - Stack Overflow, accessed April 8, 2025, https://stackoverflow.com/questions/74882061/linear-feedback-shift-register-implementation
51. Linear Feedback Shift Registers for the Uninitiated, Part VII: LFSR Implementations, Idiomatic C, and Compiler Explorer - Jason Sachs - EmbeddedRelated.com, accessed April 8, 2025, https://www.embeddedrelated.com/showarticle/1112.php

52. Linear Feedback Shift Registers for the Uninitiated, Part VIII: Matrix Methods and State Recovery - Jason Sachs - EmbeddedRelated.com, accessed April 8, 2025, https://www.embeddedrelated.com/showarticle/1114.php
53. How to Begin Development Today With the TMS320C6713 Floating-Point DSP - Texas Instruments, accessed April 8, 2025, https://www.ti.com/lit/pdf/spra809
54. Efficient Audio Encryption Algorithm for online applications using Hybrid Transposition and Multiplicative Non Binary System | Request PDF - ResearchGate, accessed April 8, 2025, https://www.researchgate.net/publication/315619083_Efficient_Audio_Encryption_Algorithm_for_online_applications_using_Hybrid_Transposition_and_Multiplicative_Non_Binary_System