MOTOROLA
Semiconductor Products Sector Engineering Bulletin

AN2113/D:
Rev. 0, 3/2001

Multichannel Voice Coding System on the RTXC Operating System

By Duberly Mazuelos, Felicia Benavidez, Iantha Scheiwe

DSP applications are moving away from assembly language and home-grown scheduling kernels to systems developed using high-level languages and running on off-the-shelf Real-Time Operating Systems (RTOSs). Assembly programming requires intimate knowledge of the device architecture and prohibits easy portability to a new architecture if cost or availability change. C programming is becoming more commonplace in the DSP market because of pressures for a fast time-to-market, low cost, and reusability. Also, C compiler technology is finally maturing to a point where the inherent benefits of a DSP architecture can be realized in the C language.

Engineers designing and programming complex systems containing DSPs have long relied on their own scheduler to determine when tasks should be handled in an application. These schedulers are often developed in-house and are application-specific. As the complexity of the systems increases, the complexity of the scheduler also increases, and the task of designing and implementing these schedulers becomes a significant portion of the system development time. However, RTOSs are available to ease the task of system integration and provide the scheduling tasks necessary to meet stringent application requirements.

Various telecommunications standards dictate specific voice coders for each telecommunications application. Because these voice coders are standard building-blocks in a system, third parties have come forward to develop highly optimized assembly language implementations of voice coders for customer use.

This application note describes a multichannel voice coding system developed in C and executing on an RTOS. The voice coding software from a third party is integrated with other tasks under the RTXC RTOS. Topics covered include the voice coding application, features of the RTXC RTOS, and a methodology for integrating this type of system. This knowledge can assist you in developing future systems using an RTOS.

Contents

1 Project Purpose

In this project, we explore the issues that arise when third-party software is integrated with an RTOS and other application software in a C language development environment. This project has better enabled us to support Motorola customers by providing in-house experience with the following:

- Voice coders/decoders
- Multichannel application
- C and assembly coding and integration
- RTOSs

The system described in this application note executes multiple channels of the IS-96-A voice coder on a Motorola DSP56307EVM. Using a variety of software tools and code, we experienced the intricacies that arise from such an approach. We developed and debugged the application software using the Tasking 2.2r2 DSP563xx Software Development Toolset. C code snippets shown in this document adhere to guidelines set forth by the 2.2r2 version of the Tasking tool set. Hand-coded assembly code interfaces to the IS-96-A[1] voice coding library supplied by Signals and Software Limited (SASL). Embedded Power Corporation’s RTXC is the underlying RTOS. The result of this project is a demo that is shown at Motorola booths in conferences and other technical events.

2 Voice Coding

Voice coders, often called *vocoders*, are technically a subset of the entire range of voice coding technology. The word “vocoder” stands for *voice coder/decoder*. Voice coders in infrastructure systems compress the voice data for transmission. Voice coders have evolved from low compression capabilities to high compression capabilities while retaining quality sound output. With improvements in voice coding technology, we can now compress voice data to 6.3 kbps or even better with toll quality results. *Toll quality* refers to the quality of sound that humans hear with analog speech. Ideally, a person cannot tell the difference between analog speech and digitized speech coming over the air interface. Through improvements in compression technology, this is becoming a reality even with high compression ratios. Voice coding algorithms are evaluated in terms of four metrics:

- *Subjective quality*. Voice coders attempt to achieve toll quality.
- *Bit rate*. The level of compression achieved.
- *Complexity of the algorithm*. Can this algorithm be implemented using today’s technology?
- *End-to-end delay*. If the algorithm adds too much delay to the signal, it becomes unusable in a real-time voice application.

Speech is separated into three categories: voiced, unvoiced, and mixed. Voiced speech is quasi-periodic in the time domain and harmonically structured in the frequency domain. Unvoiced speech is random-like. These categories are studied to determine the most effective method of compression for each. The characteristics of speech are created through the interaction of the vocal box (speech source) and the vocal tract that includes the mouth. This interaction creates the *spectral envelope*. The spectral envelope is

1. TIA/EIA/IS-96-A “Speech Services Option Standard for Wideband Spread Spectrum Digital Cellular System.”

especially important in recreating voiced speech. Peaks of the spectral envelope are called *formants*, which represent the resonant modes of the vocal tract. The research into speech coding and development of voice coders for telecommunication standards focuses heavily on understanding formants.

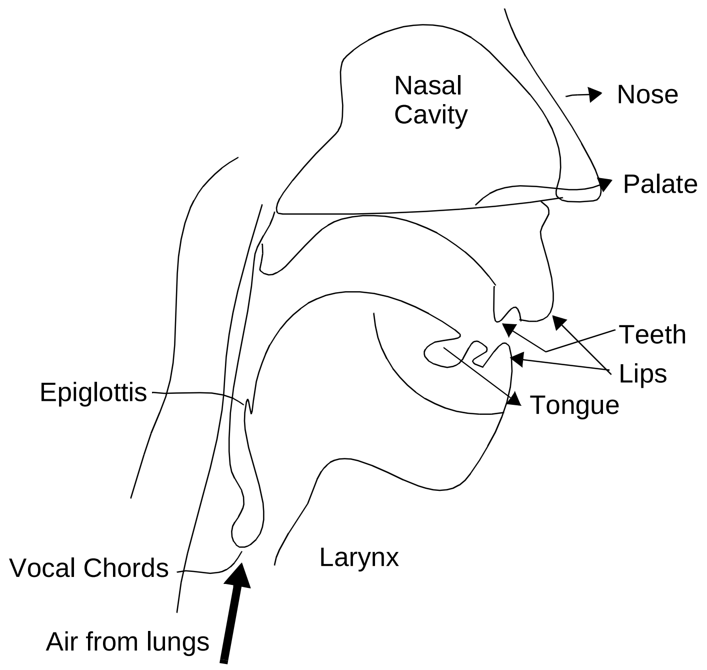


Figure 1. Vocal Tract Structure

As **Figure 1** shows, many features in the vocal tract interact to form the spectral envelope and create the formants. All of these interactions are studied to determine improved methods of compression for voice coding. **Figure 2** shows a more abstract view of the vocal tract. Notice how the space for the sound is more closed at the windpipe and lips and opens up in the vocal tract passage. Also, due to the shape of the passage, two sounds may route very differently in their travel to the lips. Differences in the travel affect the nature of the sound. All of these aspects are studied and considered in the design of voice coders.

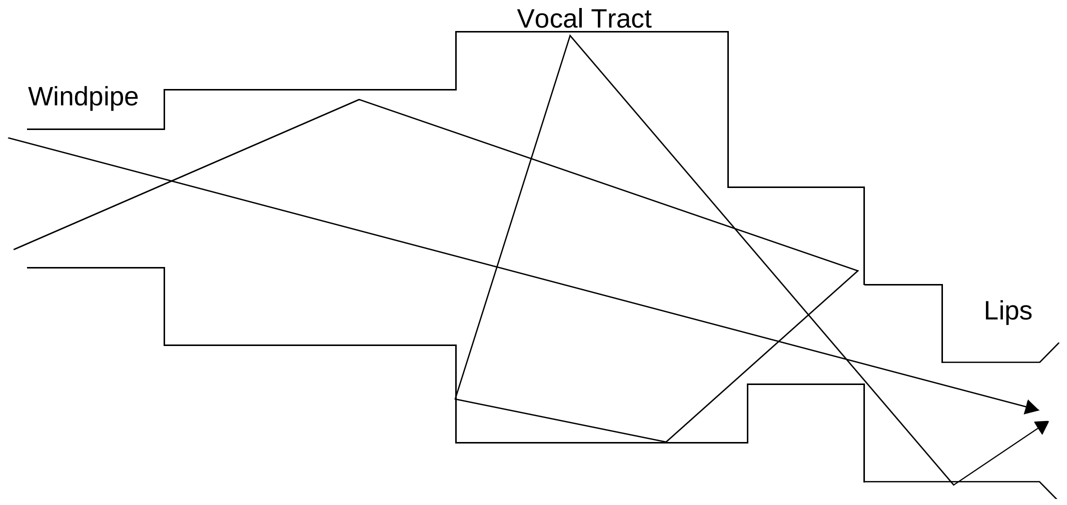


Figure 2. Abstract View of Vocal Tract

2.1 Encoding/Decoding

Each of the many different voice coding algorithms combines knowledge of the human vocal system with an understanding of the quality of speech required by the application and the processing power available to process the given voice coder in a system. The list of algorithm types includes the following:

- Pulse Coded Modulation (PCM)
- Linear Predictive Coding (LPC)
- Code Excited Linear Prediction (CELP)
- Regular Pulse Excitation (RPE)
- Vector Sum Excited Linear Prediction (VSELP)

As **Figure 3** shows, the voice coders in the "vocoder" grouping achieve the lowest bit rate but also tend to have lower subjective quality than the other coders. Perceptual and Waveform coders have high subjective quality, but the application pays a price in bandwidth since the bit rate is not as low. Hybrid coders may achieve lower bit rate and higher subjective quality, making them desirable coders for wireless systems.

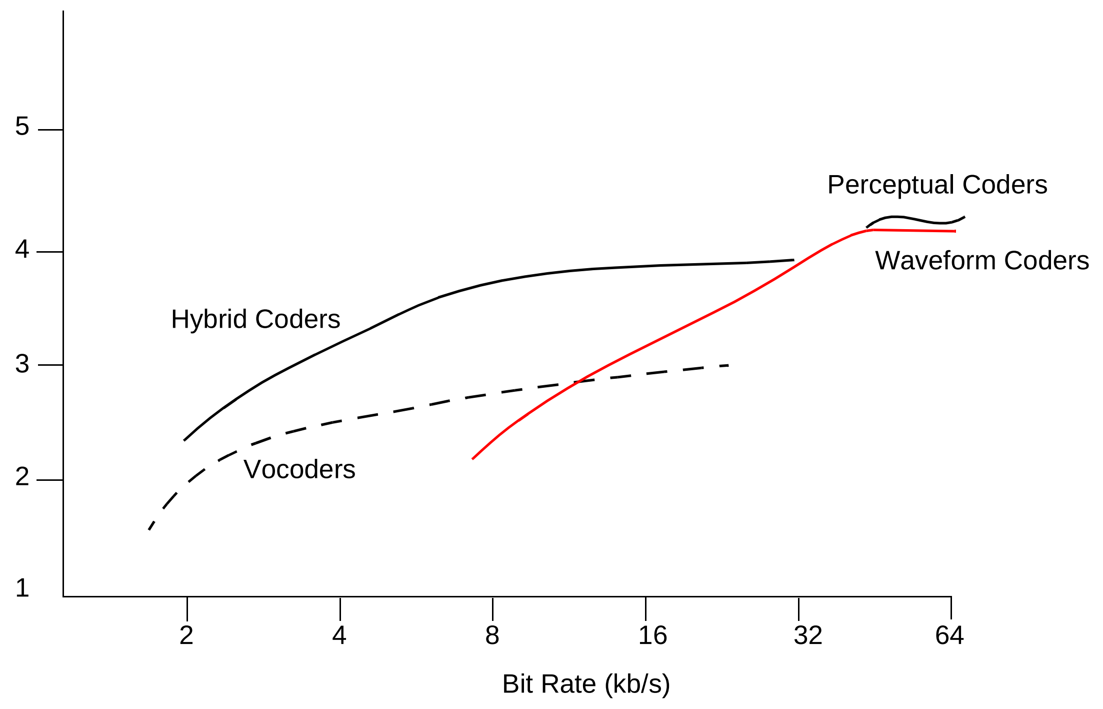


Figure 3. Comparison of Voice Coders

The voice coder used in the project discussed here is IS-96-A, a type of hybrid coder using the CELP algorithm. IS-96-A is used in the Code Division Multiple Access (CDMA) wireless standard and was standardized in 1992 and uses a codebook to assist in the voice compression.[2] The codebook used in these types of voice coders is a table of excitation parameters that define how the vocal tract is stimulated. The decoder determines the appropriate excitation parameter for a given speech sample and usually transmits

2. Motorola provides a detailed discussion of CDMA on the following website: http://www.motorola.com/NSS/Technology/cdma.html.

both a codebook index value indicating which excitation parameter is appropriate and a codebook gain indicating the strength of the excitation. The tables for IS-96-A require 6622 words of data space when implemented on the DSP56300 family.

IS-96-A is a variable-rate coder, so it senses when speech activity lessens and transmits less information during that time. This lowers the bandwidth requirements during inactive speech times and improves the efficiency of the system in general. Therefore, theoretically a given base station can handle more channels based on the statistical knowledge that not every user is at maximum speech data capacity at a given time. Also, depending on the transmission method, the variable rate cuts down the noise for other users. The maximum rate for IS-96-A is 8 kbps and the minimum rates are 4 kbps (1/2 rate), 2 kbps (1/4 rate), and 0.8 kbps (1/8 rate). The encoder sends information to the decoder when it adjusts the rate. It jumps only one rate per given sample time, so the encoder does not send a full transmission rate sample and then sense complete silence and drop down to the minimum transmission rate on the next sample. Instead, it cycles through each rate as appropriate for a given sample time and does not adjust by more than one rate at a time. This behavior helps to maintain the quality of the speech.

The encoder analyzes the input speech and transmits a set group of speech parameters to the decoder. These parameters include coefficients related to the formants that determine the resonant frequencies of the vocal tract at a given time. The encoder also transmits the codebook gain and index, pitch information, and parity check bits. The encoder determines these parameters during the process shown in **Figure 4**. In IS-96-A, the encoder implements a search procedure to recreate the input speech by comparing it to the output of the synthesizer in the encoder. For each received input sequence, the encoder attempts to synthesize the speech, comparing its output with the input speech and calculating a weighted error value. Once this error is sufficiently minimized, the parameters that create the “best” synthesized speech are transmitted to the decoder.

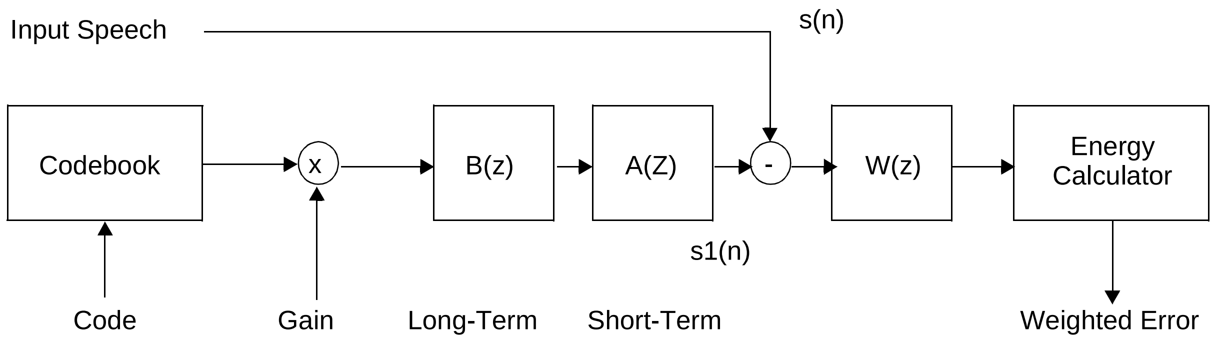


Figure 4. IS-96 Encode Block Diagram

The decoder receives the parameters transmitted by the encoder and reproduces the speech so that the person listening on the receive end can understand the person speaking. The decoding process does not require the analysis of the speech that the encoder must complete, so decoding requires less processing power for a system than encoding. **Figure 5** shows the steps required for decoding in IS-96-A.The first filter, called the *long-term filter*, reconstructs the long-term pitch periodicities of the speech in the excitation signal. The second, called the *short-term filter*, models the spectral shape of the speech.[3]

3. `http://www.msdmag.com/frameindex.htm?98/9806art4.htm`.

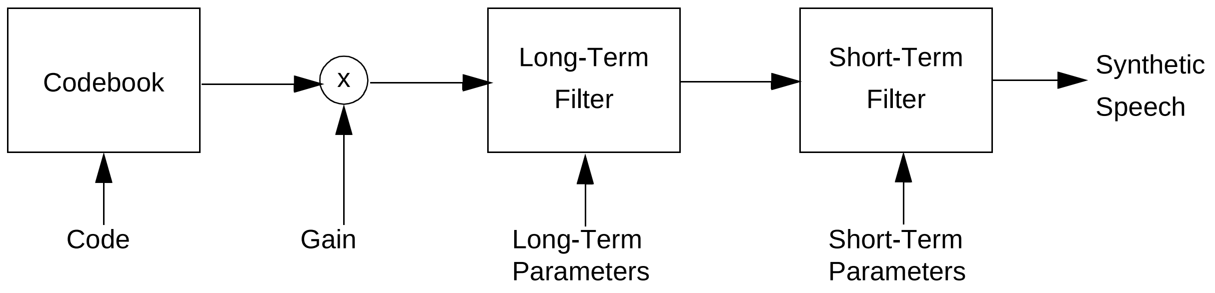


Figure 5. IS-96 Decode Block Diagram

2.2 Third-Party Voice Coding Software

Since voice coders are an integral part of systems implementing various telecommunications standards, it is essential that they be programmed efficiently. Efficient programming requires extensive use of assembly language programming to improve voice coding execution time. The smaller the voice coder execution time, the more voice the coding channels in a given system and the lower the system cost per voice channel. Programming voice coders requires intimate knowledge of both the DSP hardware and the voice coder algorithm itself. In order to develop efficient and timely voice coder solutions, third party companies focus completely on implementing telecommunication standard software on various DSP architectures. Purchasing these software modules from third parties decreases the investment an Original Equipment Manufacturer (OEM) must make in software engineers and technology, and it improves time-to-market. Thus, third parties are used almost exclusively to generate telecommunications standard software. One such company that develops software for the Motorola DSP56300 architecture is Signals and Software Limited (SASL), who developed the IS-96-A voice coder discussed here.

3 Multichannel Applications

The wireless infrastructure market requires powerful DSP architectures for processing multiple channels of information, whether it be voice information or data streams for wireless internet applications. While a subscriber device must process only a single user’s information, this information is then transmitted to a base station that processes the information of many users. Therefore, a single DSP device must have the ability to process multiple channels of information. Moreover, a “farm” of DSPs is often combined in a system to increase channel processing capacity as standards and user applications evolve to include more features for processing.

To determine how many channels a DSP can process, the system designer must look at data I/O capabilities, processing time for each task required by the standards involved (such as voice coding, interleaving, and so on), and task management. Task management is handled deterministically or via an RTOS that decides task processing based on pre-defined priority levels. While some system software remains in assembly code to operate at maximum efficiency on the DSP architecture (probably standard library code such as the voice coders provided by third parties), the use of high-level languages and an RTOS enable the system designer to modify software quickly and thus keep up with evolving standards while managing the increased number of tasks required to implement the system design.

4 C Compilers

Embedded applications are written primarily in C, which is becoming mandatory due to time-to-market constraints, reusability, and portability demands. In the quest for C programmability of DSPs, the key concern is to optimize the DSP code. Every clock cycle is precious because of the real-time environments in which DSPs operate. For example, in a wireless link the DSPs handle filtering, signal encoding, channel encoding, chip rate processing, and the inverse of all these steps within a mandatory time delay that is not noticeable to those involved in the conversation. Today's C compilers are designed to tackle DSP programmability constraints.

DSP C compilers must harness the power of the DSP architecture. Primary DSP features that must be accessible to the programmer include MAC instructions, hardware DO loops, modular addressing, efficient memory access, and parallel operation of computing units. Ideally, the compiler flexibly uses all the DSP assembly instructions to maintain highly efficient code. DSP instructions are designed for efficiency so a C compiler must strive to retain assembly efficiency while adding its own positive attributes, such as programmability and portability.

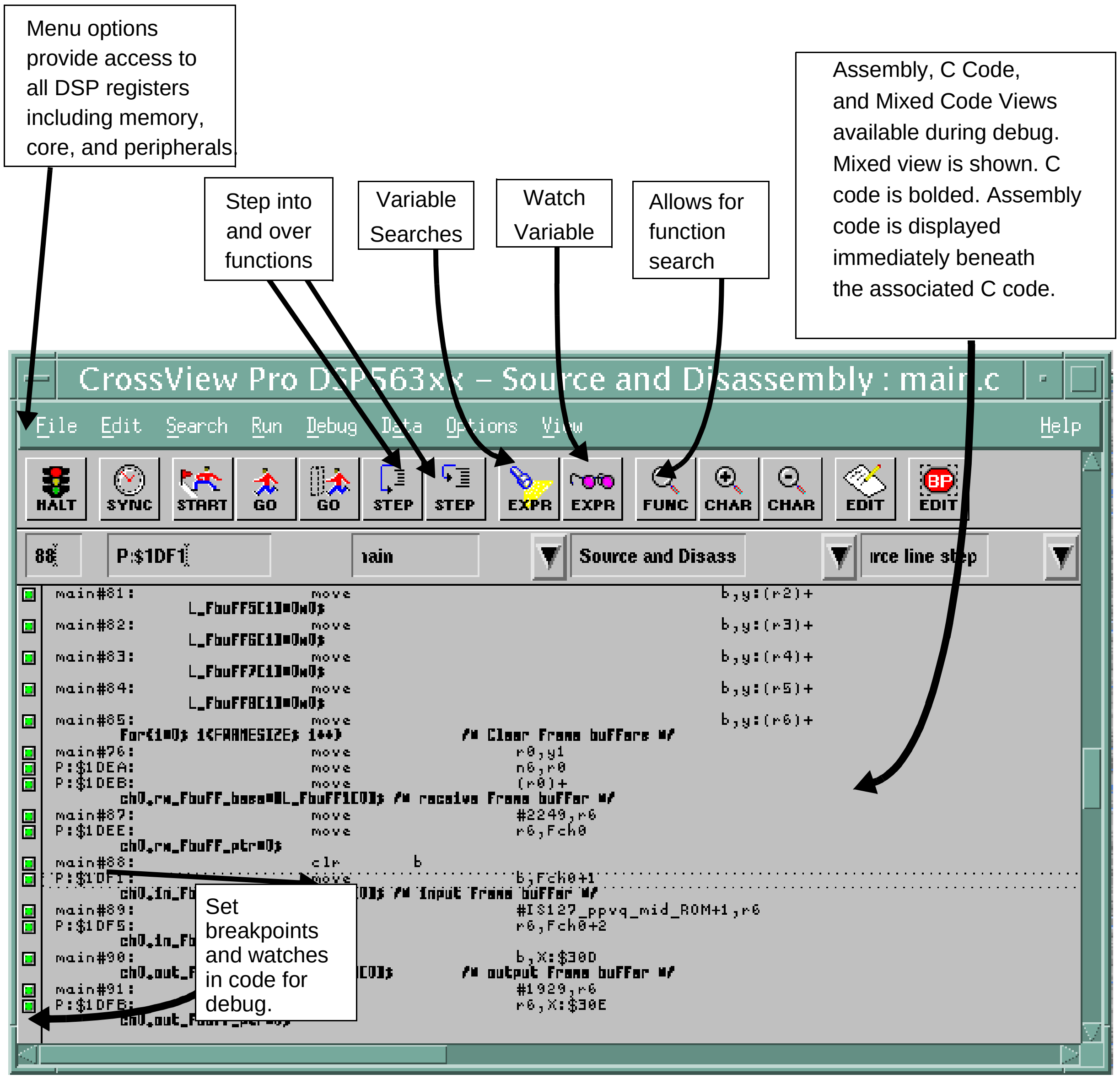
One difficulty of DSP C compiler development is the variability of DSP instruction sets themselves. For instance, each product line of DSPs has a unique instruction set, and even within a family of DSPs added device functionality may need to be considered. For example, in the Motorola DSP56300 family, each device has a different memory map due to differently sized memory. The DSP56301 has 24 KB of memory; the DSP56311 has 512 KB. The compiler must be aware of the memory constraints when building an application. Another difference among the DSP56300 family is the peripherals. Some devices contain the EFCOP, which yields a substantial increase in processing power since it has its own MAC (multiply-accumulate) unit and can handle its computational tasks independently from the core ALU. The differences in peripherals make it necessary to have new compiler header files for each DSP within a family. For this application note, we use the Tasking C compiler, which has a full line of support for the DSP56300 family—in addition to the standard tool set consisting of the macro assembler, linker/locator, libraries, Cross View Pro Debugger and Embedded Development Environment (EDE). Collectively, these tools are called the DSP56xxx software development tools. Examining some features of the Tasking tools can give you an idea of what today's C compilers must provide to be usable in real-time DSP applications.

To achieve code efficiency, the compilers apply optimization techniques. Compiler extensions allow you to program your specific applications, such as filters, in C without significant overhead. Use of the fractional data type and memory source qualifiers helps to optimize loops and exploits the parallel execution capability of the DSP. Bit field operations are optimized by use of the EXTRACT and EXTRACTU operations. The compiler supports the 16- and 24-bit modes of the DSP56300. Furthermore, the Tasking tool suite uses a calling convention that guarantees better use of registers and therefore less function call overhead.

To further aid in optimizing DSP code, C compiler tool suites must offer other features. Tasking supports in-line assembly functions that translate directly, without overhead, to specific DSP capabilities. Tasking also allows for adjustable code generation with `#pragmas`. A *pragma* is a set of instructions that can be written to control the individual compiler optimizations, to allocate character arrays, and to handle the cache. Circular buffer-type modifiers give efficient memory access. Tasking also provides floating-point libraries and memory models to support the different memory sizes across the DSP56xxx line.

Within the Tasking EDE, makefiles can be created directly from the GUI window. This is convenient for code development.

In addition to the compiler, powerful easy-to-use debuggers are required for DSP C code development. **Figure 6** shows some of the useful features of Tasking's source window, which displays the program as C, assembly, or mixed. Other useful debugger windows are the tool/status window and the register, trace, stack, and memory windows. In the tool/status window, you can load, run, step, and stop code as well as set up debugger parameters and access help files from one convenient place. In the register window, you can display, edit, and group all registers. By grouping registers you can display a set of registers as you need them. Highlighted registers indicate what has changed since the previous execution step. The trace window displays the contents of the OnCE/JTAG port trace buffer of the DSP56xxx and automatically updates each time execution halts. The stack window displays the state of the current stack frame, including function parameters. The memory window enables you to monitor and edit the current value of memory locations. Multiple memory windows for different ranges can be opened for convenience.


Menu options provide access to all DSP registers including memory, core, and peripherals.
Step into and over functions
Variable Searches
Watch Variable
Allows for function search
Assembly, C Code, and Mixed Code Views available during debug. Mixed view is shown. C code is bolded. Assembly code is displayed immediately beneath the associated C code.
Set breakpoints and watches in code for debug.
CrossView Pro DSP563xx – Source and Disassembly : main.c
File Edit Search Run Debug Data Options View Help
HALT
SYNC
START
GO
GO
STEP
STEP
EXPR
EXPR
FUNC
CHAR
CHAR
EDIT
EDIT
BP
88
P:$1DF1
main
Source and Disass
urce line step

```
main#81:                move                               b,y:(r2)+
           L_Fbuff5[i]=0x0;
main#82:                move                               b,y:(r3)+
           L_Fbuff6[i]=0x0;
main#83:                move                               b,y:(r4)+
           L_Fbuff7[i]=0x0;
main#84:                move                               b,y:(r5)+
           L_Fbuff8[i]=0x0;
main#85:                move                               b,y:(r6)+
       for(i=0; i<FRAMESIZE; i++)            /* Clear Frame buffers */
main#76:                move                  r0,y1
P:$1DEA:                move                  n6,r0
P:$1DEB:                move                  (r0)+
       ch0.rx_Fbuff_base=&L_Fbuff1[0]; /* receive Frame buffer */
main#87:                move                  #2249,r6
P:$1DEE:                move                  r6,Fch0
       ch0.rx_Fbuff_ptr=0;
main#88:                clr       b
P:$1DF1:                move                  b,Fch0+1
       ch0.in_Fb                      [0]; /* input Frame buffer */
main#89:                                      #IS127_ppvq_mid_ROM+1,r6
P:$1DF5:                                      r6,Fch0+2
       ch0.in_Fb
main#90:                                      b,X:$30D
       ch0.out_F                      [0];            /* output Frame buffer */
main#91:                                      #1929,r6
P:$1DFB:                                      r6,X:$30E
       ch0.out_F
```

Figure 6. Tasking Debugger Code Window

5 Real-time Operating Systems

As the use of higher-level languages becomes predominant in the DSP market, the use of RTOSs in these applications also increases. Though some programmers assert that an RTOS is not necessary, an RTOS eases software integration and system design. A natural relation exists between C and an RTOS. The same features that make the C language attractive for “large-scale” DSP applications also applies to the use of a multitasking RTOS. A mature RTOS from an established third-party vendor makes optimal use of system resources while providing a deterministic system behavior without the resource investment necessary to develop and support this software in-house.

The majority of off-the-shelf RTOSs that meet the real-time requirements of today’s applications use a multitasking, prioritizable, pre-emptive model. Multitasking is a technique that allows multiple chores to share the resources on a DSP. The application is partitioned into smaller tasks that are scheduled to execute by the RTOS kernel. In a pre-emptive system, each task is assigned a priority based on its relative importance. When an event occurs, such as an interrupt or a signal from another task, the kernel decides, based on task priorities, which task can use the DSP resources. A task with a higher priority can “bump,” or preempt, a lower-priority task.

Communication applications, whether they apply to the disparate subscriber (client) or infrastructure (server) markets, lend themselves to a multitasking environment since these applications can be readily partitioned into multiple, prioritizable, preemptable tasks such as I/O, data parsing, voice encoding and decoding, protocol handling, interrupts, host communication, control, and test and debug.

The RTOS for the multichannel voice coder application described here is the Real-Time eXecutive in C (RTXC) by Embedded Power Corporation.[4] RTXC is a multitasking, prioritizable, preemptive operating system designed for responsiveness and predictability. **Figure 7** shows an overview of the RTXC application. At design time the application is decomposed into tasks that use an Application Programming Interface (API) to make calls to the services provided by RTXC, as shown in **Figure 7**.

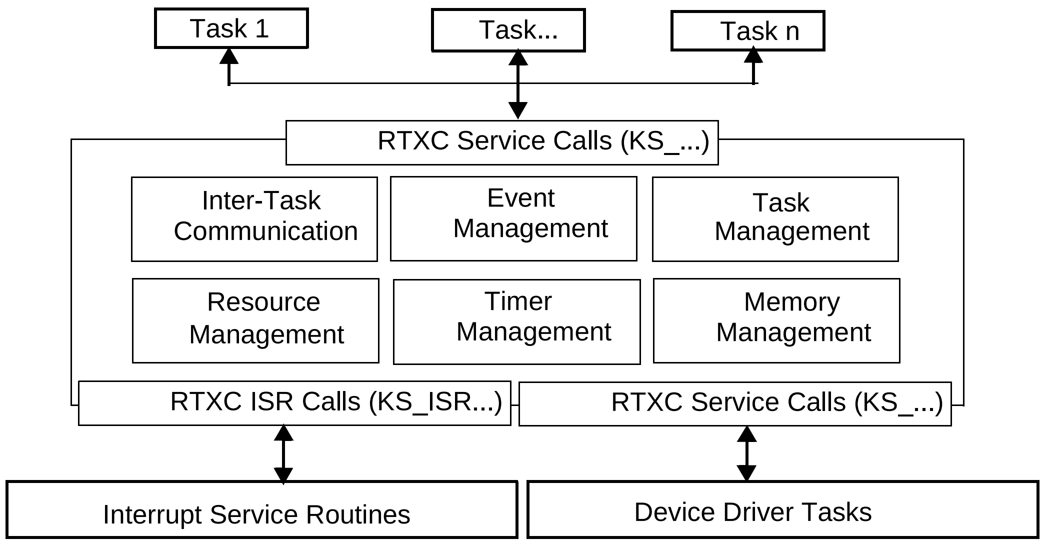


Figure 7. RTXC Services and Task Interface

4. TRXC is the recommended RTOS for the DSP56300 family. See http://www1.motorola-dsp.com/tools-info/rtos-dev.html for details.

By selecting the tasks properly, the developer can isolate the “application” tasks (labeled Task 1 through Task n) from the “device-specific” tasks (shown as interrupt service routines and device driver tasks), which makes it easier to reuse or migrate the application tasks across processors or applications. The RTXC API provides access to the following services and objects:

- Task management: starting and stopping tasks, and so on.
- Event management: semaphores.
- Inter-task communication: mailboxes, messages, and queues.
- Resource management: sharing system resources.
- Time management: timers.
- Memory management: memory partitions.

Though there is no defined way to design an application with an RTOS, there are general guidelines. Start by breaking the application into tasks by drawing data flow diagrams and determining the input, processing function, and output for each task. Define which events and conditions require a task to perform their function. Initially assign priorities based on how often each task must perform its function: the highest priority goes to the most time-critical task.

In its distribution form, RTXC is not executable. RTXC is furnished as a set of C and assembly language source files. You must first compile RTXC source code and then link it with the object files of the application programs and other system configuration files. Developers should treat RTXC as any other software library. It is not necessary to know how RTXC functions internally. You need only know what functionality RTXC provides and what RTXC kernel services achieve the desired result. Knowledge of which inputs produce which outputs is all that is needed. Additionally, not all RTXC objects are needed in an application. In fact, an effective application can be implemented using tasks and semaphores alone. Therefore, RTXC allows scaling of its software to include only the modules and API calls that the application developer wishes to use, thus improving memory and execution efficiency.

6 System Overview

The system discussed here executes multiple channels of the IS-96-A voice coder on a Motorola DSP56307EVM. This section outlines the target hardware and application software developed for this project.

6.1 Target Hardware

The target hardware is a DSP56307EVM with a connection to a daughter board that contains four Crystal CS4215 stereo audio codecs. All software described in this application note runs on the DSP56307EVM. This board can be purchased for development purposes. The EVM has an on-board DSP56307 and audio codec. You can therefore program the DSP to process voice coming from an audio source through the A/D converter and then send the processed data back out through the D/A converter to hear through headphones or speakers. The EVM also has on-board SRAM and Flash memory.

The DSP56307 is a powerful device that can process multiple channels of voice data. The audio codec on the DSP56307EVM allows processing for two channels of data. To demonstrate the processing capabilities of the DSP56307, more than one audio codec is required. We developed the multichannel board primarily to demonstrate the execution of multiple voice coding channels on a DSP56307EVM. The

DSP communicates with the multichannel board via the Enhanced Synchronous Serial Interface 0 (ESSI0) port on the EVM. Each codec on the board executes A/D or D/A operations for two channels of audio data (stereo audio requires two channels). Four audio codecs are provided for A/D and D/A conversion of the audio inputs to be processed by the DSP. Therefore, up to eight audio channels can be sent through the codec board to the DSP device. **Figure 8** shows the primary multichannel board components.

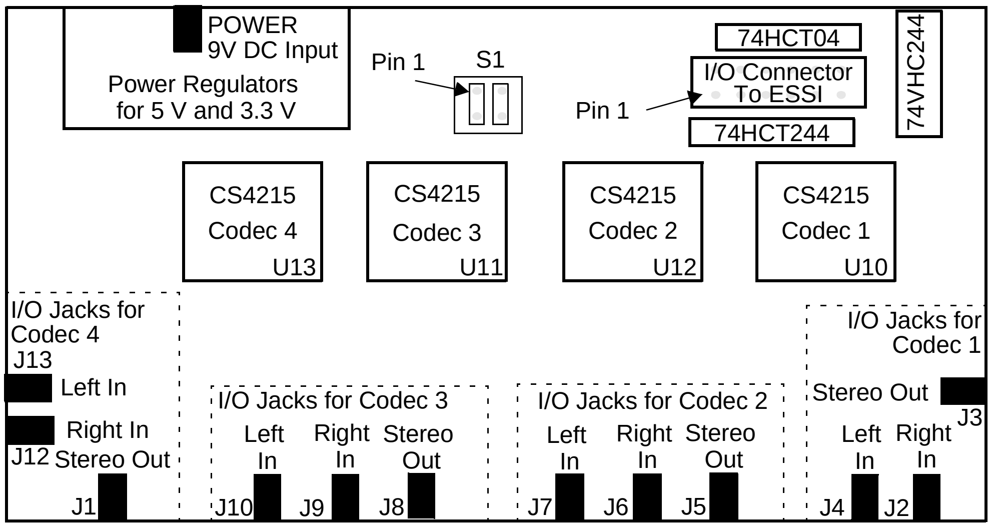


Figure 8. Multichannel Board

Table 1 shows the signal connection table between the EVM and the multichannel board. Immediately after power-up, the DSP is the master of the connection and uses its GPIO lines to signal the codec that it is sending control information. Once the master codec (Codec 1) receives the control words, the connection is reset and the master codec provides the clock for the subsequent data transmission between the EVM and the multichannel board.

Table 1. Signal Connection

EVM ESSI0 Signals	Multichannel Board Codec Signals
SCK1	SCLK
GPIO	RESET
STD1	SDIN
SRD1	SDOUT
GPIO	D/C
SC11	FSYNC
GND	GND

6.2 Application Software

The application software manipulates four independent audio data streams or channels: `ch0`, `ch1`, `ch2` and `ch3`. Each audio channel is processed on the basis of one of four user-selected modes:

- *Off*. If the channel is Off, the received audio input sample is simply ignored.
- *Pass*. In Pass mode, each received input is immediately transmitted without any further processing.
- *Delay*. In Delay mode the data received is used to fill an input 'frame' buffer that is then copied to an output 'frame' buffer for transmission. A certain amount of delay is introduced into the audio stream that is proportional to the size of the frame buffers being used.
- *IS96a*. In IS96a mode the operation is similar to Delay mode except that rather than copying the data from input frame buffer directly to the output frame buffer it is encoded (compressed) and immediately decoded (de-compressed) by the IS-96-A voice coder.

To help manage this system, a C data structure is defined that contains the information necessary to manage an individual audio channel, as follows:

```
struct CHANNEL_CONFIGURATION
{
   int _Y id;               // channel identification
   int _Y mode;             // processing mode
   _fract_Y *rx_fbuff_base; // receive frame buffer
   int _Y rx_fbuff_ptr;
   _fract_Y *in_fbuff_base; // input frame buffer
   _fract_Y *out_fbuff_base;// output frame buffer
   _fract_Y *tx_fbuff_base; // transmit frame buffer
   int _Y tx_fbuff_ptr;
   int _Y rx_fbuff_full;    // receive flag
   int _Y tx_fbuff_empty;   // transmit flag
}ch0,ch1,ch2,ch3;
```

The channel identification parameter, `id`, indicates which channel (`ch0`, `ch1`, `ch2` or `ch3`) is defined by the C structure parameters and points to a time slot in the ESSI frame (see **Section 7.2.2**) that contains the audio data associated with the channel. The channel processing mode, `mode`, defines which mode is currently used to process the channel (Off, Pass, Delay, or IS96a). The remaining elements in the `CHANNEL_CONFIGURATION` C structure implement a double buffer mechanism that manages the frame buffers in the Delay and IS96a modes. The IS-96-A voice coder is obtained from SASL.

A channel is a data stream of audio data that is accessed using the DSP56307 Enhanced Serial Synchronous Interface (ESSI). The ESSI first initializes the hardware daughter board described in **Section 6.1**, *Target Hardware*, on page 10 and then receives and transmits the audio data sampled by the multichannel board. The system connects to a host computer to allow a user to enter commands and change a channel's processing mode via a command-line interface (CLI) that provides status information and allows configuration of the four audio channels. The Serial Communication Interface (SCI) on the DSP56307 communicates with the host computer via an RS232 serial communication (COM) port.

Finally, the RTXC RTOS, version 3.2d, provides multi-tasking support, MIPS usage information, and snapshot views of the RTXC state and objects (the latter two supplied via the CLI). Use of a multitasking operating system makes it easier to integrate the various software modules (tasks, ISRs, and drivers) and allows reuse of the code.

7 Software Description

This section addresses the following topics:

- Data Input and Output (I/O)
- Voice coders
- User interface
- RTOS

First, we created a stand-alone system for data I/O using the ESSI and multichannel vocoder board, another system that uses the SASL voice coders (using the analog-to-digital (A/D) codec on the DSP56307EVM), and a third system that implements the user interface. We developed these systems without use of an RTOS and then incorporated RTXC. Use of an RTOS made it much easier to integrate the final system. When we used RTXC, each stand-alone system consisted mainly of one or more interrupt service routines (ISRs) and one or more tasks.

The integration work mostly involved creating an additional task to “dispatch” the appropriate task based on the operating mode chosen by the user and selecting the appropriate task priorities. Also, the RTXC mechanisms for task synchronization and communication were incorporated as needed.

7.1 Reentrant Requirement

At this point, the concepts of re-entrancy and the requirements of multichannel systems must be understood. A *re-entrant* computer program or routine is written so that multiple users/tasks can share the same copy of the code in memory. Re-entrant code is commonly required in RTOSs and in application programs shared in multi-user multi-tasking systems. A programmer writes a re-entrant program by making sure that no instructions modify the contents of variable values in other functions within the program. Each time the program/processor is entered for a user/task, a data area is obtained in which to keep all the variable values for that instance of the user/task. When the process is interrupted to give another user/task a turn to use the program/processor, information about the data area associated with that user is saved. When the interrupted user/task of the program recovers control of the program/processor, context information in the saved data area is recovered and the module is reentered and processing continues. If a routine follows these rules, it is re-entrant:

- All local data is allocated on the stack.
- The routine does not use any global variables.
- The routine can be interrupted at any time without affecting the execution of the routine.
- The routine calls only other re-entrant routines. It does not call non-re-entrant routines (for example, standard I/O, **malloc**, free, and so on).

7.2 Data Input/Output

In DSP applications, input/output (I/O) handling is critical to maintaining real-time data processing. Therefore, DSPs such as those in the DSP56300 family provide multiple I/O-handling peripherals and extremely fast interrupt servicing. For example, the DSP56307 has two ESSIs, the Serial Communications Interface (SCI), and the 8-bit host interface (HI08) for efficient data I/O implementation. These features provide optimal I/O solutions for various applications.

The ESSI provides a full duplex serial port for serial communications with a variety of serial devices, including industry-standard analog-to-digital codecs, other DSPs, microprocessors, and peripherals. The SCI provides a full duplex port for serial communications with other DSPs, microprocessors, or peripherals such as modems. The HI08 is a byte wide, full duplex, double-buffered parallel port that can connect directly to the data bus of a host processor. For our application, we use the ESSI for obtaining audio I/O from the multichannel codec board described in **Section 6.1**, *Target Hardware*, on page 10 and the SCI for communicating with the host computer using the command line interface. The HI08 is not required.

In the DSP56300 family, the I/O peripherals can trigger interrupts to conveniently handle processing. The DSP56300 processors have two types of interrupts:

- *Fast interrupts* are two instructions long and require no overhead for jumping to the interrupt or returning to the normal program flow. Furthermore, the fast interrupt instructions always complete without interruption. Fast interrupts are an excellent method for moving data from the I/O peripherals.
- *Long interrupts* are not limited to a certain number of instructions and can be interrupted by higher priority interrupts. However, long interrupts need to save and restore the program counter, the status register and update the stack pointer to return to the normal program flow. Long interrupts are used when more intricate data processing is required.

The ESSI software for the audio I/O handling focuses on properly initializing the hardware and efficiently using the ISRs and data structures. In our system, a double buffering technique implements efficient data transfers from the codecs to the DSP56307 core for processing of the data I/O in the Delay and IS96a modes. In the double buffering technique, two buffers and two pointers are used instead of just a single buffer for data input or data output. A single buffer that receives data requires the input data stream to “wait” while the current data in the buffer is processed. When two buffers (and two pointers) are used, one buffer can be filled while the other is processed. When the receiving buffer is full and the other is processed, the pointers are swapped so that the processing can occur with the new data and the incoming data stream can fill the alternate buffer. Swapping pointers omits the need to copy the data does from one buffer to the other. The pointer is then passed to the data input task or the processing task. A similar situation occurs for output data (the resulting data from the processing) and data transmitted via the ESSI.

7.2.1 Audio Codec Initialization

Before communications can begin between the audio codec board and the DSP56307, the codec must be initialized. There are two main steps to initialization:

1. Send control data.
2. Once the control information is sent, place the codec into data mode. Transmission may then begin.

General purpose I/O (GPIO) pins initialize the audio codec in control mode. One line sets the mode of the audio codec: control or data. The second line is tied to the codec reset line. These are the only control lines required to initialize the CS4215 codec. When the codec comes out of reset in control mode, it waits to receive control information for operation. The DSP56307 controls the interface and transmits the control words to the codec. Once the codec receives all of its control words, the codec is placed into data mode. Because the codecs on the multichannel board are daisy-chained together, one codec on the multichannel board is initialized as the interface master. When the codec is placed in data mode, the DSP56307 is no longer master of transmission between the devices. The master codec generates the clock for the ESSI.

7.2.2 Synchronous Interface

ESSI0 transfers the data to and from the codecs. The ESSI0 interface has three main modes for handling synchronous data transfers: normal, on-demand, and network. We use the network mode, which allows multiple time slots of data to be transferred to the DSP within a given time frame. This mode is extremely useful when an application requires the transfer of multiple independent channels of data through the same interface. Network mode allows the ESSI to handle up to 32 time slots. This application uses two codecs, which require a total of eight time slots per receive or transmit frame: four for audio data and four for control data (see **Figure 9**).

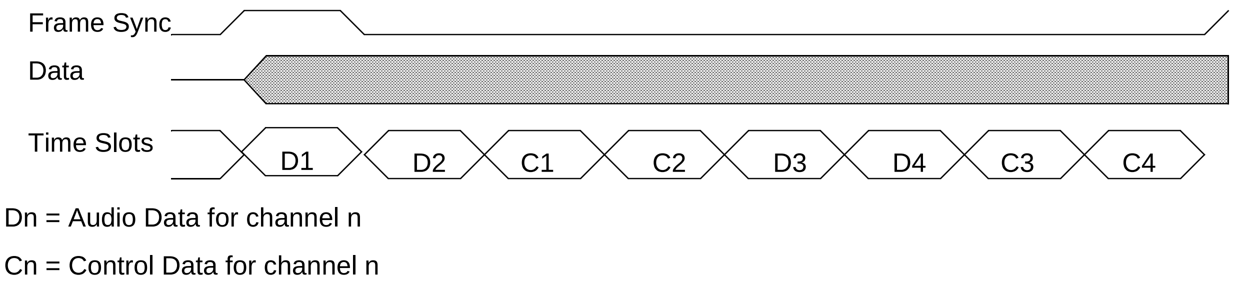

Frame Sync
Data
Time Slots
D1
D2
C1
C2
D3
D4
C3
C4
Dn = Audio Data for channel n
Cn = Control Data for channel n

Figure 9. Picture of ESSI Network Mode Usage in This Application

The first four slots contain data from/to the first codec (two for data for left and right channels and two for control information for left and right channels). The last four slots contain the data from/to the second codec. Each audio channel is sampled at 8 KHz. In total, the ESSI0 handles 128 bits every 125 µsec (16 bits per time slot for eight time slots every 8 KHz). The data received through ESSI0 is placed into a 16-element receive buffer, and the data transmitted is taken from a separate 16-element buffer.

A private (non-RTXC) ESSI receive ISR loads data/control information for each time-slot into the receive buffer. **Figure 10** shows the mechanism to receive data using the ESSI. The ESSI receive ISR simply reads the audio data from the ESSI receive register and writes it into the 16-element receive buffer called `RX_buff`.

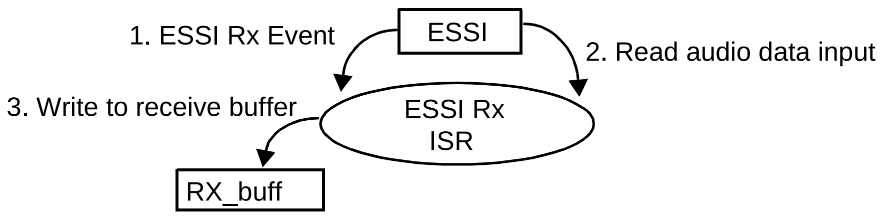


Figure 10. ESSI Data Receive Mechanism

Similarly, **Figure 10** shows the mechanism used to transmit data using the ESSI. The ESSI private transmit ISR simply writes the audio data from the 16-element receive buffer called `TX_buff` to the ESSI transmit register.

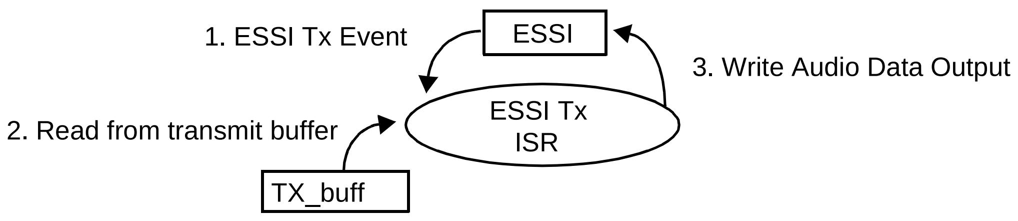


Figure 11. ESSI Data Receive Mechanism

7.3 Voice Coders

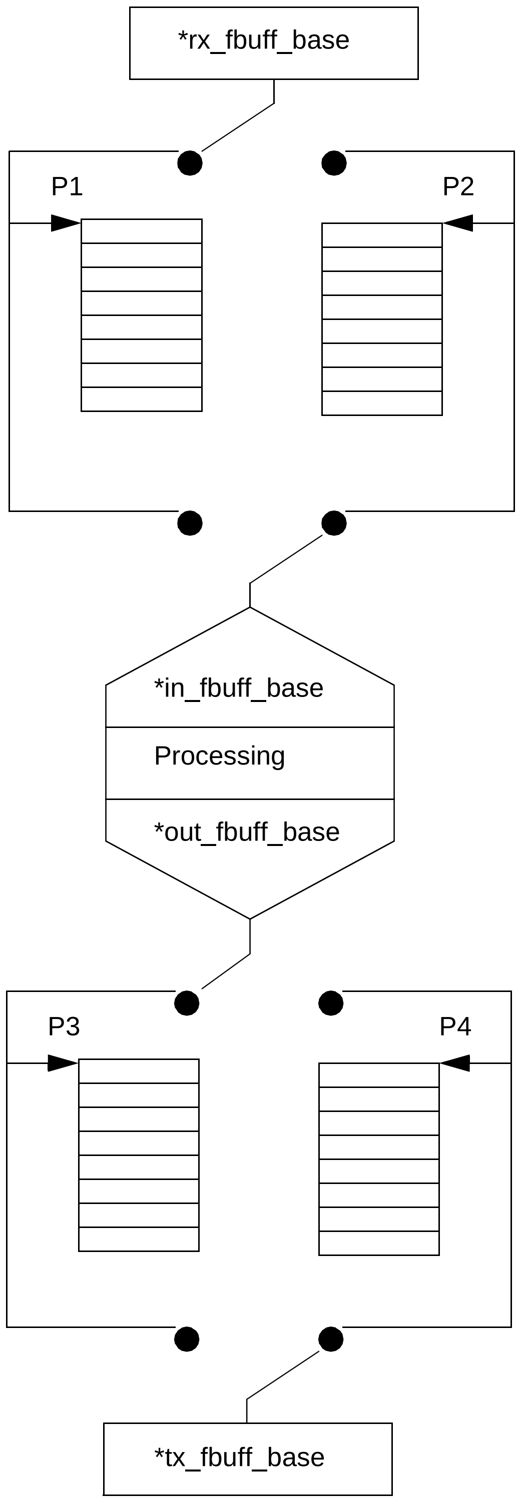
This section discusses two key features of the voice coder: the double buffering mechanism for rapidly moving data and the wrappers for integrating the voice coder software into a system.

7.3.1 Double Buffering

Our demonstration code contains a double-buffer mechanism, also known as "ping-pong buffers." Two separate buffer pairs are defined for each channel. Pointers and flags for the various buffers are maintained in the `CHANNEL_CONFIGURATION` C structure described in **Section 6.2**, *Application Software*, on page 12. Each channel uses four frame buffers so that one buffer can receive data while another buffer is processed. The third frame buffer stores the processed output, and the fourth contains the data being transmitted. When the receive frame buffer is full, the base pointers to the receive (`*rx_fbuff_base`) and input (`*in_fbuff_base`) frame buffers are swapped. Similarly, when the transmit frame buffer is empty, the base pointers to the output (`*out_fbuff_base`) and transmit (`*tx_fbuff_base`) frame buffers are swapped. The assumption is that the processing for each channel data stream completes by the time the buffers need to swapped.

Associated with each double-buffer structure is a flag that indicates whether the receive buffer is full (`rx_fbuff_full`) and another to indicate whether the transmit buffer is empty (`tx_fbuff_empty`).

The `rx_fbuff_ptr` pointer in the data structure is an index to indicate the location in the receive buffer to be loaded with the next data input. The `tx_fbuff_ptr` pointer in the data structure is an index to indicate the location in the transmit buffer of the data to be transmitted next. When the base pointers to the receive buffer and the transmit buffer are swapped (that is, the receive buffer is full and the transmit buffer is empty), the `rx_fbuff_ptr` and `tx_fbuff_ptr` index pointers are initialized to zero. **Figure 12** shows the double buffer mechanism.



The receive and input both use P1 and P2 buffer pointers. However, they use them in a mutually exclusive way. *rx_fbuff_base switches to P2 at the same time *in_fbuff_base switches to P1.

The output and transmit both use P3 and P4 buffer pointers. However, they use them in a mutually exclusive way. *out_fbuff_base switches to P4 at the same time *tx_fbuff_base switches to P3.

Figure 12. Double Buffer Diagram

7.3.2 Wrappers

Voice coding software purchased from a third-party developer includes the software to implement a given telecommunications standard on a specific DSP device. This software does not handle data input/output or other system issues for a given application. Also, each software module is generally developed to handle a single voice channel. Since most infrastructure applications handle multiple channels of voice processing, the voice coder software must be invoked multiple times, once for each channel of voice processing. To address these issues, *wrappers* are used to integrate the voice coder software into a system. The wrapper handles several tasks:

1. Sets up any initialization for data storage and pointers required by the voice coder. Generally, voice coder software is accompanied by documentation describing the information it requires to operate properly (see **Section 2.2**). The wrapper sets up this information as needed before the voice coder is invoked.
2. Handles data input and output to the vocoder modules.
3. Calls the voice coding routine when everything is ready to process.

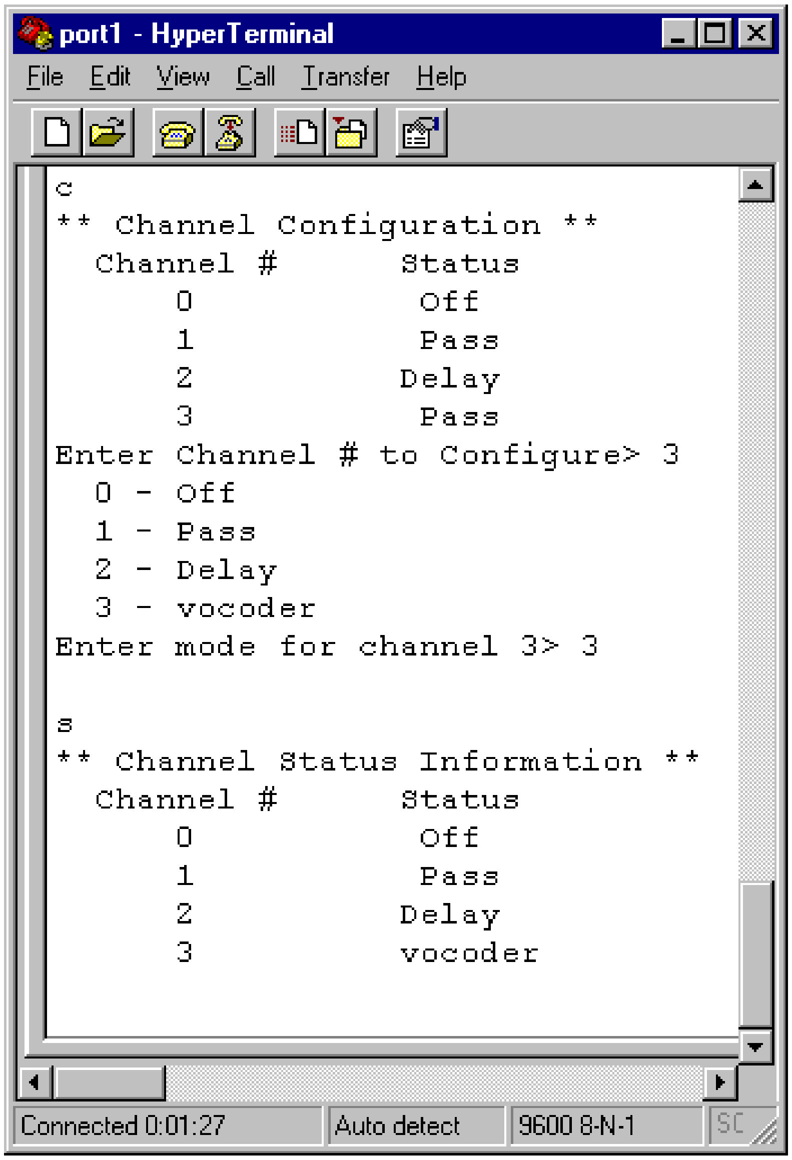
SASL provides an Interface Control Document (ICD) with the voice coding libraries. This document describes which parameters must be passed to the voice coder, registers to contain the parameters, and the form the parameters may need to take before the voice coder gets them. For IS-96-A, the parameter passing is straightforward. Before the encoder initialization routine is called, the pointer to encode data in X memory is placed into r3, and the pointer to encode data in Y memory is placed into r4. The encoder takes a block of 160 input samples and converts it into the IS-96-A packets for transmission. r1 points to the input buffer of 160 samples; r2 points to the output buffer. The r3 and r4 registers remain pointing to the static memory required by the encoder. Also, x0 holds the minimum rate allowed by the voice coder, and y0 holds the maximum rate allowed. Generally, the maximum rate is 8 kbps (y0=4) and the minimum rate is 0.8 kbps (x0=1). The encoder is called using a jump to subroutine instruction to a routine specified in the ICD.

For decoding, the parameter passing is similar. The r3 and r4 registers now point to the static memory for the decoder. r1 points to the input buffer and r2 points to the decoder output buffer. The decoder has a postfilter option, which is enabled by placing the value 1 in the y1 register.

7.4 User Interface

As noted earlier, each audio channel is processed in one of four user-selected modes: Off, Pass, Delay, and IS96a. If the channel is Off, the output does not change. In Pass mode, each input is simply sent to the ESSI output buffer for transmission. In Delay mode, the data is first buffered into a frame of 160 inputs; when this input frame is filled it is simply copied to the 160-element output frame buffer for later transmission. Finally, in IS96a mode, the operation is similar to Delay mode except that the data from input to output frame buffers it is encoded (compressed) rather than copied and immediately decoded (decompressed) by the IS96a voice coder.

The user interfaces with the target DSP56307EVM using a command-line interface (CLI) from a dumb terminal (hyperterminal) on a host computer (see **Figure 13**). The SCI peripheral on the DSP56307 connects to the COM port of the host computer via an RSR232 serial cable. Thus, you configure the individual channels by typing characters that are decoded by the application’s CLI task.



```
c
** Channel Configuration **
  Channel #       Status
      0            Off
      1            Pass
      2           Delay
      3            Pass
Enter Channel # to Configure> 3
  0 - Off
  1 - Pass
  2 - Delay
  3 - vocoder
Enter mode for channel 3> 3

s
** Channel Status Information **
  Channel #       Status
      0            Off
      1            Pass
      2           Delay
      3           vocoder
```

Figure 13. Command Line Interface

Figure 14 shows the mechanism for receiving user commands over the SCI peripheral. The SCI peripheral signals an event when it receives a character, causing the SCI receive interrupt routine to execute. The SCI receive ISR reads the character from the SCI receive register, writes it to a buffer; and signals a semaphore that enables the SCI input driver task `sci_drv`. When the SCI input driver task executes, it can read the character from the buffer and enqueue the character in the RTXC input queue.

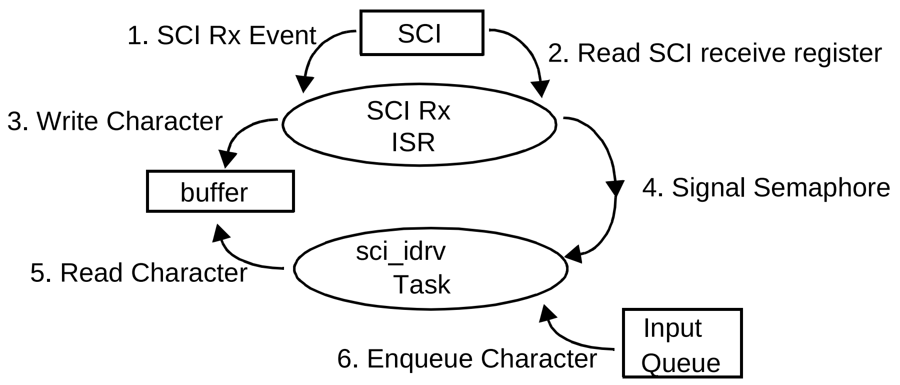


Figure 14. SCI Receiver

Using the mechanism shown in **Figure 14**, the hardware-specific ISR is isolated from the application-specific `sci_drv` task. This makes it easier to reuse or migrate the application C code across processors or applications. A similar mechanism transmits characters from the DSP56307 SCI to the host as shown in **Figure 15**. As with the receive path, the SCI peripheral signals an event when a character is ready to be transmitted, causing the SCI transmit ISR to execute. The SCI transmit ISR signals a semaphore that enables the SCI output driver task `sci_odrv`. When the SCI output task executes, it reads the output character from the RTXC output queue, which is then written to the SCI transmitter.

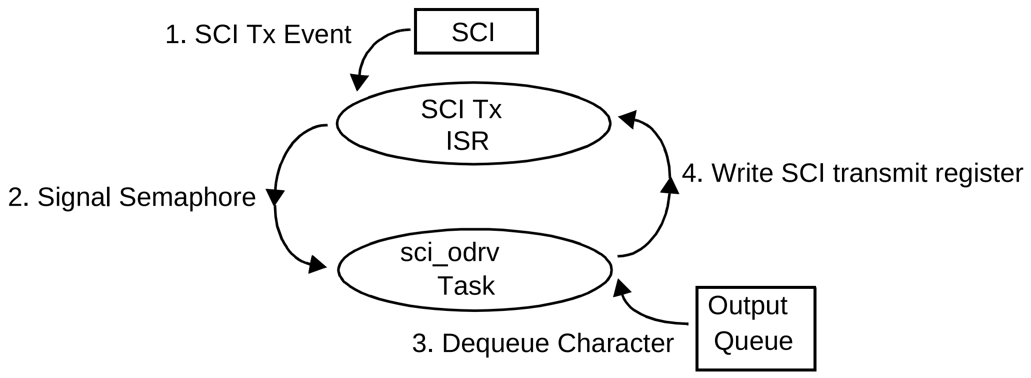


Figure 15. SCI Transmitter

7.5 RTXC

RTXC is a multitasking prioritizable, preemptive operating system. RTXC is treated as a software library, so the application tasks and ISRs access the RTXC services by making calls to the RTXC API. Knowledge of which inputs produce which outputs is all that is needed. The general form of the RTXC API call is

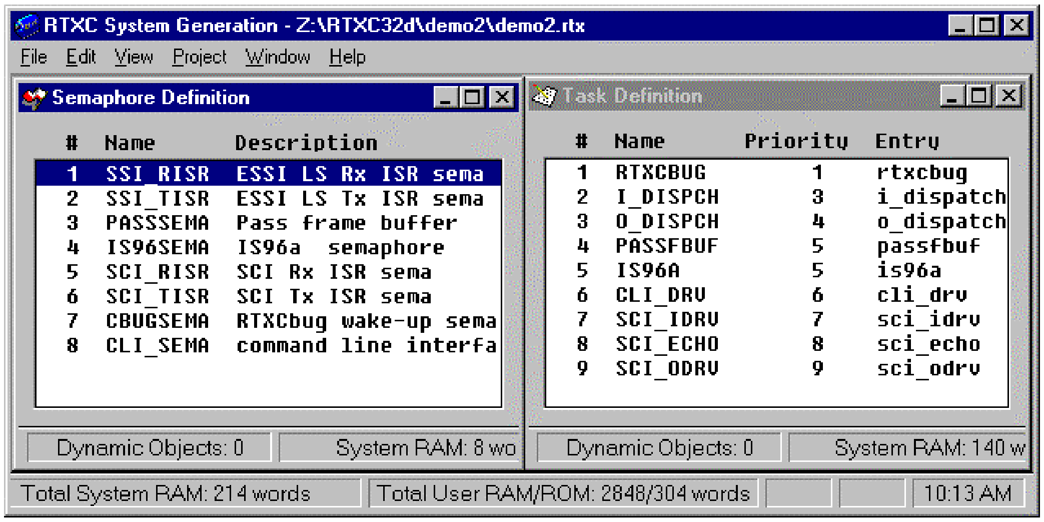
```
KS_name ([arg1],[arg2], ... , [argN])
```

The character string `KS_` identifies name as an RTXC kernel service. This prefix prevents `name` from being mistakenly identified by the linker with some similarly named function in the run-time library of the compiler. For example, the API call:

```
KS_wait(SSI_RISR)
```

blocks a task until a specified event occurs. The event is associated with the semaphore SSI_RISR, which is signaled by the ESSI receive ISR. The current task thus synchronizes its execution with the reception of a frame of data containing four audio samples, one for each of the four channels being processed. If the ESSI ISR has already signalled the semaphore, no wait occurs nor is the task blocked. Instead, the task resumes.

Once the application tasks are written, the remaining C source code files necessary to correctly build the application are automatically generated by Sysgen, a utility tool provided with RTXC for entering and editing the system configuration for the application. SYSgen accepts the definition of the configuration data through a series of interactive dialogs relative to each type of control or data structure (see **Figure 16**). SYSgen uses this information to generate header files that are then included in any application code module that uses RTXC data element to be later compiled into the application.


RTXC System Generation - Z:\RTXC32d\demo2\demo2.rtx
File Edit View Project Window Help
Semaphore Definition
Task Definition

#	Name	Description
1	SSI_RISR	ESSI LS Rx ISR sema
2	SSI_TISR	ESSI LS Tx ISR sema
3	PASSSEMA	Pass frame buffer
4	IS96SEMA	IS96a semaphore
5	SCI_RISR	SCI Rx ISR sema
6	SCI_TISR	SCI Tx ISR sema
7	CBUGSEMA	RTXCbug wake-up sema
8	CLI_SEMA	command line interfa

Dynamic Objects: 0 System RAM: 8 wo

#	Name	Priority	Entry
1	RTXCBUG	1	rtxcbug
2	I_DISPCH	3	i_dispatch
3	O_DISPCH	4	o_dispatch
4	PASSFBUF	5	passfbuf
5	IS96A	5	is96a
6	CLI_DRV	6	cli_drv
7	SCI_IDRV	7	sci_idrv
8	SCI_ECHO	8	sci_echo
9	SCI_ODRV	9	sci_odrv

Dynamic Objects: 0 System RAM: 140 w

Total System RAM: 214 words Total User RAM/ROM: 2848/304 words 10:13 AM

Figure 16. Sysgen Semaphore and Task definition windows

7.5.1 RTXCbug

RTXCbug is the RTXC system-level debugging tool. Its provides snapshots of RTXC internal data structures and performs some limited task control. RTXCbug operates as a task and is usually set up as the highest-priority task. Whenever RTXCbug runs, it freezes the rest of the system, thereby permitting coherent snapshots of RTXC components. RTXCbug is not a replacement for other debugging tools but assists you in tuning the performance of the RTXC environment or checking out problems within it. RTXCbug uses the input and output ports of a user-defined console device. In the application discussed here, the host computer interfaces with RTXC using the SCI input and output drivers. Commands are given to RTXCbug via a hyperterminal window on the host computer and transmitted to the DSP56307EVM via an RS2323 interface and a UART connection. The RTXCbug output is also displayed on the host computer’s hyperterminal window. RTXCbug is entered using two mechanisms:

1. The user enters an exclamation mark (!) on the console input.
2. A task calls a special function within RTXCbug.

Once the system enters RTXCbug, type K in the main menu to invoke the command menu.

7.6 Integration

Once the individual tasks are defined and tested, the integration simply involves properly selecting task priorities and RTXC objects to synchronize and communicate these tasks. We integrated the ESSI data I/O software with the double buffer and voice coder software using an input dispatch task called `i_dispatch` and an output dispatch task called `o_dispatch` shown in **Figure 17**.

The `i_dispatch` task is signaled via the `SSI_RISR` semaphore from the ESSI last slot ISR, indicating that all the data for a given ESSI frame has been received. This task then reads the `CHANNEL_CONFIGURATION` C structure to determine the operating mode for a given channel, and based on this information, it determines whether to ignore the input (`Off` mode), copy the received data from the receive buffer `RX_buff` to the transmit buffer `TX_buff` (`Pass` mode), or call double buffer mechanism and process the input data accordingly. If the receive frame buffer is full as indicated by the corresponding flag in the `CHANNEL_CONFIGURATION` C structure, an RTXC message is sent to either the `Delay` mode task (`passbuff`) or the `IS96a` mode task (`is96a`) indicating which channel to process.

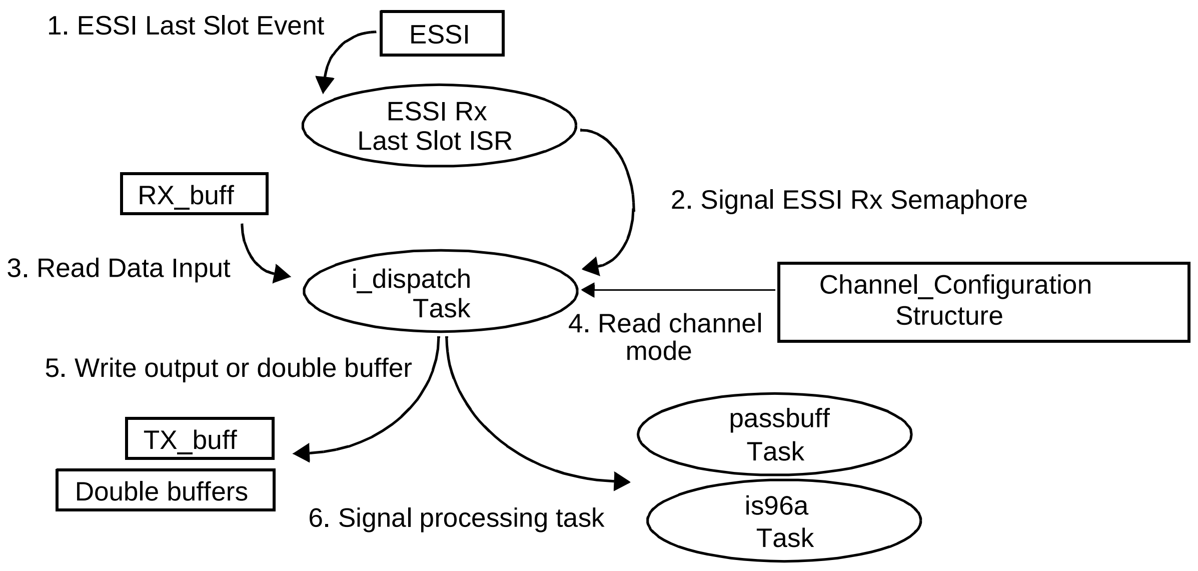


Figure 17. Input Dispatcher Task

The input dispatcher task performs the steps described for each of the four channels supported. The use of a `CHANNEL_CONFIGURATION` C structure for each channel allows the current state of the channel to be maintained independently. Thus each channel's data stream is processed independently. The same Delay and IS96a tasks are used for each channel: the messages sent to these tasks contain a pointer to the `CHANNEL_CONFIGURATION` C structure to use and if multiple messages are sent, the RTXC queues them. Similarly, the output dispatcher task is `o_dispatch` signaled by the ESSI transmit last slot ISR. However, this task simply writes output data to the transmit buffer (`TX_buff`) if `Delay` or `IS96a` modes are used for a channel. Also, the task calls the function to implement the double buffer mechanism for the output and transmit frame buffers.

The integration of the command line interface task `cli_drv` with the SCI character I/O mechanisms described in **Section 7.4** is shown in **Figure 18**. The CLI driver task `cli_drv` is signaled if a valid character command is detected in the SCI input driver task (`sci_idrv`). The `cli_drv` task then accesses the `CHANNEL_CONFIGURATION` C structure to display or change a channel's state information. Messages are sent to the user from the CLI task by enqueueing characters in the RTXC output queue. Additionally, a task that echoes characters (`sci_echo`) typed by the user is signaled when a character is placed in the RTXC input queue.

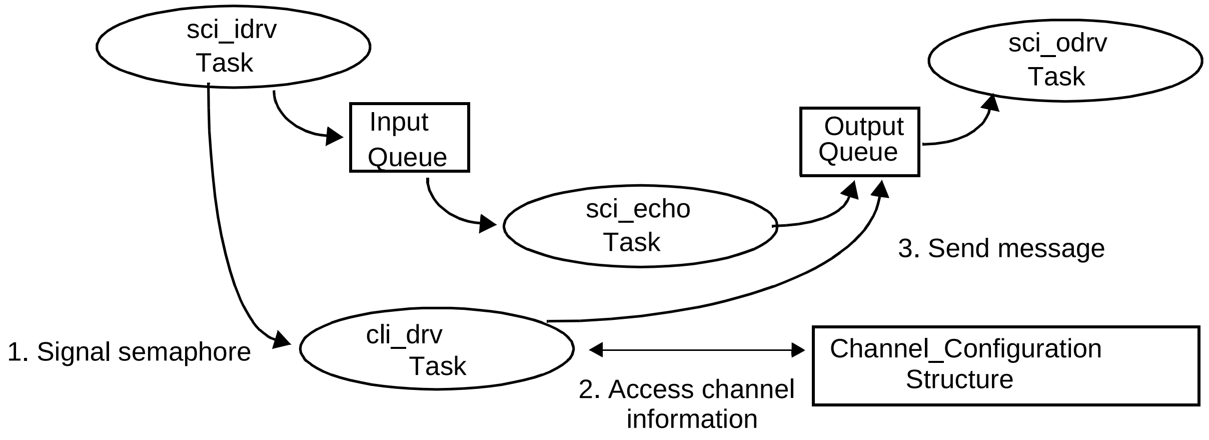


Figure 18. Command Line Interface Task

We used the Tasking make utility (mk563.exe) to build the application. Integration at this level involved proper compilation and linking of the RTXC library, the IS-96-A voice coder library, and the application object files and other the system configuration files.

8 Conclusions

As communications systems continue to increase in complexity, equipment manufacturers rely more and more upon third parties to develop standard software for a chosen DSP architecture, high-level languages to easily port code between systems, and RTOSs to integrate various system tasks to guarantee task handling. Remember that using an RTOS has the following trade-offs:

- It adds overhead, but this can be kept to a minimum by choosing a well-designed RTOS such as RTXC and designing a system efficiently.
- It is not difficult to use. A different way of looking at things is necessary, but the main step in converting existing code to use an RTOS is to divide the program into tasks that usually flow with the function organization. A robust RTOS is absolutely necessary.
- It can be used in real-time DSP applications.
- It is a part of the trend towards software portability.

Motorola is committed to continued support of RTOS usage and other innovations in DSP technology that can be used to decrease the development time required of equipment manufacturers.

9 References

1. *DSP56300 Family Manual* (DSP56300FM/D)
2. *Real-Time Kernel User's Manual: RTXC*. Embedded System Products. Version 3.2. 1986 – 1985.
3. (MA039-002-00-00) *C Cross-Compiler User's Guide*. DSP56xxx v2.3. TASKING, Inc., 1998.
4. (MA039-049-00-00) *Crossview Pro Debugger User's Guide*. TASKING, Inc., 1999.
5. Standard Speech Coding Software Training Course given by Signals and Software Ltd., July 1998.

How to reach us:

USA/EUROPE
Motorola Literature Distribution
P.O. Box 5405
Denver, Colorado 80217
1-303-675-2140
1-800-441-2447

Technical InformationCenter
1-800-521-6274

JAPAN
Motorola Japan Ltd.
SPS, Technical Information Center
3-20-1, Minami-Azabu, Minato-ku
Tokyo 106-8573 Japan
81-3-3440-3569

ASIA/PACIFIC
Motorola Semiconductors H.K. Ltd.
Silicon Harbour Centre
2 Dai King Street
Tai Po Industrial Estate
Tai Po, N.T., Hong Kong
852-26668334

Home Page
http://www.mot.com/SPS/DSP

DSP Helpline
http://www.motorola.com/SPS/DSP/support/index.html
email: dsphelp@dsp.sps.mot.com

MOTOROLA