Modem and Telephony Component Product User Manual

Miller Engineering Services, Inc.
E-mail: sales@mesi.net
Web: www.mesi.net

CONTENTS

1 Product Brief

The MESi Modem and Telephony product is a suite of integrated signal processing software components for use in telephony signalling, voice, fax, and data modem applications. Collectively, they provide the user with signal generation and detection algorithms typically found in data pump chip sets but in the form of executable software functions. The product was developed to provide portable code that addresses the current market's strongest demands: lowest cost, minimum MIPs and maximum user ease. Toward that end, algorithms were originally developed as a simulation called "VSIM" written in a restricted form of C and compiled using the Borland™ 5.0 C compiler for MS-DOS™. This restricted form of C uses only integer math functions such as multiply, shift, and modulo addressing that are available in Digital Signal Processors (DSPs). There are no divisions, transcendental functions, or floating-point operations in the portable code (the simulation uses floating point operations in the channel models). This method allowed the algorithms to be optimised to target DSPs for speed (minimum MIPs), portability, and debug utility. The memory design uses a common set of vector and structure elements for all modems and generators/detectors. The design philosophy from the was to provide the user with a minimum set of functions to call, status reports to monitor, and a unified I/O access method.

The VSIM simulator runs on IBM-PC (or compatible) platforms under the MS-DOS™ operating system or under the Microsoft Windows™ operating system in a DOS window. It allows command line options that may be listed by running VSIM with a "?" as its argument. Briefly, these options allow the user to specify the modem type, power levels, noise and gain distortions, 2 trace sweep or X-Y plot modes, line delay and echo loss parameters, and file I/O. For half duplex modems, files containing digitised samples may be input to VSIM and the demodulator operation may be monitored. This has proven to be very useful in situations where the user's hardware (the DSP) is inaccessible but samples can be acquired from the line. VSIM outputs (file and graphical) are typically altered by source code modification and re-compile - it is intended to be a tool for developers rather than a demonstration in PC programming prowess.

The assembly implementations include C interface functions such that the user need not have any knowledge of the memory structure or operation of the modems. Line output samples at 8 kHz are generated by a function called "transmitter()" and input samples are processed by a function called "receiver()". These two functions implement a pair of state machines, which the user can configure, and monitor the state progressions. For the transmitter, the user can call a function to configure a generator or modulator and then poll the state identifier to follow the progress. Users can intervene and extend states in the transmitter for various purposes (v.32 for example). For the receiver, the user can configure the detector to enable certain processors (v21, v27, and v29 for FAX as an example) and then monitor the state identifier to determine when data flow has started.

The design of the MESi products has been tailored specifically toward algorithms that reduce operations to the lowest possible rate and toward methods that are super efficient when run on most DSP chips. The C source in the VSIM simulator "looks funny" when viewed by the programming purist but in fact ports directly and efficiently to the DSP math unit and dual memory access architecture. The math operations are all 16 bit integer - no floating point are used even on floating point DSPs such as the 'C3x or SHARC. Thus the code is completely portable between 16 or 32 bit integer and floating point DSPs with exactly the same performance as in the simulator (exact comparisons of the simulator and the DSP results dumped to files are used in the porting process).

2 Product Use - Getting Started

The best way to get started with a product demonstration program is to "unpack" the software delivery package and run "make.exe" or "nmake.exe". If you don't have a make utility, then manually execute the compile/assemble/link steps in "makefile" for your particular tools. This process will generate an executable called "vmodem.out" (or vmodem.exe, vmodem.dxe, vmodem.axe depending on target device tools). Load this executable on your target (or simulator) and run it for several seconds. At this point you will have transmitted a fax CED tone and a v27 burst at 2400 bits/sec., looped these signals into the receiver, and detected and demodulated the data. The transmitter and receiver state transitions have been

captured in the “Tx_log[]” and “Rx_log[]” buffers respectively, along with the frame count values at the state transitions. The received data symbols are copied into the transmit data buffer to demonstrate “fax bypass” data handling. Vmodem.c shows minimally how you might use the transmitter() and receiver() functions, and the various memory access and configuration functions in your system as listed below:

Include - Include "common.h" and "vmodem.h" and function specific include files (such as v27.h, gendet.h, etc.) in your calling program. They have the #defines and externs that you will need to monitor and control the “transmitter()” and “receiver()” functions, and access Tx_block[] and Rx_block[] memory structures.

Initialize Tx_block and Rx_block - Call the initialization routines "Tx_block_init()" and “Rx_block_init()” once as a part of your startup procedure. These functions set up Tx_block and Rx_block minimally so you can start calling transmitter(), receiver() without risk of uninitialized pointers or essential memory context.

Configure Tx_block and Rx_block - Customize the initial configuration of the Tx_block[] and Rx_block[] control sections, if desired, to change global parameters such as Tx->scale, Rx->num_samples, etc. You can either use one of the functions or macros referenced in “vmodem.h”, or directly access Tx_block[] and Rx_block[] using a suitable pointer.

Configure transmitter() - Initialize the desired signal generator or modulator using the “Tx_init_xxx()” functions referenced in the component specific include files. Subsequent calls to transmitter() will generate samples of the specified component until you re-configure it.

Configure receiver() – For full-duplex modems (such as v22bis, v32bis, v34), the “Tx_init_v##()” function call takes care of initializing receiver() and no configuration is needed. For half-duplex modems (such as v.17, v.21, v.27, v.29), set the detector mask and initialize the detector using the functions referenced in “gendet.h”. Subsequent calls to receiver() invoke only the components you have masked on until you change the Rx->detector_mask. If you don’t use the provided FAX_DETECT_MASK or DATA_DETECT_MASK, then be sure to OR in AUTO_DETECT_MASK when setting Rx->detector_mask to enable detection.

Call receiver() - Start calling “receiver()”. Every time you call “receiver()”, it will attempt to process Rx->num_samples or more samples in Rx_sample[] at 8 kHz, and circularly advance Rx->sample_tail accordingly. If a demodulator is operating, demodulated symbols are written to Rx_data[] as indicated by displacement of Rx->data_head from Rx->data_tail.. Monitor Rx->state_ID to follow the receiver() state progression through the states #defined in each component specific include file. The same constraints apply to Rx_data[] and Rx_sample[] management as described above for the transmitter. The "receiver()" function returns a zero if there are less than Rx->num_samples samples in Rx_sample[], and a non-zero if it has processed the samples present. If you drop or add erroneous samples then you can expect large blocks of bit errors especially for the higher rate demodulators, so make sure that the continuum of samples is maintained. The demodulator symbol clock tracks that of the (other modem's) modulator that it is locked to. You can estimate the demodulated symbols produced per call of “receiver()” using the following equation:

symbols= (symbol rate in symbols per sec./8000 Hz)

Due to symbol clock offset, occasionally you may get 1 more or 1 less symbol per call than the equation indicates so you should design your data interface accordingly. You can improve your degraded line performance by enabling only the demodulator/tone detectors for those signals that you are expecting. For example, if you know that you are expecting HDLC messages for fax, then enable only v21 by calling “set_Rx_detector_mask(AUTO_DETECT_MASK|V21_CH2_MASK. Also consider that “receiver()” will probably detect and demodulate your transmitted signal from it’s return echo, so at times you might want to mask all detectors off by calling “set_Rx_detector_mask(0)”.

Call transmitter() - Start calling "transmitter()". Every time you call "transmitter()", it will attempt to generate "Tx->num_samples" samples at 8 kHz into "Tx_sample[], and circularly advance Tx->sample_head. If a modulator is operating, symbols are extracted from Tx_data[] as indicated by a displacement of Tx->data_tail to Tx->data_head. Monitor Tx->state_ID to follow the transmitter() state progression through the states #defined in each component-specific include file (v27.h, v29.h, etc.). The "transmitter()" function will return a zero if Tx->sample_tail (read pointer) is less than Tx->num_samples ahead of Tx->sample_head (write pointer), and it returns a

non-zero if it has generated Tx->num_samples samples into Tx_sample[]. For data modems, "transmitter()" inserts all ones if the user does not supply data to Tx_data[]. For fax modems, it modulates whatever is in Tx_data[] pointed to by Tx->data_tail without checking Tx->data_head, so you need to put enough symbols into Tx_data[] prior to calling "transmitter()" to ensure that it does not run dry, and you might want to "prime" it prior to reaching the MESSAGE_ID state. Use the following equation to determine the number of samples produced per symbol:

samples= (8000 Hz/symbol rate in symbols per sec.)

For example, if Tx->num_samples is 20 and the modulator is v29 (baud rate is 2400 symbols/sec.), you need to load at least 6 symbols into Tx_data[], using circular pointer addressing as follows:

```
*Tx->data_head=symbol;
if (++Tx->data_head >= ptrs->Tx_data_start+Tx-
>data_len)
Tx->data_head=ptrs->Tx_data_start;
```

Symbols are defined by the relevant Standard (Bellcore, ITU-T, etc.) modulation rate or baud rate as shown below:

Modem	**bit rate**	**baud rate**	**bits per symbol**

Baudot TTY	45.45 bits/sec.	45.45 symbols/sec.	1 bit/symbol
Bell103	300 bits/sec.	300 symbols/sec.	1 bit/symbol
Bell202	600 bits/sec.	600 symbols/sec.	1 bit/symbol
Bell202	1200 bits/sec.	1200 symbols/sec.	1 bit/symbol
Bell202	1350 bits/sec.	1350 symbols/sec.	1 bit/symbol
Bell212a	1200 bits/sec.	600 symbols/sec.	2 bits/symbol
v.17	14400 bits/sec.	2400 symbols/sec.	6 bits/symbol
v.17	12000 bits/sec.	2400 symbols/sec.	5 bits/symbol
v.17	9600 bits/sec.	2400 symbols/sec.	4 bits/symbol
v.17	7200 bits/sec.	2400 symbols/sec.	3 bits/symbol
v.21	300 bits/sec.	300 symbols/sec.	1 bit/symbol
v.22*bis*	2400 bits/sec	600 symbols/sec.	4 bits/symbol
v.22*bis*	1200 bits/sec.	600 symbols/sec.	2 bits/symbol
v.23	1200 bits/sec.	1200 symbols/sec.	1 bit/symbol
v.23	600 bits/sec.	600 symbols/sec.	1 bit/symbol
v.23	75 bits/sec.	75 symbols/sec.	1 bit/symbol
v.26	1200 bits/sec.	1200 symbols/sec.	1 bits/symbol
v.26*bis*	2400 bits/sec.	1200 symbols/sec.	2 bits/symbol
v.27*ter*	4800 bits/sec.	1200 symbols/sec.	3 bits/symbol
v.27*ter*	2400 bits/sec.	1200 symbols/sec.	2 bits/symbol
v.29	9600 bits/sec.	2400 symbols/sec.	4 bits/symbol
v.29	7200 bits/sec.	2400 symbols/sec.	3 bits/symbol
v.29	4800 bits/sec.	2400 symbols/sec.	2 bits/symbol
v.32	9600 bits/sec.	2400 symbols/sec.	4 bits/symbol
v.32	4800 bits/sec.	2400 symbols/sec.	2 bits/symbol
v.32*bis*	14400 bits/sec.	2400 symbols/sec.	6 bits/symbol
v.32*bis*	12000 bits/sec.	2400 symbols/sec.	5 bits/symbol
v.32*bis*	9600 bits/sec.	2400 symbols/sec.	4 bits/symbol
v.32*bis*	7200 bits/sec.	2400 symbols/sec.	3 bits/symbol
v.34	2400 bits/sec. to 33,600 bits/sec.	2400 symbols/sec. to 3429 symbols/sec.	16 bits/symbol for all rates

Table 1

Tx_sample[] and Tx_data[] are circular buffers. You must maintain Tx->data_head and Tx->sample_tail as circular pointers. An example follows showing an interrupt service routine that copies a sample from Tx_sample[] to a serial port transmit register and circularly wraps Tx->sample_tail. If the tail passes the end of the buffer, then wrap it around to the start.

```
void transmit_isr(void)
      {
      write_sample_to_DAC(*Tx->sample_tail);
      if (++Tx->sample_tail >= ptrs->Tx_sample_start+Tx->sample_len)
              Tx->sample_tail=ptrs->Tx_sample_start;
      }
```

3 Numeric Formats

All integers are 16 bits regardless of the DSP type. Most variables used in the modems are 16 bit signed fractional integers in Q.15 format. This means that the MSB is a sign bit and the remaining 15 bits are to the right of the decimal point. Therefore the numeric range is -32768 to +32767 corresponding to -1.0 to +0.9999695 With a Quantization Step Equivalent (QSE) of 1/32768.

3.1 Transmit Power Level

The default output power of the transmitter is -16 dB with respect to the internal numerical representation in the module. This output level corresponds to a Tx->scale value of 32767, or approximately 1.0. Lower Tx->scale value less than this will decrease the output voltage in a linear manner. Since the actual analog output voltage swing of the codec and the terminating impedance control the actual output power, the user must compute the output power of the codec and then adjust the analog circuit gain stages to achieve the desired target power level.

For sinusoidal signals, the root mean squared signal level (rms) is the peak level * sqrt(2)/2. The output power can be calculated as follows:

$$V_{rms} = \frac{V_{peak\text{-}peak}}{2} * \frac{sqrt(2)}{2}$$

The output power is the full scale output power of the codec minus the 16 dB nominal level in the transmitter.

$$P_{out} = 10*\log\left(\frac{Vrms^2}{R}\right) - 16$$

For a 2.0 volt p-p output driving a 600 ohm impedance, the output power is calculated as:

V_{rms} = 2.0 / 2 * sqrt(2) /2 = 0.707 volts

P_{out} = 10*log(0.707 * 0.707 /600) - 16 = -30.8 -16 = -46.8 dBw = -16.8 dBm into a 600 ohm impedance.

Note: It is important to use the actual or datasheet output values for the peak to peak voltage. A 3.3 voltage codec will not be able provide a 3.3 volt peak-peak voltage swing. This applies equally to the source impedance of the driver.

3.2 Receive Power Level

The input power is computed with respect to full-scale codec input and is a linear value. To compute the input power level in dB, perform the following calculation:

Rx Power (dB) = 10*log10(Rx->power/32768).

The Rx_init_measure_power() function puts receiver in a continuous power measurement mode where you can measure known input signals and calibrate your input analog gain path.

4 Memory Descriptions

4.1 Common Memory Components

There are a small number of data memory elements used for coefficient storage that are common to all channels and require only one instance in a system. These are declared in "vcoefs.c" (or “vcoefs.asm” for the assembly language ports):

sin_table[SIN_BUF_LEN] - This is a linear vector containing 1.25 cycles of sin() scaled to +/-32767. Its length is 1.25*256=320 and it is used for all complex trigonometry functions.

DFT_coef[DFT_COEF_LEN] - This is a circular vector containing 1.0 cycles of sin() scaled to +/-DFT_COEF_SCALE. Its length is 256 and it is used by the Hilbert DFT filters to implement sub-sampling band-pass filters. By taking advantage of the DSPs circular addressing mechanism, very efficient Hilbert DFT band pass sub-sampling filters can be implemented without the need for recursive memory storage, and close to 2-cycles/sample operation.

4.2 Channel Memory Elements

A channel consists of a set of linear and circular buffers associated with transmitter() and receiver(), and a "START_PTRS" structure containing the starting addresses for each of these buffers. The START_PTRS structure is defined in vmodem.h and is initialized either in code or in ROM as:

```
struct START_PTRS {
      short *Tx_block_start;
      CIRC *Tx_sample_start;
      CIRC *Tx_data_start;
      CIRC *Tx_fir_start;
      short *Rx_block_start;
      CIRC *Rx_sample_start;
      CIRC *Rx_data_start;
      CIRC *Rx_fir_start;
      short *EQ_coef_start;
      short *EC_coef_start;
      struct ENCODER_BLOCK *encoder_start;
      struct DECODER_BLOCK *decoder_start;
      CIRC *traceback_start;
      };
```

Transmitter() and receiver() each have a linear memory structure identified as Tx_block[] and Rx_block[] respectively that contains all control and component specific variables. These structures are sub-divided into CONTROL, COMMON, and ANNEX sections. The CONTROL sections (configured by Tx_block_init() and Rx_block_init()) are static in definition and function for all components, whereas the COMMON and ANNEX sections are re-configured (or recycled) depending on the particular component configuration. In the purest sense, Tx_block[] and Rx_block are actually *unions of structures* rather than arrays, as their declarations suggest. Tx_block[] and Rx_block are declared at the assembly language level as a part of a channel declaration, and referenced as externs (in vmodem.h) by the user at the C language level as C structures. Each component-specific header file (such as v27.h, gendet.h, etc.) provides a unique structure definition for Tx_block and Rx_block, and the user can cast Tx_block[] and Rx_block[] pointers as required to access component specific elements. Some examples are: in "gendet.h" the structures are defined as struct TX_GEN_BLOCK and struct RX_DET_BLOCK; in "v27.h" the structures are defined as struct TX_V27_BLOCK and struct RX_V27_BLOCK, and so on. It is not necessary for the user to access any of the members of COMMON or ANNEX for normal operation of any Vmodem component. Functions defined in "rxtx" provide access to all CONTROL members needed to operate any component, or the user can create pointers to Tx_block[] and Rx_block as struct TX_BLOCK *Tx and struct RX_BLOCK *Rx respectively.

Vmodem I/O is handled through *circular* buffers Tx_data[], Tx_sample[] in transmitter(), and Rx_sample[], and Rx_data[] in receiver(). *Circular* means that the elements are accessed sequentially until the end of the buffer and then wrapping around to the beginning in modulo fashion. Head and tail pointers are provided in the CONTROL sections of Tx_block and Rx_block for accessing these buffers. The "head" pointers are used for writing into a buffer, and the "tail" pointers are used for reading out of a buffer. The user must implement modulo modification when using these pointers. Transmitter() reads symbols from Tx_data[] and advances the Tx->data_tail pointer, and writes samples into Tx_sample[] and advances the Tx->sample_head pointer. Receiver() reads samples out of Rx_sample[] and advances Rx->sample_tail, and writes demodulated symbols into Rx_data[] using the Rx->data_head pointer. The user must write symbols into Tx_data using Tx->data_head, read outgoing samples from Tx_sample[] using Tx->sample_tail, and update these pointers in a circular fashion (see vmodem.c or other demo code for examples). Similarly, The user must write incoming samples to Rx_sample[] using Rx->sample_head, read symbols from Rx_data[] using Rx->data_tail, and update these pointers in a circular fashion. Users are advised to monitor the Rx->state_ID for MESSAGE_ID and calculate the head-to-tail displacement to qualify the arrival of data in Rx_data[]. This buffer also contains symbols during training and Rx->data_head and Rx->data_tail traverse Rx_data[] in unison during this period, but the displacement will remain 0 until message data arrives.

Transmitter() and receiver have several linear and circular buffers used internally for filters, equalizers, echo cancellers, and TCM decoding. The user should not access these elements other than to monitor their contents.

4.2.1 Transmitter Memory Structures

The complete memory requirement for the transmitter() is shown below followed by descriptions for each control block member:

Input:	CIRC Tx_data[TX_DATA_LEN]
Output:	CIRC Tx_sample[TX_SAMPLE_LEN]
Internal:	CIRC Tx_fir[TX_FIR_LEN]
Control:	short Tx_block[TX_BLOCK_LEN]

Tx_sample[] - Tx_sample[] is a circular buffer that transmitter() writes the 8 kHz samples to using the *Tx->sample_head pointer from Tx_block[]. The user reads these samples directly from Tx_sample[] using the *Tx->sample_tail pointer from Tx_block[] and usually writes them to a D/A device in hardware at an 8 kHz rate. Echo canceller modems use Tx_sample[] as the delay line for the near-end echo canceller, and also store the bulk-delayed transmit samples at the "end" of Tx_sample[]. Recall that Tx_sample[] is circular so the oldest samples in Tx_sample[] are actually replaced with bulk-delayed, re-generated samples for the far-end echo canceller to use to cancel echo from the far-end hybrid. Tx_sample[] is optionally initialized to all zero by the function Tx_block_init().

Tx_data[] - Tx_data[] is a circular buffer that transmitter() gets the transmit data symbols from using the *Tx->data_tail pointer. The user writes data symbols to be transmitted into this buffer using the *Tx->data_head pointer. It is important to note that most transmitter() functions that transmit data from Tx_data[] (v.17, v.21, v.26, v.27, v.29, DTMF, MF) DO NOT check for buffer over-run or under-run. If the user fails to supply data dynamically to Tx_data[], then whatever is currently in there will be transmitted. The data modems (v.22, v.32) monitor Tx->data_head and Tx->data_tail and if the buffer is "empty" (head=tail), then these modems will modulate all ones and not advance Tx->data_tail. Tx_data[] is optionally initialized to all zero by the function Tx_block_init().

Tx_fir[] - Tx_fir[] is a circular buffer that transmitter() modulators use to implement their Nyquist shaping interpolator/decimator re-sampling filters. The length of Tx_fir[] is dependent on the type of modulator but is hard-coded to handle the largest by default, and its contents are initialized to all zero by each modulator init function, such as Tx_init_v29(). The user does not have general access to pointers for this buffer, and it is not required for any user I/O function.

Tx_block[] – Tx_block is an array that is “cast” as a structure to provide the state memory required for a channel implementation of a particular transmitter component. The general case structure is defined in vmodem.h as:

```
struct TX_BLOCK
   {
   CONTROL;
   COMMON;
   ANNEX;
   };
```

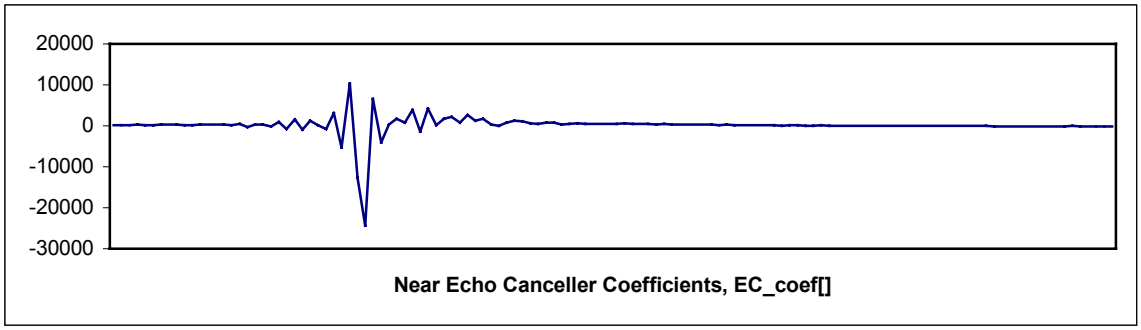
where:
CONTROL is a macro that defines a set of structure members that are common to all transmitter() implementations. The structure members defined in this macro are described in detail below.
COMMON reserves space for common-use transmitter() memory. This area is cast as different types of structures, depending on the type of device that transmitter() is configured for. For example, when transmitter() is configured for APSK modulation (such as v.29 or v.32) then COMMON would be cast as

TX_APSK_MOD_BLOCK as defined in common.h. When transmitter() is configured for GenDet tone generation, then COMMON is cast as several unique structure members as required by GenDet and defined in gendet.h.
ANNEX reserves space for device-specific memory. Most components have a device-specific structure definition that names structure members that are unique to that specific device. For example, v27 defines the variables Sguard and Sinv in v27.h uniquely since these items have no relevance to other devices.

By creating Tx_block[] as an array and then casting it as desired for specific device profile, transmitter() efficiently "recycles" its local persistent channel memory. Users can access CONTROL members by casting a pointer as a (struct TX_BLOCK *) pointer, which can be used generically across all transmitter() device types. This is usually sufficient for accessing Applications Programmer Interfaces such as sample and data buffer pointers, state_IDs, and rate information. Users can also access common and spare memory area structure members by casting a device-specific pointer, such as (struct TX_V22_BLOCK *), while transmitter() is configured for that device.

CONTROL Member Descriptions

***start_ptrs** - Pointer to the start of the "start_ptrs" structure. It allows for re-entrant code operation (i.e. multiple channels) and is used to access Rx_block[] and all buffers. It is initialized by Tx_block_init() to it's argument (void Tx_block_init(struct START_PTRS *ptrs)).
***state** - Pointer to the function call associated with the current transmitter() state. The transmitter() function simply calls whatever function this pointer is pointing to, and that function modifies Tx->state to switch to the next state. It is initialized in Tx_block_init() to the Tx_silence() state (&Tx_silence).
state_ID - State identifier. It contains a code or ID for each Tx->state and is useful in monitoring the progress of the transmitter(). The ID's are defined for each component in it's include (*.h) file and are coded to have relevance to the component type (i.e. v27 state_ID's are 0x2701, 0x2702, etc.; v32 state_ID's are 0x3201, 0x3202, etc). It is initialized in Tx_block_init() to TX_SILENCE_ID.
rate - Modulator transmitter bit rate. Specifies the current modulator transmission rate in bits per second. It is initialized in Tx_block_init() to 0.
scale - Transmitter signal level attenuator. It is a signed fractional integer and it globally attenuates all modulator and signal generator output samples. It is initialized in Tx_block_init() to 32767 or unity gain.
system_delay - Amount of delay in samples between the time a sample of the transmitted signal is produced at the output of your system's DAC to the time that signal is sampled by your ADC. It is used by the echo canceller modems (v26, v32) to align the peaks in the EC_coef[] buffer and optimizes the span of the cancellers. It is initialized in Tx_block_init() to 0. As an example, if your system copies blocks of samples from Tx_sample[] to another buffer for use by your DAC interrupt service routine, then you probably incur a fixed delay equal to the block size and you should set the Tx->system_delay accordingly. A trial-and-error method to set this would be to simply make a v32 call, wait until the modem is in the message state (TX_V32A_MESSAGE_ID), and then view the near-end taps in the EC_coef[] buffer. (The near-end taps are contained in the first half of the buffer, and the far-end taps are in the second half.) You will see a large negative peak in the sample values. Adjust Tx->system delay to position this peak at $1/4^{th}$ of the length of the near-end taps. For example if there are 128 taps in the near-end canceller, the peak should be positioned near tap 32 (EC_coef[32]). This will give more taps for the trailing echo tail, which is considerably longer than the leading tail. The figure below shows actual trained echo canceller coefficients properly positioned in the near-end portion of the EC_coef[] buffer.



***sample_head** - Tx_sample[] circular buffer write pointer. The transmitter modules modify this pointer in modulo fashion when writing their samples to Tx_sample[]. Users should not modify this pointer. It is initialized in Tx_block_init() to the start of Tx_sample[] (&Tx_sample[0]).
***sample_tail** - Tx_sample[] circular buffer read pointer. The user modifies this pointer in modulo fashion when reading samples from Tx_sample[] to send to the system DAC. Transmitter() does not modify this pointer. It is initialized in Tx_block_init() to the start of Tx_sample[] (&Tx_sample[0]).
sample_len - Length of Tx_sample[]. Caveat emptier - users can dynamically relocate and resize Tx_sample[] subject to the particular DSP's circular buffer placement rules, by modifying the Tx_sample_start member in "start_ptrs[]" and Tx->sample_len. It is initialized in Tx_block_init() to TX_SAMPLE_LEN. At a minimum, Users must ensure that Tx->sample_len is large enough to hold all of the samples generated per call as specified by Tx->num_samples. In practice, the transmitter() algorithm for computing the state of Tx_sample[] (that is, how full or empty Tx_sample[] is) seeks to keep a frame (Tx->num_samples) of samples in the buffer, but not more than 2 frames. To ensure proper real-time operation, Tx->sample_len should be greater than 2 times Tx->num_samples plus 1 (i.e. Tx->sample_len>=2*Tx->num_samples+1). Echo canceller modems (v.26, v.32, v.34) require additional space in Tx_sample[] for the near- and far-end echo canceller bulk-delay sample storage. See the relevant modem discussion for details.
***data_head** - Tx_data[] circular buffer write pointer. The user modifies this pointer in modulo fashion when writing symbols to Tx_data[]. Transmitter() does not modify this pointer. It is initialized in Tx_block_init() to the start of Tx_data[] (&Tx_data[0]).
***data_tail** - Tx_data[] circular buffer read pointer. Transmitter modules modify this pointer in modulo fashion when reading symbols from Tx_data[] for modulation. Users should not modify this pointer. It is initialized in Tx_block_init() to the start of Tx_data[] (&Tx_data[0]).
data_len- Length of Tx_ data[]. Caveat emptier - users can dynamically relocate and resize Tx_ data[] subject to the particular DSPs circular buffer placement rules, by modifying the Tx->data_start and Tx->data_len members in "start_ptrs[]". It is initialized in Tx_block_init() to TX_ DATA_LEN. Users must ensure that Tx->data_len is large enough to hold all of the symbols that will be modulated by the currently selected modulator for all of the samples specified by Tx->num_samples per call. For example, v.29 modulates one symbol per 2,400 Hz (baud rate), and if Tx->num_samples=20 then the modulator will extract 20*(2400/8000)=6 symbol per call from Tx_data[], so Tx->data_len must be 6 minimum.
sample_counter - Counts the number of samples produced per state. It is reset upon state transition, and is initialized in Tx_block_init() to 0.
symbol_counter - Counts the number of symbols produced per state. It is reset upon state transition, and is initialized in Tx_block_init() to 0.
call_counter - Counts down from Tx_num_samples the number of Tx->state calls made from transmitter.
num_samples - Specifies the number of samples generated per call to transmitter(). It is initialized in Tx_block_init() to 20. Users must ensure that Tx_sample[] is sized large enough to hold the specified number of samples generated per transmitter() call. A smaller value for Tx->num_samples results in a smaller delay from the time that transmitter() writes samples to Tx_sample[] and to the time that the sample is actually transmitted. The recommended method for invoking transmitter() is to use a Tx_sample[] buffer filling algorithm that tries to always keep Tx->sample_head ahead of Tx->sample_tail by Tx->num_samples. See "vmodem.c" for an example of a main loop transmitter(), receiver() calling algorithm based on these pointers. The delay cited above affects the full-duplex echo canceller data modems such as

v.32bis and v.34. Users should keep Tx->num_samples as low as possible to minimize the delay. A value of Tx->num_samples=20 will provide a control layer granularity of 2.5 msec. and keeps the receiver()/transmitter() handshake delay well within v.32bis and v.34 limits.
mode - Transmitter mode specification. It is used to select different operational modes for some modems. It is initialized in Tx_block_init() to 0. The bit fields are defined in vmodem.h, and are shown in the table below:

Bit	Field Name	Tx->mode Definition
0	TX_LONG_RESYNC_FIELD	0=long train, 1=resync. For fax modems (v17, v27, v29), use this bit to enable the short re-sync mode operation.
0	TX_V26_FAST_SYN_FIELD	0=long train, 1=fast synchronization. Use this bit to enable the short sync operation for v26.
1	TX_TEP_FIELD	0=disabled, 1=enabled. Setting this bit enables generation of the Talker Echo Protection (TEP) tone in v.17, v.27, and v.29 modulators.
2	Undefined	
3	TX_V32TCM_MODE_BIT	0=Trellis Ccoded Modulation encoder disabled, 1= Trellis Ccoded Modulation encoder enabled
3	TX_BELL_MODE_BIT	0=ITU modem operation (i.e. v.21, v.22bis), 1=Bellcore modem operation (i.e. Bell103, Bell212a).
4	TX_V32BIS_MODE_BIT	0=disabled, 1=enabled (v32*bis*)
5	TX_V32_SPECIAL_TRAIN	0=disabled, 1=enabled (v32, v32*bis*). This bit enables the v32 special train sequence, which users can insert to add delay in v32 training.
6	TX_SCRAMBLER_DISABLE_B IT	0=disabled, 1=enabled (v26, v32, v32*bis*). This bit disables the scrambler during the message state for STU-III compatible operation Requires V32_STU_III symbol definition at compile or assembly-time.
7	TX_V26_ALT_A_BIT	0=v26 ALT-B, 1=v26 ALT-A. This bit is used to configure the v26 modem for ALT-A or ALT-B operation.
8-15	Undefined	

Table 2

terminal_count - Specifies the maximum number of samples or symbols to be produced in the current state. Most transmitter modules compare either Tx->sample_counter or Tx->symbol_counter with Tx_terminal_count to determine when to complete the current state and switch to the next state. The user can extend the duration of a particular state by monitoring for it's state_ID and then changing the value of Tx->terminal_count as desired. In many states such as all modulator message states, Tx->terminal count is set to a negative value, which forces the state to execute forever. It is initialized in Tx_block_init() to 0. The terminal_count may be used to gracefully end a transmission. The following method is suggested as it will work for all the MESi fax modems, including Bell103, Bell202, v17, v21, v23, v26, v27, and v29.

While in the message state, determine how many symbols remain in the data buffer by comparing Tx->data_head and Tx->data_tail. Keep in mind that this is a circular buffer and wrap-around must be accounted for.
Determine how many additional symbols will be added to the buffer until the end of the data. Add this to the number found in step 1.
Read the Tx->symbol_count, AND it with 0x0003, and write the value back to Tx->symbol_count. (This preserves the 2 lsb's of the symbol_count, which are used for internal sequencing by some modems).

Add this new Tx->symbol_count value to the number of symbols determined in step 2.
Write this value to Tx->terminal_count.

Nbits – Number of bits per symbol for the current modulator. This value is related to Tx->rate by the modulation format. For example, v.17 at 14,400 is 128 QAM TCM so there are 64 points per symbol period and Nbits=6. As another example, v.27 at 2,400 bits/sec. is QAM so there are 4 points per symbol and Nbits=2.
Nmask – Bit mask corresponding to $Nbits^2 - 1$. Can be used to mask for valid bits when converting to symbols for Tx_data[]. For example, , v.17 at 14,400 has Nbits=6 so Nmask=0x3f. V.27 at 2,400 bits/sec. has Nbits=2 so Nmask=3.
bit_register – upper 16 bits of a 32-bit register used to implement a symbol-to-byte data conversion for various modems.
bit_register_low – Lower 16 bits of a 32-bit register used to implement a symbol-to-byte data conversion for various modems.
bit_index – bit-indexing variable used to implement a symbol-to-byte data conversion for various modems.

TX_APSK_MOD_BLOCK Member Descriptions

***fir_head** - Tx_fir[] circular buffer write pointer. Transmitter() modulator components, such as v27, use this pointer when writing complex values into Tx_fir[] for modulation.
***fir_tail** - Tx_fir[] circular buffer read pointer. The APSK_modulator() algorithm uses this pointer to access complex base band samples for interpolation/decimation and up-conversion.
fir_len - The length of the Tx_fir[] buffer. This is typically assigned in a device's initialization routine, such as Tx_init_v27_2400().
fir_taps - The number of taps in the current APSK_modulator() interpolator/decimator/ up-converter FIR filter.
***coef_start -** Pointer to the start of the table containing the coefficients for the current APSK_modulator() interpolator/decimator/ up-converter FIR filter.
coef_ptr - pseudo pointer used in calculating the current interpolator/decimator filter coefficient position. Normally, the coef_ptr is incremented by Tx->decimate, modulo Tx->interpolate for each modulated output sample, and a new symbol is fetched each time Tx->coef_ptr "wraps".
interpolate - Interpolation parameter for use in modulation sample rate conversion. This parameter is multiplied by the symbol rate (FSK, APSK modulators) to determine the interpolated sample rate to which the modulated signal is up-converted. For example, at 1,200 symbols per second and a modulation sample rate of 48 kHz, the interpolation factor would be 48,000/1,200=40.
decimate - Decimation parameter for use in modulation sample rate conversion. The interpolated sample rate to which the modulated signal has been converted is divided by the decimation rate to obtain the channel sample rate of 8 kHz. For example, at a sample up-converted rate of 48 kHz, the decimation rate factor would be 48,000/8,000=6.
sym_clk_offset - symbol clock frequency offset variable. This value should remain zero unless the user desires to bias the transmit symbol clock rate slightly higher or lower that the symbol clock rate derived from the 8 kHz sample clock.
sym_clk_memory – memory element for the transmit symbol clock offset loop.
sym_clk_phase – phase modifier for the interpolator/decimator filter coefficient pointer.
carrier – Modulator carrier frequency parameter.
***map_ptr** – Pointer to phase conversion map table.
***amp_ptr** – Pointer to amplitude conversion map table.
phase – Baseband carrier phase accumulator memory.
Sreg – Scrambler shift register memory (high 16 bits)
Sreg_low - Scrambler shift register memory (low 16 bits)
fir_scale – Amplitude scale parameter for the complex base band modulator symbols.
Ereg – Encoder register used for TCM encoder storage.

4.2.2 Receiver Memory Structures

The complete memory requirement for the receiver() is shown below followed by descriptions for each control block member:

Input:	CIRC Rx_sample[RX_SAMPLE_LEN]
Output:	CIRC Rx_data[RX_DATA_LEN]
Internal:	CIRC Rx_fir[RX_FIR_LEN]
	short EQ_coef[EQ_COEF_LEN]
	short EC_coef[EC_COEF_LEN
Control:	short Rx_block[RX_BLOCK_LEN]

Rx_sample[] - Rx_sample[] is a circular buffer that receiver() reads the 8 kHz samples from using the *Rx->sample_tail pointer from Rx_block[]. The user writes these samples directly to Rx_sample[] using the *Rx->sample_head pointer from Rx_block[] and usually gets them from an A/D converter device in hardware at an 8 kHz rate. Demodulator interpolator/decimator FIR filters and Hilbert band pass filters use this buffer as the sample delay line for their implementation. Rx_sample[] is optionally initialized to all zero by the function Rx_block_init().

Rx_data[] - Rx_data[] is a circular buffer that receiver() puts it's received data symbols to using the *Rx->data_head pointer. The user reads demodulated or detected data symbols from this buffer using the *Rx->data_tail pointer. It is important to note that the receiver() functions that write data to Rx_data[] (modems, DTMF, MF) DO NOT check for buffer overrun or under run. If the user fails to extract data dynamically from Rx_data[], then whatever is currently in there will be overwritten. Rx_data[] is optionally initialized to all zero by the function Rx_block_init().

Rx_fir[] - Rx_fir[] is a circular buffer that receiver() demodulators use to implement their Nyquist shaping interpolator/decimator re-sampling filters, and DTMF and MF detectors use for storage of their recursion memories. The length of Rx_fir[] is dependent on the type of demodulator but is hard-coded to handle the largest by default , and its contents are initialized to all zero by each demodulator init function. The user does not have general access to pointers for this buffer, and it is not required for any user I/O function.

Rx_block[] – Rx_block is an array that is "cast" as a structure to provide the state memory required for a channel implementation of a particular receiver() component. The general case structure is defined in vmodem.h as:

```
struct RX_BLOCK
   {
   CONTROL;
   COMMON;
   ANNEX;
   };
```

where:
CONTROL defines a set of control structure members that are common to all receiver() implementations. The structure members defined in this macro are described in detail below.
COMMON reserves space for common-use receiver() memory. This area is cast as different types of structures, depending on the type of device that receiver() is configured for. For example, when transmitter is configured for APSK demodulation (such as v.29 or v.32) then COMMON would be cast as RX_APSK_DMOD_BLOCK as defined in common.h. When receiver() is configured for GenDet tone detection, then COMMON is cast as several unique structure members as required by GenDet and defined in gendet.h.
ANNEX reserves space for device-specific memory. Most components have a device-specific structure definition which names structure members that are unique to that specific device. For example, v27 defines the variables Dguard, Dinv, etc. in v27.h uniquely since these items have no relevance to other devices.

By creating Rx_block[] as an array and then casting it as desired for specific device profile, receiver() efficiently "recycles" its local persistent channel memory. Users can access CONTROL members by casting a pointer as a (struct RX_BLOCK *) pointer, which can be used generically across all receiver() device types. This is usually sufficient for accessing Applications Programmer Interfaces such as sample and data buffer pointers, state_IDs, and rate information. Users can also access common and spare memory area structure members by casting a device-specific pointer, such as (struct RX_V22_BLOCK *), while receiver() is configured for that device.

CONTROL Member Descriptions

***start_ptrs** - Pointer to the start of the "start_ptrs" structure. It allows for re-entrant code operation (i.e. multiple channels) and is used to access Rx_block[] and all buffers. It is initialized by Rx_block_init() to it's argument (void Rx_block_init(struct START_PTRS *ptrs)).
***state** - Pointer to the function call associated with the current receiver() state. The receiver() function simply calls whatever function this pointer is pointing to, and that function modifies Rx->state to switch to the next state. It is initialized in Rx_block_init() to the Rx_idle() state (&Rx_idle).
state_ID - State identifier. It contains a code or ID for each Rx->state and is useful in monitoring the progress of the receiver(). The ID's are defined for each component in it's include (*.h) file and are coded to have relevance to the component type (i.e. v27 state_ID's are 0x2701, 0x2702, etc.; v32 state_IDs are 0x3201, 0x3202, etc). It is initialized in Rx_block_init() to TX_IDLE_ID.
status - Demodulator status report. It reports a status code defined in vmodem.h (vmodem.inc for assembly source) to indicate the current condition of the receiver() state such as LOSS_OF_LOCK or EQ_TRAINING_FAILURE. It is latched in the sense that it remains set until a new demodulator is started up. It is initialized in Rx_block_init() to 0 or STATUS_OK. The table below summarizes the possible status response messages and their meanings.

Status Response (Rx->status)	Meaning
STATUS_OK	Receiver status is OK – no events reported
DETECT_FAILURE	Generic failure to detect a state-specific event, such as start of training sequence, etc.
SYNC_FAILURE	Generic failure to achieve synchronization, such as frame sync in Caller_ID.
TRAIN_LOOPS_FAILURE	Failure to train the demodulator loops, including the carrier loop, AGC, and symbol clock recovery loops. This status is usually reported early in the training phases of fax and data demodulators and is usually the result of a time-out condition. That is, the demodulator failed to stabilize these loops and progress to the next state prior to the modem-specific timeout.
START_EQ_FAILURE	Failure to detect the start of the equalizer training sequence prior to a modem-specific timeout period. Fax and data modems usually have a specific event, such as a 180-degree phase reversal, that defines the start of equalizer training. This status would be reported if the demodulator failed to detect this event, and therefore failed to train it's equalizer.
TRAIN_EQ_FAILURE	Failure to successfully blind-train the adaptive equalizer. This status would be reported if the demodulator checks to determine if it failed to train it's equalizer within the modem-specific training sequence.
SCR1_FAILURE	Failure to detect the Scrambled Binary Ones (SCR1) associated with some modems.
LOSS_OF_LOCK	Loss of carrier lock reported if the demodulator looses carrier lock, either due to signal loss or the PLL looses carrier lock. This is the typical response seen at the end of a

	fax modulation when the signal ceases, and most half-duplex demodulators (such as v.27, v.29, v.17) switch back to the RX_ENERGY_DETECT_ID state upon detection of loss-of-lock to search for the next burst.
GAIN_HIT_STATUS	This status is reported if a gain hit or sudden drop in signal energy is detected. Unless the RX_LOS_FIELD bit in Rx->mode is set, overriding the LOS detector, the demodulator will detect a loss-of-lock during a gain hit ant take the associated action.
EXCESSIVE_MSE_STATUS	This status report indicates that the demodulator Mean Square Error measurement is excessively high.
EXCESSIVE_RTD_STATUS	This status report indicates that the Round Trip Delay measurement is excessive. Echo canceller modems, such as v.32 and v.34 may report this status if the RTD exceeds the bulk storage length (i.e. Tx->sample_len) for the far-end echo canceller. The demodulator will continue it's training, but the data may be corrupted.
RETRAIN	This status reports that the modem is engaged in a retrain sequence.
RETRAIN_FAILURE	This status indicates that the modem failed to successfully retrain.
RENEGOTIATE	This status reports that the modem is engaged in a rate-renegotiate sequence.
RENEGOTIATE_FAILURE	This status indicates that the modem failed to successfully renegotiate the rate.
V22_USB1_DETECTED	This status is reported by the v.32 modem during the Rx_v32C_detect_AC state and indicates that a v.22 USB1 signal has been detected. This status would then be used to implement v.32 to v.22 fallback or v.32bis automode processing.
V22_S1_DETECTED	This status is reported in by the v.22bis modem to indicate that the S1 signal has been successfully detected.
V22_SB1_DETECTED	This status is reported in by the v.22bis modem to indicate that the SB1 signal has been successfully detected.
V32_ANS_DETECTED	This status is reported in by the v.32/v.23bis modem to indicate that the 2100 Hz ANSWER tone signal has been successfully detected.
V32_AA_DETECTED	This status is reported in by the v.22bis modem to indicate that the v.32 CALL-mode AA signal (1800 Hz) signal has been detected.
V32_AC_DETECTED	Not implemented
GSTN_CLEARDOWN_REQUESTED	This status is reported in by the v.32bis and v.34 modems to indicate that a GSTN Clear-down request has been successfully detected during training, retrain, or renegotiate phases.

Table 3

rate - Demodulator receiver bit rate. It specifies the current demodulator reception rate in bits per second. The user must set this rate for v17 and v29 fax modems prior to allowing them to start or they won't'train or move data properly. V22 and v32 modems set the Rx->rate depending on their rate negotiations. It is initialized in Rx_block_init() to 0.

power - Received signal power level estimate. It is calculated in "gendet" upon detection of a signal enabled for detection, such as dial tone or various fax modem preambles. It is initialized in Rx_block_init() to 0, and relates to the received signal power level in decibels as $10*\log_{10}$(Rx->power/32768.0) dB.

***sample_head** - Rx_sample[] circular buffer write pointer. The user modifies this pointer in modulo fashion when writing samples to Rx_sample[]. Receiver() does not modify this pointer. It is initialized in Rx_block_init() to the start of Rx_sample[] (& Rx_sample[0]).
***sample_tail** - Rx_sample[] circular buffer read pointer. Receiver() modifies this pointer in modulo fashion when reading samples from Rx_sample[]. The user should not modify this pointer. It is initialized in Rx_block_init() to the start of Rx_sample[] (& Rx_sample[0]).
***sample_stop** - Rx_sample[] circular buffer read limit pointer. The user should not modify this pointer. It is initialized in Rx_block_init() to the start of Rx_sample[] (& Rx_sample[0]).
sample_len - Length of Rx_sample[]. Caveat emptier - users can dynamically relocate and resize Rx_sample[] subject to the particular DSPs circular buffer placement rules, by modifying the Rx_sample_start member in "start_ptrs[]" and Rx->sample_len. It is initialized in Rx_block_init() to RX_SAMPLE_LEN. Users must ensure that Rx->sample_len is large enough to hold all of the samples specified by Rx->num_samples per call plus the maximum backwards processing length required for each modem. The minimum length for detector operations in gendet is 80+Rx->num_samples, for synchronous modems (v.17, v.22, v.26, v.27, v.29, v.32, v.34, CID) is 80+Rx->num_samples samples (10 msec.), for v.21 it is 67+Rx->num_samples samples, and for digit detectors (DTMF, R1, R2) it is Rx->num_samples.
***data_head** - Rx_data[] circular buffer write pointer. Receiver() modifies this pointer in modulo fashion when writing symbols to Rx_data[]. The user should not modify this pointer. It is initialized in Rx_block_init() to the start of Rx_data[] (& Rx_data[0]).
***data_tail** - Rx_data[] circular buffer read pointer. Users modify this pointer in modulo fashion when reading symbols from Rx_data[] for modulation. Receiver() does not modify this pointer. It is initialized in Rx_block_init() to the start of Rx_data[] (& Rx_data[0]).
data_len- Length of Rx_ data[]. Caveat emptier - users can dynamically relocate and re-size Rx_data[] subject to the particular DSPs circular buffer placement rules, by modifying the Rx_ data_start member in "start_ptrs[]" and Rx->data_len. It is initialized in Rx_block_init() to RX_ DATA_LEN. Users must ensure that Rx->data_len is large enough to hold all of the symbols that will be demodulated by the selected demodulator for all of the samples specified by Rx->num_samples per call. For example, v.29 demodulates one symbol per 2,400 Hz (baud rate), and if Rx->num_samples=20 then the demodulator will produce 20*(2400/8000)=6 ±1 symbol per call, so Rx->data_len must be 7 minimum.
sample_counter - Counts the number of samples produced per state. It is reset upon state transition, and is initialized in Rx_block_init() to 0.
symbol_counter - Counts the number of symbols produced per state. It is reset upon state transition, and is initialized in Rx_block_init() to 0.
call_counter - Counts down from Rx_num_samples the number of Rx->state calls made from receiver.
num_samples - Specifies the number of samples generated per call to receiver(). It is initialized in Rx_block_init() to 20. Users must ensure that Rx_sample[] is sized large enough to hold the specified number of samples received per receiver() call, plus additional 10-msec. depth for receiver() filter processing. A smaller value for Rx->num_samples results in a smaller delay from the time that samples are written to Rx_sample[] to the time that receiver() processes them. The delay cited above affects the full-duplex echo canceller data modems such as v.32bis and v.34. Users should keep Rx->num_samples as low as possible to minimize the delay. A value of Rx->num_samples=20 will provide a control layer granularity of 2.5 msec., and keeps the receiver()/transmitter() handshake delay well within v.32bis and v.34 limits. Users must ensure that Rx->sample_len is large enough to hold all of the samples specified by Rx->num_samples per call plus the maximum backwards processing length required for each modem. The minimum length for detector operations in gendet is 80+Rx->num_samples, for synchronous modems (v.17, v.22, v.26, v.27, v.29, v.32, v.34, CID) is 80+Rx->num_samples samples (10 msec.), for v.21 it is 107+Rx->num_samples samples, and for digit detectors (DTMF, R1, R2) it is Rx->num_samples.
mode - Receiver mode specification. It is used to select different operational modes for some modems. It is initialized in Rx_block_init() to 0.

Bit	Field Name	Rx->mode Definition
0	RX_LONG_RESYNC	0=long train, 1=resync. For fax modems (v17, v27, v29), use this bit to enable the short re-sync mode operation.
	RX_V26_FAST_SYN_FIELD	0=long train, 1=fast synchronization. Use this bit to enable the short sync operation for v26.
1	RX_DETECTOR_DISABLE	Baudot: 0=start-bit detector enabled, 1=start-bit detector disabled. If disabled, then the FSK demodulator is immediately started without searching for start bits.
		Bell103: 0= startup detector enabled, 1=startup detector disabled. If disabled, then the FSK demodulator is immediately started without searching for MARKs or start bits
		V.26: 0= SYN or P1800 startup detector enabled, 1=startup detector disabled. If disabled, then the v.26 demodulator is immediately started withous searching for the ITU-T V.26/V.26bis SYN or FSVS-210 P1800 start-up sequences.
		Bellcore/ETSI Caller ID: 0=FSK detector enabled, 1=FSK detector disabled. If disabled, then the detector will only search for CAS/TAS signal (if enabled at build-time) and will not search for FSK startup
2	RX_LOS_FIELD	0=LOS terminate enabled, 1=LOS terminate inhibited. In applications where the user wants to force the demod to persist after the receive signal level has dropped (i.e. gain hit or phone line momentarily disconnected), this bit will inhibit the Loss-Of-Signal detector from terminating the demodulator.
3	RX_STU_III_BIT	0=v26 sync detector type, 1=STU-III P1800 detector type. The same detector can be re-configured using this bit to detect either v.26bis SYNC (3333 dibits) or STU-III P1800 (0202 dibits).
		0=normal v.32 operation. The modem will initiate retrain in the event of a loss of signal or loss of carrier lock. 1=STU-III operational mode. In the event of a gain hit or instantaneous phase change causing a LOS event to be detected, the demodulator will "freeze" the AGC, carrier, and symbol timing loops until the LOS event is no longer detected (i.e. the demod restores carrier lock). In this context, "freeze" means that these loops are inhibited from updating their internal memory elements and thus their pre-existing states are preserved. This allows the modem to "free-wheel" through a cellular fade or drop-out.
	RX_BELL_MODE_BIT	0=ITU modem operation (i.e. v.21, v.22bis), 1=Bellcore modem operation (i.e. Bell103, Bell212a).
4	RX_EC_COEF_SAVE_BIT	0=reset EC_coef[], 1=save existing EC_ceof[]. Setting this bit will cause subsequent Rx_init_vxx function calls to bypass the initialization of the echo canceller coefficients. The user would set this bit if the echo canceller coefficients have been previously trained and a re-initialization of the demod or detector is desired.
5	RX_EQ_COEF_SAVE_BIT	0=reset EQ_coef[], 1=save existing EQ_ceof[]. Setting this bit will cause subsequent Rx_init_vxx function calls to bypass the initialization of the equalizer coefficients. The user would set this bit if the equalizer coefficients have been previously trained and a re-initialization of the demod or detector is desired.

6	RX_DESCRAMBLER_DISABLE_BIT	0=disabled, 1=enabled (v26, v32, v32*bis*). This bit disables the descrambler during the message state for STU-III compatible operation. Requires V32_STU_III symbol definition at compile or assembly-time.
7	RX_V26_ALT_A_BIT	0=v26 ALT-B, 1=v26 ALT-A. This bit is used to configure the v26 modem for ALT-A or ALT-B operation.
	FAX_DEMOD_DISABLE_BIT	0=Fax demod initialization enabled, 1=fax demod initialization disabled. This bit is used in GenDet Fax modem (v.17, v.21 channel 2, v.27, and v.29) detectors to selectively disable the call to start the demodulator upon detection. If this bit is reset, then the demodulator's Rx_init_vxx function is called upon detection of the corresponding signal in GenDet, and Rx->state switches to that demodulator. If disabled, then GenDet Rx->state_ID reflects the modulation type detected and the state switches to Rx_tone_undetector. This mode would be used for GenDet operating only as a detector and discriminator.
8	RX_EC_DISABLE_BIT	0=echo canceller will be enabled by the enable_echo_canceller() function, 1=all echo canceller parameters will be initialized by enable_echo_canceller() function but EC_2mu will remain DISABLED. This allows the user to externally disable the echo canceller without removing it, and would be used for 4-wire applications where the canceller might not be needed.
9-15	Undefined	

Table xx. Rx->mode bit field definitions

threshold - Specifies the minimum power level for gendet and data modem signal detections. It is initialized in Rx_block_init() to the default value of -48 dB. The minimum value is -48 dB below which the demod will produce errors.

detector_mask - Enables various detectors for execution in "gendet". Mask definitions for Rx_detector_mask are provided in "gendet.h". The user can selectively enable or disable the tone and demodulator detectors in "gendet" by setting Rx->detector_mask with the appropriate mask value. The "set_detector_mask()" and "Rx_init_detector()" functions examine the Rx->detector_mask and enable the required filters. If the AUTO_DETECT_MASK bit is not set, then the filters will execute but no detectors will post-process the filtered results. The user must ensure that the AUTO_DETECT_MASK is ORed in with Rx->detector_mask to enable detection and subsequent demodulator startup. It is initialized in Rx_block_init() to 0 (no detectors enabled).

digit_CP_mask - Enables various digit and call progress detectors for execution in "gendet". Mask definitions for Rx_digit_CP_mask are provided in "gendet.h". It is initialized in Rx_block_init() to 0 (no detectors enabled).

temp0 - scratch memory.

temp1 - scratch memory.

Nbits – Number of bits per symbol for the current demodulator. This value is related to Rx->rate by the modulation format. For example, v.17 at 14,400 is 128 QAM TCM so there are 64 points per symbol period and Nbits=6. As another example, v.27 at 2,400 bits/sec. is QAM so there are 4 points per symbol and Nbits=2.

Nmask – Bit mask corresponding to $Nbits^2 - 1$. Can be used to mask for valid bits when converting symbols from Rx_data[]. For example, , v.17 at 14,400 has Nbits=6 so Nmask=0x3f. V.27 at 2,400 bits/sec. has Nbits=2 so Nmask=3.

bit_register – upper 16 bits of a 32-bit register used to implement a symbol-to-byte data conversion for various modems.

bit_register_low – Lower 16 bits of a 32-bit register used to implement a symbol-to-byte data conversion for various modems.
bit_index – bit-indexing variable used to implement a symbol-to-byte data conversion for various modems.

RX_APSK_MOD_BLOCK Member Descriptions

(*decoder_ptr)(struct RX_BLOCK *) – Function pointer for demodulator differential decoder routines.
(*slicer_ptr)(struct RX_BLOCK *) - Function pointer for demodulator hard-decision symbol slicer algorithm routines.
(*timing_ptr)(struct RX_BLOCK *) – Function pointer for demodulator symbol timing error calculation algorithm routines.
baud_counter – Counter for sampling at the symbol rate or baud rate.
***data_ptr** – Hidden pointer to used by demodulators to access symbols in Rx_data[] without "reading" them out (i.e. does not alter the Rx->data_head/Rx->data_tail relationship that the user monitors).
***sample_ptr –** Hidden pointer used by APSK_demodulator to implement 2x symbol rate base-band interpolation/decimation.
fir_taps - The number of taps in the current APSK_demodulator() interpolator/decimator/ up-converter FIR filter.
***coef_start** - Pointer to the start of the table containing the coefficients for the current APSK_modulator() interpolator/decimator/ up-converter FIR filter.
coef_ptr - pseudo pointer used in calculating the current interpolator/decimator filter coefficient position.
sym_clk_phase – phase modifier for the interpolator/decimator filter coefficient pointer.
interpolate - Interpolation parameter for demodulator sample-to-symbol rate conversion.
decimate - Decimation parameter for demodulator sample-to-symbol rate conversion.
oversample – Specifies factor above the intermediate interpolated sample rate that the current APSK_demodulator is operating at.
***timing_start** – Pointer to the start of the symbol timing adjustment table.
I – Demodulated Costas loop real base-band 2x symbol rate sample.
Q– Demodulated Costas loop imaginary base band 2x symbol rate sample.
IEQ – Adaptive equalizer in-phase output at 2x symbol rate.
QEQ – Adaptive equalizer quadrature output at 2x symbol rate.
Iprime – In-phase base band demodulated real symbol.
Qprime – Quadrature base band demodulated real symbol.
Inm1 – In-phase base band demodulated real symbol delayed by 0.5 symbol periods.
Qnm1 – Quadrature base band demodulated real symbol delayed by 0.5 symbol periods.
Inm2 – In-phase base band demodulated real symbol delayed by 1 symbol period.
Qnm2 – Quadrature base band demodulated real symbol delayed by 1 symbol period.
Inm3 – In-phase base band demodulated real symbol delayed by 1.5 symbol periods.
Qnm3 – Quadrature base band demodulated real symbol delayed by 1.5 symbol periods.
Ihat – In-phase base band demodulated real symbol slicer estimate.
Qhat – Quadrature base band demodulated real symbol slicer estimate.
Ihat_nm2 – In-phase base band demodulated real symbol slicer estimate delayed by 1 symbol period..
Qhat_nm2 – Quadrature base band demodulated real symbol slicer estimate delayed by 1 symbol period..
What – Estimate of the inverse square of Ihat and Qhat.
***fir_ptr** – Read/write pointer used for adaptive equalizer access of Rx_fir[] samples.
IEQprime_error – De-rotated real mean square error.
QEQprime_error – De-rotated imaginary mean square error.
EQ_MSE – Normalized base band symbol mean square error. This signal is the average of the square of the difference between the received symbol and it's sliced estimate. It gives a report of the current signal-to-noise ratio on the line, and is averaged over several hundred symbol periods to provide a smooth estimate.
EQ_2mu – Adaptive equalizer adaptation or "learning" constant. It is a signed value, and if negative, then the adaptive equalizer filter is bypassed and no tap update is performed (i.e. DISABLED). If it is zero, then the adaptive equalizer filter is engaged but no tap update is performed (i.e. FREEZE). If it is greater than

zero then the adaptive equalizer filter is engaged and the LMS tap update algorithm continuously updates the taps (i.e. TRAIN).
COS – Base band phase-locked loop oscillator cosine output.
SIN – Base band phase-locked loop oscillator sine output
LO_memory – Base-band Local oscillator de-rotator accumulator memory.
LO_frequency – Base-band Local Oscillator de-rotator frequency parameter.
LO_phase – Base-band Local Oscillator de-rotator phase parameter
vco_memory – Base band phase-locked loop oscillator accumulator memory
phase_error – Base band phase-locked loop phase error estimate.
loop_memory – Base band phase-locked loop filter memory (high 16 bits).
loop_memory_low – Base band phase-locked loop filter memory (low 16 bits).
loop_K1 - Baseband phase-locked loop filter constant K1.
loop_K2 - Base band phase-locked loop filter constant K1.
agc_gain – Automatic Gain Control loop memory.
agc_K – Automatic Gain Control loop constant. If this variable is set to zero, the AGC loop stops updating and continues scaling at the current value in Rx->agc_gain (i.e. FREEZE)
frequency_est – Base band phase-locked loop filter carrier frequency offset estimate.
sym_clk_memory – memory element for the transmit symbol clock offset loop
timing_threshold – Threshold parameter for symbol timing phase adjustment algorithm.
coarse_error – Accumulator memory for coarse symbol timing coarse error estimation algorithm.
LOS_counter – Consecutive events counter for Loss of Signal detection algorithm. The LOS detector reports Loss of Signal when this counter reaches a hard threshold.
LOS_monitor – Reports loss of Signal.
map_shift – Shift value used in amplitude/phase bit re-mapping.
Phat – Received signal phase estimate.
phase – phase variable used in TCM decoder.
EQ_taps – The number of complex taps in the current adaptive equalizer.
Dreg De-scrambler shift register memory (high 16 bits).
Dreg_low - De-scrambler shift register memory (low 16 bits).
pattern_reg – general-purpose register typically used for storing bit patterns.
***demod_delay_ptr** - pointer to demod_delay[] buffer associated with TCM decoder.
***trace_back_ptr** – Pointer to trace_back[] buffer associated with TCM decoder.
***signal_map_ptr** – Pointer to signal map table associated with TCM decoder.
***EC_fir_ptr** – Pointer to transmit sample storage buffer used by echo canceller algorithm.
***EC_sample_ptr** – Pointer to receive sample storage buffer used by echo canceller algorithm
EC_2mu – Adaptive echo canceller adaptation or "learning" constant. If it is zero, then the adaptive echo canceller filter is engaged but no tap update is performed (i.e. FREEZE). If it is greater than zero then the adaptive echo canceller filter is engaged and the LMS tap update algorithm continuously updates the taps (i.e. TRAIN).
EC_MSE – The mean square error of the echo canceller error term. EC_MSE is only valid for echo canceller modems, and stops updating after echo canceller training has completed. In the v.32 modem, transmitter states Tx_v32A_TRN1 and Tx_v32C_TRN1 monitor Rx->EC_MSE, and after the minimum duration of the TRN signal, if the EC_MSE is below a fixed threshold, echo canceller training ceases and these states terminate. The user can extend the training period up to the maximum allowed duration by modifying Tx->terminal_count.

EC_taps - The number of taps in the current echo canceller NEAR and FAR filters.
EC_shift – Scale value used to correct for reduction in transmit signal level.
RTD – Round Trip Delay estimate from echo canceller modems, reported in units of symbols. For v.32bis at 2,400 symbols or bauds per sec., the units would be in 1/2400 seconds.

5 Channel Creation

MESi Vmodem components use the same RAM memory structure for all control, interface, and I/O for a given channel instance. Each channel has a "struct START_PTRS" memory structure that contains vector

addresses for all RAM memory elements associates with that channel. The Vsim simulator declares 2 channels called ANS_ptrs and CALL_ptrs to simulate an end-to-end connection. The method of channel declaration differs for different DSP vendors since some of the buffers are circular and require declaration at the assembly language level, and since some components can initialize memory at load time. Channel memory is allocated by including the memory.c file (or memory.asm for assembly code) in the build. Two methods exist for channel creation, named and arrayed, as discussed below. Memory creation defaults to the arrayed channel creation method.

5.1 Arrayed Channel Creation

The arrayed method of channel creation allocates an array of identical channels. The number of channels created at build time is controlled by defining NUM_CHANNELS appropriately. The buffers are created as [NUM_CHANNELS] x [ ] arrays. For example, the Tx_sample buffer is created (approximately) as

```
CIRC Tx_sample [NUM_CHANNELS*TX_SAMPLE_LEN ];
```

The actual code uses macros to ensure that each channel's section within Tx_sample is aligned properly to support the circular buffering mechanism used by the particular DSP target. This may cause some unused space to be allocated between channel elements. Initialization of the START_PTRS structure is performed by calling the start_ptrs_init(short channel, struct START_PTRS *ptrs) function for each channel as shown below:

```
for (i=0;i<NUM_CHANNELS;i++)
        {
        ptrs=(struct START_PTRS *)&start_ptrs[i];
        start_ptrs_init(i, ptrs);
        …
```

5.2 Named Channel Creation

In the named method each channel is created and named explicitly. This method allows the user maximum flexibility in placing the channel elements into memory. It is useful for situations where the system contains multiple channels having different memory requirements. For example, a two channel system with a v32bis modem channel and a DTMF detector channel could benefit from this method. Users can declare a "START_PTRS" structure in C and then initialize it in code as shown below. The various buffers referenced by this structure must, in general, be created by the assembly language memory.asm file to allow for proper memory placement by the linker. ***CHANNEL_SPEC*** is a unique identifier created by the user for each channel. Examples are: Tx_blockA, Tx_sampleA, Rx_blockC, Rx_sampleC, etc.

```
extern short Tx_block CHANNEL_SPEC;
extern CIRC Tx_sample CHANNEL_SPEC[];
extern CIRC Tx_data CHANNEL_SPEC[];
extern CIRC Tx_fir CHANNEL_SPEC[];
extern short Rx_block CHANNEL_SPEC;
extern CIRC Rx_sample CHANNEL_SPEC[];
extern CIRC Rx_data CHANNEL_SPEC[];
extern CIRC Rx_fir CHANNEL_SPEC[];
extern short EQ_coef CHANNEL_SPEC[];
extern short encoder CHANNEL_SPEC[];
extern short decoder CHANNEL_SPEC[];
extern CIRC trace_back CHANNEL_SPEC[];

struct START_PTRS start_ptrs CHANNEL_SPEC;
```

Each channel must be explicitly initialized in the user code as:

```
ptrs=&start_ptrsCHANNEL_SPEC;
ptrs->Tx_block_start=&Tx_blockCHANNEL_SPEC[0];
ptrs->Tx_sample_start=&Tx_sampleCHANNEL_SPEC[0];
ptrs->Tx_data_start=&Tx_dataCHANNEL_SPEC[0];
ptrs->Tx_fir_start=&Tx_firCHANNEL_SPEC[0];
ptrs->Rx_block_start=&Rx_blockCHANNEL_SPEC[0];
ptrs->Rx_sample_start=&Rx_sampleCHANNEL_SPEC[0];
ptrs->Rx_data_start=&Rx_dataCHANNEL_SPEC[0];
ptrs->Rx_fir_start=&Rx_firCHANNEL_SPEC[0];
ptrs->EQ_coef_start=&EQ_coefCHANNEL_SPEC[0];
ptrs->encoder_start=&encoderCHANNEL_SPEC[0];
ptrs->decoder_start=&decoderCHANNEL_SPEC[0];
ptrs->trace_back_start=&trace_backCHANNEL_SPEC[0];
```

The memory.c file contains instructions on how to build using the named memory method. To create additional channels, the user must simply copy the single channel template, and 'stamp out' as many copies as needed for each channel, re-naming ***CHANNEL_SPEC*** as required to create unique channels. Each channel must be explicitly initialized as shown above.

5.3 Building Only the Receiver or Transmitter

For either channel creation method it may be desired to build without the transmit or receive channel to conserve memory resources. This may be accomplished in the assembly Source Code builds by defining the following values as zero in the build options:

TRANSMITTER=0
TX_BLOCK_LEN=0
TX_SAMPLE_LEN=0
TX_DATA_LEN=0
TX_FIR_LEN=0
ENCODER_BLOCK_LEN=0

or

RECEIVER=0
RX_BLOCK_LEN=0
RX_SAMPLE_LEN=0
RX_DATA_LEN=0
RX_FIR_LEN=0
EQ_COEF_LEN=0
NEC_COEF_LEN=0
FEC_COEF_LEN=0
EC_COEF_LEN=0
DECODER_BLOCK_LEN=0
TRACEBACK_LEN=0

5.4 Texas Instruments 'C54x Channel Creation

The 'C54x DSP requires manual circular buffer placement directions in the linker .cmd file. This is provided in the memory.asm file. The memory.asm file uses the .usect directive to create the following un-initialized data memory sections. Macros are used to ensure the correct memory alignment for both single and multiple channel cases. Note that the sections shown are the result of the macro expansion for the single channel case:

```
_Tx_block          .usect      "TxBlk",TX_BLOCK_LEN
_Tx_sample         .usect      "TxSmpl",TX_SAMPLE_LEN
```

```
_Tx_data          .usect      "TxData",TX_DATA_LEN
_Tx_fir           .usect      "TxFir",TX_FIR_LEN
_Rx_block         .usect      "RxBlk",RX_BLOCK_LEN
_Rx_sample        .usect      "RxSmpl",RX_SAMPLE_LEN
_Rx_data          .usect      "RxData",RX_DATA_LEN
_Rx_fir           .usect      "RxFir",RX_FIR_LEN
_EQ_coef          .usect      "EQcoef",EQ_COEF_LEN
_EC_coef          .usect      "ECcoef",EC_COEF_LEN
_encoder          .usect      "Encode",ENCODER_BLOCK_LEN
_decoder          .usect      "Decode",DECODER_BLOCK_LEN
_trace_back       .usect      "Trace",TRACEBACK_LEN
```

and linker sections in the demo code:

```
TxBlk: align=128    > DARAM
TxSmpl: align=512   > DARAM
TxData: align=64    > DARAM
TxFir: align=16     > DARAM
RxBlk: align=128    > DARAM
RxSmpl: align=512   > DARAM
RxData: align=64    > DARAM
RxFir: align=256    > DARAM
Eqcoef:             > DARAM
Eccoef:             > DARAM
Encode:             > DARAM
Decode:             > DARAM
Trace: align=256    > DARAM
```

Note that the align statements shown above are only an example. Each demo comes with its own linker description file and should be used for reference.

For the multichannel case using the arrayed memory creation, each channel must have it's own portion of the Tx_block array, Tx_sample array, etc. aligned properly for that channel to access it circularly. The memory.asm file uses macros to size the array appropriately. The start_ptrs_init() function then initializes the start_ptrs to point to the correctly aligned starting location for each channel.

When using the named memory creation method it is possible to create these memory elements in C using a #pragma to specify section names so that the linker can align them properly for circular access. An example of this would be to create the Tx_sample[] buffer as an integer array and place it in the TxSmpl section:

```
short Tx_sample[TX_SAMPLE_LEN];
#pragma DATA_SECTION (Tx_sample, "TxSmpl");
```

On the 'C54x, the Tx_block and Rx_block context structures must also be aligned on 128 word boundaries due to the data page limitations of the device.

5.5 Analog Devices ADSP21XX Channel Creation

The Analog Devices linker automatically places CIRC buffers in memory such that the DAG units will properly implement modulo wrap-around addressing. Users must create the required buffers and start_ptrs structure in ADSP 21xx assembly language, and reference them as externs in C. This is provided in the memory.asm file. The memory.asm file uses the .var directive to create the following un-initialized data memory sections. Macros are used to ensure the correct memory alignment for both single and multiple channel cases. Note that the sections shown are the result of the macro expansion for the single channel case:

```
.section/dm TxBlk ;
.var _Tx_block[TX_BLOCK_LEN];
```

```
.section/dm TxSmpl ;
.align (next power of 2 of TX_SAMPLE_LEN)        /* not used for 219x */
.var _Tx_sample[TX_SAMPLE_LEN];
.section/dm TxData ;
.align (next power of 2 of TX_DATA_LEN) /* not used for 219x */
.var _Tx_data[TX_DATA_LEN];
.section/dm TxFir ;
.align (next power of 2 of TX_FIR_LEN)           /* not used for 219x */
.var _Tx_fir[TX_FIR_LEN];
.section/dm Encode ;
.var _encoder[ENCODER_BLOCK_LEN];
.section/dm RxBlk ;
.var _Rx_block[RX_BLOCK_LEN];
.section/dm RxSmpl ;
.align (next power of 2 of RX_SAMPLE_LEN)        /* not used for 219x */
.var _Rx_sample[RX_SAMPLE_LEN];
.section/dm RxData ;
.align (next power of 2 of RX_DATA_LEN) /* not used for 219x */
.var _Rx_data[RX_DATA_LEN];
.section/dm RxFir ;
.align (next power of 2 of RX_FIR_LEN)           /* not used for 219x */
.var _Rx_fir[RX_FIR_LEN];
.section/pm EQcoef ;
.var _EQ_coef[EQ_COEF_LEN];
.section/pm ECcoef ;
.var _EC_coef[EC_COEF_LEN];
.section/dm Decode ;
.var _decoder[DECODER_BLOCK_LEN];
.section/dm Trace ;
.var _trace_back[TRACEBACK_LEN];
```

For the multichannel case using the arrayed memory creation, each channel must have it's own portion of the Tx_block array, Tx_sample array, etc. aligned properly for that channel to access it circularly. The memory.asm file uses macros to size the array appropriately. The start_ptrs_init() function then initializes the start_ptrs to point to the correctly aligned starting location for each channel. Note that the 219x processors use a base register and do not require memory alignment for circular buffering.

6 Component Descriptions

The components simulated in Vsim and ported to various DSPs include ITU-T v-series modems, digit generators/detectors, and general telephony call progress signal generators and detectors. The following sections describe each component and show their associated Vsim and DSP port files. Some of the Applications Programmer Interface functions are provided in DSP assembly code as macros where appropriate, and some are not implemented in Vsim.

6.1 Baudot TDD Modem

Simulator source files: cid.c, cid.h.
Texas Instruments source files: baudot.asm, baudot.inc, baudot.h.
Analog Devices source files: baudot.dsp, baudot.inc, baudot.h.
AT&T source files: baudot.s, baudot.inc, baudot.h.

This module modulates and demodulates Baudot Telecommunications Devices for the Deaf (TDD) FSK-modulated characters at 45.45 bits/sec. Unlike other older FSK telecom modems such as Bell103, the Baudot modulator only turns carrier on when a character is being modulated and switches to silence after

the complete character and extended stop-bit are transmitter. Thus the Baudot modem functions as a burst-mode modem where the carrier is present only during the transmission of characters, and there is no MARKing tone when there are no characters.
On the transmit side, the baudot modulator scans the transmit data buffer, Tx_data[], for the presence of a start bit (a SPACE or "0") and transmits silence until a start-bit is detected. After a start-bit is detected, the modulator then transmits the next 5 bits normally as the data payload. The sixth bit is treated as the stop-bit and is extended to 250 msec. per specification requirements. This stop-bit extension can be overridden by the user at build-time (requires source code), or at run-time by modifying the value in Tx->interpolate during the stop-bit modulation interval. It is important for users to understand that the FSK modulator is continuously running when transmitter() is configured for Baudot operation. It simply mutes the signal while in the TX_BAUDOT_SILENCE_ID state, when it is not transmitting characters and their start/stop bits. Thus, transmitter will continuously and synchronously extract data bits from Tx_data[] at the 45.5 bits/sec. rate and will continuously scan for start bits. The user must synchronously supply MARKs to the transmitter() if there are no characters to send. Because the transmitter will continue to re-modulate any old data present in Tx_data[].
The receiver searches the receive samples for a modulated start-bit (SPACE tone at 1400 Hz) and upon detection, begins data collection into a shift-register. All 5 data bits plus start and one stop bit are collected and released into Rx_data[] only if all of these bits are validated as data bits. This prevents the Baudot demodulator from false detecting Baudot characters in the presence of voice or other line impairments. The User can monitor Rx_data[] head and tail pointers, Rx->data_head and Rx->data_tail, for any displacement to determine if any bits have been received. If Rx->data_head=Rx->data_tail, then no character is present. When the complete character is received and validated, then 7 bits (one start bit, 5 data bits, and one stop bit) will appear in Rx_data[], and Rx->data_head will be ahead of Rx->data_tail by 7.

6.1.1 TIA/EIA Compliance:

The modem is fully compliant with all sections of TIA/EIA-825-2000 and with the older PN-1663 draft 9.

6.1.2 Applications Programmer Interface Functions and Macros

void Tx_init_baudot(struct START_PTRS *) – Configures Tx_block[] for Baudot character generation and sets the Tx->state to Tx_baudot.
void Rx_init_baudot(struct START_PTRS *) – Configures Rx_block for Baudot character reception, and sets the Rx->state to Rx_baudot_start_bit_detect.

6.1.3 Transmitter States (by STATE_ID)

TX_BAUDOT_SILENCE_ID – Indicates that the transmitter is generating silence and that no characters are in process. This is the default condition where no start bit (i.e. all MARKs) is present in Tx_data[].
TX_BAUDOT_MESSAGE_ID – Indicates that FSK-modulated baudot character with start and stop bits are currently being generated. The transmitter remains in this state only when a character (start bit, 5 data bits, stop bit) are present in Tx_data[], and returns to the TX_BAUDOT_SILENCE_ID state upon completion of modulating the stop bit.

6.1.4 Receiver States (by STATE_ID)

RX_BAUDOT_START_BIT_ID – Searches receive samples for a modulated start bit (1400 Hz SPACE tone).
RX_BAUDOT_MESSAGE_ID – Processes Baudot character bits and stop bit. It rejects entire character if any bits appear to be false detections. It captures only one stop bit regardless of MARKs extensions to character.

6.2 Bellcore 103 modem

Simulator source files: b103.c, b103.h.
Texas Instruments source files: b103.asm, b103.inc, b103.h.
Analog Devices source files: b103.dsp, b103.inc, b103.h.

AT&T source files: b103.s, b103.inc, b103.h.

This 2-channel modem operates at a data-signaling rate of 300 bit/s with a symbol rate of 300 symbols/sec. The modulation method is continuous phase frequency shift keying (CP-FSK). The demonstration code provides a working example of a Bell 103 modem. If run on a target platform, such as a TI DSK5402, the demonstration code will initialize the transmit and receive sections of the modem and wait for a call on the DAA. It will then answer the call and link to another Bell 103 modem looping back all characters received. If the DIGITAL_LOOPBACK option is enabled at build time, the demonstration code, will provide a digital loopback of the Tx sample output to the Rx sample input. The contents of the Rx_block[] can be examined to compare with the Tx_block[].

6.2.1 AT&T Compliance:

The modem is fully compliant with all sections of Bell System Technical Reference PUB 41101, Data Set Interface Specification 103A.

6.2.2 Applications Programmer Interface Functions and Macros

void Tx_init_bell103_F1(struct START_PTRS *) - Configures Tx_block[] for Bell 103 channel 1 transmitter (modulator) operation, and sets the Tx->state to Tx_bell103_message.
void Tx_init_bell103_F2(struct START_PTRS *) - Configures Tx_block[] for Bell 103 channel 2 transmitter (modulator) operation, and sets the Tx->state to Tx_bell103_message.
void Rx_init_bell103_F1(struct START_PTRS *) - Configures Rx_block[] for Bell 103 channel 1 receiver (demodulator) operation, and sets the Rx->state to Rx_bell103_message.
void Rx_init_bell103_F2(struct START_PTRS *) - Configures Rx_block[] for Bell 103 channel 2 receiver (demodulator) operation, and sets the Rx->state to Rx_bell103_message.

6.2.3 Transmitter States (by STATE_ID)

TX_BELL103_F1_MESSAGE_ID – Channel 1 data transmission state.
TX_BELL103_F2_MESSAGE_ID – Channel 2 data transmission state.

6.2.4 Receiver States (by STATE_ID)

RX_BELL103_F1_MARKS_ID – Detect Marks pattern on channel 1.
RX_BELL103_F1_MESSAGE_ID - Demodulate data symbols on channel 1 and sink to Rx_data[].
RX_BELL103_F2_MARKS_ID – Detect Marks pattern on channel 2.
RX_BELL103_F2_MESSAGE_ID - Demodulate data symbols on channel 2 and sink to Rx_data[].

6.3 Bellcore 202 modem

Simulator source files: bell202.c, bell202.h.
Texas Instruments source files: bell202.asm, bell202.inc, bell202.h.
Analog Devices source files: bell202.dsp, bell202.inc, bell202.h.
AT&T source files: bell202.s, bell202.inc, bell202.h.

The Bell 202 modem is a 2 channel modem with a channel data signaling rate of 600, 1200 and 1350 bit/s with symbol rates of 600, 1200 and 1350 symbols/sec. The modulation method is continuous phase frequency shift keying (CP-FSK). The demodulator does not include any preamble detection or training – it simply demodulates and dumps the data in Rx_data[]. For optimum performance the user should provide a detector for the start of a Bell 202 modulated burst (such as in the Caller ID case) and then initialize the demodulator.

6.3.1 AT&T Compliance:

The modem is fully compliant with all sections of AT&T Bulletin, Data Set 202 Interface Specification.

6.3.2 Applications Programmer Interface Functions and Macros

void Tx_init_bell202(struct START_PTRS *) - Configures Tx_block[] for Bell 202 transmitter (modulator) operation, and sets the Tx->state to Tx_bell202_message.
void Rx_init_bell202(struct START_PTRS *) - Configures Rx_block[] for Bell 202 receiver (demodulator) operation, and sets the Rx->state to Rx_Bell202_message.

6.3.3 Transmitter States (by STATE_ID)

TX_BELL202_MESSAGE_ID - Data transmission state. One-bit symbols are read from Tx_data[] and modulated.

6.3.4 Receiver States (by STATE_ID)

RX_BELL202_MESSAGE_ID - Demodulate data symbols and sink to Rx_data[]. The FSK_demodulator loss-of-signal detector will operate reliably if approximately 80 samples of the modulated signal are present in Rx_sample[] at the time Rx_init_bell202() was called. Otherwise the user must rely on demodulated data content for validation.

6.4 Bellcore 212A modem

Simulator source files: v22.c, v22.h.
Texas Instruments source files: v22.asm, v22.inc, v22.h.
Analog Devices source files: v22.dsp, v22.inc, v22.h.
AT&T source files: v22.s, v22.inc, v22.h.

This split-band modem supports full duplex data transfer at 1200 bit/s with a symbol rate of 600 symbols/sec. The modulation method is DPSK.

6.4.1 AT&T Compliance:

The modem is fully compliant with all sections of AT&T Bulletin CB109, Technical Publication 41214, Data Set 212A Interface Specification.

6.4.2 Applications Programmer Interface Functions and Macros

void Tx_init_v22A(struct START_PTRS *) - Configures Tx_block[] for v22 *bis* ANSWER side modulation starting in the Tx_v22A_silence2 state, and Rx_block[] for v22 *bis* ANSWER side demodulation.
void Tx_init_v22A_ANS(struct START_PTRS *) - Configures Tx_block[] for v22 *bis* ANSWER side modulation with the 2100 Hz answer tone, starting in the Tx_v22A_ANS state. The demodulator is initialized subsequently following the completion of the Tx_v22A_silence2 state.
void Tx_v22A_retrain(struct START_PTRS *) - Configures Tx_block[] to initiate a v22 bis retrain/rate change request, and Rx_block[] to respond to a detected retrain/rate change reply.
void Tx_init_bell212A(struct START_PTRS *) - Configures Tx_block[] for Bell212a ANSWER side modulation starting in the Tx_v22A_silence2 state, and Rx_block[] for v22 or Bellcore ANSWER side demodulation.
void Tx_init_bell212A_ANS(struct START_PTRS *) - Configures Tx_block[] for Bell212a ANSWER side modulation with the 2100 Hz answer tone, starting in the Tx_v22A_ANS state. The demodulator is initialized subsequently following the completion of the Tx_v22A_silence2 state.

void Tx_init_v22C(struct START_PTRS *) - Configures Tx_block[] for v22 *bis* CALL side modulation starting in the Tx_v22C_silence state. The demodulator startup is initiated from the v22_detector in "gendet", so be sure that the Rx->detector_mask has V22_MASK and AUTO_DETECT_MASK ORed in.
void Tx_init_bell212C(struct START_PTRS *) - Configures Tx_block[] for Bell212a CALL side modulation starting in the Tx_v22C_silence state. The demodulator startup is initiated from the v22_detector in "gendet", so be sure that the Rx->detector_mask has V22_MASK and

AUTO_DETECT_MASK ORed in. The detector discriminates between 2225 Hz and USB1 (2250 Hz plus 2850 Hz) and sets the BELLCORE_MODE_BIT in Tx->modem accordingly.

void Tx_v22C_retrain(struct START_PTRS *) - Configures Tx_block[] to initiate a v22 bis retrain/rate change request, and Rx_block[] to respond to a detected retrain/rate change reply

6.4.3 Transmitter States (by STATE_ID)

TX_V22A_SILENCE1_ID - ITU-T v.22 *bis* ANSWER 2.15 sec. Silence preceding ANS (2100 Hz).
TX_V22A_ANS_ID - ITU-T v.22 *bis* ANSWER 2100 Hz tone for 3.3 sec.
TX_V22A_SILENCE2_ID - ITU-T v.22 *bis* ANSWER 75 msec. Silence preceding USB1.
TX_BELL212A_2225_ID – Bellcore Bell212a ANSWER tone at 2225 Hz.
TX_V22A_UB1_ID - ITU-T v.22 *bis* ANSWER Unscrambled binary 1 at 1200 bit/s.
TX_V22A_S1_ID - ITU-T v.22 *bis* ANSWER S1 alternations for 100 msec.
TX_V22A_SCR1_ID - ITU-T v.22 *bis* ANSWER Scrambled R2 at 1200 bit/se
TX_V22A_SB1_R2_ID - ITU-T v.22 *bis* ANSWER Scrambled R2 at R2 rate.
TX_V22A_MESSAGE_ID - ANSWER modulate data symbols from Tx_data buffer.

TX_V22C_SILENCE_ID - ITU-T v.22 *bis* CALL Silence preceding S1.
TX_V22C_S1_ID - ITU-T v.22 *bis* CALL S1 alternations for 100 msec.
TX_V22C_SCR1_ID - ITU-T v.22 *bis* CALL Scrambled binary 1 at 1200 bit/s.
TX_V22C_SB1_R2_ID - ITU-T v.22 *bis* CALL Scrambled R2 at R2 rate.
TX_V22C_MESSAGE_ID - CALL modulate data symbols from Tx_data buffer.

6.4.4 Receiver States (by STATE_ID)

RX_V22A_START_DETECT_ID – ITU-T v.22 *bis* ANSWER Detect S1, Scrambled binary 1 (SB1), or V.32 AA.
RX_V22A_TRAIN_LOOPS_ID – ITU-T v.22 *bis* ANSWER Train carrier, symbol timing, and agc loops.
RX_V22A_TRAIN_EQ_ ID – ITU-T v.22 *bis* ANSWER Train adaptive equalizer.
RX_V22A_MESSAGE_ID – ITU-T v.22 *bis* ANSWER Demodulate data symbols and sink to Rx_data buffer.
RX_V22A_RC_RESPOND_ID – ITU-T v.22 *bis* ANSWER responding to rate change request.
RX_V22A_RC_INITIATE_ID – ITU-T v.22 *bis* ANSWER Initiating rate change request.

RX_V22C_START_DETECT_ID - ITU-T v.22 *bis* CALL Detect S1, Scrambled binary 1 (SB1), or V.32 AC.
RX_V22C_TRAIN_LOOPS_ID - ITU-T v.22 *bis* CALL Train carrier, symbol timing, and agc loops.
RX_V22C_TRAIN_EQ_ID - ITU-T v.22 *bis* CALL Train adaptive equalizer**.**
RX_V22C_MESSAGE_ID - ITU-T v.22 *bis* CALL Demodulate data symbols and sink to Rx_data buffer ITU-T v.22 *bis*.
RX_V22C_RC_RESPOND_ID - ITU-T v.22 *bis* CALL responding to rate change request.
RX_V22C_RC_INITIATE_ID - ITU-T v.22 *bis* CALL Initiating rate change request.

6.5 Bellcore-ETSI Caller ID

Simulator source files: cid.c, cid.h.
Texas Instruments source files: cid.asm, cid.inc, cid.h.
Analog Devices source files: cid.dsp, cid.inc, cid.h.
AT&T source files: cid.s, cid.inc, cid.h.

This module generates and detects the Calling Number Delivery service specified in Bellcore GR-30-CORE and in the European Telecommunications Standards Institute ETS 300 778 standards.

The transmitter can generate Type 1 (on-hook) or Type 2 (off-hook) signals. The Type 1 signals can be with or without the Caller Alert Signal (Bellcore CAS) or Terminal Equipment Alert Signal (ETSI TAS). Both Type 1 and Type 2 signals may be generated with either Bellcore or ETSI frequencies, selectable at run-time by the TX_BELLCORE_MODE_BIT of Tx->mode (0 is ETSI, 1 is Bellcore). Type 1 or Type 2 format is selectable with the TX_CID_TYPE2_BIT of Tx->mode (0 is Type 1, 1 is Type 2). Note that for simplification any reference to CAS below also applies to the ETSI TAS signal.

For Type 1 signals the receiver detects the start and end of CAS if present, and sets Rx->state_ID to **RX_CID_TRACK_CAS_ID** for the duration of the CAS signal. It simultaneously searches for the seizure signal and if found, will switch states to search for MARKs (1's).

The Type 2 receiver searches for and reports the CAS signal as above. If CAS is found it will switch states to search for MARKs.

For both Type 1 and Type 2 the receiver automatically detects and demodulates the caller ID presentation message layer and validates the checksum. Bellcore or ETSI operation is selectable at run-time with the RX_BELLCORE_MODE_BIT of Rx->mode, and Type 1 or Type 2 operation is selectable with the RX_CID_TYPE2_BIT. Assembly-time options let you build the object with only transmitter or receiver code generation if desired to minimize the size. The demonstration program provides a digital loopback that loops the Tx sample outputs back into the Rx sample inputs. Standard output provides an indication of the detection of a Caller ID burst along with the received data.
The RX_DETECTOR_DISABLE bit in Rx->mode can be set to disable the FSK detection in the Rx_CID_detect_burst state. If disabled, then the receiver() will only function as a CAS dual tone signal detector, and will report Rx->state_ID=RX_CID_TRACK_CAS_ID when the signal is present, and RX_CID_DETECT_BURST_ID when it is not detected.

6.5.1 Bellcore Compliance:

The modem is fully compliant with all sections of GR-30-CORE.

6.5.2 ETSI Compliance:

The modem is fully compliant with all sections ETS 300 778-1 and ETS 300 778-2.

6.5.3 Applications Programmer Interface Functions and Macros

void Tx_init_CAS(struct START_PTRS *) - Configures Tx_block[] for Caller Alert Signal (CAS) generation followed by a Caller ID burst, and sets the Tx->state to Tx_silence1. Transmitted format is determined by Tx->mode.
void Tx_init_CID(struct START_PTRS *) - Configures Tx_block[] for Caller ID FSK burst generation, and sets the Tx->state to Tx_silence2. Transmitted format is determined by Tx->mode.
void Tx_init_CAS1(struct START_PTRS *) - Macro which configures Tx->mode for ETSI CID Type 1 generation and calls Tx_init_CAS( ).
void Tx_init_CID1(struct START_PTRS *) - Macro which configures Tx->mode for ETSI CID Type 1 generation and calls Tx_init_CID( ). The CAS/TAS segment is not transmitted.
void Tx_init_CID2(struct START_PTRS *) - Macro which configures Tx->mode for ETSI CID Type 2 generation and calls Tx_init_CAS( ).
void Tx_init_CAS1B(struct START_PTRS *) - Macro which configures Tx->mode for BELLCORE CID Type 1 generation and calls Tx_init_CAS( ).
void Tx_init_CID1B(struct START_PTRS *) - Macro which configures Tx->mode for BELLCORE CID Type 1 generation and calls Tx_init_CID( ). The CAS/TAS segment is not transmitted.
void Tx_init_CID2B(struct START_PTRS *) - Macro which configures Tx->mode for BELLCORE CID Type 2 generation and calls Tx_init_CAS( ).

void Rx_init_CID(struct START_PTRS *) - Configures Rx_block[] for CAS detection and alternations (start of caller ID burst) detection, and sets the Rx->state to Rx_CID_detect_burst. Format is determined by Rx->mode.
void Rx_init_CID1(struct START_PTRS *) - Macro which configures Rx->mode for ETSI CID Type 1 detection and calls Rx_init_CID( ).
void Rx_init_CID2(struct START_PTRS *) - Macro which configures Rx->mode for ETSI CID Type 2 detection and calls Rx_init_CID( ).
void Rx_init_CID1B(struct START_PTRS *) - Macro which configures Rx->mode for Bellcore CID Type 1 detection and calls Rx_init_CID( ).
void Rx_init_CID2B(struct START_PTRS *) - Macro which configures Rx->mode for Bellcore CID Type 2 detection and calls Rx_init_CID( ).

6.5.4 Transmitter States (by STATE_ID)

TX_CID_SILENCE1_ID – Generates a silence period prior to CAS tone generation to allow the user to key up CAS tones in the X1 interval between SAS and CAS. The default is zero msec. and transmitter() switches to TX_CID_CAS_ID when Tx->sample_counter>=Tx->terminal_count. Immediately after calling the Tx_init_CAS() function, the user can re-program Tx->terminal_count to the number of samples of silence desired before CAS is generated.
TX_CAS_ID – Generates the CAS (Bellcore) or TAS (ETSI) dual tone signal for 100 msec. This state switches to TX_CID_SILENCE2_ID when Tx->sample_counter>=Tx->terminal_count.
TX_CAS_SILENCE2_ID - Generates a 45-msec. Silence period prior to caller ID burst Seizure generation. This state switches to TX_CID_SEIZURE_ID when Tx->sample_counter>=Tx->terminal_count.
TX_CID_SILENCE2_ID - Generates a silence period to allow the user to key up the Caller ID burst immediately after removal of power ringing, if desired. The default is zero msec. When Tx->sample_counter>=Tx->terminal_count the transmitter switches to TX_CID_SEIZURE_ID if Type 1 is enabled, or to TX_CID_MARKS_ID for Type 2. Immediately after calling the Tx_init_CID() function, the user can re-program Tx->terminal_count to the number of samples of silence desired before Caller ID FSK burst is generated.
TX_CID_SEIZURE_ID – For Type 2 formates only, generates the Caller ID burst "Channel Seizure" signal consisting of alternating Marks and Spaces for 300 bits. This state switches to TX_CID_MARKS_ID when Tx->symbol_counter>=Tx->terminal_count.
TX_CID_MARKS_ID - Generates the Caller ID burst "Marks" signal consisting of Marks for 180 bits. This state switches to TX_CID_MESSAGE_ID when Tx->symbol_counter>=Tx->terminal_count.
TX_CID_MESSAGE_ID - Generates the Caller ID Message segment. The Tx_fir[] buffer is the source for the Message bytes including the Message header, Message body, and checksum. The checksum is calculated in Tx_init_CID() so the user must fill Tx_fir[] with the desired Message prior to initialization. Tx_fir[] is treated as a linear buffer and the data format is 8-bit bytes oriented in the lower 8 bits of each Tx_fir[] location. If compile-time option TX_CID_SEGMENTA id #defined, then this state switches to TX_CID_SEGMENTA_ID when Tx->symbol_counter>=Tx->terminal_count. Otherwise, Tx_init_silence() is called and the state switches to TX_SILENCE_STATE.
TX_CID_SEGMENTA_ID - Generates a post-amble" signal consisting of Marks for 10 bits intended to extend the end of the Message to ensure that the last checksum bit is detectable. This state calls Tx_init_silence() and switches to TX_SILENCE_ID when Tx->symbol_counter>=Tx->terminal_count.

6.5.5 Receiver States (by STATE_ID)

RX_CID_DETECT_BURST_ID – Executes detectors for both CAS and the Channel Seizure signal associated with a Caller ID FSK burst. The detectors consist of a set of band-pass filters, a broadband energy estimator, and a series of level and ratio tests. If CAS is detected, then the state switches to RX_CID_TRACK_CAS_ID. If Channel Seizure is detected, then the receive power is calculated, the FSK demodulator is initialized, and the state switches to RX_CID_DETECT_SEIZURE_ID. In VSIM, the traces are WHITE in this state.
RX_CID_TRACK_CAS_ID – Executes CAS detection algorithm but searches for the end of the CAS dual tone signal. If the end of CAS is detected then the state switches back to RX_CID_DETECT_BURST_ID

when Type 1 is enabled, or to RX_CID_MARKS_ID if Type 2 is enabled. In VSIM the traces are LIGHTBLUE in this state

RX_CID_DETECT_SEIZURE_ID – The FSK demodulated data is correlated against the Channel Seizure (0101 alternations) pattern. If detected, then the state switches to RX_CID_DETECT_MARKS_ID. If Loss of Lock (Rx->status-LOSS_OF_LOCK) is detected then Rx_init_CID() is called and the state switches back to RX_CID_DETECT_BURST_ID. In VSIM the traces are YELLOW in this state.

RX_CID_MARKS_ID - For Type 2 only, executes detector for the presence of MARKS associated with the beginning of a Caller ID burst. The detector consists of band-pass filter, a broadband energy estimator, and a series of level and ratio tests. If MARKS are detected, then the state switches to RX_CID_DETECT_MARKS_ID. In VSIM, the traces are WHITE in this state.

RX_CID_DETECT_MARKS_ID - The FSK demodulated data is correlated against the Marks (all ones) pattern. If detected, then the state switches to RX_CID_DETECT_MESSAGE_ID. If Loss of Lock (Rx->status-LOSS_OF_LOCK) is detected then Rx_init_CID() is called and the state switches back to RX_CID_DETECT_BURST_ID. In VSIM the traces are LIGHTMAGENTA in this state.

RX_CID_DETECT_MESSAGE_ID - The FSK demodulated data is correlated against the Marks (all ones) pattern. If the end of Marks (i.e. a start bit) is detected, then the state switches to RX_CID_MESSAGE_ID. If Loss of Lock (Rx->status-LOSS_OF_LOCK) is detected then Rx_init_CID() is called and the state switches back to RX_CID_DETECT_BURST_ID. In VSIM the traces are LIGHTCYAN in this state.

RX_CID_MESSAGE_ID - The FSK demodulated data stripped of start and stop bits, byte-aligned, and written to Rx_fir[]. The start bit is always validated and byte framing is re-acquired for each byte so that marking between message bytes is accommodated. The checksum is continuously computed for all bytes indicated in the Message Length burst header parameter. After all of the Message Bytes have been processed, then Rx_fir[] will contain the complete Message including header and checksum. If the calculated checksum is not zero, then a checksum failure is reported (Rx->status=CHECKSUM_FAILURE) and the state switches to RX_IDLE_ID. If the checksum is zero Rx->status is set to CID_DETECTED and the state switches to RX_IDLE_ID. In both cases the user must call Rx_init_CID() before another CID message can be detected. If Loss of Lock (Rx->status=LOSS_OF_LOCK) is detected then Rx_init_CID() is called and the state switches back to RX_CID_MESSAGE_ID. In VSIM the traces are LIGHTGREEN in this state. Also in VSIM, upon detection of a good checksum the received Message is parsed for valid content and is displayed in the "x" dump file.

6.6 Japan Caller ID

Simulator source files: cidjapan.c, cidjapan.h.
Texas Instruments source files: cidjapan.asm, cidjapan.inc, cidjapan.h.
Analog Devices source files: cidjapan.dsp, cidjapan.inc, cidjapan.h.
AT&T source files: cidjapan.s, cidjapan.inc, cidjapan.h.

This module generates and detects the Number Display service as specified in Nippon Telegraph and Telephone Corporation (NTT) Technical Reference for Telephone Service Interfaces, Edition 5. The transmitter appends the preamble and postamble, calculates parity and the CRC. The receiver detects the beginning and end of the preamble sequence and automatically demodulates the caller ID presentation message layer, strips parity and validates the CRC. Assembly-time options let you build the object with only transmitter or receiver code generation if desired to minimize the size. The demonstration program provides a digital loopback that loops the Tx sample outputs back into the Rx sample inputs. Standard output provides an indication of the detection of a Caller ID burst along with the received data.

6.6.1 NTT Compliance:

The modem is fully compliant with all applicable sections of NTT Technical Reference for Telephohne Service Interfaces, Edition 5.

6.6.2 Applications Programmer Interface Functions and Macros

void Tx_init_Japan_CID(struct START_PTRS *) - Configures Tx_block[] for v.23 burst generation, and sets the Tx->state to Tx_Japan_CID_marks.
void Rx_init_Japan_CID(struct START_PTRS *) - Configures Rx_block[] for v.23 detection, and sets the Rx->state to Rx_Japan_CID_detect_burst .

6.6.3 Transmitter States (by STATE_ID)

TX_JAPAN_CID_MARKS_ID - Generates the Caller ID burst "Marks" signal consisting of Marks for 72 bits. This state switches to TX_JAPAN_CID_MESSAGE_ID when Tx->symbol_counter>=Tx->terminal_count.
TX_JAPAN_CID_MESSAGE_ID - Generates the Number Display Message segment. The Tx_fir[] buffer is the source for the Message bytes including the Message preamble, Message body, Message postamble, and CRC. Both the parity and CRC are computed in Tx_init_CID() so the user must fill Tx_fir[] with the desired Message prior to initialization. Tx_fir[] is treated as a linear buffer and the data format is 8-bit bytes oriented in the lower 8 bits of each Tx_fir[] location. Tx_init_silence() is called and the state switches to TX_SILENCE_ID.

6.6.4 Receiver States (by STATE_ID)

RX_JAPAN_CID_DETECT_BURST_ID – Executes detector for the presence of v.23 MARKS associated with the beginning of a Caller ID burst. The detector consists of band-pass filter, a broadband energy estimator, and a series of level and ratio tests. If v.23 MARKS are detected, then the state switches to RX_JAPAN_CID_DETECT_START_BIT_ID. In VSIM, the traces are WHITE in this state.
RX_JAPAN_CID_DETECT_START_BIT_ID – The FSK demodulated data is correlated against the MARKS (all ones) pattern. If detected, then the state switches to RX_JAPAN_CID_DETECT_PREAMBLE_ID. In VSIM the traces are CYAN in this state.
RX_JAPAN_CID_DETECT_PREAMBLE_ID - The FSK demodulated data is correlated against the proper preamble pattern. If detected, the state switches to RX_JAPAN_CID_MESSAGE_ID. If Loss of Lock (Rx->status=LOSS_OF_LOCK) is detected then Rx_init_Japan_CID() is called and the state switches back to RX_JAPAN_CID_DETECT_BURST_ID. In VSIM the traces are LIGHTMAGENTA in this state.
RX_JAPAN_CID_MESSAGE_ID - The FSK demodulated data is stripped of start and stop bits, parity, byte-aligned, and written to Rx_fir[]. The start bit is always validated and byte framing is re-acquired for each byte so that marking between message bytes is accommodated. The CRC is continuously computed for all bytes indicated in the Message Length burst header parameter. After all of the Message Bytes have been processed, then Rx_fir[] will contain the complete Message including header and CRC. If the calculated CRC is not zero, then a CRC failure is reported (Rx->status=CRC_FAILURE) and the state switches to RX_IDLE_ID. If the CRC is zero Rx->status is set to CID_DETECTED and the state switches to RX_IDLE_ID. In both cases the user must call Rx_init_Japan_CID() before another CID message can be detected. If Loss of Lock (Rx->status=LOSS_OF_LOCK) is detected then Rx_init_Japan_CID() is called and the state switches back to RX_JAPAN_CID_DETECT_BURST_ID. In VSIM the traces are LIGHTGREEN in this state. Also in VSIM, upon detection of a good CRC the received Message is parsed for valid content and is displayed in the "x" dump file.

6.7 DTMF Caller ID

Simulator source files: ciddtmf.c, ciddtmf.h.
Texas Instruments source files: ciddtmf.asm, ciddtmf.inc, ciddtmf.h.
Analog Devices source files: ciddtmf.dsp, ciddtmf.inc, ciddtmf.h.
AT&T source files: ciddtmf.s, ciddtmf.inc, ciddtmf.h.

This module generates and detects the Calling Line Identification Presentation (CLIP) as specified in Tele Danmark Technical Specification TDK-TS 900 216. Countries other than Denmark using this form of Caller ID, either similar to or a variation of this implementation, are Finland, Iceland, the Netherlands, India, Belgium, Sweden, Brazil, Saudi Arabia and Uruguay. The transmitter appends the preamble and postamble characters. The receiver detects the preamble character and automatically demodulates the caller

ID presentation message layer and end of information character. Assembly-time options let you build the object with only transmitter or receiver code generation if desired to minimize the size. The demonstration program provides a digital loopback that loops the Tx sample outputs back into the Rx sample inputs. Standard output provides an indication of the detection of a Caller ID burst along with the received data.

6.7.1 TeleDanmark Compliance:

The modem is fully compliant with all sections of the Committee for Specification and Standardization (SSU) Technical Specification TDK-TS 900 216.

6.7.2 Applications Programmer Interface Functions and Macros

void Tx_init_DTMF_CID(struct START_PTRS *) - Configures Tx_block[] for DTMF burst generation.
void Rx_init_DTMF_CID(struct START_PTRS *) - Configures Rx_block[] for DTMF detection, and sets the Rx->state to Rx_DTMF_CID_message.

6.7.3 Transmitter States (by STATE_ID)

TX_DTMF_CID_MESSAGE_ID - Generates the CLIP Message segment. The Tx_fir[] buffer is the source for the Message bytes including the Message preamble (type of message character), Message body, and Message postamble (end of information character). Tx_fir[] is treated as a linear buffer and the data format is 8-bit bytes oriented in the lower 8 bits of each Tx_fir[] location.

6.7.4 Receiver States (by STATE_ID)

RX_DTMF_CID_MESSAGE_ID - After the proper Message preamble character is detected, the DTMF data is decoded, the ASCIIII data is converted to DTMF digits and written to Rx_fir[]. After all of the Message Bytes have been processed, then Rx_fir[] will contain the complete Message. The Rx->status is set to CID_DETECTED and the state is then switched to RX_IDLE_ID. The user must call Rx_init_DTMF_CID() before another CID message can be detected. In VSIM the traces are LIGHTBLUE when the DTMF digit detector is in DTMF_analysis state and the traces are LIGHTGREEN when the DTMF digit detector is in DTMF_undetector state.

6.8 DTMF Generation and Detection

Simulator source files: dtmfc, dtmf.h.
Texas Instruments source files: dtmf.asm, dtmf.inc, dtmf.h.
Analog Devices source files: dtmf.dsp, dtmf.inc, dtmf.h.
AT&T source files: dtmf.s, dtmf.inc, dtmf.h.

This module generates and detects 16 DTMF digits with configurable parameters and talk-off abatement. The generator gets 4 bit digits from Tx_data[] until the number of digits transmits equals Tx->terminal_count and then transmitter switched to Tx_silence_state. Setting Tx->terminal_count to a negative number causes transmitter to remain in Tx_DTMF_state forever, continuously reading the digits in Tx_data[]. The detector reports detected DTMF digits as 4 bit nibbles in Rx_data[] and in the 4 LSBs of Rx->state_ID. The detector is enabled by ORing DTMF_MASK into the Rx->digit_CP_mask and then calling Rx_init_detector.
By default the two DTMF tones are generated with equal amplitudes for NORTH AMERICAN PSTN service. The user can compile or assemble using an international option such as BRAZIL_PSTN to generate with the higher frequency tone level 2 dB lower than the lower frequency tone.
The 4 bit Tx_data[] and Rx_data[] digits map to the 16 DTMF standard digits as follows:

4 bit value	DTMF digit
0x0	DTMF 0
0x1	DTMF 1
0x2	DTMF 2

0x3	DTMF 3
0x4	DTMF 4
0x5	DTMF 5
0x6	DTMF 6
0x7	DTMF 7
0x8	DTMF 8
0x9	DTMF 9
0xa	DTMF A
0xb	DTMF B
0xc	DTMF C
0xd	DTMF D
0xe	DTMF *
0xf	DTMF #

Table 4 DTMF Detector Digit Codes

Note that the DTMF detector operates in parallel with the call progress and modem detectors so you can listen for DTMF digits at any time when GenDet call progress and fax modem detectors are operating. Also note that the DTMF detector uses the Rx_fir[] memory buffer for its memory storage.

6.8.1 ITU-T Compliance:

The DTMF generator/detector is fully compliant with all sections of ITU-T Q.23 and Q.24 with the exception of minimum digit duration. The DTMF detector will correctly detect digits less than 10 msec. in duration.

6.8.2 Applications Programmer Interface Functions and Macros

void Tx_init_DTMF(struct START_PTRS *) - Configures Tx_block[] for generation of a DTMF digit. The number of digits to be generated is specified by Tx->terminal_count (default is 1). The digit on time is specified by Tx->cad_on_time (default is 500 samples or 62.5 msec.), and the total digit period or cadence (on time + off time) is specified by Tx->cad_period (default is 1000 samples or 125 msec.).
void Rx_init_DTMF(struct START_PTRS *) - Configures Rx_block[] for detection of DTMF digits. Detected digits are written to Rx_data[] and also ORed into the Rx->state_ID.

6.8.3 Transmitter States (by STATE_ID)

TX_DTMF_ID - DTMF digit generation state.

6.8.4 Receiver States (by STATE_ID)

RX_DTMF_ID - DTMF digit detection state. Upon detection, the digit code is ORed with the state_ID. For example, if a DTMF 1 is currently present, the Rx_state_ID would be RX_DTMF_ID|1 or 0x1301.

6.9 GenDet Signal Generation and Detection

Simulator source files: gendet.c, gendet.h.
Texas Instruments source files: gendet.asm, gendet.inc, gendet.h.
Analog Devices source files: gendet.dsp, gendet.inc, gendet.h.
AT&T source files: gendet.s, gendet.inc, gendet.h.

The Gendet module provides for the generation and detection of a variety of signals associated with call progress, digit processing, data modem, and fax modem startup. When transmitter() is configured to be a GenDet generator function, Tx_block is cast as TX_GEN_BLOCK (defined in gendet.h) where the common members become generator variables, and Rx_block is cast as RX_DET_BLOCK (defined in gendet.h) where the common members become detector variables.

The GenDet generator module provides a generic dual-tone generator module with configuration functions, such as Tx_init_dialtone(), to produce the following signals:

- Dialtone 350 Hz to 650 Hz continuous tones (default NORTH AMERICAN).
- Ringback - 440 Hz + 480 Hz tones, 2.0 sec. ON, 4.0 sec. OFF (default NORTH AMERICAN).
- Busy - 480 Hz + 620 Hz tones, 0.5 sec. ON, 0.5 sec. OFF (default NORTH AMERICAN).
- Reorder (congestion) - 480 Hz + 620 Hz tones, 0.25 sec. ON, 0.25 sec. OFF (default NORTH AMERICAN).
- AA – V.32 Call-mode ranging signal, 1800 Hz tone.
- AC – V.32 ANSWER-mode ranging signal, 600 Hz + 3000 Hz tones.
- CNG - Fax 1100 Hz tone.
- CED – Fax 2100 Hz tone.
- ECSD (or ANS) - 2100 Hz tone with phase reversals at .55 sec.

Users can generate other single and dual-tone signals by calling the Tx_init_tone_gen() function described below, and then modify the oscillator parameters to select frequencies, power levels, cadence, and phase reversal intervals. Oscillator frequencies are set by the TX_GEN_BLOCK structure members Tx->frequency1 and Tx->frequency2 for oscillator2 and 2 respectively using the following equation:

Tx->frequency#=8.192*(desired frequency in Hz)

Where:

Tx=(struct TX_GEN_BLOCK *)&Tx_block;

Similarly, the power levels for each oscillator are set by structure members Tx->scale1 and Tx->scale2 using the following equation:

Tx->scale#=32768*10exp(power level in dB/20.0)

The cadence is set by default to always on (100% duty cycle) with Tx->cad_period=Tx->on_time=0. The period of the generated waveform is set by Tx->cad_period in samples at 8kHz, and the "on-time" (the time that a tone is being generated) is set by Tx->on_time in samples. For example, to generate a tone that is on for 1 second and off for 2 seconds, set Tx->on_time to 8000, and Tx->cad_period to 8000+16000=24000. Phase reversals only work for oscillator 1 and the interval between reversals is set by Tx->rev_period in samples at 8kHz. For example, to program oscillator 1 for phase reversals every 450 msec., set Tx->rev_period to 0.45*8000 or 3600.

The Gendet detector module is a 2-state machine that switches between an energy detection state and a signal spectral analysis/detection state. The Rx_energy_det_state function monitors the received samples for the presence of energy above a specified threshold (Rx->threshold), and if detected switches to the Rx_analysis_state. In Rx_analysis_state, a frequency-selective DFT with a 10-msec. window is executed and the spectral magnitudes are written to frequency bins in Rx_block (cast as the structure RX_DET_BLOCK). A bank of signal-specific detectors then processes the spectral information along with a broadband energy estimate. If no mask-enabled signal is detected, then Rx->state switches back to Rx_energy_det to search for another signal.

Gendet includes detection and auto-startup for the following types of signals:

- Dialtone 350 Hz to 650 Hz continuous tone detection and tone tracker.
- Ringback - (350 Hz to 650 Hz tone ending with at least 0.8 msec of silence) detection and tone tracker.
- Busy/Reorder - (350 Hz to 650 Hz 50% duty cycle tone over 1 to 1.5 sec) detection and tone tracker.
- CNG (1100 Hz fax tone) detection and tone tracker.
- CED (2100 Hz fax tone) detection and tone tracker.
- ECSD (2100 Hz with phase reversals) discriminator.
- TEP 1700 fax tone (v.29) detection and tone tracker.
- TEP 1800 fax tone (v.27, v.29) detection and tone tracker.
- v29 detection and demodulator startup.
- v21 detection and demodulator startup.
- v22 USB1detection and Rx_v22A (Side-side) demodulator startup.
- v27 detection and demodulator startup.

- v29 detection and demodulator startup.
- ANS, ECSD, AA, AC, and USB1 detection for v32 auto mode operation.

Gendet.h provides a set of masks which are used with the gendet detector mask, Rx->detector_mask, to selectively enable or disable detection of the above signals.
AUTO_DETECT_MASK – Global enable bit that enables the operation of the complete suite of GenDet detectors upon completion of the DFT frequency bin calculations. Unmask this bit to disallow the execution of any enabled detectors after Rx_analysis_state has calculated the frequency bin values. This would be the case if the user only wanted to make spectral measurements on the line continuously without any detector processing or subsequent receiver module startup.

- V21_CH1_MASK - Enables v.21 channel 1 signal detection and startup of the v.21 demodulator configured for v.21 channel 1 (low channel).
- V21_CH2_MASK Enables v.21 channel 2 HDLC flag detection and startup of the v.21 demodulator configured for v.21 channel 2 (high channel, used for fax).
- V22_MASK - Enables v.21 USB1 and 2225 Hz detections and startup of v.22 Answer side demodulator.
- V27_2400_MASK - Enables detection of v.27 alternations at rate 2,400 bits/sec., and startup of v.27 demodulator configured for 2,400-bits/sec. demodulation.
- V27_4800_MASK - Enables detection of v.27 alternations at rate 4,800 bits/sec., and startup of v.27 demodulator configured for 4,800-bits/sec. demodulation.
- V29_MASK - Enables detection of v.29 alternations and startup of v.29 demodulator.
- V29_MASK - Enables detection of v.29 alternations and startup of v.29 demodulator.
- CED_MASK - Enables detection of fax CED tone. This tone is only detected and tracked - no subsequent processing is activated.
- CNG_MASK - Enables detection of fax CNG tone. This tone is only detected and tracked - no subsequent processing is activated.
- TEP_MASK - Enables detection of fax TEP_2900 and TEP_1800 tones. These tones are only detected and tracked - no subsequent processing is activated.
- CALL_PROGRESS_MASK - The call progress tone detectors, Dialtone, Ringback, and Busy are collectively enabled. Detection and tracking of these tones are reported through the Rx->state_ID. There is no discrimination between reorder and busy.
- V32_AUTOMODE_MASK - Enables detection of ANS (2100 Hz), ECSD (2100 Hz + reversals at 450 msec.), AA (1800 Hz), AC (600 Hz + 3000 Hz), and USB1(Unscrambled Binary Ones modulated by v.22 high channel) signals as required for v.32 auto mode operation. These signals are reported through the Rx->state_ID. These tones are only detected and tracked - no subsequent processing is activated. The user might operate this detector in conjunction with his clear-voice processors at call startup to determine when to switch over to "data" in a fax/data relay system.

Gendet.h provides a set of masks which are used with the DTMF and MF digit detector mask, Rx->digit_CP_mask, to selectively enable or disable detection of DTMF, R1, R2 Forward, or R2 Backward digits. Gendet searches for digits concurrently with the DFT analysis section described above. If a digit is detected, the Rx->state switches to Rx_DTMF_undetector or Rx_MF_undetector until the digit terminates, and then goes back to Rx_energy_det to search for new signals. Similarly, if a tone or modem signal has been detected, then the Rx_state will automatically switch either to a tone tracker or the specific modem demodulator, and digit detection will cease until Rx->state returns to the gendet receiver() states, Rx_energy_det_state or Rx_analysis_state.

6.9.1 ITU-T Compliance:

The tones generated and detected are fully compliant with all sections of ITU-T TBD.

6.9.2 Applications Programmer Interface Functions and Macros

void Tx_init_tone_gen(struct START_PTRS *) - Configures Tx_block[] for dual tone generation with both frequencies set to 0), and switches the Tx->state to Tx_tone_gen.

void Tx_init_tone(short, struct START_PTRS *) - Configures Tx_block[] for single tone generation at the specified frequency calculated by F=8.192*(desired_ frequency in Hz)), and switches the Tx->state to Tx_tone_gen.
void Tx_init_AA(struct START_PTRS *) - Configures Tx_block[] for generation of AA (1800 Hz), and switches the Tx->state to Tx_tone_gen.
void Tx_init_AC(struct START_PTRS *) - Configures Tx_block[] for generation of AC (600 Hz + 3000 Hz), and switches the Tx->state to Tx_tone_gen.
void Tx_init_CED(struct START_PTRS *) - Configures Tx_block[] for generation of CED (2100 Hz), and switches the Tx->state to Tx_tone_gen.
void Tx_init_CNG(struct START_PTRS *) - Configures Tx_block[] for generation of CNG (1100 Hz), and switches the Tx->state to Tx_tone_gen.
void Tx_init_ECSD(struct START_PTRS *) - Configures Tx_block[] for generation of ECSD (2100 Hz with phase reversals every 450 msec.), and switches the Tx->state to Tx_tone_gen.
void Tx_init_dialtone(struct START_PTRS *) - Configures Tx_block[] for generation of dialtone (350 Hz + 440 Hz), and switches the Tx->state to Tx_tone_gen.
void Tx_init_ringback(struct START_PTRS *) - Configures Tx_block[] for generation of ringback (440 Hz + 480 Hz, 2 sec. On, 4 sec. Off), and switches the Tx->state to Tx_tone_gen.
void Tx_init_reorder(struct START_PTRS *) - Configures Tx_block[] for generation of reorder (480 Hz + 620 Hz, 0.25 sec. On, 0.25 sec. Off), and switches the Tx->state to Tx_tone_gen.
void Tx_init_busy(struct START_PTRS *) - Configures Tx_block[] for generation of busy (480 Hz + 620 Hz, 0.5 sec. On, 0.5 sec. Off), and switches the Tx->state to Tx_tone_gen.
void Rx_init_detector(struct START_PTRS *) - Configures Rx_block[] for gendet signal detection according to Rx->detector_mask and Rx->digit_CP_mask. This function sets Rx->digit_detector function pointer to either no_digit_detector(), Rx_DTMF_detector(), Rx_R1_detector(), Rx_R2F_detector(), or Rx_R2B_detector() depending on the contents of Rx->digit_CP_mask. It also calls an internal function, set_Rx_filter_mask() which enables the required band pass filters based on which signal types are masked on in Rx->detector_mask. Finally, Rx_init_detector() switches Rx->state to Rx_energy_det_state.
void set_Rx_detector_mask(short, struct START_PTRS *) - This function writes the first argument to the Rx->detector mask, and if receiver() is currently in either Rx_energy_det_state or Rx_analysis_state, it updates the Rx->filter_mask according to the signals enabled in Rx->detector_mask.
void set_Rx_digit_CP_mask(short, struct START_PTRS *) - This function writes the first argument to the Rx->digit_CP mask.

6.9.3 Transmitter States (by STATE_ID)

TX_TONE_GEN_ID - Dual tone generator state.
TX_CNG_ID - 1100 Hz fax CNG tone generator state.
TX_CED_ID - 2100 Hz fax CED tone generator state.
TX_ECSD_ID - 2100 Hz Echo canceller-suppresser tone with phase reversals every 450 msec.
TX_DIALTONE_ID - ITU-T dial-tone (350 Hz + 440 Hz).
TX_RINGBACK_ID - ITU-T ring-back (440 Hz + 480 Hz, 2 sec. On, 4 sec. Off).
TX_REORDER_ID - ITU-T reorder or congestion (480 Hz + 620 Hz, 0.25 sec. On, 0.25 sec. Off).
TX_BUSY_ID - ITU-T busy (480 Hz + 620 Hz, 0.5 sec. On, 0.5 sec. Off).

6.9.4 Receiver States (by STATE_ID)

RX_ENERGY_DET_ID - Detect increase in average signal energy above -43 dBm0 (RMS).
RX_SIG_ANALYSIS_ID - Perform selective DFT and execute masked detectors.
RX_TONE_ID - Tone detected, search for decrease in signal level to detect end of tone.
RX_CNG_ID - CNG detected, search for decrease in signal level to detect end of tone.
RX_CED_ID - CED detected, search for reversals and decrease in signal level to detect end of tone.
RX_ECSD_ID - ECSD detected, search for decrease in signal level to detect end of tone.
RX_TEP_1700_ID - 1700 Hz FAX Talker Echo Protection tone detected, search for decrease in signal level to detect end of tone.

RX_TEP_1800_ID - 1800 Hz FAX Talker Echo Protection tone detected, search for decrease in signal level to detect end of tone.
RX_DIALTONE_ID – Continuous dial-tone signal detected, search for decrease in signal level to detect end of signal.
RX_RINGBACK_ID – Ring-back signal detected, search for decrease in signal level to detect end of signal.
RX_BUSY_ID - Busy or Reorder (Congestion) signal detected, search for decrease in signal level to detect end of signal.

RX_V32_AUTOMODE_ID - ITU-T v.32 auto mode detector set operating.
RX_V32_CED_ID - 2100 Hz tone present.
RX_V32_ECSD_ID - Phase reversal detected in 2100 Hz tone.
RX_V32_AA_ID - 1800 Hz AA tone present.
RX_V32_AC_ID - 600 Hz + 3000 Hz AC reversals present.
RX_V32_USB1_ID - V22 unscrambled binary 1 signal present.

RX_DETECT_V17_ID – ITU-T v.17 modulation was detected and is being tracked.
RX_DETECT_V21_CH2_ID – ITU-T v.21 channel 2 HDLC modulation was detected and is being tracked
RX_DETECT_V22C_ID – ITU-T v.22bis USB1 modulation was detected and is being tracked
RX_DETECT_BELL_2225_ID – Bellcore 2225 Hz answer tone was detected and is being tracked
RX_DETECT_V27_2400_ID – ITU-T v.27 2400 bits/sec. modulation was detected and is being tracked
RX_DETECT_V27_4800_ID – ITU-T v.27 4800 bits/sec. modulation was detected and is being tracked
RX_DETECT_V29_ID – ITU-T v.29 modulation was detected and is being tracked

6.10 MF (R1/R2F/R2B) Generation and Detection

Simulator source files: mf.c, mf.h.
Texas Instruments source files: mf.asm,mf.inc, mf.h.
Analog Devices source files: mf.dsp,mf.inc, mf.h.
AT&T source files: mf.s,mf.inc, mf.h.

This module generates and detects 15-R1, 15-R2 forward, and 15-R2 backward digits per ITU Recommendation Q.320. The user can configure many generation and detection parameters either at compile-time or at run-time, including transmit tone pair twist and receive detection power level. The generator extracts 4-bit digits from Tx_data[] until the number of digits transmits equals Tx->terminal_count and then transmitter switched to Tx_silence_state. Setting Tx->terminal_count to a negative number causes transmitter to remain in Tx_MF_state forever, continuously reading the digits in Tx_data[]. The detector reports detected R1, R2 forward, or R2 backward digits in Rx_data[] and in the 4 LSBs of Rx->state_ID. The type of MF detector is determined and enabled by ORing R1_MASK, R2F_MASK, or R2B_MASK into the Rx->digit_CP_mask and then calling Rx_init_detector().
The 4 bit Tx_data[] and Rx_data[] digits map to the 15 MF standard digits as follows:

4 bit value	R1 digit	4 bit value	R2F digit	4 bit value	R2B digit
0x0	R1 1	0x0	R2F 0	0x0	R2B 0
0x1	R1 2	0x1	R2F 1	0x1	R2B 1
0x2	R1 3	0x2	R2F 2	0x2	R2B 2
0x3	R1 4	0x3	R2F 3	0x3	R2B 3
0x4	R1 5	0x4	R2F 4	0x4	R2B 4
0x5	R1 6	0x5	R2F 5	0x5	R2B 5
0x6	R1 7	0x6	R2F 6	0x6	R2B 6
0x7	R1 8	0x7	R2F 7	0x7	R2B 7
0x8	R1 9	0x8	R2F 8	0x8	R2B 8
0x9	R1 0	0x9	R2F 9	0x9	R2B 9
0xa	R1 code 11	0xa	R2F A	0xa	R2B A

0xb	R1 code 12	0xb	R2F B	0xb	R2B B
0xc	R1 KP	0xc	R2F C	0xc	R2B C
0xd	R1 KP2	0xd	R2F D	0xd	R2B D
0xe	R1 ST	0xe	R2F E	0xe	R2B E

Table 5 MF Detector Digit Codes

Note that the MF detector operates in parallel with the call progress and modem detectors so you can listen for MF digits at any time while GenDet call progress and fax modem detectors are operating. Also note that the MF detector uses the Rx_fir[] memory buffer for its memory storage.

6.10.1 ITU-T Compliance:

The MF generator/detector is fully compliant with all sections of ITU-T Q.320.

6.10.2 Applications Programmer Interface Functions and Macros

void Tx_init_R1(struct START_PTRS *) - Configures Tx_block[] for generation of a MF R1digit. The number of digits to be generated is specified by Tx->terminal_count (default is 1). The digit on time is specified by Tx->cad_on_time (default is 500 samples or 62.5 msec.), and the total digit period or cadence (on time + off time) is specified by Tx->cad_period (default is 1000 samples or 125 msec.).
void Tx_init_R2F(struct START_PTRS *) - Configures Tx_block[] for generation of a MF R2F digit.
void Tx_init_R2B(struct START_PTRS *) - Configures Tx_block[] for generation of a MF R2B digit.
void Rx_init_R1(struct START_PTRS *) - Configures Rx_block[] for detection of MF R1 digits. Detected digits are written to Rx_data[] and also ORed into the Rx->state_ID.
void Rx_init_R2F(struct START_PTRS *) - Configures Rx_block[] for detection of MF R2F digits. Detected digits are written to Rx_data[] and also ORed into the Rx->state_ID.
void Rx_init_R2B(struct START_PTRS *) - Configures Rx_block[] for detection of MF R2B digits. Detected digits are written to Rx_data[] and also ORed into the Rx->state_ID.

6.10.3 Transmitter States (by STATE_ID)

TX_MF_ID - MF digit generation state.
TX_R1_ID - MF R1 digit generation state.
TX_R2F_ID- MF R2 Forward digit generation state
TX_R2B_ID- MF R2 Backward digit generation state

6.10.4 Receiver States (by STATE_ID)

RX_MF_ID - MF digit detection state.
RX_R1_ID - MF R1 digit detection state. Upon detection, the digit code is ORed with the state_ID. For example, if an R1digit "1" is currently detected, the Rx_state_ID would be RX_R1_ID|1 or 0x1411.
RX_R2F_ID - MF R2F digit detection state. Upon detection, the digit code is ORed with the state_ID. For example, if an R2F digit "1" is currently detected, the Rx_state_ID would be RX_R2F_ID|1 or 0x1421.
RX_R2B_ID - MF R2B digit detection state. Upon detection, the digit code is ORed with the state_ID. For example, if an R2B digit "1" is currently detected, the Rx_state_ID would be RX_R2B_ID|1 or 0x14a1.

6.11 Rxtx State Machine Interface

Simulator source files: rxtx.c, vmodem.h.
Texas Instruments source files: rxtx.asm, vmodem.inc.
Analog Devices source files: rxtx.dsp, vmodem.inc.
AT&T source files: rxtx.s, vmodem.inc.

This module contains the transmit and receive state machine entry points and initialization functions called by users to invoke all generators, detectors, and modems. It also contains the default Tx_silence() and Rx_idle() states, which produce silence samples and discard receive samples respectively.

6.11.1 Applications Programmer Interface Functions

void transmitter(struct START_PTRS *) - This is the transmitter state machine entry point. A call to this function will execute the current transmitter component (specified in Tx->state with state identifier in Tx->state_ID) and, if possible, generates the number of samples specified in Tx->num_samples into the transmitter sample buffer, Tx_sample[] taking symbols from Tx_data[] as required. The transmitter() function contains conditional logic to check the condition of Tx->sample_head and Tx->sample_tail to determine if the number of samples specified by Tx->num_samples can be placed in the Tx_sample[] buffer yet. Transmitter() tries to minimize the delay from where *Tx->sample_head has written samples into the buffer to where *Tx->sample_tail reads them out by checking if the distance between the two pointers is equal to or less than Tx->num_samples. If so, calls are made to Tx->state to produce Tx->num_samples samples into Tx_sample[]. In this way, the transmit sample buffer delay is kept to within Tx->num_samples. Transmitter() returns non-zero if any samples were written to *Tx->sample_head, and zero otherwise. The C Source for transmitter() is shown below:

```
short transmitter(struct START_PTRS *ptrs)
{
      short k;
      struct TX_BLOCK *Tx;

      Tx=(struct TX_BLOCK *)ptrs->Tx_block_start;
      k=(short)(Tx->sample_head-Tx->sample_tail);
      if (k<0)
            k+=Tx->sample_len;
      if (k>Tx->num_samples)
            return(0);

      Tx->call_counter=Tx->num_samples;
      do
            {
            (Tx->state)(Tx);
            }while (--Tx->call_counter>0);
      return(1);
}
```

void Tx_block_init(struct START_PTRS *) - Function to initialize the control portion of the Tx_block structure as follows:

```
Tx->state=&Tx_silence_state;
Tx->state_ID=TX_SILENCE_ID;
Tx->num_samples=NUM_SAMPLES;
Tx->scale=TX_MINUS_16DBM0;
Tx->sample_head=&Tx_sample[0];
Tx->sample_tail=&Tx_sample[0];
Tx->sample_len=TX_SAMPLE_LEN;
Tx->call_counter=0;
Tx->data_head=&Tx_data[0];
Tx->data_tail=&Tx_data[0];
Tx->data_len=TX_DATA_LEN;
Tx->sample_counter=0;
Tx->symbol_counter=0;
Tx->rate=0;
Tx->mode=0;
Tx->system_delay=0;
Tx->terminal_count=0;
```

The user MUST call this function before calling transmitter() or any other functions, but it is only necessary to call it once from system reset.

void Tx_init_silence(struct START_PTRS *) - Configures transmitter to generate silence. This is the default state from Tx_block_init().

void receiver(struct START_PTRS *) - This is the receiver state machine entry point. A call to this function will execute the current receiver component (specified in Rx->state with state identifier in Rx->state_ID) and processes minimally the number of samples specified in Rx->num_samples from the receive sample buffer, Rx_sample[], placing symbols into Rx_data[] as detected. The receiver() function contains conditional logic to check the condition of Rx->sample_head and Rx->sample_tail to determine if the number of samples specified by Rx->num_samples are present in the Rx_sample[] buffer yet. Receiver() tries to consume all samples from where *Rx->sample_head has written samples into the buffer to at the time of entry , to where *Rx->sample_tail is by checking if the distance between the two pointers is equal to or less than Rx->num_samples. If so, calls are made to Rx->state to process all of the samples in Rx_sample[] up to Rx->sample_stop. In this way receiver() always tries to process all samples in Rx_sample[] regardless of Rx->num_samples. Receiver() returns non-zero if the number of samples in Rx_sample[] is equal to or greater than Rx->num_samples, and zero otherwise. The C Source for receiver() is shown below:

```
short receiver(struct START_PTRS *ptrs)
{
      short k;
      struct RX_BLOCK *Rx;

      Rx=(struct RX_BLOCK *)ptrs->Rx_block_start;
      k=(short)(Rx->sample_head-Rx->sample_tail);
      if (k<0)
            k+=Rx->sample_len;
      if (k<Rx->num_samples)
            return(0);

      /**** call the receiver function num_samples times ****/

      Rx->call_counter=Rx->num_samples;
      Rx->sample_stop=Rx->sample_head;
      do
            {
            (Rx->state)(Rx);
            } while (--Rx->call_counter>0);
      return(1);
}
```

void Rx_block_init(struct START_PTRS *) - Function to initialize the control portion of the Rx_block structure as follows:

```
Rx->state=&Rx_idle_state;
Rx->state_ID=RX_IDLE_ID;
Rx->num_samples=NUM_SAMPLES;
Rx->status=0;
Rx->sample_head=&Rx_sample[0];
Rx->sample_tail=&Rx_sample[0];
Rx->sample_len=RX_SAMPLE_LEN;
Rx->call_counter=0;
Rx->data_head=&Rx_data[0];
Rx->data_tail=&Rx_data[0];
Rx->data_ptr=&Rx_data[0];
Rx->data_len=RX_DATA_LEN;
Rx->sample_counter=0;
Rx->symbol_counter=0;
Rx->rate=0;
Rx->mode=0;
Rx->detector_mask=0;
```

The user MUST call this function before calling receiver() or any other Rx functions, but it is only necessary to call it once from system reset.

void Rx_init_measure_power(struct START_PTRS *ptrs) - Function to configure Rx_block to continuously measure the average receive signal power.

6.11.2 Transmitter States (by STATE_ID)

TX_SILENCE_STATE - Generates silence (samples of 0) into Tx_sample[], and circularly updates Tx->sample_head..

6.11.3 Receiver States (by STATE_ID)

RX_IDLE_STATE - Reads and discards samples from Rx_sample[], and circularly updates Rx->sample_tail.

RX_MEASURE_POWER_STATE – Calculates the average power of the receive samples in Rx_sample[] and writes the result to Rx->power.

6.12 V.14 Sync/Async Converter

Simulator source files: v14.c, v14.h, dh.c, dh.h.
Texas Instruments source files: v14.c, v14.h, dh.c, dh.h.
Analog Devices source files: v14.c, v14.h, dh.c, dh.h.
AT&T source files: v14.c, v14.h, dh.c, dh.h.

This module implements the synchronous-to-asynchronous conversion protocol specified in ITU-T Recommendation v.14. On the transmit side, it inserts a zero start bit and a one stop bit to a transmit byte, slices it to the modem's symbol data format, and writes it to the modem's Tx_data[] transmit symbol buffer. On the receive side, it searches the incoming bit stream for a zero start bit. Upon detection, it de-slices the received symbols from the modem's Rx_data[] receive symbol buffer to form an output byte, and strips out the start and stop bits. V14 can be built to create it's own input and output byte buffers, or to make use of existing input and output byte buffers in a software UART or other byte-driven I/O device. The number of data bits is configurable from 5 to 8 bits at run-time.

6.12.1 ITU-T Compliance

The modem is fully compliant with all sections of ITU-T recommendation v.14 except:
Section 4 - no support for 9 data bits.
Section 7.3 - no support for Break signal.
Annex A - no interchange circuits.

6.12.2 Applications Programmer Interface Functions and Macros

void init_v14(struct DH_START_PTRS *) - Configures the DH_block structure members to assume Tx_v14 and Rx_v14 operational mode. After this function executes, call to the Data Handlers will go to Tx_v14() and Rx_v14(). If the number of data bits is not eight (the default) then DH_block->mode must be configured BEFORE the call to this function, as shown below:

```
DH->mode&=~DH_V14_NDATA_MASK;
DH->mode|=DH_V14_5_DATA_BITS;       // see dh.h for definitions
init_v14 (ptrs);
```

short Tx_v14(struct DH_START_PTRS *) - This function implements the transmit v.14 async-to-sync conversion. It checks the Tx_DTE[] buffer for available characters, and if one or more are present, it processes them to produce a symbol data stream that is compatable with the symbol-per-entry format of the transmitter's modulator, and writes the symbol data to the Tx_data[] buffer.

short Rx_v14(struct DH_START_PTRS *) - This function monitors the receiver demodulator's symbol data stream in Rx_data[] for the presence of a start bit. If a start bit is detected, the received symbol data is processed to strip the start and stop bits, and the character is written to Rx_DTE[]. If Rx_DTE[] is full and cannot accept data, the character is discarded.

6.12.3 Transmitter States (by STATE_ID)

Not Applicable

6.12.4 Receiver States (by STATE_ID)

Not Applicable

6.13 V.17 modem

Simulator source files: v17.c, v17.h.
Texas Instruments source files: v17.asm, v17.inc, v17.h.
Analog Devices source files: v17.dsp, v17.inc, v17.h.
AT&T source files: v17.s, v17.inc, v17.h.

This half-duplex modem is intended for use in FAX applications. It employs trellis-coded modulation at data signaling rates of 7.2, 9.6, 12.0, and 14.4 kbit/s with a symbol rate of 2400 symbols per sec. The modulation methods are rectangular 16-QAM at 7.2 kbit/s, modified cross 32-QAM at 9.6 kbit/s, rectangular 64 -QAM at 12 kbit/s, and modified cross 128-QAM at 14.4 kbit/s.

6.13.1 ITU-T Compliance

The modem is fully compliant with all sections of ITU-T recommendation v.17 except:
Section 3 - no interchange circuits.

6.13.2 Applications Programmer Interface Functions and Macros

void Tx_init_v17(struct START_PTRS *) - Configures Tx_block[] for v.17 transmitter operation at the rate specified by Tx->rate or 14,400 bits/s if no valid v.17 rate is selected. If the LONG_RESYNC_FIELD bit of Tx->mode is cleared (set to zero) then the modulator is configured for LONG_TRAIN_MODE, and if this field is set to one, then the modulator is configured for RESYNC_MODE. If the TEP_FIELD bit of Tx->mode is disabled (set to zero), then Tx->state is set to Tx_v17_segment1. If this field is enabled (set to one), then Tx->state is set to Tx_v17_TEP and the modulator generates 200 msec. of un-modulated carrier (Talker Echo Protection).
Tx_init_v17_14400(struct START_PTRS *) - Macro that configures Tx_block[] for v17 modulation at 14400 bits/sec. without Talker Echo Protection (TEP).
Tx_init_v17_14400_TEP(struct START_PTRS *) - Macro that configures Tx_block[] for v17 modulation at 14400 bits/sec. with Talker Echo Protection (TEP) enabled.
Tx_init_v17_12000(struct START_PTRS *) - Macro that configures Tx_block[] for v17 modulation at 12000 bits/sec. without Talker Echo Protection (TEP).
Tx_init_v17_12000_TEP(struct START_PTRS *) - Macro that configures Tx_block[] for v17 modulation at 12000 bits/sec. with Talker Echo Protection (TEP) enabled.
Tx_init_v17_9600(struct START_PTRS *) - Macro that configures Tx_block[] for v17 modulation at 9600 bits/sec. without Talker Echo Protection (TEP).
Tx_init_v17_9600_TEP(struct START_PTRS *) - Macro that configures Tx_block[] for v17 modulation at 9600 bits/sec. with Talker Echo Protection (TEP) enabled.
Tx_init_v17_7200(struct START_PTRS *) - Macro that configures Tx_block[] for v17 modulation at 7200 bits/sec. without Talker Echo Protection (TEP).
Tx_init_v17_7200_TEP(struct START_PTRS *) - Macro that configures Tx_block[] for v17 modulation at 7200 bits/sec. with Talker Echo Protection (TEP) enabled.

enable_Tx_v17_TEP(struct START_PTRS *) - Macro that sets the TX_TEP_FIELD in Tx->mode, thereby enabling TEP, and configures Tx_block[] for v17 modulation at Tx->rate or 9600 bits/sec. if no valid v.17 rate is selected.
disable_Tx_v17_TEP(struct START_PTRS *) - Macro that clears the TX_TEP_FIELD in Tx->mode, thereby disabling TEP, and configures Tx_block[] for v17 modulation at Tx->rate or 9600 bits/sec. if no valid v.17 rate is selected.
enable_Tx_v17_long_train(struct START_PTRS *) - Macro that clears the TX_LONG_RESYNC_FIELD in Tx->mode, thereby enabling v.17 long train mode. This is the default mode for ITU-I V.17.
enable_Tx_v17_resync(struct START_PTRS *) - Macro that sets the TX_LONG_RESYNC_FIELD in Tx->mode, thereby enabling v.17 re-sync or short train mode.
enable_Rx_v17_long_train(struct START_PTRS *) - Macro that clears the RX_LONG_RESYNC_FIELD in Tx->mode, thereby enabling v.17 long train mode. This is the default mode for ITU-I V.17.
enable_Rx_v17_resync(struct START_PTRS *) - Macro that sets the RX_LONG_RESYNC_FIELD in Tx->mode, thereby enabling v.17 re-sync or short train mode. In this mode, the equalizer coefficients are not reset to zero and the demodulator operates with pre-trained equalizer coefficients.

6.13.3 Transmitter States (by STATE_ID)

TX_V17_TEP_ID - 200 msec. of unmodulated carrier (1800 Hz).
TX_V17_SILENCE1_ID - 20 msec. Of silence.
TX_V17_SEGMENT1_ID - ITU-T v.17 segment 1.
TX_V17_SEGMENT2_ID - ITU-T v.17 segment 2.
TX_V17_SEGMENT3_ID- ITU-T v.17 segment 3.
TX_V17_SEGMENT4_ID- ITU-T v.17 segment 4.
TX_V17_MESSAGE_ID - Data transmission state.
TX_V17_SEGMENTA_ID - ITU-T v.17 segment A.
TX_V17_SEGMENTB_ID - ITU-T v.17 segment B.

6.13.4 Receiver States (by STATE_ID)

RX_V17_TRAIN_LOOPS_ID - Train carrier, symbol timing, and AGC loops.
RX_V17_DETECT_EQ_ID - Search for start of equalizer training sequence.
RX_V17_TRAIN_EQ_ID - Train adaptive equalizer.
RX_V17_SCR1_ID - De-scramble and discard SCR1 sequence.
RX_V17_MESSAGE_ID - Demodulate data symbols and sink to Rx_data[].

6.14 V.21 modem

Simulator source files: v21.c, v21.h.
Texas Instruments source files: v21.asm, v21.inc, v21.h.
Analog Devices source files: v21.dsp, v21.inc, v21.h.
AT&T source files: v21.s, v21.inc, v21.h.

This modem is used in fax, v.34 startup, and data applications. It is a 2-channel modem operating at a data-signaling rate of 300 bit/s with a symbol rate of 300 symbols/sec. The modulation method is continuous phase frequency shift keying (CP-FSK). This modem will operate in fax mode (the default) or in data mode if V21_DATA_MODEM is be defined as ENABLED (1) in the options file.

6.14.1 ITU-T Compliance:

The modem is fully compliant with all sections of ITU-T recommendation v.21 except:
Section 8 - no interchange circuits.

6.14.2 Applications Programmer Interface Functions and Macros

void Tx_init_v21_ch1(struct START_PTRS *) - Configures Tx_block[] for v.21 channel 1 transmitter (modulator) operation, and sets the Tx->state to Tx_v21_message.
void Tx_init_v21_ch2(struct START_PTRS *) - Configures Tx_block[] for v.21 channel 2 transmitter (modulator) operation, and sets the Tx->state to Tx_v21_message.
void Rx_init_v21_ch1(struct START_PTRS *) - Configures Rx_block[] for v.21 channel 1 receiver (demodulator) operation, and sets the Rx->state to Rx_v21_message.
void Rx_init_v21_ch2(struct START_PTRS *) - Configures Rx_block[] for v.21 channel 2 receiver (demodulator) operation, and sets the Rx->state to Rx_v21_message.

6.14.3 Transmitter States (by STATE_ID)

TX_V21_CH1_SILENCE_ID - Generates silence until the receiver detects marks (data mode only).
TX_V21_CH2_SILENCE1_ID - Generates silence for 2 seconds (data mode only).
TX_V21_CH2_ANS_ID - Generates 2100 Hz answer tone for 4 seconds (data mode only).
TX_V21_CH2_SILENCE2_ID - Generates silence for 75 msec (data mode only).
TX_V21_MESSAGE_ID - Data transmission state.

6.14.4 Receiver States (by STATE_ID)

RX_V21_CH1_MARKS_ID - Detect Marks pattern on channel 1 (data mode only).
RX_V21_CH1_START_BIT_ID - Searches for a start bit on channel 1 and sinks MARKs to Rx_data[] (data mode only).
RX_V21_CH2_MARKS_ID - Detect Marks pattern on channel 2 (data mode only).
RX_V21_CH2_START_BIT_ID - Searches for a start bit on channel 2 and sinks MARKs to Rx_data[] (data mode only)
RX_V21_MESSAGE_ID - Demodulate data symbols and sink to Rx_data[]. In data mode only, switches back to **RX_V21_CH1_START_BIT_ID** or **RX_V21_CH2_START_BIT_ID** after the detection of a MARK bit.

6.15 V.22 bis modem

Simulator source files: v22.c, v22.h.
Texas Instruments source files: v22.asm, v22.inc, v22.h.
Analog Devices source files: v22.dsp, v22.inc, v22.h.
AT&T source files: v22.s, v22.inc, v22.h.

This split-band modem supports full duplex data transfer at 1200 and 2400 bit/s with a symbol rate of 600 symbols/sec. The modulation methods are QPSK at 1200 bit/s, and rectangular 16-QAM at 2400 bit/s.

6.15.1 ITU-T Compliance:

The modem is fully compliant with all sections of ITU-T recommendation v.22 bis except:
Sections 2.1 and 2.2 - No guard tones.
Section 3 - no interchange circuits.
Section 4 - synchronous modes only.
Section 7 - not supported

6.15.2 Applications Programmer Interface Functions and Macros

void Tx_init_v22A(struct START_PTRS *) - Configures Tx_block[] for v22 *bis* ANSWER side modulation starting in the Tx_v22A_silence2 state, and Rx_block[] for v22 *bis* ANSWER side demodulation.
void Tx_init_v22A_ANS(struct START_PTRS *) - Configures Tx_block[] for v22 *bis* ANSWER side modulation with the 2100 Hz answer tone, starting in the Tx_v22A_ANS state. The demodulator is initialized subsequently following the completion of the Tx_v22A_silence2 state.

void Tx_v22A_retrain(struct START_PTRS *) - Configures Tx_block[] to initiate a v22 bis retrain/rate change request, and Rx_block[] to respond to a detected retrain/rate change reply.
void Tx_init_bell212A(struct START_PTRS *) - Configures Tx_block[] for Bell212a ANSWER side modulation starting in the Tx_v22A_silence2 state, and Rx_block[] for v22 or Bellcore ANSWER side demodulation.
void Tx_init_bell212A_ANS(struct START_PTRS *) - Configures Tx_block[] for Bell212a ANSWER side modulation with the 2100 Hz answer tone, starting in the Tx_v22A_ANS state. The demodulator is initialized subsequently following the completion of the Tx_v22A_silence2 state.

void Tx_init_v22C(struct START_PTRS *) - Configures Tx_block[] for v22 *bis* CALL side modulation starting in the Tx_v22C_silence state. The demodulator startup is initiated from the v22_detector in "gendet", so be sure that the Rx->detector_mask has V22_MASK and AUTO_DETECT_MASK ORed in.
void Tx_init_bell212C(struct START_PTRS *) - Configures Tx_block[] for Bell212a CALL side modulation starting in the Tx_v22C_silence state. The demodulator startup is initiated from the v22_detector in "gendet", so be sure that the Rx->detector_mask has V22_MASK and AUTO_DETECT_MASK ORed in. The detector discriminates between 2225 Hz and USB1 (2250 Hz plus 2850 Hz) and sets the BELLCORE_MODE_BIT in Tx->modem accordingly.

void Tx_v22C_retrain(struct START_PTRS *) - Configures Tx_block[] to initiate a v22 bis retrain/rate change request, and Rx_block[] to respond to a detected retrain/rate change reply

6.15.3 Transmitter States (by STATE_ID)

TX_V22A_SILENCE1_ID - ITU-T v.22 *bis* ANSWER 2.15 sec. Silence preceding ANS (2100 Hz).
TX_V22A_ANS_ID - ITU-T v.22 *bis* ANSWER 2100 Hz tone for 3.3 sec.
TX_V22A_SILENCE2_ID - ITU-T v.22 *bis* ANSWER 75 msec. Silence preceding USB1.
TX_BELL212A_2225_ID – Bellcore Bell212a ANSWER tone at 2225 Hz.
TX_V22A_UB1_ID - ITU-T v.22 *bis* ANSWER Unscrambled binary 1 at 1200 bit/s.
TX_V22A_S1_ID - ITU-T v.22 *bis* ANSWER S1 alternations for 100 msec.
TX_V22A_SCR1_ID - ITU-T v.22 *bis* ANSWER Scrambled R2 at 1200 bit/se
TX_V22A_SB1_R2_ID - ITU-T v.22 *bis* ANSWER Scrambled R2 at R2 rate.
TX_V22A_MESSAGE_ID - ANSWER modulate data symbols from Tx_data buffer.

TX_V22C_SILENCE_ID - ITU-T v.22 *bis* CALL Silence preceding S1.
TX_V22C_S1_ID - ITU-T v.22 *bis* CALL S1 alternations for 100 msec.
TX_V22C_SCR1_ID - ITU-T v.22 *bis* CALL Scrambled binary 1 at 1200 bit/s.
TX_V22C_SB1_R2_ID - ITU-T v.22 *bis* CALL Scrambled R2 at R2 rate.
TX_V22C_MESSAGE_ID - CALL modulate data symbols from Tx_data buffer.

6.15.4 Receiver States (by STATE_ID)

RX_V22A_START_DETECT_ID – ITU-T v.22 *bis* ANSWER Detect S1, Scrambled binary 1 (SB1), or V.32 AA.
RX_V22A_TRAIN_LOOPS_ID – ITU-T v.22 *bis* ANSWER Train carrier, symbol timing, and agc loops.
RX_V22A_TRAIN_EQ_ ID – ITU-T v.22 *bis* ANSWER Train adaptive equalizer.
RX_V22A_MESSAGE_ID – ITU-T v.22 *bis* ANSWER Demodulate data symbols and sink to Rx_data buffer.
RX_V22A_RC_RESPOND_ID – ITU-T v.22 *bis* ANSWER responding to rate change request.
RX_V22A_RC_INITIATE_ID – ITU-T v.22 *bis* ANSWER Initiating rate change request.

RX_V22C_START_DETECT_ID - ITU-T v.22 *bis* CALL Detect S1, Scrambled binary 1 (SB1), or V.32 AC.
RX_V22C_TRAIN_LOOPS_ID - ITU-T v.22 *bis* CALL Train carrier, symbol timing, and agc loops.
RX_V22C_TRAIN_EQ_ID - ITU-T v.22 *bis* CALL Train adaptive equalizer**.**

RX_V22C_MESSAGE_ID - ITU-T v.22 *bis* CALL Demodulate data symbols and sink to Rx_data buffer ITU-T v.22 *bis*.
RX_V22C_RC_RESPOND_ID - ITU-T v.22 *bis* CALL responding to rate change request.
RX_V22C_RC_INITIATE_ID - ITU-T v.22 *bis* CALL Initiating rate change request.

6.16 V.23 modem

Simulator source files: v23.c, v23.h.
Texas Instruments source files: v23.asm, v23.inc, v23.h.
Analog Devices source files: v23.dsp, v23.inc, v23.h.
AT&T source files: v23.s, v23.inc, v23.h.

This modem is used generic data and caller ID applications. It is a 2 channel modem with the forward channel operating at a data signaling rate of 600 and 1200 bit/s with symbol rates of 600 and 1200 symbols/sec., and the backward channel operating at a data signaling rate of 75 bit/s with a symbol rate of 75 symbols/sec. The modulation method is continuous phase frequency shift keying (CP-FSK).
The demodulator does not include any preamble detection or training – it simply demodulates and dumps the data in Rx_data[]. For optimum performance the user should provide a detector for the start of a v.23 modulated burst (such as in the Caller ID case) and then initialize the demodulator.

6.16.1 ITU-T Compliance:

The modem is fully compliant with all sections of ITU-T recommendation v.23 except:
Section 8 - no interchange circuits.

6.16.2 Applications Programmer Interface Functions and Macros

void Tx_init_v23(struct START_PTRS *) - Configures Tx_block[] for v.23 forward channel transmitter (modulator) operation, and sets the Tx->state to Tx_v23_message.
void Tx_init_v23B(struct START_PTRS *) - Configures Tx_block[] for v.23 backward channel transmitter (modulator) operation, and sets the Tx->state to Tx_v23_message.
void Rx_init_v23(struct START_PTRS *) - Configures Rx_block[] for v.23 forward channel receiver (demodulator) operation, and sets the Rx->state to Rx_v23_message.
void Rx_init_v23B(struct START_PTRS *) - Configures Rx_block[] for v.23 backward channel receiver (demodulator) operation, and sets the Rx->state to Rx_v23_message.

6.16.3 Transmitter States (by STATE_ID)

TX_V23_MESSAGE_ID - Data transmission state in the forward channel. One-bit symbols are read from Tx_data[] and modulated.
TX_V23B_MESSAGE_ID - Data transmission state in the backward channel. One-bit symbols are read from Tx_data[] and modulated

6.16.4 Receiver States (by STATE_ID)

RX_V23_MESSAGE_ID - Demodulate data symbols from the forward channel and sink to Rx_data[]. The FSK_demodulator loss-of-signal detector will operate reliably if approximately 80 samples of the modulated signal are present in Rx_sample[] at the time Rx_init_v23() was called. Otherwise the user must rely on demodulated data content for validation.
RX_V23_MESSAGE_ID - Demodulate data symbols from the backward channel and sink to Rx_data[].The FSK_demodulator loss-of-signal detector will operate reliably if approximately 80 samples of the modulated signal are present in Rx_sample[] at the time Rx_init_v23() was called. Otherwise the user must rely on demodulated data content for validation.

6.17 V.26 modem

Simulator source files: v26.c, v26.h.
Texas Instruments source files: v26.asm, v26.inc, v26.h.

Analog Devices source files: v26.dsp, v26.inc, v26.h.
AT&T source files: v26.s, v26.inc, v26.h.

This full duplex echo canceller modem implements the v.26 and v.26*bis* standards and operates at 1.2 and 2.4 kbits/sec, and is used primarily for STU-III applications. The modulation method is essentially QPSK and has two-phase plans - Alternative A which has 90-degree increments, and Alternative B, which has 45 and 135-degree increments. By default, the Rx_init_v26() functions enable a detector for the synchronizing signal to detect start-up and discriminate 1.2 and .4 kbits/sec. rates. The user can set the RX_DETECTOR_DISABLE bit in Rx->mode to bypass the synchronizing signal detector but must ensure that Rx->power contains a valid estimate of received signal power prior to Rx_init_v26() call. In normal operation the user supplies the synchronization signal as part of the data to the transmitter. The modem may be configured for a fast synchronization mode. In this case the transmitter will output six symbols of alternations for synchronization before transmitting the user data, and the receiver will obtain sync within six symbol periods.

6.17.1 ITU-T Compliance:

The modem is fully compliant with all sections of ITU-T recommendation v.26 except:
Section 1.0 c), 5 - backward channel not supported.
Section 6 - no interchange circuits.
The modem is fully compliant with all sections of ITU-T recommendation v. 26*bis* except:
Section 1 - backward channel not supported.
Section 5 - no interchange circuits.

6.17.2 Applications Programmer Interface Functions and Macros

void Tx_init_v26(struct START_PTRS *) - Configures Tx_block[] for v26 modulation and sets the Tx->state to Tx_v26_message or to Tx_v26_SYN based on the TX_V26_FAST_SYN_FIELD mode bit.
Tx_init_v26_ALT_A(struct START_PTRS *) - Macro that configures Tx_block[] for v26 modulation in Alternate A phase plan and sets the Tx->state to Tx_v26_message or to Tx_v26_SYN based on the TX_V26_FAST_SYN_FIELD mode bit.
Tx_init_v26_ALT_B(struct START_PTRS *) - Macro that configures Tx_block[] for v26 modulation in Alternate B phase plan and sets the Tx->state to Tx_v26_message or to Tx_v26_SYN based on the TX_V26_FAST_SYN_FIELD mode bit.
Tx_init_v26_fast(struct START_PTRS *) - Macro that configures Tx_block[] for fast sync v26 modulation in Alternate B phase plan. It sets the TX_V26_FAST_SYN_FIELD mode bit so that the Tx->state will be set to Tx_v26_SYN.
void Rx_init_v26(struct START_PTRS *) - Configures Rx_block[] for v26 demodulation and sets the Rx->state to Rx_v26_detect_sync or to Rx_v26_message based on the RX_DETECTOR_DISABLE mode bit.
Rx_init_v26_ALT_A(struct START_PTRS *) - Macro that configures Rx_block[] for v26 demodulation in Alternate A phase plan and sets the Rx->state to Rx_v26_detect_sync or to Rx_v26_message based on the RX_DETECTOR_DISABLE mode bit.
Rx_init_v26_ALT_B(struct START_PTRS *) - Macro that configures rx_block[] for v26 demodulation in Alternate B phase plan and sets the Rx->state to Rx_v26_detect_sync or to Rx_v26_message based on the RX_DETECTOR_DISABLE mode bit.
Rx_init_v26_fast(struct START_PTRS *) - Macro that configures rx_block[] for fast sync v26 demodulation in Alternate B phase plan. It sets the RX_V26_FAST_SYN_FIELD mode bit to enable the fast sync functionality. Rx->state will be set to Rx_v26_detect_sync or to Rx_v26_message based on the RX_DETECTOR_DISABLE mode bit. Note that the RX_DETECTOR_DISABLE bit overrides the RX_V26_FAST_SYN_FIELD mode bit, and must be cleared (the default) for the fast sync mode.
short v26_GPA_scrambler(short, struct START_PTRS *) - Scrambles the data supplied as the first argument using GPA polynomial.
short v26_GPC_scrambler(short, struct START_PTRS *) - Scrambles the data supplied as the first argument using GPC polynomial.

short v26_GPA_descrambler(short, struct START_PTRS *) - De-scrambles the data supplied as the first argument using GPA polynomial.
short v26_GPC_descrambler(short, struct START_PTRS *) - De-scrambles the data supplied as the first argument using GPA polynomial.
void enable_echo_canceller(struct START_PTRS *) - Initializes and enables the echo canceller

6.17.3 Transmitter States (by STATE_ID)

TX_V26_SYNC_ID - Modulate the sync pattern (alternations) if in fast mode.
TX_V26_POSTAMBLE_ID - Modulates to flush the Tx filters.
TX_V26_MESSAGE_ID - Modulate data symbols sourced from Tx_data[].

6.17.4 Receiver States (by STATE_ID)

RX_V26_DETECT_SYNC_ID - Detect the "11" synchronizing signal and initiate the demodulator upon detection.
RX_V26_TRAIN_LOOPS_ID - Obtains carrier and symbol sync in fast mode.
RX_V26_MESSAGE_ID - Demodulate data symbols and sink to Rx_data[].

6.18 V.27 ter modem

Simulator source files: v27.c, v27.h.
Texas Instruments source files: v27.asm, v27.inc, v27.h.
Analog Devices source files: v27.dsp, v27.inc, v27.h.
AT&T source files: v27.s, v27.inc, v27.h.

This half/full duplex modem is used in half duplex FAX applications, and full duplex in leased-line applications. It employs QPSK and 8-PSK modulation at data signaling rates of 2.4 and 4.8 kbits/sec., with symbol rates of 1200 and 1600 symbols/sec respectively. It includes an automatic adaptive equalizer, long and short training sequences, and optional Talker Echo Protection tone generation.

6.18.1 ITU-T Compliance:

The modem is fully compliant with all sections of ITU-T recommendation v.27 ter except:
Section 4 - not supported.
Section 5 - no interchange circuits.

6.18.2 Applications Programmer Interface Functions and Macros

void Tx_init_v27(struct START_PTRS *) - Configures Tx_block[] for v27 *ter* QPSK modulation at the rate specified by Tx->rate or 4800 bit/s if no valid v.27 rate is selected, and the Tx->state is set to Tx_v27_segment1or Tx_v27_segment3 based on the TEP_FIELD bit of Tx->mode.
void enable_Tx_v27_startup_seq(struct START_PTRS *) - Sets the LONG_RESYNC_FIELD bit of Tx->mode to STARTUP (set to zero) which causes the transmitter to sequence through the longer startup training segments.
void enable_Tx_v27_turnaround_seq(struct START_PTRS *) - Sets the LONG_RESYNC_FIELD bit of Tx->mode to TURNAROUND (set to 1) which causes the transmitter to sequence through the shorter turnaround training segments.
void enable_Tx_v27_TEP(struct START_PTRS *) - Sets the TEP_FIELD bit of Tx->mode (set to one) to enable Talker Echo Protection tone generation**.**
void disable_Tx_v27_TEP(struct START_PTRS *) - Resets the TEP_FIELD bit of Tx->mode (set to zero) to disable Talker Echo Protection tone generation**.**
void enable_Rx_v27_startup_seq(struct START_PTRS *) - Sets the LONG_RESYNC_FIELD bit of Rx->mode to STARTUP (set to zero) which causes the demodulator to expect the longer startup training segments.

void enable_Rx_v27_turnaround_seq(struct START_PTRS *) - Sets the LONG_RESYNC_FIELD bit of Rx->mode to TURNAROUND (set to 1) which causes the demodulator to expect the shorter turnaround training segments.

6.18.3 Transmitter States (by STATE_ID)

TX_V27_SEGMENT1_ID - ITU-T v.27 *ter* segment 1.
TX_V27_SEGMENT2_ID - ITU-T v.27 *ter* segment 2.
TX_V27_SEGMENT3_ID - ITU-T v.27 *ter* segment 3.
TX_V27_SEGMENT4_ID - ITU-T v.27 *ter* segment 4.
TX_V27_SEGMENT5_ID - ITU-T v.27 *ter* segment 5.
TX_V27_MESSAGE_ID - data transmission state.
TX_V27_SEGMENTA_ID - ITU-T v.27 *ter* segment A.
TX_V27_SEGMENTB_ID - ITU-T v.27 *ter* segment B

6.18.4 Receiver States (by STATE_ID)

RX_V27_TRAIN_LOOPS_ID - Train carrier, symbol timing, and agc loops.
RX_V27_DETECT_EQ_ID - Search for start of equalizer training sequence.
RX_V27_TRAIN_EQ_ID - Train adaptive equalizer.
RX_V27_SCR1_ID - De-scramble and discard SCR1 sequence.
RX_V27_MESSAGE_ID - Demodulate data symbols and sink to Rx_data[].

6.19 V.29 modem

Simulator source files: v29.c, v29.h.
Texas Instruments source files: v29.asm, v29.inc, v29.h.
Analog Devices source files: v29.dsp, v29.inc, v29.h.
AT&T source files: v29.s, v29.inc, v29.h.

This half duplex modem was originally intended for 4-wire leased line operation but is most commonly used in FAX applications. It employs an unusual amplitude-phase shift modulation scheme at 4.8, 7.2, and 9.6 kbits/sec with a symbol rate of 2400 symbols/sec. If the LONG_RESYNC_FIELD bit of Rx->mode is cleared (set to zero) then the demodulator is configured for LONG_TRAIN_MODE which is the standard ITU-T v.29 training sequence for Segments 2 and 3. If this field is set to one, then the demodulator is configured for RESYNC_MODE where Segments 2 and 3 are reduced to 27 msec. and 27 msec. respectively. V29 resync operation proceeds through the same training segments as for long train except that the existing equalizer coefficients are retained and the Alternations and Equalizer training segments are shortened. Fax users could use the long train mode for TCF and then use resync for the page data to get better performance in gain-impaired lines. If the TEP_FIELD bit of Tx->mode is disabled (set to zero), then Tx->state is set to Tx_v29_segment1. If this field is enabled (set to one), then Tx->state is set to Tx_v29_TEP and the modulator generates 200 msec. of unmodulated carrier (Talker Echo Protection).

6.19.1 ITU-T Compliance:

The modem is fully compliant with all sections of ITU-T recommendation v.29 except:
Section 5 - no interchange circuits Section 12 - not supported

6.19.2 Applications Programmer Interface Functions and Macros

void Tx_init_v29(struct START_PTRS *) - Configures Tx_block[] for v29 modulation at Tx->rate or 9600 bits/sec. if no valid v.29 rate is selected. If the TEP_FIELD bit of Tx->mode is disabled (set to zero), then Tx->state is set to Tx_v29_segment1. If this field is enabled (set to one), then Tx->state is set to Tx_v29_TEP and the modulator generates 200 msec. of unmodulated carrier (Talker Echo Protection).
Tx_init_v29_9600(struct START_PTRS *) - Macro that configures Tx_block[] for v29 modulation at 9600 bits/sec. without Talker Echo Protection (TEP).

Tx_init_v29_9600_TEP(struct START_PTRS *) - Macro that configures Tx_block[] for v29 modulation at 9600 bits/sec. with Talker Echo Protection (TEP) enabled.
Tx_init_v29_7200(struct START_PTRS *) - Macro that configures Tx_block[] for v29 modulation at 7200 bits/sec. without Talker Echo Protection (TEP).
Tx_init_v29_7200_TEP(struct START_PTRS *) - Macro that configures Tx_block[] for v29 modulation at 7200 bits/sec. with Talker Echo Protection (TEP) enabled.
Tx_init_v29_4800(struct START_PTRS *) - Macro that configures Tx_block[] for v29 modulation at 4800 bits/sec. without Talker Echo Protection (TEP).
Tx_init_v29_4800_TEP(struct START_PTRS *) - Macro that configures Tx_block[] for v29 modulation at 4800 bits/sec. with Talker Echo Protection (TEP) enabled.
enable_Tx_v29_TEP(struct START_PTRS *) - Macro that sets the TX_TEP_FIELD in Tx->mode, thereby enabling TEP, and configures Tx_block[] for v29 modulation at Tx->rate or 9600 bits/sec. if no valid v.29 rate is selected.
disable_Tx_v29_TEP(struct START_PTRS *) - Macro that clears the TX_TEP_FIELD in Tx->mode, thereby disabling TEP, and configures Tx_block[] for v29 modulation at Tx->rate or 9600 bits/sec. if no valid v.29 rate is selected.
enable_Tx_v29_long_train(struct START_PTRS *) - Macro that clears the TX_LONG_RESYNC_FIELD in Tx->mode, thereby enabling v.29 long train mode. Segment 2 is 128 symbols and Segment 3 is 384 symbols in duration as per ITU-T v.29. This is the default mode for ITU-I V.29.
enable_Tx_v29_resync(struct START_PTRS *) - Macro that sets the TX_LONG_RESYNC_FIELD in Tx->mode, thereby enabling v.29 resync or short train mode. Segment 2 is reduced to 64 symbols and Segment 3 is reduced to symbols in duration.
enable_Rx_v29_long_train(struct START_PTRS *) - Macro that clears the RX_LONG_RESYNC_FIELD in Rx->mode, thereby enabling v.29 long train mode. This is the default mode for ITU-I V.29.
enable_Rx_v29_resync(struct START_PTRS *) - Macro that sets the RX_LONG_RESYNC_FIELD in Rx->mode, thereby enabling v.29 resync or short train mode. In this mode, the equalizer coefficients are not reset to zero and the demodulator operates with pre-trained equalizer coefficients. It is essential that the user first train the equalizer coefficients using the long-train mode, and then switch both the modulator and demodulator for resync operation.

6.19.3 Transmitter States (by STATE_ID)

TX_V29_TEP_ID - 200 msec. of unmodulated carrier (2900 Hz).
TX_V29_SEGMENT1_ID - ITU-T v.29 segment 1, silence for 48 symbols.
TX_V29_SEGMENT2_ID - ITU-T v.29 segment 2, alternations for 128 symbols (long train) or 64 symbols (resync).
TX_V29_SEGMENT3_ID - ITU-T v.29 segment 3, equalizer training for 384 symbols (long train) or 64 symbols (resync).
TX_V29_SEGMENT4_ID - ITU-T v.29 segment 4, scrambled ONEs for 48 symbols.
TX_V29_MESSAGE_ID - Data transmission state.
TX_V29_SEGMENT6_ID – 3 symbols of silence to flush the modulator.

6.19.4 Receiver States (by STATE_ID)

RX_V29_TRAIN_AGC_ID - Estimate Rx power and seed agc_gain.
RX_V29_TRAIN_LOOPS_ID - Train carrier, symbol timing, and agc loops.
RX_V29_DETECT_EQ_ID - Search for start of equalizer training sequence.
RX_V29_TRAIN_EQ_ID - Train adaptive equalizer.
RX_V29_SCR1_ID - De-scramble and discard SCR1 sequence.
RX_V29_MESSAGE_ID - Demodulate data symbols and sink to Rx_data[].

6.20 V.32 modem

Simulator source files: v32.c, v32.h.

Texas Instruments source files: v32.asm, v32.inc, v32.h.
Analog Devices source files: v32.dsp, v32.inc, v32.h.
AT&T source files: v32.s, v32.inc, v32.h.

This full duplex echo canceller modem operates at 4.8 and 9.6 kbits/sec. The symbol rate for all data signaling rates is 2400 symbols/sec. The modulation methods are QPSK at 4.8 kbit/s, rectangular 16-QAM at 9.6 kbit/s non-TCM.

6.20.1 ITU-T Compliance:

The modem is fully compliant with all sections of ITU-T recommendation v.32 except:
Section 3 - no specific v.24 interchange circuits, however, circuits 133, 106, 107, 108, and 109 can be derived from accessible modem status.
Section 6 - not supported
Interworking Procedure for Echo Canceling Modems not supported. That is, there is no mechanism to operate as v.26*te*r.
Annex A - Not automatically supported directly. This means that the v32 modem cannot automatically initiate a v22 modem upon timed USB1 detection. However, the presence of the USB1 signal is reported in Rx->status if detected, and the user can fall back to v22bis if desired.

6.20.2 Applications Programmer Interface Functions and Macros

void Tx_init_v32A(struct START_PTRS *) - Configures Tx_block[] for v32 ANSWER side operation starting in the Tx_v32A_AC1 state generating AC reversals, and Rx_block[] for v32 ANSWER side path measurement in the Rx_v32A_detect_AA state. If the V32TCM_MODE bit is set in Tx->mode AND the modem has been built with TCM_9600 ENABLED (in the config.inc configuration file), then TCM is enabled for 9,600 bits/sec. rate.

void Tx_init_v32A_ANS(struct START_PTRS *) - Configures Tx_block[] for v32 ANSWER-side operation starting in the Tx_v32A_ANS state generating the ANS signal. The demodulator is initialized subsequently following the completion of the Tx_v32A_silence state. If the Tx->rate is not 4800 then TCM is enabled.

void Tx_v32A_retrain(struct START_PTRS *) - Configures Tx_block[] to initiate a v32 ANSWER-side retrain request by switching to Tx_v32A_AC1 state.

void Tx_init_v32C(struct START_PTRS *) - Configures Tx_block[] for v32 CALL side operation starting in the Tx_v32A_AA state generating the AA tone, and Rx_block[] for v32 CALL side path measurement in the Rx_v32C_detect_AC state. If the Tx->rate is 9,600 then TCM is enabled.

void Tx_v32C_retrain(struct START_PTRS *) - Configures Tx_block[] to initiate a v32 CALL side retrain request by switching to Tx_v32A_AA state.

#define Tx_v32_enable_special_TRN(struct START_PTRS *) - Sets the V32_SPECIAL_TRAIN_BIT in Tx->mode. This enables the Tx_v32_special_TRN segment to be inserted prior to Tx_v32A_S1 in ANSWER mode, and Tx_v32C_S in CALL mode.

#define Tx_v32_enable_TCM(struct START_PTRS *) - Sets the V32TCM_MODE bit in Tx->mode. The modem can operate at all rates in this mode, but the 9,600 bits/sec. will be TCM.

#define Tx_v32_disable_TCM(struct START_PTRS *) - Resets the V32TCM_MODE bit in Tx->mode and disables TCM operation. The modem can only operate at 4,800 or 9,600 bits/sec. non-TCM in this mode.

#define GSTN_cleardownA(struct START_PTRS *) – Forces the V32A Tx->max_rate variable to zero and causes the modem to either renegotiate (if v.32bis) or retrain. The user can then monitor for Rx->status=GSTN_CLEARDOWN_REQUESTED from the other modem, and then hang up and execute Tx_init_silence().The user can also implement a timeout for the subsequent retrain or renegotiate.

#define GSTN_cleardownC(struct START_PTRS *) – Forces the V32C Tx->max_rate variable to zero and causes the modem to either renegotiate (if v.32bis) or retrain. The user can then monitor for Rx->status=GSTN_CLEARDOWN_REQUESTED from the other modem, and then hang up and execute Tx_init_silence().The user can also implement a timeout for the subsequent retrain or renegotiate.

6.20.3 Transmitter States (by STATE_ID)

TX_V32A_SILENCE1_ID - 1.8 sec. gap preceding ANS
TX_V32A_ANS_ID - 2100 Hz w/phase reversals for 3.3 sec.
TX_V32A_SILENCE2_ID - 75 msec. gap following ANS
TX_V32A_AC1_ID - AC reversals
TX_V32A_CA_ID - CA reversals
TX_V32A_AC2_ID - 2nd AC reversals
TX_V32A_SILENCE3_ID - 16T silence gap preceding S
TX_V32A_SPECIAL_TRN1_ID - special TRN preceding S.
TX_V32A_S1_ID - 1st S reversal sequence.
TX_V32A_SBAR1_ID - 1st inverted S sequence.
TX_V32A_TRN1_ID - 1st TRN sequence
TX_V32A_R1_ID - R1 rate signal repetitions.
TX_V32A_SILENCE4_ID - silence preceding 2nd S.
TX_V32A_S2_ID - 2nd S reversal sequence.
TX_V32A_SBAR2_ID - 2nd inverted S sequence.
TX_V32A_TRN2_ID - 2nd TRN sequence.
TX_V32A_R3_ID - R3 rate signal repetitions.
TX_V32A_E_ID - E rate signal terminator.
TX_V32A_B1_ID - scrambled binary 1 sequence.
TX_V32A_MESSAGE_ID - data transmission state.
TX_V32A_RC_PREAMBLE_ID - rate re-negotiation preamble (AC, CA).
TX_V32A_R4_ID - rate re-negotiation R4 (initiate) rate signal.
TX_V32A_R5_ID - rate re-negotiation R5 (respond) rate signal.

TX_V32C_SILENCE1_ID -1.0 sec. silence preceding AA.
TX_V32C_AA_ID -AA tone.
TX_V32C_CC_ID - CC tone.
TX_V32C_SILENCE2_ID - silence preceding S.
TX_V32C_S_DELAY_ID - S reversals for NT symbols.
TX_V32C_SPECIAL_TRN1_ID - special TRN preceding S.
TX_V32C_S1_ID - S reversals.
TX_V32C_SBAR1_ID - inverted S reversals.
TX_V32C_TRN1_ID - TRN sequence.
TX_V32C_R2_ID - R2 rate signal repetitions.
TX_V32C_E_ID - E rate signal terminator.
TX_V32C_B1_ID - scrambled binary 1 sequence.
TX_V32C_MESSAGE_ID - data transmission state.
TX_V32C_RC_PREAMBLE_ID - rate re-negotiation preamble (AA, CC). rate re-negotiation
TX_V32C_R4_ID - rate re-negotiation R4 (initiate) rate signal.
TX_V32C_R5_ID - rate re-negotiation R5 (respond) rate signal.

6.20.4 Receiver States (by STATE_ID)

RX_V32A_DETECT_AA_ID - Detect presence of AA tone.
RX_V32A_DETECT_AACC_ID - Detect AA-to-CC transition.
RX_V32A_DETECT_CC_END_ID - Detect end of CC tone.
RX_V32A_TRAIN_EC_ID - Train adaptive echo canceller.
RX_V32A_S_DETECT_ID - Detect S, AA, and AC signals.
RX_V32A_TRAIN_LOOPS_ID - Train carrier, symbol timing, and agc loops.
RX_V32A_DETECT_EQ_ID - Search for start of equalizer training sequence.
RX_V32A_TRAIN_EQ_ID - Train adaptive equalizer.
RX_V32A_RATE_ID - Search for valid rate signal or E terminator**.**
RX_V32A_B1_ID - De-scramble and discard SCR1 sequence.

RX_V32A_MESSAGE_ID - Demodulate data symbols and sink to Rx_data[].
RX_V32A_RC_PREAMBLE_ID - Track AACC, search for rate signal.
RX_V32A_R4_ID - Search for E in rate re-negotiation R4 (initiate) signal
RX_V32A_R5_ID - Search for E in rate re-negotiation R5 (respond) signal.

RX_V32C_DETECT_AC_ID - Detect presence of AC reversals.
RX_V32C_DETECT_ACCA_ID - Detect AC-to-CA transition.
RX_V32C_DETECT_CAAC_ID - Detect CA-to-AC transition**.**
RX_V32C_DETECT_AC_END_ID - Detect end of AC reversals.
RX_V32C_TRAIN_EC_ID - Train adaptive echo canceller.
RX_V32C_S_DETECT_ID - Detect S, AA, and AC signals.
RX_V32C_TRAIN_LOOPS_ID - Train carrier, symbol timing, and agc loops.
RX_V32C_DETECT_EQ_ID - Search for start of equalizer training sequence.
RX_V32C_TRAIN_EQ_ID - Train adaptive equalizer.
RX_V32C_RATE_ID - Search for valid rate signal or E terminator**.**
RX_V32C_B1_ID - De-scramble and discard SCR1 sequence.
RX_V32C_MESSAGE_ID - Demodulate data symbols and sink to Rx_data[].
RX_V32C_RC_PREAMBLE_ID - Track ACCA, search for rate signal.
RX_V32C_R4_ID - Search for E in rate re-negotiation R4 (initiate) signal
RX_V32C_R5_ID - Search for E in rate re-negotiation R5 (respond) signal.

6.21 V.32 bis modem

Simulator source files: v32.c, v32.h.
Texas Instruments source files: v32.asm, v32.inc, v32.h.
Analog Devices source files: v32.dsp, v32.inc, v32.h.
AT&T source files: v32.s, v32.inc, v32.h.

This full duplex echo canceller modem operates with trellis-coded modulation at 7.2, 9.6, 12.0, and 14.4 kbit/sec. The symbol rate for all data signaling rates is 2400 symbols/sec. The modulation methods are rectangular 16-QAM at 7.2 kbit/s, modified cross 32-QAM at 9.6 kbit/s, rectangular 64 -QAM at 12 kbit/s, and modified cross 128-QAM at 14.4 kbit/s.
Fallback to v.22bis and v32 auto mode are not supported directly. The user can monitor the Rx->status during a CALL mode connection to determine if the ANSwering modem is a v.22bis or a v32 auto mode modem by checking for V22_USB1_DETECTED and V32_ANS_DETECTED. The duration of ANS can be determined by the user from the Rx->status=V32_ANS_DETECTED condition.

6.21.1 ITU-T Compliance:

The modem is fully compliant with all sections of ITU-T recommendation v.32 bis except:
Section 3 - no specific v.24 interchange circuits, however, circuits 133, 106, 107, 108, and 109 can be derived from accessible modem status.
Annex A - Not automatically supported directly. This means that the v32 modem cannot automatically initiate a v22 modem upon timed USB1 detection. However, the presence of the USB1 signal is reported in Rx->status if detected, and the user can fall back to v22bis as desired by initializing the v.22 modem (i.e. Tx_init_v22). .

6.21.2 Applications Programmer Interface Functions and Macros

void Tx_init_v32bisA(rate, struct START_PTRS *) - Same as for v32, but specifies rate.
void Tx_init_v32bisA_ANS(rate, struct START_PTRS *) - Same as for v32, but specifies rate.
void Tx_v32A_retrain(struct START_PTRS *) - Same as for v32.
void Tx_v32A_renegotiate(struct START_PTRS *) - Configures Tx_block[] to initiate a v32 ANSWER side rate re-negotiation request by switching to Tx_v32A_RC_preamble state.
void Tx_init_v32bisC(rate, struct START_PTRS *) - Same as for v32, but specifies rate.
void Tx_v32C_retrain(struct START_PTRS *) - Same as for v32.

void Tx_v32C_renegotiate(struct START_PTRS *) - Configures Tx_block[] to initiate a v32 CALL side rate re-negotiation request by switching to Tx_v32C_RC_preamble state.
#define Tx_v32_enable_special_TRN(ptrs) - Same as for v32.
#define Tx_init_v32bisA(rate, ptrs) - Sets the Tx->rate as specified and calls Tx_init_v32A().
#define Tx_init_v32bisC(rate, ptrs) - Sets the Tx->rate as specified and calls Tx_init_v32C().
#define get_EC_MSE(ptrs) - Same as for v32.

6.21.3 Transmitter States (by STATE_ID)

Same as for v32.

6.21.4 Receiver States (by STATE_ID)

Same as for v32.

6.22 V.42 Error Control

Simulator source files: v42.c, v42.h.
Texas Instruments source files: v42.c, v42.h.
Analog Devices source files: v42.c, v42.h.
AT&T source files: v42.c, v42.h.

This module implements an HDLC-based protocol for modem error management known as Link Access Procedure for Modems (LAPM). It detects channel bit errors through the use of a cyclic redundancy check (CRC), and corrects for any detected errors by automatic data retransmission.

The V42 module is fully independent from the modem and signal processing modules that use the transmitter() and receiver() interface. Users can call Tx_v42() and Rx_v42() directly, in tandem with transmitter() and receiver(), and the v42 module will generate and process symbol data from the configured modem (usually v.22bis or v.32bis). The interface between the v42 module and the data modem is referred to as the DCE interface. A compile-time option allows the v42 module to create a pair of circular symbol buffers for this interface (Tx_MODEM[] and Rx_MODEM[]), or to make use of existing symbol buffers in the modems (i.e. Tx_data[] and Rx_data[] in vmodem). The interface between the v42 module and the byte-data source/sink, or data terminal equipment, is referred to as the DTE interface. A compile-time option allows the v42 module to create a pair of circular byte data buffers for this interface (Tx_UART[] and Rx_UART[]), or to make use of existing byte-data buffers in the source/sink module (i.e. software UART TxU_data[] and RxU_data[] buffers). A seamless processing stream can be created linking the MESi software UART, v42, and Vmodem data modems without the need for wasteful data buffer copying.

6.22.1 ITU-T Compliance:

The modem is fully compliant with all sections of ITU-T Recommendation v.42 except:
Annex A – not supported.

6.22.2 Applications Programmer Interface Functions and Macros

void Tx_init_v42(struct DH_START_PTRS *) - Configures DH_block[] for v.42 error control transmit-side operation.
void Tx_v42(struct DH_START_PTRS *) – Implements the current v.42 transmit state.
void Rx_init_v42(struct DH_START_PTRS *) - Configures DH_block[] for v.42 error control receive-side operation.
void Rx_v42(struct DH_START_PTRS *) – Implements the current v.42 receive state.

6.23 V.42bis Compression

Simulator source files: v42.c, v42.h, v42bis.c, v42bis.h.
Texas Instruments source files: v42.c, v42.h, v42bis.c, v42bih.h.

Analog Devices source files: v42.c, v42.h, v42bis.c, v42bis.h.
AT&T source files: v42.c, v42.h, v42bis.c, v42bis.h.

This module implements V42bis compression. It can be used in conjunction with V42 Error Control, or as a stand-alone compressor/de-compressor. However, the use of compression without some means of error control is not recommended as errors will result in improper decompression.
For combined V.42 and V.42bis operation, the user interface using is identical to the V42 user interface. The V42 Error Control logical calls the compressor/de-compressor during normal operations. V42bis parameter negotiations are handled by the V42 XID frames during call initialization. There are two bits in the mode register that permit/disallow compression. Compression can be enabled from the initiator to the responder, from the responder to the initiator or in both directions.
To enable V42bis operation in a combined system application, both V42 and V42bis must be compiled with V42BIS defined on the compiler command line. In this case, V42bis compression is an integral part of the V42 error control function and is completely transparent to the user. As data bytes are received in the HDLC frames, they are then de-compressed and placed in the user's UART transmit buffer. Similarly, transmit bytes from the UART receive buffer are first given to the compressor, and the compressors output is placed into HDLC frames. This logic is enabled at build time in V42.
Since V42bis is a table look-up compressor, the size of the tables can be controlled at build time in the event the is insufficient memory for the default size. The V42BIS_MAX_CODEWORD is set to the ITU recommended size of 2048 code words. It can be any number greater than or equal to 512. Setting the size to powers of two maximizes the efficiency of the compression.
Provided that the codeword size is limited to 2048 or smaller, V42BIS_COMPACT can be defined. This packs the codeword, child, parent and sibling links into one 32-bit integer to minimize the memory required. It does this at a very small execution premium and is the recommended method of operation. If however, a codeword size of greater than 2048 is needed, then the compact mode of operation is not allowed. The V42 module is fully independent from the modem and signal processing modules that use the transmitter() and receiver() interface. Users can call Tx_v42() and Rx_v42() directly, in tandem with transmitter() and receiver(), and the v42 module will generate and process symbol data from the configured modem (usually v.22bis or v.32bis). The interface between the v42 module and the data modem is referred to as the DCE interface. A compile-time option allows the v42 module to create a pair of circular symbol buffers for this interface (Tx_MODEM[] and Rx_MODEM[]), or to make use of existing symbol buffers in the modems (i.e. Tx_data[] and Rx_data[] in vmodem). The interface between the v42 module and the byte-data source/sink, or data terminal equipment, is referred to as the DTE interface. A compile-time option allows the v42 module to create a pair of circular byte data buffers for this interface (Tx_UART[] and Rx_UART[]), or to make use of existing byte-data buffers in the source/sink module (i.e. software UART TxU_data[] and RxU_data[] buffers). A seamless processing stream can be created linking the MESi software UART, v42, and Vmodem data modems without the need for wasteful data buffer copying.

6.23.1 ITU-T Compliance:

The modem is fully compliant with all sections of ITU-T Recommendation v.42bis.

6.23.2 Applications Programmer Interface Functions and Macros

void Init_v42bis(struct V42BIS_BLOCK *b,short MaxCodeWordSize, short MaxStringLen) -
Configures DH_block[] for v.42bis compression and decompression operation.
Input: **V42BIS_BLOCK *b -** pointer to V42 Decompression Block
void V42bis_ResetDictionary(struct V42BIS_BLOCK *b, short dictsize) - Initializes the v42bis dictionary to it's base state.
Input: **V42BIS_BLOCK *b -** pointer to V42 Decompression Block
short V42bis_Flush(struct V42BIS_BLOCK *b,unsigned char *OutBuf) - Flushes the compressor buffer, and writes a partial word to the output buffer.
Input: **V42BIS_BLOCK *b -** pointer to V42 Decompression Block
OutBuf * - pointer to array of compressed data
Returns: number of compressed bytes in OutBuf, possibly zero

short V42bis_Compress(struct V42BIS_BLOCK *b, unsigned char *InBuf, short sizeInputBuf, unsigned char *OutBuf)– Implements the current v.42bis compressor state.
Input: **V42BIS_BLOCK *b -** pointer to V42 Decompression Block
Inbuf * - pointer to array of chars to compress
SizeInputBuf - number of bytes to compress
OutBuf * - pointer to array of compressed data
Returns: number of compressed bytes in OutBuf, possibly zero
short V42bis_Decompress(struct V42BIS_BLOCK *b, unsigned char *InBuf, short sizeInputBuf,unsigned char *OutBuf, short sOutbufMax) – Implements the current v.42bis de-compressor state.
Input: **V42BIS_BLOCK *b -** pointer to V42 Decompression Block
Inbuf * - pointer to array of chars to decompress
SizeInputBuf - number of bytes to decompress
OutBuf * - pointer to array of decompressed data
SOutbufMax - max number of bytes to write- extra data lost
Returns: number of decompressed bytes in OutBuf, -1 if data was discarded

6.24 Software UART

Simulator source files: uart.c, uart.h.
Texas Instruments source files: uart.asm, uart.inc.
Analog Devices source files: uart.asm, uart.inc.
AT&T source files: not applicable.
Demo source files: uartdemo.c.

This module implements a software Universal Asynchronous Receiver/Transmitter using one of the DSP serial ports. It can operate at buad rates up to 115,200. In most applications it can derive suitable timing from the CPU clock and no external baud clock is required. The parallel data interface consists of two circular, byte-wide buffers, TxU_data[] and RxU_data[]. The hardware flow control signals RTS and CTS are optionally supported through a variety of DSP pin options or an external I/O port. There are several choices for mapping the CTS and RTS flow control signals since these signals are not synchronous nor are they tightly coupled with the TXD/RXD data conditions.

The C source consists of two functions (Tx_UART and Rx_UART) which implement the UART core, and stubs for the hardware interface functions. The assembly ports include the target interface functions.

The UART is configured for no parity, 8 data bits, and 1 stop bit (N-8-1). The baud rate is build-time selectable. See the uart.inc or uart.h files for available baud rates.

The user APIs and a functional description are presented below.

6.24.1 ITU-T Compliance

Not applicable.

6.24.2 Applications Programmer Interface Functions and Macros

void UART_block_init(struct UART_BLOCK *) - This function initializes the UART_block structure members for default operation. The user MUST call this function before any UART operations are enabled, and it is recommended that it be called immediately after reset or entry into
main(). However, it only needs to be called once.

void init_UART_hardware(void) - This function initializes the serial port peripheral as configured by the compile-time options for TXD/RXD operations. It also initializes, as necessary, the hardware device used for RTS and CTS. Note that the interrupt vector is "hooked" and replaced withg a vector to the

UART_ISR() routine so the interrupt vectors must be write-able.

short Tx_UART(struct UART_BLOCK *) - This function implements the UART transmitter, and it returns the 16 bit integer digital sample stream to be written to the serial interface. The digital sample rate is hardware dependent, but is a minimum of 4*baud rate. In a typical application, this function is called in the serial port interrupt service routine to produce the next 16 bit transmit word. It can also be called in a foreground loop where sample buffering is provided, such as in serial port auto-buffering or TDM-slotted applications.

short Rx_UART(short, struct UART_BLOCK *) - This function implements the UART receiver. The first argument is the 16 bit integer digital sample stream read from the serial interface. The sample rate of this stream is hardware dependent, but is a minimum of 4*baud rate. In a typical application, this function is called in the serial port interrupt service routine to process the current 16 bit receive word. A single ISR can call both Tx_UART() and Rx_UART(). It can also be called in a foreground loop where sample buffering is provided, such as in serial port auto-buffering or TDM-slotted applications.

interrupt void UART_ISR(void) - This function is the serial port interrupt service routine. The minimum interrupt rate is (4*baud rate)/16, depending on the supported hardware. It calls both the Tx_UART() and Rx_UART() routines to process the serial stream bits.

6.24.3 Transmitter States (by STATE_ID)

Not Applicable

6.24.4 Receiver States (by STATE_ID)

Not Applicable

6.25 Common Functions

Simulator source files: common.c, common.h, fsk.c, fsk.h, filter.c, filter.h, tcm.c, tcm.h, vsim.h, plot.c, channel.c.
Texas Instruments source files: common.asm, common.inc, fsk.asm, fsk.inc, filter.asm, filter.inc, tcm.asm, tcm.inc, tcmcoef.inc, tcm.h.
Analog Devices source files: common.dsp, common.inc, tcm.asm, tcm.inc, tcmcoef.inc, tcm.h.
AT&T source files: common.s, common.inc, fsk.s, fsk.inc, filter.s, filter.inc, tcm.s, tcm.inc, , tcm.h.
These modules contain common functions such as filters, generic modulators and demodulators, TCM encoders and decoders, simulator channel model and graphics. Many of the algorithm functions are 'under-the-hood' functions and are not available for general use in the Object Code ports.

6.25.1 Bandpass_filter()

This band-pass filter implementation is a Hilbert band-pass sub-sampling filter constructed as a transversal (i.e. a FIR filter). It returns the magnitude of the filtered signal's real and imaginary parts, which is its Hilbert Transform, and so it can be sub-sampled.

6.25.2 Broadband_estimator()

The broadband_estimator() filter estimates the envelope of the broadband received signal, and returns the magnitude of the envelope. It is approximately equivalent to the square root of the average broadband power level.

6.25.3 Ratio_test()

This function compares the ratio of the two supplied arguments and compares it with the supplied threshold argument. The return value is the result of the comparison and is positive if the ratio is less than the absolute value of the threshold.

6.25.4 Magnitude()

This function returns an approximation of the magnitude, or the square root of the two arguments squared.

6.25.5 Goertzel_bank()

This function executes a bank of Goertzel DFT filters. It is called on a sample-by-sample basis and it calculates the denominator recursion only. When the full DFT block of samples is processed, then the function computes the amplitude square of the complete DFT bins. It returns the value of the Goertzel Filter down counter which decrements to zero upon completion of a block.

6.25.6 FSK_modulator()

This function implements a continuous phase Frequency Shift Keyed modulator using an oscillator. The interpolator and decimator ratio relates the sample rate and the symbol rate, and can be programmed to implement a variety of frequency plans. It includes a sub-sampler that reads 1-bit data from Tx_data[] using Tx->data_tail, and converts the data bits for use by the modulator.

6.25.7 FSK_demodulator()

This function implements a Frequency Shift Keyed demodulator using a pair of Hilbert band-pass sub-sampling filters and ratio detection. The interpolator and decimator ratio relates the sample rate and the symbol rate, and can be programmed to implement a variety of frequency plans. It sinks the received bits to the Rx_data[] buffer using Rx->data_head. It includes a loss of signal (LOS) detector that calculates the difference between the current signal level and the initial signal level. It reports Rx->status=LOSS_OF_LOCK if the signal level drops by 6 dB.

6.25.8 APSK_modulator()

This function implements an Amplitude-Phase Shift Keyed modulator and interpolator/decimator operation to produce 8 kHz samples. It modulates complex symbols in Tx_fir[] using the Tx->fir_tail pointer. By default the symbol clock is derived from the transmit sample clock, but the user can adjust the transmit symbol rate using the Tx->sym_clk_offset variable.

6.25.9 APSK_demodulator()

This function implements an Amplitude-Phase Shift Keyed demodulator and interpolator/decimator operation operating on 8 kHz samples. The algorithm includes: matched filtering, AGC gain level normalization, a complex linear T/2 fractional-spaced adaptive equalizer, Costas PLL for carrier recovery, variable-constellation slicer interface, symbol clock recovery and tracking, mean-square error estimator, and loss-of-lock detector. The received symbol hard-decision slicer, symbol clock timing error estimator, and data decoder operations are all programmable function calls such that APSK_demodulator() can be programmed to demodulate virtually any form of phase-shift and amplitude-shift keyed signals.

6.25.10 AGC_gain_estimator()

This function calculates the appropriate AGC_gain level from Rx->power recursively, and is used to seed the AGC for more rapid convergence.

6.25.11 Rx_fir_autocorrelator()

This function calculates the complex autocorrelation of the received sub-sampled signal and can be used to detect the presence of a phase reversal. It returns the scaled autocorrelation power.

6.26 DSP Demo Functions

Texas Instruments, Analog Devices, AT&T: vmodem.c, vmodem.cmd, vdata.c, vdata2.c, vdata.cmd, vfax.c, vfax.cmd, makefile

These modules contain demo code and linker commands for DSP evaluation boards. The demo code is set up so that users can observe the operation of various components either in loop back or in a connection with a live data modem. Texas Instruments EVM30 require hardware modifications to sample at 8.0 kHz. With the modification to 8 kHz, these boards can be connected with fax/data modems using a 2-to-4-wire converter. It is important to note that all component functions have been and are tested using the demo code and appropriate hardware, and this mechanism is the baseline for all warranty support determinations. Users are strongly urged to start out with the demo code alone and then build upon its successful operation, migrating toward integration with the user's hardware/software environment.

Vfax.c shows how one might use the fax components (v29, v21, v27, v29, gendet) in a loop back configuration or with a fax signal generator. In this scenario, all of the fax modem types are invoked for a data transmission of 10,000 symbols. The data patterns are 0x7e for v21 and all zeros for v29, v27, and v29 thereby emulating fax HDLC messages and TCF training sets under T.30.

Vdata.c demonstrates the v22 and v32 modems interoperating with a data modem (with 8 kHz sampling and 2-to4 wire conversion. The received data is looped back to the transmitter so that one can connect with a data modem and observe typed characters being echoed back.

Vdata2 demonstrates two data modem channels (ANS and CALL) in a digital connection (no analog samples) similar to the Vsim implementation. This shows how multiple channels are set up in "memory2.c", and provides a DSP equivalent so that Vsim-generated test vectors can be evaluated.

7 Warranty Statement

Please see www.mesi.net/license.htm on the Worldwide Web for the End User License Agreement and Warranty. For all program products, this document and the ITU-T recommendations referenced herein constitute the complete and total specification of performance of the products. The vsim.exe DOS simulation program and vmodem.c, vdata.c, and vfax.c demonstration programs included in all MESi deliveries demonstrate full specification compliance when specifically configured and executed on a properly configured or modified (configured or modified to sample at 8.0 kHz) evaluation board supplied by the relevant DSP chip vendor and a 2-to-4 wire telephone line interface. In no instance and under no circumstance shall integrated product operation, on the purchaser's hardware platform or with purchaser's software system, be considered in a warranty determination.

8 Trademarks and Patents

The use of various corporation names, product names, and trademarks in this manual has been done for identification purposes only. No endorsement, accreditation, warranty of fitness, or affiliation with any of these corporations is expressed or implied herein. Specific product and process names used herein are recognized to be registered trademarks of the following corporations:

Analog Devices Corporation
Borland Corporation
DSP Research Corporation
Lucent Technologies Corporation
Microsoft Corporation
Rockwell International Corporation
Texas Instruments Corporation
The ITU-T

Some of the Standards and ITU-T Recommendations referenced herein may contain patented "Intellectual Properties". Users need to be aware that their use of the software Products referenced herein may be construed as an infringement of one or more patents, and must do so at their own risk. MESi does not, by selling, providing, or describing Products herein, make any claims regarding fitness of use, suitability of use, or inability to use the Products; nor does MESi intentionally or indirectly induce any party to infringe upon any applicable patents.

9 Discrepancies and Known Bugs

This section lists the current discrepancies and known bugs in the Vsim simulator and DSP ports. Since these items are known, they are currently being worked on and should be fixed in upcoming releases.

1. V.29 does not auto baud. ITU-T v.29 recommends a modem for leased lines and v29 operation on gain impaired PSTN lines is not reliable (i.e. you won't be able to train properly). Since fax always determines the rate before v.29 transmission, the user will know what rate to set for v29. Therefore, v29 auto baud will not be supported.
2. The far-end echo canceller in v.32 does not compensate for frequency offset in the far-end echo signal. This might occur if one path were to remain in copper and the other go over a satellite link. The Phase II design will include a PLL to be able to track frequency offset.
3. V.26 and V.32 echo canceller modems are not designed to operate over cellular radio links and will require echo canceller re-training in the event of signal loss due to fade or instantaneous phase change due to cell site hand-off.