Preprint No. 4825

A Real Time DSP Kernel for Concurrent Audio Tasks

David Reinhardt and Robert C. Maher

EuPhonics, Inc.

4840 Pearl East Circle, Suite 201E
Boulder, CO 80301-6115 USA

http://www.euphonics.com

Abstract:

It is desirable to share the resources of a single DSP microprocessor among multiple concurrent audio tasks, but this poses a variety of practical problems. In this paper the features of general-purpose operating systems are contrasted with the needs of typical audio DSP processes, and the architecture of a simple yet powerful real time DSP kernel is proposed.

1. Introduction

In this paper a description is given of a real time DSP operating system. The system is intended to facilitate the simultaneous execution of multiple audio DSP tasks using a single DSP microprocessor. The definition of the DSP operating system includes means for loading and unloading DSP code modules, interconnecting processes and devices, scheduling tasks for execution, and conveying messages between a host processor and the tasks running on the DSP microprocessor.

The need for concurrent audio task execution is increasingly important for multimedia audio accelerators. Computer-based games require multiple streams of 3D-localized audio to be processed and mixed, soundtrack audio to be presented, and music synthesis to be performed simultaneously and in real time. In the near future there will be a need in the market for concurrent telephony and audio compression/decompression for internet gaming and entertainment. Also, because specific technical and marketing requirements change very rapidly, there is a need to avoid redesigning the entire audio DSP code base every time a feature is added or modified. A DSP operating system enables the required concurrent execution while also insulating the details of one task from the others [1].

The features and performance characteristics of a multi-processing audio DSP operating system must be different from those of a conventional CPU. Audio DSP tasks have input and/or output requirements that are continuous and high-bandwidth, and the allowable processing delay, or *latency*, is usually tightly constrained. While conventional operating systems are often driven by asynchronous events like mouse movements and key presses, the audio DSP operating system must be more attuned to the arrival and departure of audio sample data at a relatively constant rate [2].

DSP microprocessors are typically noted for the speed at which they execute repetitive calculations (such as digital filters and FFTs), and for the amount of on-chip, fast memory available. There is significant pressure on DSP programmers to minimize execution time so that more concurrent tasks can be accommodated, and to minimize on-chip memory requirements due to the cost of static RAM. Thus, the DSP operating system must be optimized carefully to support only the most essential features in order to minimize its memory footprint and processing overhead.

In the following sections we describe the data structures and algorithms necessary to implement an audio DSP operating system. To draw a clear distinction between the features and characteristics of a general-purpose operating system and the specific requirements of a real time DSP, we use the term *real time kernel* (RTK), instead of operating system, when describing the DSP control functions.

2. Characteristics of Audio DSP Tasks

As mentioned above, audio DSP tasks differ from general-purpose data processing tasks because DSP processes must handle uninterruptible streams of data. Loss of data occurs if the DSP process falls behind real time, producing an unacceptable click or pop in the audio output. Even with careful buffering, the DSP RTK must provide a guaranteed rate of execution for each task to avoid this problem.

The time-slicing paradigm of operating systems for general-purpose CPUs is generally not satisfactory for DSP tasks. With time-slicing, each concurrent program is allowed to run for a short slice of time before being suspended so that another program can run for its time slice, and so on. It is quite difficult in such a preemptive system to guarantee some minimum rate of execution for each program--especially when asynchronous interruptions occur while the operating system handles keyboard activity, etc. The result is that a program may become stalled for an indefinite period, which is not tolerable for most audio DSP tasks.

Audio DSP programs are usually designed to process more than one data sample at a time in order to spread the overhead of switching context and initializing registers across a group of samples. The required number of input samples, the input *block size*, must therefore be available before the DSP task can perform its block iteration. There must also be enough space for the DSP task to store the corresponding block of output samples. In other words, in order to execute one iteration the DSP task must have its input/output requirements satisfied. Choosing a large block size typically increases computational efficiency at the expense of requiring greater latency and more RAM. Conversely, choosing a small block size reduces latency and memory requirements while increasing processing overhead. This tradeoff is part of the careful engineering required for concurrent audio processing.

Audio DSP tasks may require a synchronous or asynchronous control stream in addition to the I/O data streams. For example, the coefficients of a digital filter may need to be

updated or the attenuation applied to samples entering a mixer may require a change. The control stream is generally of much lower bandwidth than the data streams, but the DSP task must still be able to respond quickly and correctly to the control messages.

3. Real Time Kernel (RTK) Features

The DSP RTK is responsible for managing the hardware resources, scheduling process execution, and generally coordinating the movement of data through the DSP system. The essential features provided by the RTK include:

- Managing hardware I/O devices
- Loading and unloading DSP processes
- Interconnecting processes
- Scheduling process execution
- Transferring control information to and from DSP processes
- Maintaining data structures

In order to minimize the size of the RTK, we adopt a *trusted task* model. The run-time version of the RTK does not include extensive error detection and bounds checking. It is the responsibility of the DSP programmer to ensure that the process obeys the conventions for access to system registers, system memory, and other shared resources. Furthermore, the programmer must avoid writing code that takes excessive time to execute, locks out interrupts for extended periods, and other "antisocial" behavior. Although the trusted task model may seem to place extra pressure on the DSP programmer, it is actually the normal state of affairs for programmers whose code typically ends up in a ROM: the code simply has to be right.

A simple audio DSP system is depicted in Figure 1. The example contains three hardware *devices* (ADC, DAC, UART), and four DSP software *processes* (MIDI parser, synthesizer, sample rate converter, mixer). Note that the data rates may differ from one process to another and from input to output of a process. The MIDI data arriving at the UART is asynchronous and low bandwidth, while the audio data streams from the ADC and the synthesizer must be mixed synchronously.

The DSP RTK becomes active when the DSP is reset. The RTK may either contain hardwired instructions to set up the specific DSP processes and devices, or the instructions can be passed from a host processor. In any case, once the RTK has completed the loading and initialization tasks it can begin executing the various processes, enabling the transfer of data among the processes and devices, and dispatching control messages from the host to the affected DSP processes.

4. Interconnection Using Data Streams

To implement a DSP system like the example of Figure 1, it is necessary for the data stream out of a device or process to be conveyed to the input of another device or process. In our non-preemptive RTK, only one process is actually executing during a

block iteration so the output block of the active process must be stored temporarily until the process destined to receive the data becomes active. Furthermore, the block sizes of the generating and receiving processes may not necessarily be the same, so the two processes may execute a different number of times in any real time interval. We handle these issues by defining a circular buffer FIFO (first-in first-out queue) in RAM to hold the data between each pair of processes. The computational expense of the data buffers is minimal, since most DSP architectures and instruction sets include excellent support for FIFO modulo buffering.

The processes attached to a circular buffer maintain data structures describing the buffer size, location, and current read/write positions. The process or device writing data into the buffer has a *write stream* structure, and each process or device reading data from the buffer has a *read stream* structure. This concept is shown in Figure 2 for an example in which two read streams are attached to a buffer.

The write stream structure contains:

- Memory space flag (e.g., program or data, X or Y, etc.)
- Buffer base address
- Buffer current write address
- Buffer size
- Pointer to associated read stream structure

The memory space, base address and buffer size define the location and extent of the modulo buffer. The current write address is updated as new data is written into the buffer.

The read stream structure contains:

- Memory space flag
- Buffer base address
- Buffer current read address
- Buffer size
- Data available to be read
- Pointer to associated write stream
- Pointer to next read stream structure (or null)

The memory space, base address, current address, and size elements correspond to the information in the write stream structure. The *Data available to be read* structure member contains the number of data elements in the buffer that have not yet been read by this stream. The available data count must be incremented when the associated write stream places new data into the buffer, and decremented when the read stream takes data out of the buffer.

If more than one read stream is attached to the buffer, a pointer (address) is provided to identify the companion read stream. This allows the write stream to monitor the *data*

available counts for each read stream so that no new data is written over old data until all of the read streams have had access to it.

The RTK contains functions to create and allocate write streams, buffers, and read streams. In a typical implementation the RTK manages a fixed number of available buffers with a range of pre-specified sizes.

In Figure 3 the simple example of Figure 1 is redrawn to show the interconnecting FIFOs and streams.

5. Input/Output Hardware Devices

The DSP *devices* managed by the RTK require data structures to identify how to open, initialize, perform data transfers, and close the device. The device data structures contain:

- Pointer to Configuration function
- Pointer to Open function
- Pointer to Close function
- Pointer to associated read or write stream structure
- Pointer to interrupt service routine (ISR)

The function pointers are filled in with the address of the corresponding subroutine supplied by the DSP programmer. The RTK retrieves the address from the structure and jumps to that subroutine.

Similarly, the RTK jumps to the specified interrupt service routine when the device asserts its interrupt. The RTK maintains a list identifying which interrupt corresponds to which device. When an interrupt occurs the RTK determines which device requires attention, then calls the corresponding ISR. The RTK passes the stream structure pointer to the ISR so that data can be moved to or from the associated stream buffer.

6. Processes and Process Scheduling

Each DSP program is referred to as a *process*. The RTK represents each DSP process using a linked list of process structures. Each process structure contains:

- Pointer to next structure in the process linked list
- Process ID number
- Process priority level
- List of Read Stream structures and minimum read block sizes
- List of Write Stream structures and minimum write block sizes
- Pointer to iteration function
- Pointer to host message function

The process ID number is a unique integer assigned to each active process by the host. The priority level is an integer used to sort the active processes according to their required response time. The RTK organizes the process linked list in order of decreasing priority level. The read stream and write stream lists contain pointers to the structures attached to this process, along with the minimum amount of input data and output space that must be present before the process can run one iteration (the input and output block sizes). The addresses of the iteration subroutine and the host message subroutine are also included in the process structure. The process linked list is depicted in Figure 4.

If a new DSP process is started a process structure is created and inserted as a new element in the process linked list. Similarly, a process to be deactivated has its process structure removed from the linked list.

The RTK schedules process execution as follows. Beginning with the head of the process linked list, the RTK compares the read and write stream pointers for the process with the corresponding minimum read and write block sizes to see whether there is sufficient space for the process to run one iteration. If there is sufficient input data and output space, the RTK calls the process' iteration function, passing along the pointers to the read and write streams. Once the iteration function is complete it returns control to the RTK, which proceeds to examine the next process in the linked list. In this way, the RTK executes each active process whenever sufficient data and space is available. Process execution is throttled naturally by the rate at which data enters and leaves the DSP system, so processes with high sample rate streams are executed more frequently than processes with lower rate streams.

The process priority level can be used by the RTK to alter the simple round robin execution algorithm. Since the process list is sorted in order of priority, the RTK monitors and dispatches each process in turn until all of the processes at the highest priority level are serviced. When the RTK determines that none of the highest priority processes can run (due to insufficient input data or output space), the round robin continues with the processes at the next lower priority level. After handling the processes at that priority level, the RTK immediately returns to the head of the process list and monitors the highest priority processes again. In other words, the higher priority processes are given more frequent monitoring for execution. For a lower priority process to be executed, the RTK must determine that all processes with higher priority have currently satisfied their input/output requirements.

This priority level arrangement was developed to accommodate DSP systems in which certain processes have very tight latency requirements to meet a certain performance specification, such as loop or phase delay. It is also possible to adjust the execution priority through carefully choosing the read and write stream FIFO sizes and minimum block lengths.

It is important to note that no matter how clever the RTK scheduling algorithm may be, the actual instruction rate of the DSP microprocessor must be sufficient to handle all of the concurrent tasks. The DSP programmer and system integrator must still use good

engineering analysis to determine the worst-case processing load and latency for each task [1].

7. Host Messages and Control

The RTK provides a uniform mechanism for the host system to communicate with the kernel and each of the DSP processes. The first word of the host message contains the process ID number (zero for the RTK itself, greater than zero for each DSP process). The RTK decodes the first word of the message, then dispatches the host message function from the process linked list. The host message function receives a pointer to the buffer containing the remainder of the host message. Other than the first word being the ID number, the message format is left entirely up to the programmer.

The manner in which the host informs the DSP system that a message is waiting to be processed will vary from system to system. The host may communicate with the DSP via a unidirectional host command port, a serial port, a bidirectional mailbox, or some other arrangement. The RTK can be configured to handle the host message immediately if its arrival generated an interrupt, or to defer acting on the host message until control is passed back to the kernel at the completion of the currently executing process.

8. Discussion

DSP programs intended to run under the Real Time Kernel must obey the conventions for passing parameters, updating read and write streams, receiving host messages, and so forth. The DSP programmer needs to ensure that any processor state information, register contents, and arithmetic modes be initialized at the start of the process iteration function, since there is no guarantee regarding the order in which the RTK will execute the active processes, nor is there usually any restriction on the processor resources available to each concurrent task.

Since the RTK does not support memory allocation, task swapping, or other types of dynamic processing, it is necessary for the concurrent tasks to be coordinated to all fit within the available DSP system memory. This requires the various program modules to be linked carefully to make efficient use of data and program RAM. We have found that the details and constraints associated with using the RTK are small, and DSP programmers can port their code to run under the RTK with minimal inconvenience.

The uniform manner in which the RTK treats each DSP process and device allows each DSP program to be largely autonomous and self-contained. Each DSP process only needs to know its read stream and write stream interfaces: it does not need to be aware of where the streams are coming from or going to. This independence is significant for several reasons. First, it allows the DSP system to be reorganized with new modules and new interconnections without the need to rewrite large sections of code. Second, individual modules can be debugged and tested by temporarily connecting the read and write streams to a host port or some other off-line data logging mechanism. Finally, the independence of each process enables the integration of object code modules from

different vendors without the need to divulge source code or other intellectual property. This allows each vendor to test its own code modules under the RTK without the need to coordinate closely with the other vendors. It also allows modules to be tuned and updated individually.

In addition to the run-time version of the RTK, we have developed a profiling version that includes the ability to count instruction cycles, monitor FIFO depths, and determine processor idle time. These features are useful during the debugging and module integration phases of development.

The run-time version of the RTK typically requires less than 1500 instructions and enough data memory for the control structures and FIFOs (approximately 1000 words for the system of Figure 3). The computational overhead for a typical system is about 0.5 MIPS.

9. Conclusions

In this paper we have described the features of a multitasking DSP Real Time Kernel. The system allows concurrent execution of several audio tasks using a single DSP microprocessor. The RTK is compact and efficient in order to minimize the processing overhead and memory requirements. The process scheduling algorithms, stream handling, and host message facilities are very general in nature, and suitable for use with any standard DSP microprocessor architecture.

10. Acknowledgement

Portions of this paper are based on proof-of-concept development work and a confidential report prepared by Duane K. Wise.

11. References

[1] Lapsley, P., Bier, J., Shoham, A., and Lee, E., *DSP Processor Fundamentals*, New York, NY: IEEE Press, 1997.

[2] Berkeley Design Technology, Inc., *DSP Design Tools and Methodologies*, Fremont, CA: Berkeley Design Technology, Inc., 1995.

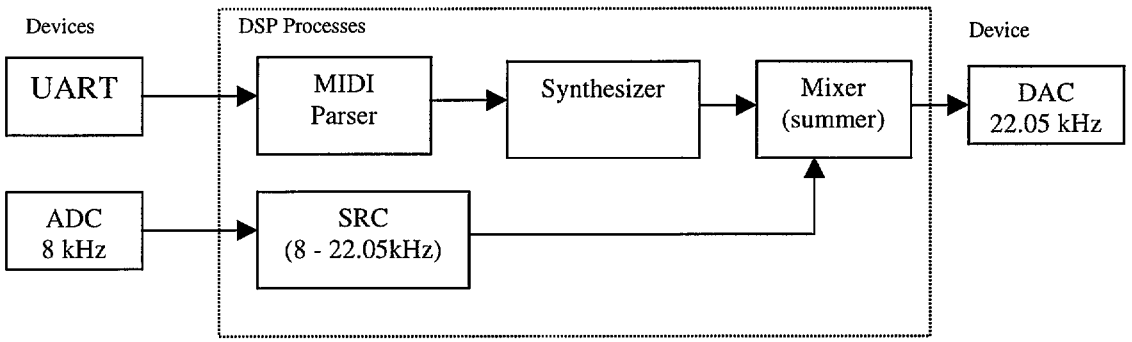

Figure 1: An example multitasking DSP system with 3 devices (UART, A/D converter, and D/A converter) and 4 processes (MIDI parser, synthesizer, sample rate converter, and mixer) running on the DSP microprocessor.

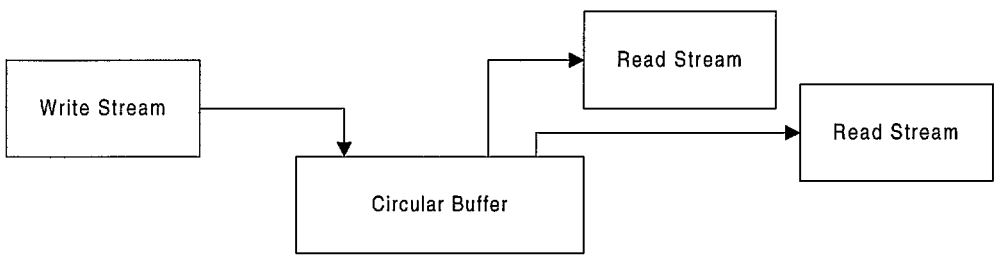

Figure 2: The DSP RTK uses circular buffers (FIFOs) to interconnect DSP processes and devices. The Read Stream and Write Stream structures contain the current read and write positions within the FIFO.

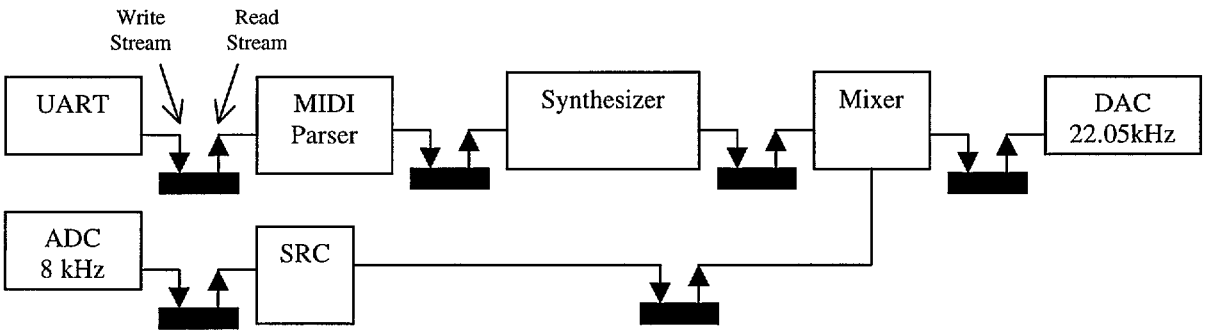

Figure 3: The example DSP system of Figure 1 with the circular buffers and read/write streams shown explicitly.

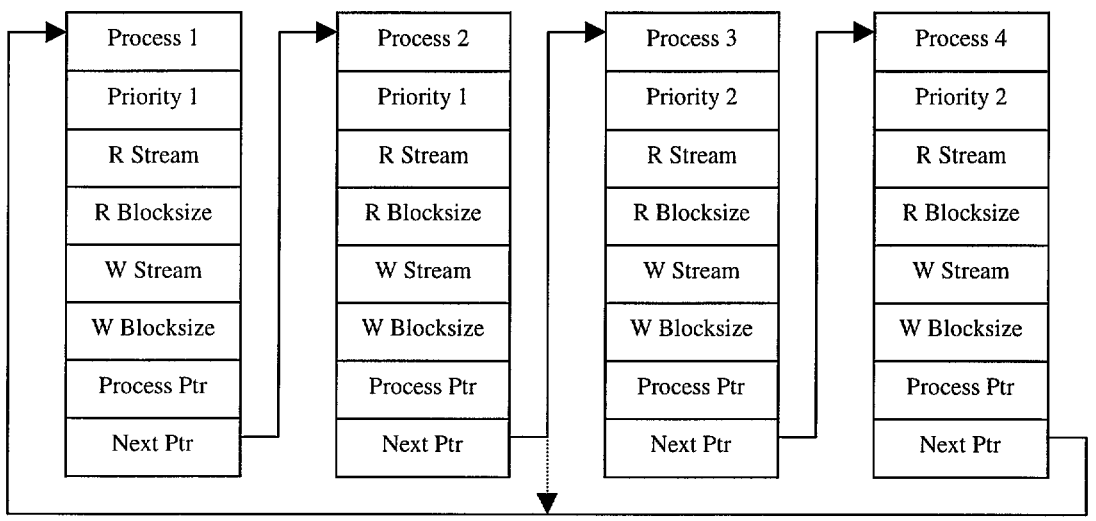

Figure 4: A schematic representation of the process linked list. Each active process is identified by a structure containing task information, priority, and a pointer to the next process structure. The RTK does not execute the lower priority processes until all the higher priority processes have been satisfied.