A CORBA-Based DAMA Modem for the Digital Modular Radio (DMR)

John D. Bard, Ph.D., TJ Mears, II
Mnemonics, Inc.
Melbourne, Florida

Abstract

The advent of the Software Defined Radio (SDR) concept has offered a new paradigm for the modem designer. Consider the requirements of implementing a MIL-STD-188 UHF DAMA modem in a CORBA-based software environment. Here direct access to the hardware is denied. Interfaces to the synthesizer and AGC functions are wrapped in CORBA-based calls. In addition, no hardware timing signals exist to help establish frame synchronization. These and others are the requirements that were successfully met in our implementation of the UHF SATCOM modem for the Digital Modular Radio (DMR) program. This CORBA-based software modem solution is not attached to any particular piece of modem hardware and, as such, is highly portable. This paper reveals the architectural features that allow the inherently slow CORBA-implementation to synchronize and respond to the real-time requirements of the DAMA network.

DMR Hardware Configuration

The DMR is a software re-programmable, multi-channel transceiver system covering the 2 MHz to 2 GHz frequency range and is capable of operating on four independent full-duplex, channels simultaneously. The hardware is designed to be general purpose and to support a wide variety of both military and commercial waveforms. Any combination of supported waveforms, and even multiple instances of the same waveform, can be running simultaneously within the platform. New waveforms are enabled via software download. A view of a single channel in the system is shown in Figure 1

DMR Software Configuration

The DMR SATCOM is composed of the following major Computer Software Configuration Items (CSCI):

- Transmitter/Receiver (T/R) CSCI
- Radio Control CSCI
- Modem CSCI
- Satellite Communications Controller (SCC) CSCI
- WITS INFOSEC Module (WIM) CSCI
- Serial Red I/O Manager (SRIOM) CSCI
- Audio CSCI
- Red Controller CSCI

The CSCI's are composed of CORBA objects that are distributed across the various processors within the WITS system. The focus of this paper is the CORBA objects within the Modem CSCI and their interaction with other SATCOM objects.

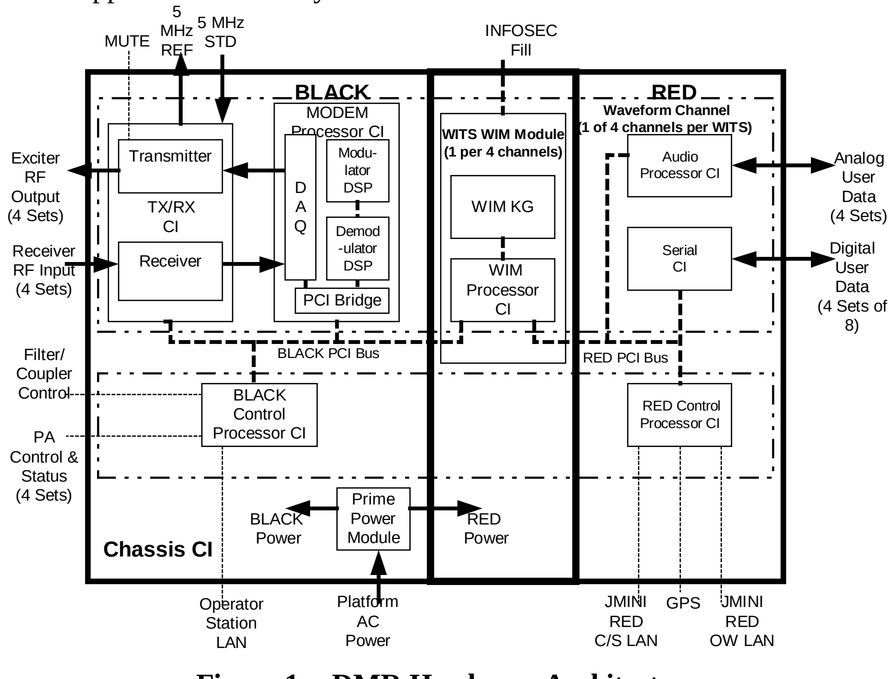

Figure 1 - DMR Hardware Architecture

Modem Objects

The Modem CSCI consists of four CORBA objects: the Modulator Object, the Demodulator Object, the Tagger Object and the Frequency Control Object. These objects provide the functions necessary to inter-operate with the UHF SATCOM DAMA waveform. These functions include carrier acquisition, baud synchronization, frame synch, message synch, interleaving and de-interleaving, coding and decoding control, frequency control and modulation. Figure 2 shows the notional interactions between the Modem CSCI and other SATCOM objects.

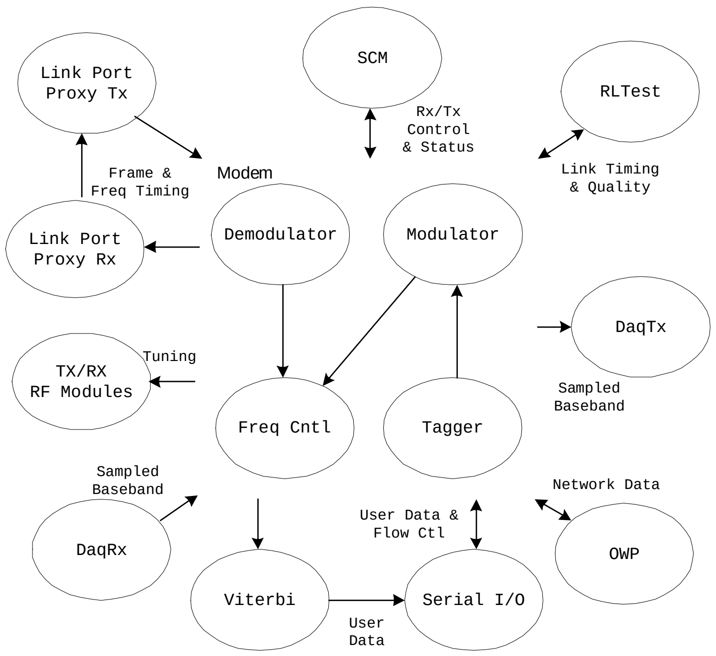

Figure 2 Modem CSCI Interactions

The entire modem is under the control of the Slot Connect Manager (SCM) object. Recall that DAMA is really a TDMA multiplexer on the uplink and demultiplexer on the downlink [1]. The SCM object also manages the mapping of which serial ports (Serial I/O object) go to which TDMA slots in both the automatic control (AC) and distributed control (DC) modes of operation. For receive, the SCM object tells the Demodulator object when (and on what frequency) to listen, what to listen for and where to pipe the data it hears. For transmit, the SCM object tells the modulator on a frame-by-frame basis where to get its data from, how to modulate it and when (and on what frequency) to actually transmit it. We call this the "dumb modem" approach. The Modem object does not have any knowledge of the TDMA link protocol it merely does what it is told to do.

The other objects in Figure 2 are described as follows. The DaqRx object is the analog-to-digital converter. The DaqRx object produces a constant rate In-phase (I) and Quadrature (Q) data stream at a very low Intermediate Frequency (IF). This data stream is more accurately defined as near baseband. Conversely, the DaqTx object converts a digital data stream to analog for upconversion. Since the Modulator object and Demodulator object execute on different processors (dual ADSP 21060's), they are able to invoke CORBA methods on each other through the Link Port Proxies. The Viterbi object, Orderwire Processor (OWP) object and Range/Link-Test (RLTest) objects are all destinations and/or sources of specialized data bursts. The modem does not treat these objects any differently than a Serial I/O object. The SCM tells the Demodulator object to send data to only a particular CORBA object reference. The Demodulator doesn't care if that's an OWP object, a Viterbi object or Serial I/O instance 3.

Whereas the de-multiplexing function of the receive chain is nicely handled via indirection provided by the CORBA object reference, the DMR framework cannot easily support the reverse. When a source object uses the DMR framework to push data into a destination object, the destination object does not have access to the source's CORBA object reference. The function of the Tagger object is to insert an identifier into a data stream so that the Modulator object knows where the data stream came from. This supports the use of virtual channels within the radio and allows the user port connections to be re-assigned during operation. If the system has 4 instances of Serial I/O object there are a corresponding 4 instances of Tagger object to insert an identifying tag into the data streams. This mapping is known by the Slot Connect Manager. The TDMA multiplexing function is then easily performed by the Modulator when the Slot Connect Manager tells it to take "n" bits from data stream 3 and to transmit it at time "y".

The Demodulator Object

The Demodulator object accepts in-phase and quadrature inputs and produces soft-decision or hard-decision output as commanded. Control of the Demodulator object is accepted via the getBurst() method. Up to 30 getBurst's can be queued at one time. Contents of a getBurst define everything the Demodulator object needs to know in order to process any Mil-Std 188 series phase-shift keyed (PSK) burst.

The Demodulator object spans three threads of execution. The Demodulator object only explicitly creates (and destroys) one thread. The other threads are managed by the DaqRX object and the ORB. These threads are identified as follows:

Thread 1) demodHigh - the purpose of this thread is to take I and Q samples originating from the analog-to-digital converters and present them to the Demodulator object. The flow of data through this thread is a sustained (and very accurate) “n” samples per second. No matter what state the Demodulator is in all samples are counted, thus the Demodulator derives its timing from this data stream. When the Demodulator is executing a getBurst(), I&Q data from this thread are time-tagged and buffered in a semaphore-controlled ring buffer. When the Demodulator is idle, I and Q data from this thread are counted and then "thrown on the floor". By design, this thread runs at a high priority thus insuring the integrity of the data stream, i.e., consecutive data samples are in fact temporally related. This thread is created and destroyed by the daqRX object.

Thread 2) demodLow - the demodLow thread executes getBurst() commands. After initialization, demodLow pends on the getBurst message queue waiting for a getBurst() to arrive. Upon retrieving a getBurst() from the message queue, demodLow notifies demodHigh (via shared memory), when to start writing IQT (in-phase, quadrature and time-tag) samples to the IQT ring buffer. demodLow() then reads and processes these samples in blocks called atoms. As data is processed demodLow() advances through an acquisition and track state machine culminating in the output of data decisions (soft or hard as commanded by the SCM object) to the "Next" object in the receive chain. When burst processing is completed demodLow resets itself and goes back to the message queue for more getBurst()'s. The demodLow thread is created and destroyed by the pskDemodulator object.

The body of pskDemodulator is contained in the thread demodLow(). All inputs and control relative to demodLow are asynchronous. This alleviates the need for demodLow to complete processing on a foreground basis. Furthermore, continuous feed of new control information and IQT data can be received by demodLow() without apparent interruption to current processing. All inputs and control are buffered. IQT data is buffered in a ring buffer - access to which is arbitrated by a mutex semaphore. Control data (a.k.a. getBurst requests) are buffered in a 30 deep POSIX message queue.

Output and status from the Demodulator are synchronous. That is demodLow will not process any new incoming data until it has successfully passed data on to the “Next”. Similarly, demodLow is blocked while registering the status of the current burst processing by invoking the burstFound() method on the Slot Connect Manager object. This blocking is minimized in that these are “one way” CORBA calls.

Thread 3) getBurst() – The thread pool concurrency model [2] is employed to keep inter-processor ORB communication overhead to a minimum. This thread of execution is launched by the ORB the first time the getBurst() method is invoked. The result is that the contents of a getBurst are written to a POSIX message queue.

The set of interactions required of the Demodulator object is most complex when a TDMA frame synchronization event occurs (as signaled by the SCM object). This series of events is described in the transaction diagram, Figure 3.

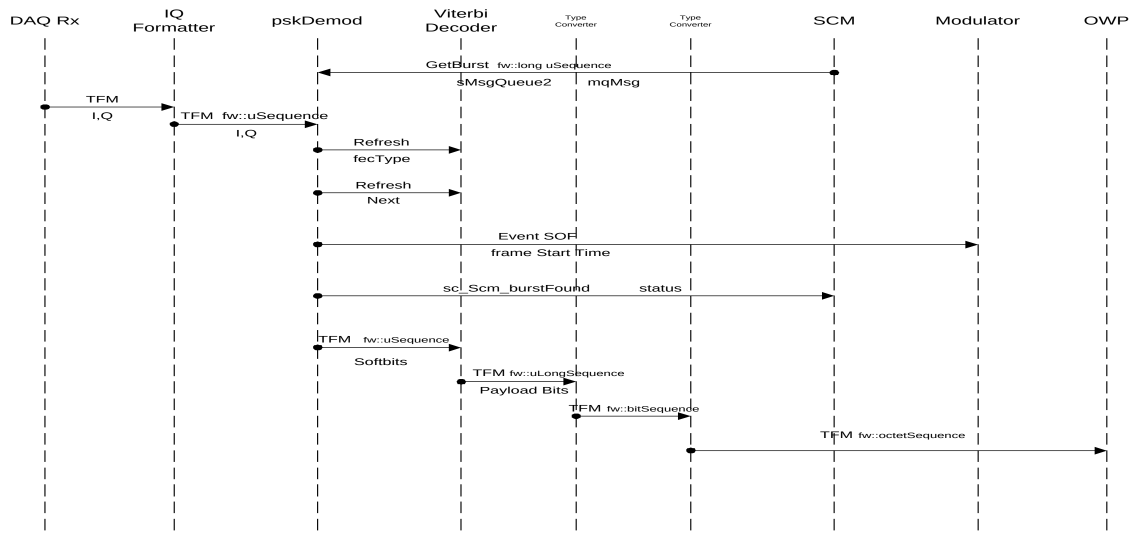

Figure 3 Transaction Diagram – Frame Synchronization Burst

The Modulator Object

The Modulator object accepts binary data in from multiple data sources and produces modulated in-phase and quadrature output samples for the D/A converter. Control input to the PSK Modulator object is accepted via the putBurst() method. Up to 12 putBurst's can be input at one time - 8 for user data bursts and 4 for network control bursts. Contents of a putBurst define everything the Modulator object needs to know in order to output any MIL-STD 188 series PSK burst. Eight slots of user data are input via the Taggers and stored in individual data buffers. Network control data doesn't require a Tagger since its source is not changeable.

The Modulator object spans six threads of execution. When a client invokes the Modulator's Activate() method, the threads are launched. Conversely when the Modulator is Deactivated() the threads are destroyed. These threads are identified as follows:

Thread 1) DaqTxFlowControlThread – This thread, executing at the highest priority, runs in response to a semaphore posted in the Daq interrupt service routine and provides a packet of I/Q samples to the D/A. The thread invokes the Modulator On() method which performs the actual, encoding ,interleaving, and modulation of the data. An interrupt is generated every m^{th} sample clock at which time m I and Q samples are transferred to a DMA double buffer to be clocked into the D/A. The packet size is chosen to allow time between interrupts for lower priority threads to run. The challenge of the Modulator object is to keep the DMA double buffer full. In the event no modulation occurs during an interrupt cycle, "guard" samples are output. These result in no output signal from the D/A and correspond to the guard time between bursts. Outputting a continuous stream of samples allows the Modulator to maintain timing via sample count.

Transmit burst processing is handled by a state machine at each invocation of the On() method. State transitions for an example transmission are shown in Figure 4. Six states are indicated:

- Guard Generation – guard samples are output between bursts.
- Guard Generation/Preamble Modulation – includes both guard and modulated preambles bits.
- Preamble Modulation – only preamble
- Preamble Modulation/Data Processing – preamble modulation continues and encoding, interleaving of user data begins.
- Data Modulation/Data Processing – user data processing continues and modulation of the payload begins.
- Data Modulation/Guard – transitions from data modulation back to guard, i.e, the burst ends.

Thread 2) putBurst() – This thread is identical to the getBurst() of the demodulator and is used convey all information needed by the Modulator for a transmission.

Thread 3) burstTxStatusThread – This primary job of this thread is to convey the burst completion status to the SCM. It does this by invoking the burstSent() method of the Slot Connect Manager at the end of a burst. . A second burstSent call will be made if there is no more data to be transmitted for this data source, an indication that the user has completed the transmission.

Thread 4) ModControl – The purpose of this thread is to "throttle" the data flow from the Serial I/O objects to the Modulator. This throttling is achieved through calls to the On() or Off() methods of each I/O object. The decision as to what method to call is made based on the fill-status of the buffer associated with each data source object.

Thread 5) The FrequencyControlThread is used to send the frequencies to the Frequency Control object and add frequency tuning events to the event queue when frequency hopping is enabled. The frequencies and events are added at the start of each modulator frame.

This thread waits for a semaphore to be posted from the On() method. When the semaphore is posted the thread transfers a table of transmit frequencies to the Frequency Control Object (for later transfer to the Transmitter objects). These frequencies are collected during the previous frame from the various putBurst messages.

Thread 6) User Data – user data is conveyed to the bit buffers on the Modulator from the Taggers via on-board Orb call. Packets sizes are chosen as a trade-off between efficiency and data latency. This thread runs at a lower priority than the DaqTxFlowControlThread, meaning it is executed between interrupts.

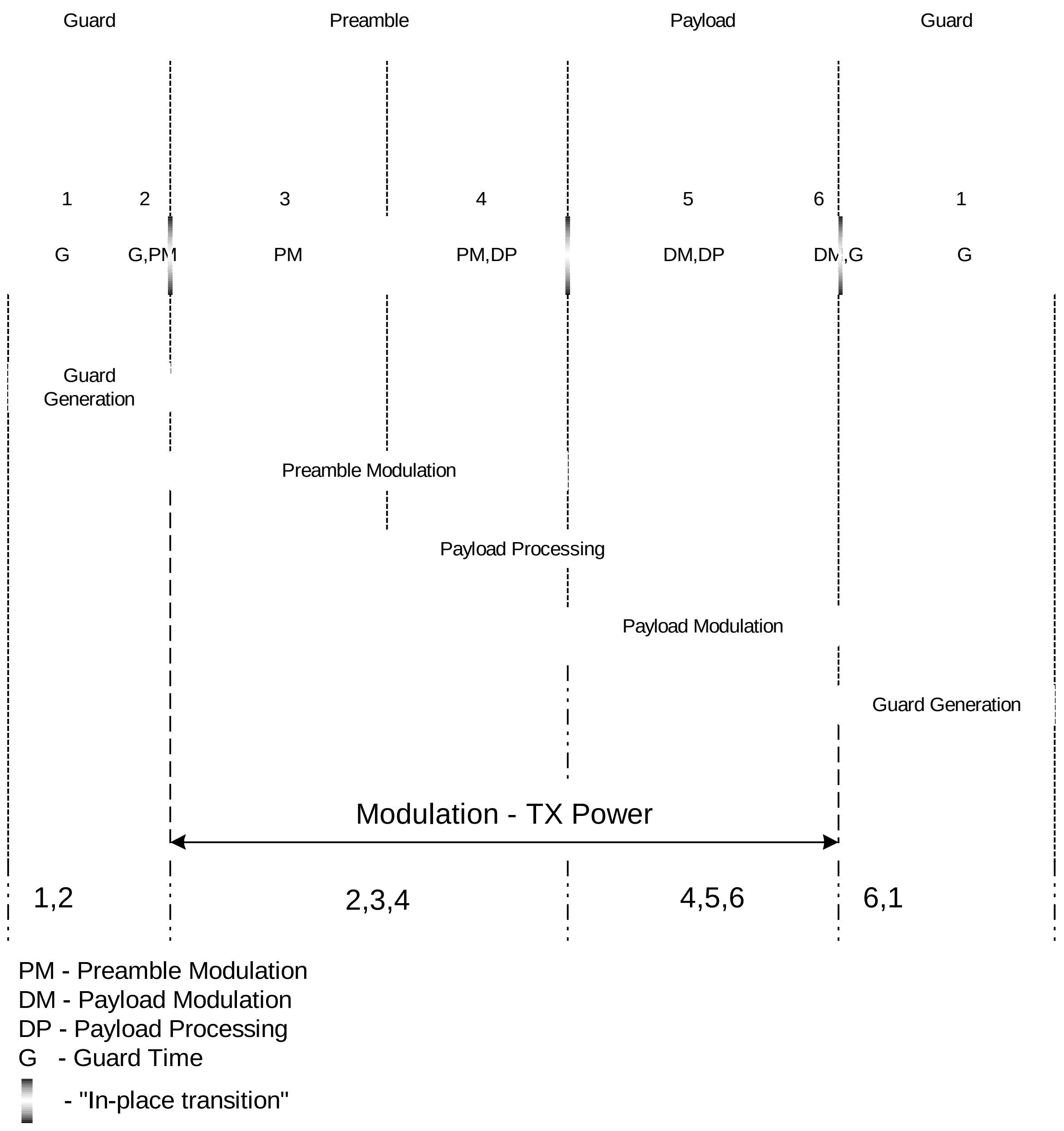

Figure 4 Transaction Diagram – User Transmit

Included with the samples transferred to the D/A is timing information used for the generation of the transmit keyline and frequency hop hardware strobes. Tying these to the samples allows precise control over the timing of these events, as is required in the DAMA/TDMA waveform. This also provides a convenient mechanism for the compensation of the satellite range delay and the adjustment can be made with sample count precision. Since the Link Port Proxy that relays the startOfFrame event is interrupt driven, there is very little uncertainty in the slave synchronization of the Modulator object to the Demodulator object and the precision can be maintained for extended periods.

Conclusion

In this paper, an implementation of a CORBA-based software modem has been presented. It is highly portable, has a minimum number of direct hardware interfaces and, can be readily upgraded as new requirements emerge.

References

[1] MIL-STD 188-183, Interoperability Standard for 25-KHZ TDMA/DAMA Terminal Waveform, DoD-DISA

[2] ORBacus For C++ and Java, Object-Oriented Concepts, 1998, pp. 114-117