A Low-Cost Soft Modem Using the Freescale Digital Signal Controller MC56F802x/3x Series Using On-Chip ADC and DAC

by:  John L. Winters
Systems and Applications Engineering
Tempe, Arizona, USA

The hardware and software design of a low cost V.22/V.22bis soft modem is presented. The design does not include a traditional telecommunications PCM CODEC (pulse code modulation coder/decoder), but rather, uses the ADC and DAC of the Freescale 56F802x/3x series to implement a less complex solution. An optional serial port with AT command set is included as a test fixture. Included for reference are modem performance figures measured on the implementation, as well as complete implementation details.
1 System Specification and Design

This section describes the system used to implement the soft modem, as well as the motivation for such a system. Then the system specification is followed by a block diagram and discussion of implementation.

1.1 Soft Modem System Concept

To begin with, it will help to define the term soft modem. A soft modem is one that can be used to modulate/demodulate data to be sent serially over an analog channel directly, without the need for a serial data path to another entity to supply data or control signals. It includes a simple way to dial phone numbers; detect ringing signals; control the hook relay of the DAA; control input and output analog data; connect to remote modems; and communicate with remote modems. Complete control of the modem is embedded within the same host computer performing other system functions, such as alarm monitoring or motion control. A soft modem is the enabling ingredient that allows combining the data terminal equipment (DTE) and data communication equipment (DCE) into one entity.

This differs from traditional communication systems in which the DTE and DCE are two separate parts. Traditionally, the DTE and DCE communicated via a 25-pin serial interface that carries signals such as those described in the V.14, V.25, and V.22bis (and many other) specifications. These signals were used for flow control, and complex signalling between the DTE and DCE as to what respective states they might be in. With the advent of the AT command set in the 1980s many of these signals fell into disuse, and the 25-pin DTE/DCE interface is now more typically found as a 9-pin interface. The difference is made up by providing a complex set of commands and responses that attempt to keep the DTE and DCE in sync. This is depicted in Figure 1.
In the 1990s this interface all but disappeared, when the modem designed to be used only with the most common personal computer operating system was developed. This spoofed the serial communications to the DCE, actually performing much of the modem function on the host processor. Only the digital signal processing was done on another processor, which was closely coupled to the personal computer on its bus. The DTE and DCE continued to be two intelligent processors, but the old serial interface was gone.

When the DTE and DCE were merged into one processor around the turn of the century, the need for such a cable or even bus vanished, as well as the need for a “serial port.” This is in fact how modems in PCs are implemented today. The host processor is powerful enough to implement the modem algorithms and the DCE as a separate intelligent device does not exist. All that is needed is a way to send and receive analog signals over the phone line, and to protect telephone equipment, using the DAA. The serial ports, used only for PC communication software compatibility, are complete spoofs.

Now it is easy for this same elegant architecture, shown in Figure 2, to exist in embedded systems.

In this document we will show how simple such a design is, how few resources of the processor it takes, and how well it performs on USA average lines. This design even omits the standard telecommunications CODEC, using instead a DAC (digital-to-analog convertor) for output and ADC (analog-to-digital converter) for input. Because both of these peripherals are readily available from the many peripherals on one 56F802x/3x series device, along with more than the digital signal processing power required from the single core, the design is a true one-chip, one-core system that includes telecommunications ability while leaving room for much more system functionality.
This approach has many applications in embedded systems. One example would be a security system that consists of a single processor, which is charged with both monitoring local security sensors and communicating directly over phone lines to a central reporting location.

Another application would be a portable medical device, such as a heart monitor, that could periodically interface directly to a phone line without the need for an external modem (or costly modem chip onboard) of any kind. Without the DCE/DTE split such a device would be more reliable, less expensive, and consume less power.

While it would be possible to split the design into a DCE and a DTE, this would add cost due to several factors:

- the additional processor
- the two serial ports required to connect the DCE and DTE
- software in both the DCE and DTE needed for communication between the two
- the interrupts and context saving/restoring that such communication requires

It would also add a potential failure scenario when the DCE and DTE fail to observe each other’s states in a timely manner.

### 1.2 Soft Modem Specification and Design

Figure 3 shows how the basic modem, DTMF, and call progress detection are implemented. Not shown are the DAA control signal, the ring-detect signal, the flow control signal (for testing only), and the serial port (for testing only).

The modem incorporates portions of the following standards as they apply to a soft modem: V.25, V.22, and V.22bis, all implemented on the host digital signal controller (DSC). The portions of these standards related to the DTE/DCE interface are simply not required for a soft modem. DAC and ADC peripherals
on the DSC pass digital samples at a rate of 7200 samples per second (SPS) for the modems and the DTMF generation software. For the call progress software, used to detect dial tone, a sampling rate of 8000 SPS is used. The smaller the sampling rate, the less work for the processor, and subsequently the less power used. The rate of 7200 SPS is ideal for these modems. Fixed rate codecs, commonly used in other solutions, may require the use of sample rate conversion algorithms; these are not needed to interface this modem, because complete control over the codec (which is a soft codec) is intrinsic to the design.

1.3 Test Harness Specification and Design

The test harness for the soft modem application, as depicted in Figure 4, consists of some of the resources of the MC56F8037 and some external test equipment. The DSC and the telco connection to the DSC are the only parts of this figure that are not purely test equipment.

The resources of the MC56F8037 DSC used for testing consist of an asynchronous serial communication port and associated beans and software. This port is used to support an AT command set that is used only for testing — it is not an essential element of the soft modem design. Data and commands are communicated alternately through this test channel. There are two kinds of AT commands supported: online commands and offline commands. The online commands may be issued after the escape sequence puts the test fixture into the online command state. The offline commands may be issued when no connection is in progress.

The online commands supported are:

- ato — return to online data state
- ath — hang up the phone
- atz — hang up the phone and soft reset the modem
- the +++ escape sequence, with three second pre- and post-guard times

The offline commands supported are:

- ata — causes the modem to go off-hook and answer (this command is not required, because the test fixture will automatically answer two seconds after the first ring).
- atd<string> — sets the number to dial string. When this string is set to a non-empty condition, it causes the soft modem to dial that number in the string and attempt a connection. (A production program interfacing to the soft modem would simply set this string to achieve a dialed connection from the soft modem.)
- ati — issues the modem test fixture model and version number.
System Specification and Design

- `atq1` — puts the AT command set into quiet mode, where it will operate without responding.
- `atq0` — undoes the `atq1` command.
- `atz` — performs a soft reset of the soft modem. This hangs up the phone and frees RAM resources.
- `at+0` — puts the modem into V.21 mode (if it is conditioned in), where it will attempt to force all subsequent connections. A modem soft reset is also performed.
- `at+2` — puts the modem into V.22/V.22bis mode, where the modem will attempt to force all subsequent connections. A modem soft reset is also performed.

The online data state of the test fixture is attained with the industry standard escape sequence, which consists of:

- a delay period of three seconds with no traffic on the serial test channel
- three “plus” characters (+++)
- another three seconds of no activity on the serial test channel

Connect messages are issued indicating the line speed. The serial test channel operates at a fixed 2400 baud.

Characters are buffered into the test fixture for transmission and reception to and from the test fixture queues.

![Figure 4. Test Harness for Soft Modem Application](image-url)

The external test equipment consists of two types: one is connected to the RJ11 telco jack of the soft modem, and the other is connected to a USB async serial port on the EVM. The telco jack is connected to equipment that simulates the USA average public switched telephone network (PSTN) connection to an off-the-shelf modem, a Hayes Accura V.92 modem. This PSTN simulation equipment consists of the TAS Series 2 Plus unit and the TAS model 240 unit.

1.3.1 Terminal Emulation One

Terminal emulation one runs HyperTerminal on a PC at 2400 BPS, eight bits data, no parity, one stop bit on a communication port. It runs a binary file transfer using one of the protocols included with
HyperTerminal. It is connected to a Hayes Accura modem where the AT command set is used to directly control the modem.

### 1.3.2 Terminal Emulation Two

Terminal emulation number two is a HyperTerminal session at 9600 baud used to control the TAS Series 2 Plus unit. This unit is programmed to simulate USA average lines.

### 1.3.3 Terminal Emulation Three

Terminal emulation three controls the TAS model 240, which in these tests is used to supply the NULL line. The use of the TAS model 240 is optional, because all functions are available with the series 2 unit.

### 1.3.4 Terminal Emulation Four

Terminal emulation four is HyperTerminal on a PC at 2400 BPS eight bits data, no parity, one stop bit, no flow control. It is the peer of terminal emulation one. It runs the other end of the binary file transfer. The active signals supplied to the serial-to-USB converter on the EVM include tx data, rx data, and no hardware flow control signal.

### 1.4 Hardware Implementation, Setup, and Operation

A standard off-the-shelf Freescale MC56F8037EVM is used for this project. A Freescale Low Cost Modem Daughter Card (LCMDC) is also used, together with a peripheral expansion card to allow easy access to each signal on the LCMDC. The two boards are cross-wired together.

Power is conveyed from the EVM in the form of digital 3.3 V power. It is used for both digital and analog power on the LCMDC. Twisted-pair wiring is used to convey the analog signal from the LCMDC to an ADC input on the MC56F8037EVM, and another twisted pair is used to take one of the DAC outputs from the EVM to the LCMDC card. Ring-indicate and hook-control signals are single wires connecting the LCMDC and the EVM.

Five-volt power is brought from a bench supply to the LCMDC. It is needed only for the DAA circuit. (Other DAA circuits are available that will run on 3.3 V, such as the XE0092 available from Xecom, but the LCMDC card was readily available for this project. A model with the Xecom unit is under construction.)

### 1.4.1 DAC Application to Telephony Baseband Transmission

The DAC is used to develop the analog signal in this project. The DAC updates the voltage output at a rate four times the sample rate of 7200 SPS, or 28800 updates per second. The software only needs to calculate a new delta each 1/7200th second. This allows us to combine the DAC and ADC interrupts into one interrupt, that runs at 1/7200 interrupts per second, on the same clock derived from 3x the IPBUS clock. Other designs might require five times the interrupts per second.

The output of the DAC is a piece-wise approximation of the desired analog signal. Some filtering is needed to limit its bandwidth.
This DAC signal enters from the left of Figure 5 where it is limited to 4 kHz by a 4th order active low-pass filter. The filtered signal is then passed on to the left of Figure 6 for level setting.

The output, XMIT, from Figure 6 is then passed to the XMIT input of a Cermetek CH 1837A DAA whose output level is set to –10 dBm via adjustment of R17. R18 is not populated.

1.4.2 Telephony Baseband Reception to Analog/Digital Convertor

Level adjustment and signal offset calibration are achieved with the circuit shown in Figure 7. The REC signal is from the DAA, the received telephony signal. After level adjustment and offset adjustment, the signal is passed directly to the ADC of the MC56F8037, where it is sampled per the bean configuration.
shown in Figure 13 through Figure 15. The gain of R16 is set to its maximum, allowing communication over null lines at –51dBm at V.22bis speed.

![Figure 7. REC to Offset and Gain to ANA6](image)

Complete schematics for the Low Cost Modem Daughter Card are included in Appendix A, “Low-Cost Modem Daughter Card Schematics.”

## 2 Processor Expert Bean Use in the Soft Modem

Version 8.2 of Metrowerks CodeWarrior for the Freescale DSC family was used to develop all of the code. Much of the code was generated automatically from GUI selections made from simple menus, after the basic system building blocks (software/hardware combinations) were selected. Processor Expert beans encapsulate these basic system blocks.

All of the beans are documented in this section, so that the modem may be constructed directly from these beans together with the code supplied in the next section. (The Processor Expert modem library will also be needed, to supply the modem beans.) A bean has properties, methods, and events. After a bean is added to a project, it must be configured in these three areas. For each bean in the project the complete list of properties, methods, and events is illustrated, along with a commentary on the bean’s usage. When Processor Expert is then called upon to generate code, it will use the configured beans to generate most of the code required for the Soft Modem. The user-written code that must be added to complete the project is documented in 3, “Software Design Details.”

Processor Expert simplified the construction of the software by encapsulating peripheral support into beans. For example, there is a bean for an asynchronous serial port, used to form a test channel over which the AT commands are interfaced to the modem. The properties include such items as baud rate, in this case chosen to be 2400 baud. Another property is used to associate the bean with a particular set of pins on the device. The methods are ways to interface your main program with the peripheral. Code for methods can be drag-and-dropped into your program. Events of the bean facilitate installing user-defined functions to
Processor Expert Bean Use in the Soft Modem

be performed by an interrupt service routine (ISR) used with the bean. The modem data pumps are also added as beans.

The software project is formed by first selecting blank Processor Expert Stationery, adding the beans, and then configuring them. Then the user-written software programs are added as a final step. Pop-up help on the methods for using the beans, along with the drag-and-drop feature, makes using the processor simple.

The bean name (given at the time the bean is added to a project by the designer) is followed by a colon and then the generic bean type. The soft modem software project consists of the beans depicted in Figure 8.

The soft modem has been implemented on several members of the DSC family, including 56F8323, 56F8346, 56F8357, 56F8367, and now on the MC56F8037 as in this project.

![Figure 8. Modem Project Bean List](image)

2.1 CPU Bean

Under CPUs in Figure 8, there is just one CPU listed. The CPU bean has the check mark. This is the one used in the active target, sdm pROM xRAM, or small memory model wherein the constants in data xRAM are initialized from program flash (pROM). The CPU bean has the default settings with the external crystal option. Double-clicking on the CPU bean shows its properties as in Figure 9.

One of the important tabs of the CPU bean is the build-options tab. Note the dynamic memory allocation at the bottom of the build-options list. This allows most of the memory for the soft modem to be deallocated when the soft modem is not in use.
### Processor Expert Bean Use in the Soft Modem

<table>
<thead>
<tr>
<th>Bean name</th>
<th>Cou</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU type</td>
<td>56F8037</td>
</tr>
<tr>
<td>Oscillator frequency [MHz]</td>
<td>8</td>
</tr>
<tr>
<td>Clock source</td>
<td>External crystal</td>
</tr>
<tr>
<td>Initialization priority</td>
<td>minimal priority</td>
</tr>
<tr>
<td>Saturation mode</td>
<td>Disabled</td>
</tr>
<tr>
<td>Initialize shadow registers</td>
<td>no</td>
</tr>
<tr>
<td>Initialize unused I/O pins</td>
<td>no initialization</td>
</tr>
</tbody>
</table>

#### Internal peripherals

- **SIM module**
  - Wait disable mode: Enabled
  - Stop disable mode: Enabled
  - OnCE clock to core: Enabled when core TAP is enabled
  - CLK0 select: 32 kHz system clock
  - CLKOUT mode: Disabled
- **Flash security & protection**: Disabled
- **Peripheral clocks**
- **Peripheral clock rates**
- **I/O module**
- **CPU interrupts**

#### Enabled speed modes

- **High speed mode**: Enabled
  - System clock (IP Bus): 32 MHz
- **PLL clock**: Enabled
  - PLL clock frequency: 132 MHz
  - Enable PLL after reset: yes
- **Low speed mode**: Disabled
- **Slow speed mode**: Disabled
2.2 MEM1: DSP_MEM Bean

This bean defines internal xRAM for use by the other DSP-related beans. Internal memory is much faster than external memory. The most notable feature on this configuration is the _x_Dynmem at the bottom of Figure 10. This must be present to provide dynamic memory services for the soft modem. Because the modem is not the only function in a real application, it is good for it to give back all of its RAM memory when it is not in operation. This can be most easily accomplished with the help of the dynamic memory services provided by Processor Expert.
<table>
<thead>
<tr>
<th>Memory Area</th>
<th>Name</th>
<th>Address</th>
<th>Size</th>
<th>Qualifier</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemoryArea0</td>
<td>.p_Interrupts</td>
<td>0</td>
<td>80</td>
<td>RWX</td>
</tr>
<tr>
<td>MemoryArea1</td>
<td>.p_Code</td>
<td>80</td>
<td>FF7F</td>
<td>RWX</td>
</tr>
<tr>
<td>MemoryArea2</td>
<td>.x_Data</td>
<td>1</td>
<td>FF7F</td>
<td>RW'</td>
</tr>
<tr>
<td>MemoryArea3</td>
<td>.x_DynMem</td>
<td>300</td>
<td>INTERNAL_DYNAMIC</td>
<td>RW'</td>
</tr>
</tbody>
</table>

Figure 10. MEM1:DSP_MEM Properties
These are the various methods available for the DSP algorithms to allocate memory. A major advantage of having so many classes of memory available is that it is possible to allow the algorithm to specify the arrangement of operands in memory, which minimizes wait states.

### 2.3 ADC1:ADC Bean

Note the InputTimer is used to trigger the ADC conversion. After it is triggered, it takes eight rapid fire samples. The fifth sample is used. A special low-level bean is used just to initialize the ADC peripheral. This results in more efficient code, especially because no events are used and the ISR is directed to the user code with no intermediate processing. Also, the only method is to initialize.

The fifth sample is used because it was observed to have the least noise in this application. Averaging the eight samples resulted in more noise, resulting in poor mode performance.
### Processor Expert Bean Use in the Soft Modem

#### A Low-Cost Soft Modem Using the Freescale Digital Signal Controller MC56F802x/3x Series, Rev. 0

<table>
<thead>
<tr>
<th>Bean name</th>
<th>ADC_Analog_RX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device</td>
<td>ADC</td>
</tr>
</tbody>
</table>

#### Settings

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock setting</td>
<td></td>
</tr>
<tr>
<td>Clock divisor select value</td>
<td>3</td>
</tr>
<tr>
<td>A/D Frequency</td>
<td>5.33333333333333 MHz</td>
</tr>
<tr>
<td>Stop mode (0)</td>
<td>no</td>
</tr>
<tr>
<td>Stop mode 1</td>
<td>yes</td>
</tr>
<tr>
<td>ADC mode</td>
<td>Triggered Sequential</td>
</tr>
<tr>
<td>Parallel mode</td>
<td>simultaneous</td>
</tr>
<tr>
<td>Trigger mode (0)</td>
<td>SYNC input or START bit</td>
</tr>
<tr>
<td>Trigger mode 1</td>
<td>SYNC input or START bit</td>
</tr>
<tr>
<td>Power savings mode</td>
<td>Disabled</td>
</tr>
<tr>
<td>Power-up delay</td>
<td>13</td>
</tr>
<tr>
<td>Auto standby</td>
<td>Disabled</td>
</tr>
<tr>
<td>High volt. ref. source 0</td>
<td>external</td>
</tr>
<tr>
<td>Low volt. ref. source 0</td>
<td>internal</td>
</tr>
<tr>
<td>High volt. ref. source 1</td>
<td>internal</td>
</tr>
<tr>
<td>Low volt. ref. source 1</td>
<td>internal</td>
</tr>
</tbody>
</table>

#### A/D Channels

<table>
<thead>
<tr>
<th>Channel</th>
<th>Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANA0-ANA1</td>
<td>Single ended mode</td>
</tr>
<tr>
<td>ANA2-ANA3</td>
<td>Single ended mode</td>
</tr>
<tr>
<td>ANB0-ANB1</td>
<td>Single ended mode</td>
</tr>
<tr>
<td>ANB2-ANB3</td>
<td>Single ended mode</td>
</tr>
<tr>
<td>ANA4-ANA5</td>
<td>Single ended mode</td>
</tr>
<tr>
<td>ANA6-ANA7</td>
<td>Single ended mode</td>
</tr>
<tr>
<td>ANB4-ANB5</td>
<td>Single ended mode</td>
</tr>
<tr>
<td>ANB6-ANB7</td>
<td>Single ended mode</td>
</tr>
</tbody>
</table>
## Processor Expert Bean Use in the Soft Modem

### Sample 0
- Enabled
- Channel: GPIOC13_ANB5

### Sample 1
- Enabled
- Channel: GPIOC13_ANB5

### Sample 2
- Enabled
- Channel: GPIOC13_ANB5

### Sample 3
- Enabled
- Channel: GPIOC13_ANB5

### Sample 4
- Enabled
- Channel: GPIOC13_ANB5

### Sample 5
- Enabled
- Channel: GPIOC13_ANB5

### Sample 6
- Enabled
- Channel: GPIOC13_ANB5

### Sample 7
- Enabled
- Channel: GPIOC13_ANB5

### Sample 8
- Disabled

### Sample 9
- Disabled

### Sample 10
- Disabled

### Sample 11
- Disabled

### Sample 12
- Disabled

### Sample 13
- Disabled

### Sample 14
- Disabled

### Sample 15
- Disabled

### Zero crossing
- Sample 0: disabled
- Sample 1: disabled
- Sample 2: disabled
- Sample 3: disabled
- Sample 4: disabled
- Sample 5: disabled
- Sample 6: disabled
- Sample 7: disabled

### Interrupts
#### Conversion complete (0)
- Interrupt: INT_ADCA_Complete
- End of scan interrupt: Enabled
- Interrupt priority: medium priority
- ISR name: ADC_EOS_INT

#### Conversion complete 1
- Interrupt: INT_ADCB_Complete
- End of scan interrupt: Disabled
- Interrupt priority: medium priority
- ISR name:
### Registers

<table>
<thead>
<tr>
<th>Low Limit registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low limit register 0</td>
</tr>
<tr>
<td>Low limit register 1</td>
</tr>
<tr>
<td>Low limit register 2</td>
</tr>
<tr>
<td>Low limit register 3</td>
</tr>
<tr>
<td>Low limit register 4</td>
</tr>
<tr>
<td>Low limit register 5</td>
</tr>
<tr>
<td>Low limit register 6</td>
</tr>
<tr>
<td>Low limit register 7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>High Limit registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>High limit register 0</td>
</tr>
<tr>
<td>High limit register 1</td>
</tr>
<tr>
<td>High limit register 2</td>
</tr>
<tr>
<td>High limit register 3</td>
</tr>
<tr>
<td>High limit register 4</td>
</tr>
<tr>
<td>High limit register 5</td>
</tr>
<tr>
<td>High limit register 6</td>
</tr>
<tr>
<td>High limit register 7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Offset registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offset register 0</td>
</tr>
<tr>
<td>Offset register 1</td>
</tr>
</tbody>
</table>

**Figure 13. AD1:ADC Properties (Four Parts)**

![Bean Inspector ADC_Analog_RX:Init_ADC](image)

**Figure 14. AD1:ADC Methods**

![Bean Inspector ADC_Analog_RX:Init_ADC](image)

**Figure 15. AD1:ADC Events**

### 2.4 DAC_Step_Clock:Init_TMR Bean

This clock runs at a frequency of 4 times 7200, or 28800 cycles per second. The V.22bis implementation uses 7200 samples per second, for both input and output processing.
This is the bean that supplies the clock signal that causes the DAC to increment or decrement by its assigned delta step. This results in the signal construction of the TX modem signal. This is the signal that is next taken by the low-pass filter, then passed to the DAA and on to the telephone circuit.

The frequency of this clock is four times the sample rate, because it has been shown in the previous modem project that a linear interpolation into four line segments is enough to construct a good modem signal for the low-pass filter to process. Thus, between each of the 7200 samples per second, four equal steps are taken by the DAC to connect between two adjacent output samples.

The advantage of this is that the DAC can do this on its own, and needs to be adjusted with a new delta only once each sample period. This clock is divided down by four to produce the sample rate clock. Because the DAC and ADC are running off the same time base, it is possible to use one ISR to handle both the ADC and the DAC function, once per 7200 samples per second.

So, once each sample time, a new delta is calculated for the DAC, and the ADC sample is obtained, both in the same ISR.

The new delta is calculated based on where the DAC will end up from the previous application four times of the previous delta, versus where the DAC needs to go. This process keeps the DAC tracking the desired signal. The full range of the DAC is used.

Note that the time base for this clock is three times the IPBus clock rate. These frequencies allow the system to run at its maximum rate of 32 MHz.

This bean has a pin output for diagnostic purposes only. Also, it is not required to run any function at the rate of this timer, because only the DAC needs to step its output when the timer runs down. The output of this timer feeds the sample rate timer.
### Processor Expert Bean Use in the Soft Modem

<table>
<thead>
<tr>
<th>Bean name</th>
<th>DAC_Step_Clock</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device</td>
<td>TMRA1</td>
</tr>
<tr>
<td><strong>Settings</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Clock settings</strong></td>
<td></td>
</tr>
<tr>
<td>TMRA clock rate</td>
<td>TMRA_3x_System_clock</td>
</tr>
<tr>
<td><strong>Primary source</strong></td>
<td></td>
</tr>
<tr>
<td>Primary source</td>
<td>prescaler (IP BUS clock)</td>
</tr>
<tr>
<td><strong>Secondary source</strong></td>
<td></td>
</tr>
<tr>
<td>Secondary source</td>
<td>counter 0 input pin</td>
</tr>
<tr>
<td><strong>Operation mode</strong></td>
<td></td>
</tr>
<tr>
<td>Operation mode</td>
<td>Count mode</td>
</tr>
<tr>
<td>Count once</td>
<td>count repeatedly</td>
</tr>
<tr>
<td>Count length</td>
<td>count till compare, then reinitialize</td>
</tr>
<tr>
<td>Master mode</td>
<td>Enabled</td>
</tr>
<tr>
<td>External OFLAG force</td>
<td>Disabled</td>
</tr>
<tr>
<td>Forced OFLAG value</td>
<td>0</td>
</tr>
<tr>
<td>Force OFLAG output</td>
<td>no</td>
</tr>
<tr>
<td>Output enable</td>
<td>yes</td>
</tr>
<tr>
<td>Output polarity</td>
<td>true</td>
</tr>
<tr>
<td>Input polarity</td>
<td>true</td>
</tr>
<tr>
<td>Co-channel initialization</td>
<td>Disabled</td>
</tr>
<tr>
<td><strong>Input capture mode</strong></td>
<td></td>
</tr>
<tr>
<td>Input capture mode</td>
<td>Disabled</td>
</tr>
<tr>
<td>Load capture register</td>
<td>on rising edge of input</td>
</tr>
<tr>
<td>OutputMode</td>
<td>toggle OFLAG output on successful cor</td>
</tr>
<tr>
<td><strong>Compare load control 1</strong></td>
<td></td>
</tr>
<tr>
<td>Load upon successful compare</td>
<td>with the value in TMRx_CMP1</td>
</tr>
<tr>
<td><strong>Compare load control 2</strong></td>
<td></td>
</tr>
<tr>
<td>Debug mode action</td>
<td>Halt TMR counter</td>
</tr>
<tr>
<td><strong>Input filter</strong></td>
<td>Disabled</td>
</tr>
<tr>
<td><strong>TMR pin</strong></td>
<td></td>
</tr>
<tr>
<td>Pin name</td>
<td>GPIOA12_TBI1_SCLK1_TA1</td>
</tr>
<tr>
<td>Pin direction</td>
<td>Output</td>
</tr>
<tr>
<td>Pin signal</td>
<td></td>
</tr>
</tbody>
</table>
2.5 IV1:InterruptVector Bean

The misaligned LongWordISR is supplied only to detect a misaligned LongWord interrupt. This does not occur in the project as provided, but is useful to illustrate how to hook an ISR. Also, should any bugs be introduced into the project, they may be trapped, in some cases, by this ISR.
This bean does not generate any place to put event code in Events.c. Instead, it is up to the programmer to locate his ISR code.

2.6 Mdm1:DSP_v22bis Bean

This software bean generates the modem code for the V22bis/V22 modem data pump. Most of the code is written in optimized assembly language. The API in C is also generated. The memory management makes it possible to set up the modem on a per-call basis. When no call is active, no modem is taking RAM resources. That is why the Memory Management feature is used.

Rather than the AT command set, the API of this bean is used to control the V.22bis modem.
The v22bisControl method is for future expansion of the data pump. The other methods used are illustrated in the next section. They are also documented as part of Processor Expert documentation included with the IDE.

No events are needed. The modem is called. When it has data or needs data, it calls the application program back at a designated function to deliver or collect data. Because no event is needed, there is no context switch and the code is more efficient. The modem code operation is gated by the collection of samples for the receiver.

When twelve samples are available, the modem code must be called (or task released if using OS).

The codec (ADC and DAC) code is designed so that it will not have to wait to transmit samples, only for the reception. Because the DAC and ADC are in sync, this is not only possible, but simple to achieve.

It is also arranged in such a way that only 12 samples are obtained, and no more, prior to calling the modem code. This assures that any handshaking between modems is completed in a timely manner, assuring robust connection.

The modem bean code itself is very robust, adapting quickly to changing line signal levels, for example. The main thrust of this note is to show how best to use this modem bean.

2.7 **SampleRateTimer:Init_TMR Bean**

This bean is responsible for timing the ADC sample reads. It normally operates at 7200 samples per second. Using the PESL commands, the peripheral is modified to sample at 8000 samples per second during call progress. This dynamic sampling rate avoids the use of any sample rate conversion code.
<table>
<thead>
<tr>
<th>Feature</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bean name</strong></td>
<td>SampleRateTimer</td>
</tr>
<tr>
<td><strong>Device</strong></td>
<td>TMRA3</td>
</tr>
<tr>
<td><strong>Settings</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Clock settings</strong></td>
<td></td>
</tr>
<tr>
<td>TMRA clock rate</td>
<td>TMRA_3x_System_clock</td>
</tr>
<tr>
<td><strong>Primary source</strong></td>
<td></td>
</tr>
<tr>
<td>Primary source</td>
<td>counter 1 output pin</td>
</tr>
<tr>
<td><strong>Secondary source</strong></td>
<td></td>
</tr>
<tr>
<td>Secondary source</td>
<td>counter 0 input pin</td>
</tr>
<tr>
<td><strong>Operation mode</strong></td>
<td>Count mode</td>
</tr>
<tr>
<td>Count once</td>
<td>count repeatedly</td>
</tr>
<tr>
<td>Count length</td>
<td>count till compare, then reinitialize</td>
</tr>
<tr>
<td>Count direction</td>
<td>up</td>
</tr>
<tr>
<td>Master mode</td>
<td>Disabled</td>
</tr>
<tr>
<td>External OFLAG force</td>
<td>Disabled</td>
</tr>
<tr>
<td>Forced OFLAG value</td>
<td>0</td>
</tr>
<tr>
<td>Force OFLAG output</td>
<td>no</td>
</tr>
<tr>
<td>Output enable</td>
<td>yes</td>
</tr>
<tr>
<td>Output polarity</td>
<td>true</td>
</tr>
<tr>
<td>Input polarity</td>
<td>true</td>
</tr>
<tr>
<td>Co-channel initialization</td>
<td>Disabled</td>
</tr>
<tr>
<td><strong>Input capture mode</strong></td>
<td>Disabled</td>
</tr>
<tr>
<td>OutputMode</td>
<td>toggle OFLAG output on successful</td>
</tr>
<tr>
<td><strong>Compare load control 1</strong></td>
<td>Enabled</td>
</tr>
<tr>
<td>Load upon successful compare</td>
<td>with the value in TMRx_CMP1</td>
</tr>
<tr>
<td><strong>Compare load control 2</strong></td>
<td>Disabled</td>
</tr>
<tr>
<td>Debug mode action</td>
<td>Halt TMR counter</td>
</tr>
<tr>
<td><strong>Input filter</strong></td>
<td>Disabled</td>
</tr>
<tr>
<td><strong>Pin</strong></td>
<td></td>
</tr>
<tr>
<td>Pin</td>
<td>1</td>
</tr>
<tr>
<td><strong>TMR pin</strong></td>
<td></td>
</tr>
<tr>
<td>Pin name</td>
<td>GPIOB3_MOSI0_TA3_PSRC1</td>
</tr>
<tr>
<td>Pin direction</td>
<td>Output</td>
</tr>
<tr>
<td>Pin signal</td>
<td></td>
</tr>
</tbody>
</table>
To alter the sampling rate without having to execute a stop or a reset, the Timer Compare register 1 is changed to a smaller number on the fly, achieving 8000 samples per second with PESL commands during call progress detection. Be sure to enable the PESL commands in the project, because the soft modem uses these. The compiler will report no prototype if this is not enabled.

**Figure 25. SampleRateTimer:Init_TMR Properties (Two Parts)**

---

2.8 **TestHarnessDCE:AsynchroSerial Bean**

This bean was set up for 2400 baud. It actually does not matter what baud rate is used as long as it is at least 2400 baud. The buffering is done in the modem. One of the SCI ports on the device is used for this purpose. It connects to a chip that converts this signal to USB, so that computers without serial ports may
easily interface (and power the EVM) via USB cable here. Such a device is transparent to applications, other than that a special device driver is needed on the PC (acting as the DTE) to allow a terminal emulation program to assign a virtual device driver to this phantom port presented on the USB. On my computer it showed up as COMM 5, but results may vary on different PC configurations.

Remember, it is intended that this interface be removed for the actual application (such as calling a pre-programmed number to report a voltage change on a sensor). The embedded modem may be accessed directly with the APIs illustrated here, rather than resort to an external DTE.

<table>
<thead>
<tr>
<th>Bean name</th>
<th>TestHarnessDCE</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel</td>
<td>QSC10</td>
<td>QSC10</td>
</tr>
<tr>
<td><strong>Interrupt service/event</strong></td>
<td>Enabled</td>
<td></td>
</tr>
<tr>
<td>Interrupt RxD</td>
<td>INT_QSC10_RxFull</td>
<td>INT_QSC10_RxFull</td>
</tr>
<tr>
<td>Interrupt RxD priority</td>
<td>medium priority</td>
<td>1</td>
</tr>
<tr>
<td>Interrupt RxD preserve registers</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Interrupt TxD</td>
<td>INT_QSC10_TxEmpty</td>
<td>INT_QSC10_TxEmpty</td>
</tr>
<tr>
<td>Interrupt TxD priority</td>
<td>medium priority</td>
<td>1</td>
</tr>
<tr>
<td>Interrupt TxD preserve registers</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Interrupt Error</td>
<td>INT_QSC10_Error</td>
<td>INT_QSC10_Error</td>
</tr>
<tr>
<td>Interrupt Error priority</td>
<td>medium priority</td>
<td>1</td>
</tr>
<tr>
<td>Interrupt Error preserve registers</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Interrupt Idle</td>
<td>INT_QSC10_Idle</td>
<td>INT_QSC10_Idle</td>
</tr>
<tr>
<td>Interrupt Idle priority</td>
<td>medium priority</td>
<td>1</td>
</tr>
<tr>
<td>Interrupt Idle preserve registers</td>
<td>yes</td>
<td></td>
</tr>
<tr>
<td>Input buffer size</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>Output buffer size</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td><strong>Handshake</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CTS</td>
<td>Disabled</td>
<td></td>
</tr>
<tr>
<td>RTS</td>
<td>Disabled</td>
<td></td>
</tr>
<tr>
<td><strong>Settings</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parity</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>Width</td>
<td>8 bits</td>
<td>8 bits</td>
</tr>
<tr>
<td>Stop bit</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>SCI output mode</td>
<td>Normal</td>
<td></td>
</tr>
<tr>
<td>LIN slave mode</td>
<td>Disabled</td>
<td></td>
</tr>
<tr>
<td>Receiver</td>
<td>Enabled</td>
<td></td>
</tr>
</tbody>
</table>
Figure 28. TestHarnessDCE:AsynchroSerial Properties (Two Parts)

Notice that block, not character, input/output is used. All interrupts are at the same priority throughout the project.

Figure 29. TestHarnessDCE:AsynchroSerial Methods
To send a block of characters to the serial port, use SendBlock. To receive a block of characters, use RecvBlock.

These methods were simply drag-and-dropped into the application code as required.

---

<table>
<thead>
<tr>
<th>Event module name</th>
<th>Events</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>BeforeNewSpeed</strong></td>
<td>don’t generate code</td>
</tr>
<tr>
<td><strong>AfterNewSpeed</strong></td>
<td>don’t generate code</td>
</tr>
<tr>
<td><strong>OnError</strong></td>
<td>generate code</td>
</tr>
<tr>
<td><strong>OnRxChar</strong></td>
<td>generate code</td>
</tr>
<tr>
<td><strong>OnRxCharExt</strong></td>
<td>don’t generate code</td>
</tr>
<tr>
<td><strong>OnTxChar</strong></td>
<td>generate code</td>
</tr>
<tr>
<td><strong>OnFullRxBuf</strong></td>
<td>generate code</td>
</tr>
<tr>
<td><strong>OnFreeTxBuf</strong></td>
<td>generate code</td>
</tr>
<tr>
<td><strong>OnBreak</strong></td>
<td>don’t generate code</td>
</tr>
<tr>
<td><strong>OnTxComplete</strong></td>
<td>don’t generate code</td>
</tr>
</tbody>
</table>

**Figure 30. TestHarnessDCE:AsynchroSerial Events**

### 2.9 **RingDetect: PulseAccumulator Bean**

The ring detect uses timer hardware to pre-process the ring signal. This reduces the MIP load when processing it, and filters out spurious ring signals.
## Processor Expert Bean Use in the Soft Modem

<table>
<thead>
<tr>
<th>Bean name</th>
<th>RingDetect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Counter</td>
<td>TMRA0_PACNT</td>
</tr>
<tr>
<td>Interrupt service/event</td>
<td>Enabled</td>
</tr>
<tr>
<td>Interrupt</td>
<td>INT_TMRA0</td>
</tr>
<tr>
<td>Interrupt priority</td>
<td>low priority</td>
</tr>
<tr>
<td>Interrupt preserve registers</td>
<td>yes</td>
</tr>
<tr>
<td>Mode</td>
<td>Count</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input filter</th>
<th>Disabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sample count</td>
<td>10</td>
</tr>
<tr>
<td>Sample period</td>
<td>01</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Primary input</th>
<th>GPIOA6_FAULT0_TA0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input pin signal</td>
<td>RingIndicateUnconditioned</td>
</tr>
<tr>
<td>Pull resistor</td>
<td>pull up</td>
</tr>
<tr>
<td>Edge</td>
<td>rising edge</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Input source</th>
<th>external</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Initialization</th>
<th>Bean is not selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enabled in init. code</td>
<td>no</td>
</tr>
<tr>
<td>Events enabled in init.</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Figure 31. RingDetect:PulseAccumulator Properties**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable</td>
<td>generate code</td>
</tr>
<tr>
<td>Disable</td>
<td>generate code</td>
</tr>
<tr>
<td>EnableEvent</td>
<td>don't generate code</td>
</tr>
<tr>
<td>DisableEvent</td>
<td>don't generate code</td>
</tr>
<tr>
<td>ResetCounter</td>
<td>generate code</td>
</tr>
<tr>
<td>SetCounter</td>
<td>don't generate code</td>
</tr>
<tr>
<td>GetCounterValue</td>
<td>generate code</td>
</tr>
<tr>
<td>SetCompare1</td>
<td>don't generate code</td>
</tr>
<tr>
<td>SetCompare2</td>
<td>don't generate code</td>
</tr>
<tr>
<td>GetCompare1Value</td>
<td>don't generate code</td>
</tr>
<tr>
<td>GetCompare2Value</td>
<td>don't generate code</td>
</tr>
<tr>
<td>ConnectPin</td>
<td>don't generate code</td>
</tr>
</tbody>
</table>

**Figure 32. RingDetect:PulseAccumulator Methods**

<table>
<thead>
<tr>
<th>Event name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BeforeNewSpeed</td>
<td>don't generate code</td>
</tr>
<tr>
<td>AfterNewSpeed</td>
<td>don't generate code</td>
</tr>
<tr>
<td>OnOverflow</td>
<td>don't generate code</td>
</tr>
<tr>
<td>OnEnd</td>
<td>don't generate code</td>
</tr>
<tr>
<td>OnCompare1</td>
<td>don't generate code</td>
</tr>
<tr>
<td>OnCompare2</td>
<td>don't generate code</td>
</tr>
</tbody>
</table>

**Figure 33. RingDetect:PulseAccumulator Events**
2.10 OffHook:BitIO Bean

This bitIO is used to control the on-hook/off-hook status of the DAA. Never take the phone from the on-hook to the off-hook condition while the phone is ringing, as this may damage the DAA circuits. The auto-answer feature of the soft modem assures that this will not happen.

<table>
<thead>
<tr>
<th>Bean name</th>
<th>OffHook</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pin for I/O</td>
<td>GPIOA10_CINA2_T82</td>
</tr>
<tr>
<td>Pin signal</td>
<td>OFFHK</td>
</tr>
<tr>
<td>Pull resistor</td>
<td>no pull resistor</td>
</tr>
<tr>
<td>Open drain</td>
<td>no open drain</td>
</tr>
<tr>
<td>Drive strength for</td>
<td>GPIOA10</td>
</tr>
<tr>
<td>Direction</td>
<td>Output</td>
</tr>
</tbody>
</table>

**Initialization**
- Init. direction: Output
- Init. value: 0
- Sale mode: yes
- Optimization for code size

**Figure 34. OffHook:BitIO Properties**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetDir</td>
<td>don't generate code</td>
</tr>
<tr>
<td>SetDir</td>
<td>don't generate code</td>
</tr>
<tr>
<td>Setinput</td>
<td>don't generate code</td>
</tr>
<tr>
<td>SetOutput</td>
<td>don't generate code</td>
</tr>
<tr>
<td>GetVal</td>
<td>don't generate code</td>
</tr>
<tr>
<td>PutVal</td>
<td>don't generate code</td>
</tr>
<tr>
<td>ClrVal</td>
<td>generate code</td>
</tr>
<tr>
<td>SetVal</td>
<td>generate code</td>
</tr>
<tr>
<td>NegVal</td>
<td>don't generate code</td>
</tr>
<tr>
<td>ConnectPin</td>
<td>don't generate code</td>
</tr>
<tr>
<td>GetRawVal</td>
<td>don't generate code</td>
</tr>
</tbody>
</table>

**Figure 35. OffHook:BitIO Methods**

ClrVal and SetVal are methods used to take the modem on and off the hook. When on hook, the modem is not connected to the phone line. When off hook, it is connected. This term goes back to the antique telephone design.

**Figure 36. OffHook:BitIO Events**

This bean does not contain any event.
2.11 TwentythSecInt:TimerInt Bean

This bean is used to generate an interrupt once every twentieth second for the ring-detection and AT command set state machines. Also, the AT command set needs to run a three-second timer when it is online, connected to another modem, to prevent data of +++ being construed as a request to enter the online command state.

![Figure 37. TwentythSecInt:TimerInt Properties](image)

![Figure 38. TwentythSecInt:TimerInt Methods](image)
2.12 TEL1: CallProgressToneDetection Bean

This is a software bean that generates the Call Progress Tone Detection code used to detect dial tone. Note the available methods. When they are moused over in the project window, each method’s documentation pops up.

2.13 TEL2: DTMFGenerate Bean

This bean generates the software to generate the samples needed for DTMF dialing. This bean supports both 7200 and 8000 samples per second. The soft modem uses 7200 SPS. This consumes fewer cycles and less energy. Also, no sample rate convertor is needed to interface to the codec. This is really a soft codec included with the MC56F8037 in the form of its ADC and DAC, and their timing may be reconfigured on the fly.

The resources of this bean are used (RAM allocated) only while the bean is in use, due to the use of Memory Management.
2.14 Mdm2 DSP_v21 Modem Software Bean

One of the two modem beans available for this project, Mdm2, generates the V.21 modem data pump code. V.21 is not the subject of this note, but the code is conditionally assembled into the project.

2.15 CTS1:56F8367EVM_LED_Yellow2 Bean

This Clear-to-Send (of hardware flow control application) bean takes a bit IO and employs it for the Clear-to-Send signal to the USB device on the EVM, which is not pinned out to this signal.

It is included as a hook for the user who may want to use this signal.

Without this hardware flow control mechanism, the PC could overrun the modem during binary file transfers using XMODEM, or YMODEM, or ZMODEM binary file transfer protocols with HyperTerminal.
Processor Expert Bean Use in the Soft Modem

A Low-Cost Soft Modem Using the Freescale Digital Signal Controller MC56F802x/3x Series, Rev. 0

Freescale Semiconductor

33

2.16 DAC_Analog_TX Bean

The DAC can generate user-defined waveforms. We define the waveform to be the V.22bis TX signal. To do this, we have to keep giving the updated delta to the DAC at exactly the required time.

Note the right angle bracket on the Output PIN on the top element of Figure 46. This indicates that this bean encapsulates another bean. By clicking on the right angle bracket, another screen is presented. That is shown on the second part (of two) of Figure 46 and shows the actual pin used for the Clear-to-Send signal which is sent to the PC serial port.

This on initial value actually means that flow is restricted initially.

The on method would stop the flow from the PC, whereas the off method would allow it to resume.

The DAC can generate user-defined waveforms. We define the waveform to be the V.22bis TX signal. To do this, we have to keep giving the updated delta to the DAC at exactly the required time.

This on initial value actually means that flow is restricted initially.

The on method would stop the flow from the PC, whereas the off method would allow it to resume.

The DAC can generate user-defined waveforms. We define the waveform to be the V.22bis TX signal. To do this, we have to keep giving the updated delta to the DAC at exactly the required time.
Figure 49. DAC_Analog_TX Bean Properties

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAC_Analog_TX</td>
<td>DAC</td>
</tr>
<tr>
<td>D/A channels</td>
<td>1</td>
</tr>
<tr>
<td>Channel output pin</td>
<td>Enabled</td>
</tr>
<tr>
<td>D/A channel (pin)</td>
<td>GPIOD6_DACC</td>
</tr>
<tr>
<td>D/A channel (pin) signal</td>
<td>Analog_TX</td>
</tr>
<tr>
<td>Init value</td>
<td>000000000000</td>
</tr>
<tr>
<td>D/A resolution</td>
<td>12 bits</td>
</tr>
<tr>
<td>Data mode</td>
<td>Left justified</td>
</tr>
<tr>
<td>Glitch filter</td>
<td>Enabled</td>
</tr>
<tr>
<td>Filter value</td>
<td>7</td>
</tr>
<tr>
<td>Synchronous mode</td>
<td>Enabled</td>
</tr>
<tr>
<td>SYNC input pin</td>
<td>TA1_output</td>
</tr>
<tr>
<td>SYNC input pin signal</td>
<td>internal</td>
</tr>
<tr>
<td>Source bean</td>
<td>DAC_Step_Clock</td>
</tr>
<tr>
<td>Automatic waveform generation</td>
<td>Enabled</td>
</tr>
<tr>
<td>Generated shape</td>
<td>User</td>
</tr>
<tr>
<td>Counting up</td>
<td>yes</td>
</tr>
<tr>
<td>Counting down</td>
<td>yes</td>
</tr>
<tr>
<td>Maximum value</td>
<td>FFF</td>
</tr>
<tr>
<td>Minimum value</td>
<td>000</td>
</tr>
<tr>
<td>Step size</td>
<td>000</td>
</tr>
<tr>
<td>Initialization</td>
<td></td>
</tr>
<tr>
<td>Enabled in init. code</td>
<td>yes</td>
</tr>
</tbody>
</table>

Figure 50. DAC_Analog_TX Bean Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enable</td>
<td>don't generate code</td>
</tr>
<tr>
<td>Disable</td>
<td>don't generate code</td>
</tr>
<tr>
<td>SetValue</td>
<td>generate code</td>
</tr>
<tr>
<td>SetValue8</td>
<td>don't generate code</td>
</tr>
<tr>
<td>SetValue16</td>
<td>don't generate code</td>
</tr>
<tr>
<td>SetMaxValue</td>
<td>generate code</td>
</tr>
<tr>
<td>SetMinValue</td>
<td>generate code</td>
</tr>
<tr>
<td>SetStep</td>
<td>generate code</td>
</tr>
<tr>
<td>ConnectPin</td>
<td>don't generate code</td>
</tr>
</tbody>
</table>

Figure 51. DAC_Analog_TX Bean Events

<table>
<thead>
<tr>
<th>Properties</th>
<th>Methods</th>
<th>Events</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>This bean does not contain any event.</td>
</tr>
</tbody>
</table>
3 Software Design Details

All of the code described here is readily available by downloading the modem project from the Freescale
website. However, it is included here for ease of reference, explanation of function, and discussion. The
actual modem code for the V.22bis and the V.21 is not published here.

This section includes the actual code written to implement the modem, which is separate from any code
generated by the beans. The actual code and the discussion of design details are included together. The
following sections relate to the following list of files written during the implementation of this project:

1. Modem.C
2. modem.h
3. Events.c
4. V21_processA.c (included in case the V.21 option is desired)

Modem.C is a file containing C code — the main program as well as most of the top level functionality of
the modem.

The modem.h header file contains mainly state machines for the ring detection and AT command set.

The Events.c program is generated by Processor Expert and is used to connect functions with ISRs that are
associated with beans. These are called events in Processor Expert terminology.

The V21_processA.c file is a customized version of the V21_process.c file automatically generated for the
V.21 modem bean. This is required because the bean only supports synchronous communications with
V.21. By replicating this file and making modifications to its contents, and then calling the methods that
are in V21_processA.c rather than the ones in V21_process.c, an async interface is implemented. The
previous method names are retained with the simple addition of a capital A suffix. The effect of this is a
new, customized API for the V.21 modem.

3.1 Modem.C

Modem.C contains the functions required to use the beans which will implement both the V.22bis/V.22
modem and the V.21 modem (although the V.21 is optioned out for this note).

The sections that follow describe these functions and are interspersed with the code from main.c that
implements them.

3.1.1 main.c Preamble

The preamble is largely generated by the various beans that add the include file references. It also contains
static RAM variables needed for global access and state definition for the modem.

```c
/**
 ** Filename : modem.C
 ** Project : modem
 ** Processor : 56F8367
 ** Version : Driver 01.09
 ** Compiler : Metrowerks DSP C Compiler
 ** Date/Time : 4/5/2005, 2:52 PM
 **
 * A Low-Cost Soft Modem Using the Freescale Digital Signal Controller MC56F802x/3x Series, Rev. 0
 */
```
Software Design Details

** ** Main module.
** ** Here is to be placed user's code.
** ** Settings :
** ** Contents :
** ** No public methods
** **
** ** (c) Copyright UNIS, spol. s r.o. 1997-2004
** ** UNIS, spol. s r.o.
** ** Jundrovska 33
** ** 624 00 Brno
** ** Czech Republic
** ** http : www.processorexpert.com
** ** mail : info@processorexpert.com
** **
** ** ###################################################################*/
/* MODULE Modem */

//#define SAMPLE_RATE 8000
//#define ANALOG_LOOPBACK
//#define FORCE_DATA_MODE
/* Including used modules for compiling procedure */
#include "Cpu.h"
#include "Events.h"
#include "MEM1.h"
#include "CTS1.h"
#include "IV1.h"
#include "SampleRateTimer.h"
#include "TestHarnessDCE.h"
#include "RingDetect.h"
#include "OffHook.h"
#include "TwentythSecInt.h"
#include "TEL1.h"
#include "TEL2.h"
#include "Inhr1.h"
#include "STCK1.h"
#include "Mdm1.h"
#include "DAC_Analog_TX.h"
#include "ADC_Analog_RX.h"
#include "WDogl1.h"
#include "DAC_Step_Clock.h"
/* Include shared modules, which are used for whole project */
#include "PE_Types.h"
#include "PE_Error.h"
#include "PE_Const.h"
#include "IO_Map.h"
#include "v22bis_app.h"
#include "modem.h"
//#define V21LOOPBACK

extern int CountTime ;
extern Result dtmfGenerate (dtmf_sHandle *pDTMF,
  Int16 *pData, /* Pointer to output buffer */
  UWord16 NumSamples);extern void TXCallbackRoutine (void *
pCallbackArg, v22bis_eStatus Status,
  Word16 * pSamples, UWord16 NumberSamples);
extern void RXCallbackRoutine (void * pCallbackArg, v22bis_eStatus Status,
  char * pBits, UWord16 NumberBits);
UWord16 ReadAnalogRxData(Word16 * pRxBuffer, UWord16 Size);
UWord16 WriteAnalogTxData(Word16 * pTxBuffer, UWord16 Size);
void InitAnalogRxChannel(void);
void InitAnalogTxChannel(void);
void InitPoorMansCodec(void);
void CPTDetCallback (void *, UWord16);
UWord16 v22bis_connection_established, connection_lost;
UWord16 rate_negotiated;
v22bis_sTXCallback v22bis_TXCallback;
v22bis_sRXCallback v22bis_RXCallback;

word Connecting_v21,Caller_Modem,call_phone_number,AT_z_flag,AT_q_flag,
V21_Mode,Ans_Tone_Start,Ans_Tone_Detect;
#define AC0tr 0x3ffff     // energy threshold for ANS detection
#define NUM_ANS_SAMPLESd4 12
#define NUM_ANS_SAMPLESd2 24
#define NUM_ANS_SAMPLES 48 // number of samples to collect integral number of periods of
ANS
// based on 2100 hertz tone sampled at 7200 SPS.
// 2100/7200 * 7 , seven samples in 24 cycles.
// twice that to calculate autocorrellation
// function number 24, as well as 0 and 12 to compare.
/* Number of samples to be collected before calling the V22bis
receiver */
#define NUMRX_SAMPLES_V22bis 12 // v22 bis modem
#define NUM_SAMPLES_V21 24
/* Number of samples to be collected before calling the Call Progress Detect
receiver */
#define NUM_SAMPLES 100 // call progress
Word16 CodecTxBuffer[NUMSAMPLES]; /* Samples generated by the transmitter */
Word16 CodecRxBuffer[NUM_SAMPLES]; /* Samples for the receiver */
unsigned char tty_in[100];
unsigned char modem_in[100]; // modem has private buffer, cannot share.
byte tty_in_status;
unsigned char ModemRxBuffer[MODEM_RX_BUFF_SIZE];
volatile unsigned char *pModemRxWrt = ModemRxBuffer;
volatile unsigned char *pModemRxRead = ModemRxBuffer;
volatile UWord16 Line_Tones;  // call progress detected
volatile UWord16 is_time_to_shake = FALSE ; // between rings
int AnalogRxBuffer[RX_BUFFER_SIZE];
volatile int *pAnalogRxWrite=0;
volatile int *pAnalogRxRead=0;
volatile unsigned int AnalogTxBuffer[TX_BUFFER_SIZE];
volatile unsigned int *pAnalogTxWrite = 0;
volatile unsigned int *pAnalogTxRead = 0;
Word16 PreviousSample = 0;
word CumCnt = 0 ;
word SaveCnt = 0 ;
/* Number of bytes transmitted at a time */
extern word ring_pulse_count;
extern RING_STATE ring_state;
volatile UWord16 Line_Tones; // call progress detected
signed long ACa,AC0,AC12,AC24; // autocorrelation function accumulators for ANS
detect.
#ifdef CALIBRATE
signed long DCavg;
#endif
int h_parm ;
byte *p_phone_number ;
byte phone_number[41]; //ascii characters.. terminated with 0x00 as in a string.
   //the phone number to call is stored here.. up to 40
digits. Calling is activated via "call_phone_number".
AT_OFF_STATES AT_off_state ;
AT_ON_STATES AT_on_state ;
ESCAPEMENT_STATES state_of_escape ;
word Last_Dac_Value = 0;
long CombinedAnalogRxSample ;
  // V.21

#ifndef __V21_H
v21_sHandle *pV21;
v21_sConfigure ConfigV21;
void TxCallbackV21 (void *pCallbackArg, Word16 *pSamples, UWord16 NumberSamples);
void RxCallbackV21 (void *pCallbackArg, char *pChars, UWord16 NumberBits);
Result v21TxProcessA (v21_sHandle *pV21, unsigned char *pBytes,UWord16 NumBytes);
Result v21RxProcessA (v21_sHandle *pV21, Word16 *pSamples, UWord16 NumSamples);
Result resTx=V21_TX_FREE;
Result resRx=V21_RX_PASS;
#endif
word TestTone[4] =
{
   0xC471, // don't make it too regular or too easy
   0x7876,
   0x8763,
   0x79f
};
Frac16 V25ansTone[24] =
{
   0,
   3967,
   -2054,
   -2904,
   3557,
   1063,
   -4107,
   1063,
   3557,
   -2904,
   -2054,
   3967,
   0,
   -3967,
   2054,
   2904,
   -3557,
   -1063,
   4107,
   -1063,
   -3557,
   2904,
   2054,
   -3967
};
unsigned int Blue_DAC_Scale ;
word This_Dac_Value ;
3.1.2 main()

The main() function is the control code for the modem. It is organized around the sequence of events needed to form one data connection to another modem connected over the PSTN. The initialization block that is executed only at system reset or power-up is first. This is followed by the start of a large loop that represents the establishment and disestablishment of a call to another modem.

One time through this large loop represents one call. The modem is re-initialized to the extent necessary after each call, retaining no information about the previous call. Various stages of the process differ depending on how the call is originated. The process begins, after initializing the per-call variables, with waiting for one of two events to happen:

- the phone rings
- there is a command to do one of these:
  - establish a call
  - adjust one of the command-configurable parameters
  - display product information

The main program can be broken conceptually into several parts that flow in sequence within this large loop.

3.1.2.1 “On Stack” Main Data Definitions

These definitions are allocated on the stack portion reserved when the main.c program is called. Because main() never returns by design, the variables herein may be considered to be static, but not global.

```c
void main(void)
{
    #ifdef CALIBRATE
    int DCcount = 0 ;
    signed long DC=0   ;
    #endif
    v22bis_sHandle *V22bisInstance=0;
    CPTDet_sHandle *pCPTDet=0;
    dtmf_sHandle DTMFhandle;
    v22bis_sConfigure *pConfig=0;
    CPTDet_sConfigure *pCPTDet_Config=0;
    dtmf_sConfigure ConfigDTMFg;
    int resultv22,i;
    int resultcpd;
    int resultDTMFg;
    UWord16 modem_config;
    // there is no public configuration flag word for the call progress bean's structures
    UWord16 num_words;
    UWord16 message_transmitted;
    int numtxbytes ;
```
Software Design Details

3.1.2.2 Actual Entry Point to Main and One-Time-Only Initialization Block

The Processor Expert adds the following line to initialize the beans, as directed in the properties in the GUI bean interfaces. Following that is the one-time initialization for the modem itself.

```c
/*** Processor Expert internal initialization. DON'T REMOVE THIS CODE!!! ***/
PE_low_level_init();
WDog1_Disable();  // make sure none of this
PESL(DAC0, DAC_WRITE_MAXVAL_REG_LEFT, 0xfff0); // PE uses 0xffff!
numtxbytes = 0 ;
/*** End of Processor Expert internal initialization.                    ***/
pModemRxWrt = ModemRxBuffer;
pModemRxRead = ModemRxBuffer;
PreviousSample = 0;
V21_Mode = FALSE ;
```

3.1.2.3 Call Establishment/Disestablishment while Loop Runs Once Per Call

This loop consists of the various stages of call establishment and disestablishment. The various stages are delineated and described in the sub-sections that follow.

```c
while (1)  // start / restart modem.
    // the modem must return to this loop top when it restarts. The modem restarts
    // after a call is lost.
{
    byte Key;
```

3.1.2.3.1 Per-Call Initialization

Prior to the first and subsequent calls, variables are initialized. No record of prior calls is maintained. By clearing the call_phone_number flag, the modem is prevented from repeatedly calling the stored phone number until a connection is established.

```c
OffHook_ClrVal();
for (i = 0; i< NUM_SAMPLES ; i++)  CodecRxBuffer[i] = 0;
AT_q_flag = AT_z_flag = FALSE ;
AT_off_state = AT_off_idle ;
AT_on_state = AT_on_idle ;
is_time_to_shake = FALSE;
call_phone_number = FALSE;
//CTS1_On();  // no data flow yet. CTS.. Clear To Send.. used to throttle the
DTE, which is
    // most often a PC used for testing. When CTS1_On is called, it has
    // the effect of stopping the
    // the DTE from sending further data. This is commonly known as
hardware flow control
    // such as is found in the hyperterminal program of Windows. This
program does not implement
    // flow control in the other direction.. towards the DTE, which
is assumed to be capable of
    // recieving the data without pause.
CumCnt = 0 ;    // This is the accumulation of characters in this programs buffer
  of data from the DTE.
    // It is monitored for the purposes of flow control which is
  effected thru the CTS1 bean.
```
Caller_Modem = FALSE ; // When the modem bean is initialized, it must be initialized in caller or callee mode.
// We start with the assumption that we are not making a call, so we define the boolean
// variable, Caller_Modem, to be FALSE.

pCPTDet_Config = (CPTDet_sConfigure *) memMallocEM (sizeof (CPTDet_sConfigure)); // In order to use call progress
tone detection, memory must be allocated in advance.
if (pCPTDet_Config == NULL) // Once debugging is complete, this check may perhaps be removed.
{
    asm(debughlt);
    while(1){};
}
pCPTDet_Config->CPTDetCallback.pCallback = CPTDetCallback; // The call progress
detection will call us back when
call progress tones are detected.
pCPTDet = CPTDetCreate (pCPTDet_Config); // Prepare to use the Call Progress Detection API with this creation.
if (pCPTDet == NULL) // Once debugging is complete, this check may perhaps be removed.
{
    asm(debughlt);
    while(1){};
}
// Initialize and start the ring detection state machine:
ring_state = ring_idle ; // The ring detector is a state machine in conjunction with the
// bean. The initial or reset state of the ring detector is
"ring_idle".
ring_pulse_count = 0;    // The ring detector
OffHookClrVal();         // ONHOOK, or not offhook.. rest condition of
DAA.. not connected.
RingDetect_Disable();
RingDetect_Enable();
RingDetect_ResetCounter();
archEnableInt();
#ifdef FORCE_DATA_MODE    // force data mode ... handshake even if no ring and no
    is_time_to_shake = TRUE;
#endif
// CTS1_Off(); // allow flow to modem from DTE for commands.

3.1.2.3.2 Connection Establishment Loop
This loop waits for a ring or command to establish a connection. Origination is triggered via the
"call_phone_number " flag, which is set by the offline AT command function, called at the top of the loop.
As characters come in, the command is assembled and executed to set this flag. Or, the ATA command can
force the modem into answer mode. Other commands can change the modem to operate in V.21 mode, to
display the version of the code, or to suppress output of response to commands and connection status
changes.
while ((is_time_to_shake == FALSE) && (AT_z_flag == FALSE))   // stay in this loop until modem goes off hook and completes a connection.
{
  AT_offline();
  if(call_phone_number)

3.1.2.3.3 Search for Dial Tone

The modem is now off-hook to dial and must find dial tone. The call progress bean is used to search for dial tone. In addition to the methods defined for the beans, PESL commands allow the selection of particular useful commands for direct manipulation of the peripherals. In this case, QTIMER_A1 is associated with the sample rate. This bean is used to time the samples on the ADC to achieve a net sample rate of 7200 samples per second. However, for the dial tone search, a bean is used that needs to be provided samples at 8000 samples per second. By changing the starting point of the timer up a few positions, the time is reduced and the number of samples is increased. The following PESL command changes the sample rate (of just the input, because the output is not used during dial tone search):

PESL(QTIMER_A1, QT_WRITE_LOAD_REG, 168);  // 802x set for 8000 sps output

When the bean is re-initialized, this value is reset to zero. That is why there is no PESL command to put the value back to zero. The bean is simply re-initialized when a return to 7200 samples per second is required.

// Start monitoring call progress.
    int Time_left_to_ANS, Time_left_to_DIALTONE ;
    byte * pKey;
    OffHook_SetVal();
    InitPoorMansCodec();
    PESL(QTIMER_A1, QT_WRITE_LOAD_REG, 168);  // 802x set for 8000 sps output
    Line_Tones = 0;
    Time_left_to_DIALTONE = (7200/NUM_SAMPLES) * 10; // 10 seconds

3.1.2.3.4 Wait-for-Dial-Tone Loop is Timed and User-Abortable

This complex while statement looks for dial tone, a timer expiration (nine second sledge-hammer timer), or any character to be entered from the test fixture, to abort the attempt to obtain dial tone. The sledge-hammer timer times the entire process of dialing, including the dial tone search, up to the point of answer tone detection.

while ((Line_Tones != DIAL_TONE_DETECTED) && Time_left_to_DIALTONE-- &&
(TestHarnessDCE_GetCharsInRxBuf() == 0))
{
    num_words = 0 ;
    do
    {
        num_words += ReadAnalogRxData((CodecRxBuffer + num_words),NUM_SAMPLES - num_words);
    } while (num_words < NUM_SAMPLES);  // 100 for Call Progress Detect

3.1.2.3.5 CALIBRATE — For Calibration of the Daughter Cards

Normally, CALIBRATE is not defined. When it is, the code will calculate the energy in the dial tone. This value may be observed directly with CodeWarrior using the Data Visualization feature of the IDE. Then
the value of the potentiometer on the LCMDC (low cost modem daughter card) may be adjusted and real-time feedback obtained on the offset of the signal.

```c
#ifdef CALIBRATE // try to adjust LCMDC R15 for zero offset.
    Word16 *pCodecB;
    int i;
    pCodecB = CodecRxBuffer;
    for (i = 0 ; i < NUMRX_SAMPLES_V22bis ; i++) DC += *pCodecB++;
    DCcount += NUMRX_SAMPLES_V22bis;
    if (DCcount > 500) // can be larger if desired smoother.
    {
        DCcount = 0;
        DCavg = DC; // no need to divide, just show the number. Use data visualization!
        DC = 0;
    }
#else // if calibrating (above), holds on dialtone or whatever tone is incoming.
    resultcpd = CPTDetection(pCPTDet, CodecRxBuffer, NUM_SAMPLES);
#endif
CPTDetDestroy(pCPTDet); free(pCPTDet_Config);
if ((Time_left_to_DIALTONE <= 0) || TestHarnessDCEGetCharsInRxBuf())
{
    unsigned int n;
    call_phone_number = FALSE; // don't keep trying to call the number forever automatically ;
    AT_off_state = AT_off_idle; // allow new commands.
    if (!AT_q_flag) TestHarnessDCESendBlock((byte *)"\nNO DIALTONE\r\n", 14, &n); // indicate to the DTE the problem.
    goto RESTART; // K&R approve use of goto in cases where break is not effective.
}
```

### 3.1.2.3.6 After Dial Tone is Obtained, Rapid DTMF Dialing Ensues

The dialer parameters are set for the fastest dialing that should be recognized by standard telephony equipment. To slow down this rate of dialing by a factor of two, increase ConfigDTMFg.OnDuration and ConfigDTMFg.OffDuration contents by a factor of two.

The phone number to dial is simply a string pointed to by phone_number. This dialer can dial the digits 0 through 9 as well as the letters A through D, not normally provided on telephone handsets.

Note below the presence of additional diagnostics that are conditionally available when bringing the modem up on hardware. These diagnostics will output various tones and DTMF digits.

```c
// dial-tone detected .. dial as fast as possible
InitPoorMansCodec();
CPTDetDestroy(pCPTDet); pCPTDet = 0;
free(pCPTDet_Config); pCPTDet_Config = 0;
if ((Time_left_to_DIALTONE <= 0) || TestHarnessDCEGetCharsInRxBuf())
{
    unsigned int n;
    call_phone_number = FALSE; // don't keep trying to call the number forever automatically ;
    AT_off_state = AT_off_idle; // allow new commands.
    if (!AT_q_flag) TestHarnessDCESendBlock((byte *)"\nNO DIALTONE\r\n", 14, &n); // indicate to the DTE the problem.
    goto RESTART; // K&R approve use of goto in cases where break is not effective.
}
```
Blue_DAC_Scale = BLUE_DAC_SCALE_DTMF;
ConfigDTMFg.OnDuration = 288;  /* No. of samples for on duration */
ConfigDTMFg.OffDuration = 252;  /* No. of samples for off duration */
ConfigDTMFg.SampleRate = 7200;  /* Sampling Frequency */
ConfigDTMFg.amp = 0x1fff;  /* Amplitude, should be less than 0.5 */
/* Call dtmfCreate */
TELE2_dtmfInit (&DTMFhandle, &ConfigDTMFg);
pKey = phone_number ;

#ifdef OUTPUT_TEST_TONE
#endif
#ifdef OUTPUT_ANS_TEST_TONE
#endif
#ifdef DIAL_FIRST_REPEAT_DEBUG
#endif

for (;;)
{
    num_words = 0;
    do
    {
        num_words += WriteAnalogTxData((short *) &TestTone[0] + num_words), 4 - num_words);
    }
    while (num_words < 4);
}

for (;;)
{
    num_words = 0;
    do
    {
        num_words += WriteAnalogTxData((&V25ansTone[0] + num_words), 24 - num_words);
    }
    while (num_words < 24);
}

for (;;)
{
    Key = *pKey;  // dial first digit repeatedly
    for (;;)
    {
        TEL2_dtmfSetKey(&DTMFhandle, (char)Key);
        resultDTMFg = PASS;
        while (resultDTMFg == PASS)
        {
            resultDTMFg = dtmfGenerate(&DTMFhandle, (Int16 *) CodecTxBuffer, NUMSAMPLES);
            num_words = 0;
            do
            {
                num_words += WriteAnalogTxData((CodecTxBuffer + num_words), NUMSAMPLES - num_words);
            }
            while (num_words < NUMSAMPLES);
        }
    }
} // number dialed.
3.1.2.3.7 Search for Answer Tone

To achieve the best possible results, the timing of the hand-off to the V.22bis bean is critical. The V.22bis bean should take over as soon as answer tone is detected. This provides the largest connect percentage.

In the case of V.21, the entire answer tone should be detected all the way to the end, prior to handing control of the line to the V.21 code.

For the application program to determine this, a simple autocorrelation function is used. Due to the sampling rate and frequency being detected, the following code will quickly detect the answer tone or any periodic function with the same period as the answer tone. Prior to doing the autocorrelation, a dc offset correction is performed.

The modem code itself contains a similar function to detect answer tone in a similar way.

Again, any key typed at the async interface will abort the attempt to find answer tone and connect.

```c
InitPoorMansCodec();
//#define MODEM_DET_ANS_TONE  // give the modem a chance to see if it can do it better than this code can.
#else MODEM_DET_ANS_TONE
    Time_left_to_ANS = (7200/NUM_ANS_SAMPLES) * 30; // 30 seconds
    Ans_Tone_Start = FALSE ;
    Ans_Tone_Detect = FALSE ;
    while
        {           // until this is false
            (   (TestHarnessDCE_GetCharsInRxBuf() == 0)       &&   // or this is false
                    (Time_left_to_ANS > 0)             &&   // or all this false
                    ( V21_Mode || !Ans_Tone_Start)     &&   // or all this false
                    (!V21_Mode ||!Ans_Tone_Start || Ans_Tone_Detect ) // or all this false
                )
        }
    
    Time_left_to_ANS--;
    num_words = 0;
    // we stay in this do loop a long time perhaps..ReadAnalogRxData may give zero..do
    do
        {num_words += ReadAnalogRxData((CodecRxBuffer + num_words),NUM_ANS_SAMPLES - num_words);
    }
    while (num_words < NUM_ANS_SAMPLES);
#endif
```

// Detect ANS
// autocorrelation function 0 should equal autocorrelation function 24.
// autocorrelation function 12 should be the negative of both for 2100 hertz at 7200SPS.
// scaling by shift right 5 (32) will allow use of long for accumulation without overflow.
ACa=AC0=AC12=AC24 = 0;
for (i=0; i < NUM_ANS_SAMPLESd2; i++)
    {
        ACa += (long) CodecRxBuffer[i];            // note, ACa will be small for well adjusted offset on RX
    }
ACa /= NUM_ANS_SAMPLESd2;
for (i=0; i < NUM_ANS_SAMPLESd2; i++)
    {
        AC0 += ((((long)(CodecRxBuffer[i ])-ACa) *
(long)(CodecRxBuffer[i   ]-ACa))    / NUM_ANS_SAMPLESd2;
AC12   += (((long)(CodecRxBuffer[i   ]-ACa) * (long)(CodecRxBuffer[i+NUM_ANS_SAMPLESd4]-ACa)))    / NUM_ANS_SAMPLESd2;
AC24   += (((long)(CodecRxBuffer[i   ]-ACa) * (long)(CodecRxBuffer[i+NUM_ANS_SAMPLESd2]-ACa)))    / NUM_ANS_SAMPLESd2;
}
if ((AC0  >=  AC0tr) && (AC24 >=  AC0tr) && (AC12 <= -AC0tr) &&
  ( (50  * (AC0+AC12)) < AC0) && ( (50  * (AC0-AC24)) < AC0)
  ){
  Ans_Tone_Start= TRUE ;
  Ans_Tone_Detect = TRUE ;
} else {
  Ans_Tone_Detect = FALSE ;
}
if ((Time_left_to_ANS <= 0) || TestHarnessDCE_GetCharsInRxBuf() ) {
  unsigned int n;
  call_phone_number = FALSE ; // don't keep trying to call the number forever.. automatically ;)
  AT_off_state = AT_off_idle ; // allow new commands.
  if (!AT_q_flag) TestHarnessDCE_SendBlock((byte *)"\nNO CARRIER\r\n", 13, &n); // indicate to the DTE the problem.
  goto RESTART ; // K&R approve use of goto in cases where break is not effective.
} #endif
Caller_Modem = TRUE ;
// launch originate modem here.
is_time_to_shake = TRUE ; // ANS detected, originate mode selected, go to V.xx protocol.
} // end (if call_phone_number)
}; // End check for phone ringing to answer
is_time_to_shake = FALSE ;
if (AT_z_flag)
  goto RESTART ;
}
OffHook_SetVal();

3.1.2.3.8 Modem Handshake — V.21 Case — Provide ANS Tone if Answering

Now that the number has been dialed and answer tone has been detected, or we have answered the phone and must provide answer tone, first we check for the case of V_21. In this case, the answer tone must be provided by the application. This is done very easily in the small loop below, which illustrates well the technique for outputting samples to the telephone interface.

// DATA PUMPING STARTS HERE
Blue_DAC_Scale = BLUE_DAC_SCALE_V22bis ; //0 also works
Software Design Details

The V.21 modem is dynamically instantiated here based on the calling mode, which is either answer or originate. Note the callback routines. These are routines that are provided by the application programmer and are called back by the modem data pump code when the modulation or demodulation of a “chunk” of data is obtained. In the case of the V.21 modem, a chunk is just one bit. The application program is called back for each received bit. This makes it a task of the application program to put the received characters together. The final else statement below is “if not V.21.” The #endif is the demarcation of the end of the V.21 code that is not compiled because it is conditioned off.

```c
ConfigV21.Gain = 0x07ff; // EXPERIMENT WITH THIS!
// ConfigV21.Gain = 0x01ff; // original
ConfigV21.TxCallback.pCallback = TxCallbackV21;
ConfigV21.TxCallback.pCallbackArg = NULL;
ConfigV21.RxCallback.pCallback = RxCallbackV21;
ConfigV21.RxCallback.pCallbackArg = NULL;
pV21 = v21Create (&ConfigV21);
if (pV21 == NULL)
{
    asm(debughlt);
    while(1){}
}
    // assert (!"Out of memory");
} else
    #endif
```

A Low-Cost Soft Modem Using the Freescale Digital Signal Controller MC56F802x/3x Series, Rev. 0
3.1.2.3.9 V.22bis Dynamic Instantiation on a Per-Call Basis

In this case the V.22bis data pump is activated, not the V.21 data pump. If required, it will generate the answer tone. It also looks for answer tone, so if we are calling, the loop above has dropped us here while the majority of the answer tone is yet to be heard.

```c
/* Initialize the flow control parameters for the demo */
v22bis_connection_established = FALSE;     // Will be set in the TX Callback routine
connection_lost = FALSE;                  // Will be set in the RX Callback routine if there
is an error
message_transmitted = (unsigned short)FAIL;
/* Set the rate_negotiated flag to none (-1) before starting
V22bis / V22 transaction. The flag "rate_negotiated will
take either V22BIS_1200BPS_CONNECTION_ESTABLISHED or
V22BIS_2400BPS_CONNECTION_ESTABLISHED which are a part of
"v22bis_eStatus" enum defined in "v22bis.h" file */
rate_negotiated = (unsigned short)-1;
```

This code manages the initialization for the V.22bis modem, the parameters that are used to control flow to and from the actual V.22bis data pump bean. Next, depending on whether the call has been originated or is being answered, the modem parameters for the data pump itself are determined. The boolean flag Caller_Modem is true if we have dialed or otherwise originated this call.

```c
/* Modem configuration parameters */
if (Caller_Modem)
{
    modem_config = (V22BIS_CALL_MODEM |   V22BIS_GUARD_TONE_DISABLE |
V22BIS_SELF_RETRAIN_ENABLE | V22BIS_V14_ENABLE_ASYNC_MODE);
} else
{
#define DISABLE_2100_ANSWERTONE
    modem_config = (V22BIS_ANSWER_MODEM | V22BIS_GUARD_TONE_DISABLE |
V22BIS_SELF_RETRAIN_ENABLE | V22BIS_V14_ENABLE_ASYNC_MODE |
V22BIS_DISABLE_2100_ANSWERTONE);
#undef DISABLE_2100_ANSWERTONE
    modem_config = (V22BIS_ANSWER_MODEM | V22BIS_GUARD_TONE_DISABLE |
V22BIS_SELF_RETRAIN_ENABLE | V22BIS_V14_ENABLE_ASYNC_MODE);
}
/* Allocate memory for the init structure */
pConfig = memMallocEM( sizeof(v22bis_sConfigure));
if (pConfig == NULL)
{
    asm(debughlt);
    while(1){}
}
```

The callback routines are called when, in the course of the data pump either handling samples it has received or else generating samples to transmit, it determines that characters have been received or that characters may be given for transmission. Other status may also be conveyed in a callback routine. Note that the CTS functions are commented out because they were not used with this test fixture. In an embedded modem such a signal is not even needed, because there is no interface with a DTE.

```c
v22bis_TXCallback.pCallback = TXCallbackRoutine;
```
Software Design Details

A Low-Cost Soft Modem Using the Freescale Digital Signal Controller MC56F802x/3x Series, Rev. 0

3.1.2.3.10 Online Data Loop

After the connection, V.21 or V.22bis/V.22 is established, the code simply loops back to this point. When
the call is over, it falls through this loop and back up to the outermost loop in the program. The AS1 bean
is tied to the serial port, which is connected to HyperTerminal for testing. AT commands and data come
from that source. The online command mode escapement machine follows, as well as the normal data input
point. Data flow is similar for V.21 and V.22bis, but not identical, because V.21 is more of a bit-oriented
device. This V.21 operates at a fixed 300 bits per second. However, even when V.21 is operating, the AS1
bean, or async terminal connection, remains at 2400 baud. The data is buffered. Flow control signaling
from the EVM to the PC assures that the PC will not overrun the modem with data.

The ATZ command will set a flag that will cause the online loop to terminate and the modem to perform
a soft reset.

3.1.2.3.10 Online Data Loop

After the connection, V.21 or V.22bis/V.22 is established, the code simply loops back to this point. When
the call is over, it falls through this loop and back up to the outermost loop in the program. The AS1 bean
is tied to the serial port, which is connected to HyperTerminal for testing. AT commands and data come
from that source. The online command mode escapement machine follows, as well as the normal data input
point. Data flow is similar for V.21 and V.22bis, but not identical, because V.21 is more of a bit-oriented
device. This V.21 operates at a fixed 300 bits per second. However, even when V.21 is operating, the AS1
bean, or async terminal connection, remains at 2400 baud. The data is buffered. Flow control signaling
from the EVM to the PC assures that the PC will not overrun the modem with data.

The ATZ command will set a flag that will cause the online loop to terminate and the modem to perform
a soft reset.

3.1.2.3.10 Online Data Loop

After the connection, V.21 or V.22bis/V.22 is established, the code simply loops back to this point. When
the call is over, it falls through this loop and back up to the outermost loop in the program. The AS1 bean
is tied to the serial port, which is connected to HyperTerminal for testing. AT commands and data come
from that source. The online command mode escapement machine follows, as well as the normal data input
point. Data flow is similar for V.21 and V.22bis, but not identical, because V.21 is more of a bit-oriented
device. This V.21 operates at a fixed 300 bits per second. However, even when V.21 is operating, the AS1
bean, or async terminal connection, remains at 2400 baud. The data is buffered. Flow control signaling
from the EVM to the PC assures that the PC will not overrun the modem with data.

The ATZ command will set a flag that will cause the online loop to terminate and the modem to perform
a soft reset.
This code implements flow control. The PC test fixture would possibly overrun the serial port interface during binary file transfers if there were no flow control mechanism. Data from the serial port is buffered in tty_in, with the cumulative count, CumCnt, recorded up to a maximum of 100.

The following code checks for escape from the online data mode (entered after a connection is established) to the online command mode. The operator can type “+++” to leave the online data mode and enter the online command mode. The data passes into the serial port, and even to the other modem. There is a guard time before the +++ and another guard time after it, to ensure that it has not been entered by mistake. A state machine implemented with a simple case statement keeps track of this feature’s state.

```
switch (state_of_escape)
{
    case Escape_idle_zero:
        if (CountTime >= 30) state_of_escape = Escape_pre_idle_1;
        break;
    case Escape_pre_idle_1: // no data for a while.. look for plus
        if (CumCnt > 3) state_of_escape = Escape_idle_zero ;
        else if (CumCnt == 1) {if (tty_in[0]=='+') state_of_escape = Escape_p1 ; else
            state_of_escape = Escape_idle_zero ;}
        else if (CumCnt == 2) {if ((tty_in[0]=='+' && (tty_in[1]=='+')) state_of_escape = Escape_p2; else state_of_escape = Escape_idle_zero ;}
        else if (CumCnt == 3) {if ((tty_in[0]=='+' && (tty_in[1]=='+')) && (tty_in[2]=='+'))
            state_of_escape = Escape_p3; else state_of_escape = Escape_idle_zero ;}
        break;
    case Escape_p1:    // the first + is received.
        if (CumCnt > 2) state_of_escape = Escape_idle_zero ;
        else if (CumCnt == 1) {if (tty_in[0]=='+') state_of_escape = Escape_p2 ; else
            state_of_escape = Escape_idle_zero ;}
        else if (CumCnt == 2) {if ((tty_in[0]=='+') && (tty_in[1]=='+')) state_of_escape =
            Escape_p2; else state_of_escape = Escape_idle_zero ;}
        break;
    case Escape_p2:    // the second + is received.
        if (CumCnt > 1) state_of_escape = Escape_idle_zero ;
        else if (CumCnt == 1) {if (tty_in[0]=='+') state_of_escape = Escape_p3 ; else
            state_of_escape = Escape_idle_zero ;}
        break;
    case Escape_p3:    // the third + is received. Wait to make sure no data.
        if (CountTime >=30)
        {
            state_of_escape = Escape_post_idle;
            if (!AT_q_flag) TestHarnessDCE_SendBlock((byte *)&"\n\nOK\n\n", 6, &n);
            CumCnt =0 ;
        }
        break;
    case Escape_post_idle:    // Escape is in effect.. process online AT commands.
        AT_online();
    }    // end switch
```

The data flow for the V.21 beans and the V.22bis beans differ. First, the V.21 bean case is coded.

```
#ifdef __V21_H
    if (V21_Mode)
    {
        if (Connecting_v21)
        {
            if (!AT_q_flag) TestHarnessDCE_SendBlock((byte *)&"\n\n\nCONNECT 300\n\n", 15, &n);
            Connecting_v21 = FALSE ;
```
num_words = 0;
do {
    num_words += ReadAnalogRxData((CodecRxBuffer + num_words),Num_Samples_V21 - num_words);
} while (num_words < Num_Samples_V21); // 24 at least for v.21
if (resTx == V21_TX_FREE ) {
    if ((CumCnt >= 1) && (resRx != V21_RX_CARRIER_LOST ) ) // can't use tty_in_status which comes back empty after draining chars
    for (i=0;i<CumCnt;i++) modem_in[i] = tty_in[i] ;
    SaveCnt = CumCnt ;
    resTx = v21TxProcessA (pV21, (unsigned char *) &modem_in[0], CumCnt);
    CumCnt = 0 ;
} else resTx = v21TxProcessA (pV21, (unsigned char *) NULL, 0); // mark some time.. one bit.
else resTx = v21TxProcessA (pV21, (unsigned char *) &modem_in[0], SaveCnt);
if (pModemRxRead != pModemRxWrt) {
    unsigned char RxChar = *pModemRxRead++;
    TestHarnessDCE_SendChar(RxChar); // if no analog loopback.
    if (pModemRxRead >= &ModemRxBuffer[MODEM_RX_BUFF_SIZE])
        pModemRxRead = ModemRxBuffer;
}
#endif V21LOOPBACK
for (i = 0; i < NUM_SAMPLES_V21; i++)
    CodecRxBuffer[i] = CodecTxBuffer[i];
#else
    resRx = v21RxProcessA (pV21, CodecRxBuffer, Num_Samples_V21);
    if (resRx == V21_RX_CARRIER_LOST)
        if (!AT_q_flag) TestHarnessDCE_SendBlock((byte *)"\r\nNO CARRIER\r\n", 14, &n);
    if (resRx == V21_RX_CARRIER_LOST )
        break;
#endif

The other case would be V.22bis, because only these two modes are implemented.
else // if not V21, then V.22bis..or V.22
{ }
Software Design Details

if (message_transmitted == FAIL)
{
  if (CumCnt >= 1) // can't use tty_in_status which comes back empty after draining
    chars
    {
      for (i=0;i<CumCnt;i++) modem_in[i] = tty_in[i];
      v22bisTXDataInit (V22bisInstance, (char *) &modem_in[0], CumCnt);
      CumCnt = 0;
    }
}
message_transmitted = (unsigned short) v22bisTX (V22bisInstance);
if (pModemRxRead != pModemRxWrt)
{
  unsigned char RxChar = *pModemRxRead++;
  TestHarnessDCE_SendChar(RxChar); // if no analog loopback.
  if (pModemRxRead >= &ModemRxBuffer[MODEM_RX_BUFF_SIZE])
  {
    pModemRxRead = ModemRxBuffer;
  }
}
/* Call V22bis receiver */
resultv22 = v22bisRX (V22bisInstance, CodecRxBuffer, NUMRX_SAMPLES_V22bis);
if (resultv22 == FAIL)
{
  connection_lost = TRUE; // Terminate the loop */
}
if (connection_lost == TRUE)
{
  if (!AT_q_flag) TestHarnessDCE_SendBlock((byte *)"\r\nNO CARRIER\r\n", 14, &n);
  break;
}
} // end block
} // end while atz flag is false

3.1.2.3.11 Call Disestablishment and Modem Recycling to Re-Initialize for Next Call

There is only one tag used in this program, because a call can be aborted from within several of the loops that have been described. Resources incidental to a particular call setup are freed.

RESTART:
{
  InitPoorMansCodec();
  if (pCPTDet) CPTDetDestroy(pCPTDet);
  if (pCPTDet_Config) free (pCPTDet_Config);
  if (V22bisInstance) resultv22 = v22bisDestroy (V22bisInstance);
  if (pConfig) free (pConfig);
  #ifdef __V21_H
  if (pV21) v21Destroy (pV21);
  #endif
  pCPTDet =NULL;
  pCPTDet_Config = NULL;
  V22bisInstance =NULL ;
  pConfig =NULL;
  #ifdef __V21_H
  pV21 = NULL ;
  #endif
  } // restart modem
3.1.3 AT_offline()

This routine parses and executes the AT commands that are directed from the test fixture, the AS1 bean, to the modem when it is not connected to another modem. Commands can be used to dial or change the options of the modem. A simple state machine is used to parse the commands one character at a time in real time. Because this is just a test harness, no command editing features are included.

---

```c
void AT_offline (void)
{
  unsigned int n ;
  byte c;
  if (TestHarnessDCE_GetCharsInRxBuf())
  {
    tty_in_status = TestHarnessDCE_RecvBlock(&c,1,&n) ;
    switch (AT_off_state)
    {
      case AT_off_idle:
        if ((c=='a') || (c=='A'))
        {
          AT_off_state = AT_off_a ;
          TestHarnessDCE_SendChar(c) ;
        }
        break;
      case AT_off_a:
        if ((c=='t') || (c=='T'))
        {
          AT_off_state = AT_off_at ;
          TestHarnessDCE_SendChar(c) ;
        }
        else if ((c=='a') || (c=='A'))
        {
          TestHarnessDCE_SendChar(c) ;
        }
        else AT_off_state = AT_off_idle ;
        break;
      case AT_off_at:
        if ((c=='a') || (c=='A'))
        {
          TestHarnessDCE_SendChar(c) ;
        }
        else break;
    }
  }
}
```
Software Design Details

```c
TestHarnessDCE_SendChar(c) ;
if (c== '\r')
{
    AT_off_state = AT_off_idle ;
    if (!AT_q_flag) TestHarnessDCE_SendBlock((byte *)"\nOK\r\n", 5, &n);
} else if ((c=='a') || (c=='A'))
{
    AT_off_state = AT_off_at ;
} else if ((c=='d') || (c=='D'))
{
    AT_off_state = AT_off_atd ;
    p_phone_number = phone_number ;
} else if ((c=='i') || (c=='I'))
{
    AT_off_state = AT_off_ati ;
} else if ((c=='q') || (c=='Q'))
{
    AT_off_state = AT_off_atq ;
    AT_q_flag = FALSE ;
} else if ((c=='z') || (c=='Z'))
{
    AT_off_state = AT_off_atz ;
} else if (c=='+')
{
    AT_off_state = AT_off_plus ;
    V21_Mode = TRUE ;
} break;

case AT_off_ata:
TestHarnessDCE_SendChar(c) ;
if (c== '\r')
{
    call_phone_number = FALSE ;
    AT_off_state = AT_off_idle ;
    is_time_to_shake = TRUE;
} break;

case AT_off_atd:
TestHarnessDCE_SendChar(c) ;
if (c > '0' && c <= '9') ||(c > 'a' && c <= 'd') ||(c > 'A' && c <= 'D') || c == '+'
|| c == '#')
{
    if (p_phone_number != &phone_number[39]) *p_phone_number++ = c;
} else if (c== '\r')
{
    *p_phone_number = '\0';
    call_phone_number = TRUE ;
    AT_off_state = AT_off_idle ;
} break;
```
case AT_off_atq:
    TestHarnessDCE_SendChar(c);
    if (c == '0') AT_q_flag = FALSE ; else if (c == '1') AT_q_flag = TRUE ;
    if (c == '\r')
    {
        if (!AT_q_flag) TestHarnessDCE_SendBlock((byte *)"\n\rOK\r\n", 6, &n);
        AT_off_state = AT_off_idle ;
    } break;
    case AT_off_ati:
        TestHarnessDCE_SendChar(c);
        if (c == '\r')
        {
            if (!AT_q_flag) TestHarnessDCE_SendBlock((byte *)"\n\rFreescale Modem Model DSCAT8037 Serial 0001 Version at003gb16\r\nV.22bis \r\nOK\r\n", 78, &n);
#ifdef STCK1_stackcheckSizeAllocated
    if ( STCK1_stackcheckSizeUsed() > STCK1_stackcheckSizeAllocated() )
        asm(debughlt);
#endif
        AT_off_state = AT_off_idle ;
    } break;
    case AT_off_atz:
        TestHarnessDCE_SendChar(c);
        if (c == '\r')
        {
            if (!AT_q_flag) TestHarnessDCE_SendBlock((byte *)"\n\rOK\r\n", 6, &n);
            AT_off_state = AT_off_idle ;
            AT_z_flag = TRUE ;
        } break;
    case AT_off_plus:
        TestHarnessDCE_SendChar(c);
        if (c == '\r')
        {
            if (!AT_q_flag) TestHarnessDCE_SendBlock((byte *)"\n\rOK\r\n", 6, &n);
            AT_off_state = AT_off_idle ;
            AT_z_flag = TRUE ;
        } else if (c == '0')
        {
            V21_Mode = TRUE ;
        } else if (c == '2')
        {
            V21_Mode = FALSE ;
        } // else if
        // case
    } // switch
} // if
} // function
3.1.4 AT_online()

When the modem is connected to another modem, there is no need to dial. All that can be done is to end
the call. One way provided to end the call is simply to hang up the phone, while keeping the data pump
running. This is the case for the ath command provided. The other way is the zap command, atz, which
both hangs up the call and totally resets the call from the application.

```c
#include <stdio.h>

void AT_online (void)
{
    unsigned int n;
    byte c;
    if (TestHarnessDCE_GetCharsInRxBuf())
    {
        tty_in_status = TestHarnessDCE_RecvBlock(&c,1,&n) ;
        switch (AT_on_state)
        {
            case AT_on_idle:
                if ((c=='a') || (c=='A'))
                {
                    AT_on_state = AT_on_a ;
                    TestHarnessDCE_SendChar(c) ;
                }
                break;
            case AT_on_a:
                if ((c=='t') || (c=='T'))
                {
                    AT_on_state = AT_on_at ;
                    TestHarnessDCE_SendChar(c) ;
                }
                else if ((c=='a') || (c=='A'))
                {
                    TestHarnessDCE_SendChar(c) ;
                }
                else AT_on_state = AT_on_idle ;
                break;
            case AT_on_at:
                TestHarnessDCE_SendChar(c) ;
                if (c== '\r')
                {
                    AT_on_state = AT_on_idle ;
                }
        }
    }
}
```
if (!AT_q_flag) TestHarnessDCE_SendBlock((byte *)"\nOK\r\n", 5, &n);
} else if ((c=='h') || (c=='H'))
{
    AT_on_state = AT_on_ath;
    h_parm = 0;
}
else if ((c=='o') || (c=='O'))
{
    AT_on_state = AT_on_ato;
}
else if ((c=='z') || (c=='Z'))
{
    AT_on_state = AT_on_atz;
}
break;
case AT_on_ath:
    TestHarnessDCE_SendChar(c);
    if (c == '0')
    {
        h_parm = 0;
    }
    if (c == '1')
    {
        h_parm = 1;
    }
    if (c == '\r')
    {
        if (h_parm) OffHook_SetVal(); else OffHook_ClrVa();
        AT_on_state = AT_on_idle;
        if (!AT_q_flag) TestHarnessDCE_SendBlock((byte *)"\n\r\n\r\n", 6, &n);
    }
    break;
case AT_on_ato:
    TestHarnessDCE_SendChar(c);
    if (c == '\r')
    {
        state_of_escape = Escape_idle_zero;
        AT_on_state = AT_on_idle;
        if (!AT_q_flag) TestHarnessDCE_SendBlock((byte *)"\n\r\r\n\r\n", 11, &n);
    }
    break;
case AT_on_atz:
    TestHarnessDCE_SendChar(c);
    if (c == '\r')
    {
        state_of_escape = Escape_idle_zero;
        AT_on_state = AT_on_idle;
        if (!AT_q_flag) TestHarnessDCE_SendBlock((byte *)"\n\r\n\r\n", 6, &n);
        AT_z_flag = TRUE;
    }
}
### 3.1.5 CPTDetCallback()

This is the callback function used in conjunction with dial tone detection.

```c
void CPTDetCallback (void *pCallbackArg, UWord16 return_value)
{
    Line_Tones = return_value ;
    if (pCallbackArg) return ;
}
```

### 3.1.6 TxCallbackV21()

This function is called when the V.21 data pump is being called from the main online data loop. It dumps the samples that it has formed by modulation of the bit it is sending. Note that __V21_H is not defined.

```c
#ifdef __V21_H

void TxCallbackV21 (void *pCallbackArg, Word16 *pSamples, UWord16 NumberSamples)
{
    Word16  i;
```
UWord16 num_words;
if (pCallbackArg != NULL)
{
    asm(debughlt);
    while(1){}
}
for ( i = 0; i < NumberSamples; i++)
{
    CodecTxBuffer[i] = pSamples[i];
}
num_words = 0;
do
{
    num_words += WriteAnalogTxData((CodecTxBuffer + num_words), NumberSamples - num_words);
}while (num_words < NumberSamples);
#endif

3.2 TXCallbackRoutine()

In the case of V.22bis, samples are delivered as the modem modulates them. This interface also indicates
the status of the modem connection in the case of a V.22bis or V.22 connection.

/*******************************************************/
/*
* Module: TXCallbackRoutine ()
*
* Description: This is a V22 transmit callback routine. This module is
called by V22bis library as and when it has samples
to be sent across to the remote modem.
*
* Returns: None
*
* Arguments: pCallbackArg -> Supplied by the user in the
*              v22bis_TXCallback structure; this value is
*              passed back to the user during the call
to the Callback procedure. pCallbackArg
typically points to context information
used by the user's callback procedure
(user has to write his/her own callback
function)
* Status - This is an enum containing the following fields
* V22BIS_1200BPS_CONNECTION_ESTABLISHED,
* V22BIS_2400BPS_CONNECTION_ESTABLISHED,
* V22BIS_CONNECTION_LOST,
* V22BIS_DATA_AVAILABLE,
* V22BIS_RETRAINING
*
* pSamples -> Pointer to the buffer containing 16-bit
* linear samples for transmission to the
* remote modem
*
* NumberSamples - Number of samples in the sample
* buffer pointer by pSamples
*
* Range Issues: None

A Low-Cost Soft Modem Using the Freescale Digital Signal Controller MC56F802x/3x Series, Rev. 0
Software Design Details

* * Special Issues: None *
* Test Method: interopans.mcp *
******************************************************************************/

void TXCallbackRoutine (void * pCallbackArg, v22bis_eStatus Status, 
                      Word16 * pSamples, UWord16 NumberSamples)
{
    Word16  i;
    UWord16 num_words;
    unsigned int n;
    /* Modem is in re-training mode, and hence, user should not call the transmitter to send 
       the data */
    if ( Status == V22BIS_RETRAINING)
    {
        v22bis_connection_established = FALSE;
        if (!AT_q_flag) TestHarnessDCE_SendBlock((byte *)&"\n\rTRAINING\r\n", 12, &n);
    }
    /* Connection established. This could happen in 2 ways. 
       During power-up, after the handshake this flag is set to 
       indicate the user that, the data can be sent to the remote modem. 
       During data transfer, modem can enter re-training mode due 
       to high noise on the channel. Once the re-training is over, 
       this flag is set to indicate that user can now send the 
       data. During re-training user should not call the transmitter 
       for sending the data */
    else if ( Status == V22BIS_1200BPS_CONNECTION_ESTABLISHED)
    {
        v22bis_connection_established = TRUE;
        rate_negotiated = V22BIS_1200BPS_CONNECTION_ESTABLISHED;
        if (!AT_q_flag) TestHarnessDCE_SendBlock((byte *)&"\n\rCONNECT 1200\r\n", 16, &n);
    }
    /* Above comment holds good here too */
    else if ( Status == V22BIS_2400BPS_CONNECTION_ESTABLISHED)
    {
        v22bis_connection_established = TRUE;
        rate_negotiated = V22BIS_2400BPS_CONNECTION_ESTABLISHED;
        if (!AT_q_flag) TestHarnessDCE_SendBlock((byte *)&"\n\rCONNECT 2400\r\n", 16, &n);
    }
    else if ( Status == V22BIS_DATA_AVAILABLE)
    {
        for  ( i = 0; i < NumberSamples; i++)
        {
            CodecTxBuffer[i] = pSamples[i];
        }
        num_words = 0;
        do
        {
            num_words += WriteAnalogTxData((CodecTxBuffer + num_words), NumberSamples - num_words);
        } while (num_words < NumberSamples);
    }
}
3.2.1 RxCallbackV21()

Unlike V.22bis, the V.21 RxCallback function is called each time just one bit is received. This makes it possible to perform the search for the character’s start bit in this function. After a zero bit is detected (start bit), then the next 8 bits are considered as data, and the following bit as a stop bit. However, this stop bit is optional in the implementation below. Notice that it is really looking for the next start bit as soon as the eight data bits are done. This makes operation at zero stop bits possible. Any number of stop bits may be used, as long as they are an integral number equal to or greater than zero.

```c
#ifndef __V21_H

/******************************************************************************
 *
 * Module: RxCallbackV21()
 *
 * Description: The demodulated bits are given to the user through this
 *              function once zero is detected
 *              and once for each subsequent bit that is detected.
 *              (assume async start 8 data no parity one stop) .. other formats may be supported
 *              by making modifications herein.
 *
 * Arguments: pCallbackArg - not used
 *            pChars - pointer to demodulated byte.
 *            NumberBytes - Number of bytes received/demodulated at Rx.
 *
*******************************************************************************/

#define V21_NUM_BITS 8  /* 8 bits per byte for 8N1 async format */

word StartBitRXd = FALSE;
unsigned char MsgByteRx1 = 0;
int BitCounterRx1 = V21_NUM_BITS;

void RxCallbackV21(void *pCallbackArg, char *pChars, UWord16 NumberBits)
{
    Word16 tempBit;

    if (NumberBits !=1)   {
        asm(debughlt);
        while(1){}
    }
    // assumption check
    if (pCallbackArg != NULL )   {
        asm(debughlt);
        while(1){}
    }
    // assumption check
    tempBit = *pChars & 0x01;        // The bit we are being delivered is here
    if (StartBitRXd)                 // ignore stop bits .. as many as may be present .. if any.
    {
        MsgByteRx1 >>= 1;               // Make way for the next bit to enter .. LSB's received
        tempBit <<= (V21_NUM_BITS - 1); // Seven
        MsgByteRx1 |= tempBit;          // collect the bit for the developing character
        BitCounterRx1--;
        if (BitCounterRx1 == 0)        // If byte is formed ...
        {
```

A Low-Cost Soft Modem Using the Freescale Digital Signal Controller MC56F802x/3x Series, Rev. 0

Freescale Semiconductor  61
Software Design Details

StartBitRXd = FALSE ;       // look for another start bit.. stop bits are OPTIONAL here.
BitCounterRx1 = V21_NUM_BITS; // Reinit. bit counter
*pModemRxWrt++ = MsgByteRx1;
if (pModemRxWrt>= &ModemRxBuffer[MODEM_RX_BUFF_SIZE])
{
    pModemRxWrt = ModemRxBuffer;
}
MsgByteRx1 = 0;           // Reset, for collecting next data byte
}
else
{
    if (tempBit == 0) StartBitRXd = TRUE ;
}
#endif

3.2.2 RXCallbackRoutine()

The V.22bis data pump puts the characters together for the application, so it is somewhat simple. This function is not called for each bit, but deals only in bytes. That is because this data pump performs the function of sync to async for the application.

/******************************************************************************
* Module: RXCallbackRoutine ()
* Description: This is a V22 recive callback routine. This module is
called by V22bis library as and when it receives
few bits .
*
* Returns: None
*
* Arguments: pCallbackArg -> Supplied by the user in the
* v22bis_TXCallback structure; this value is
* passed back to the user during the call
* to the Callback procedure. pCallbackArg
* typically points to context information
* used by the user's callback procedure
* (user has to write his/her own callback
* function)
* Status - This is an enum containing the following fields
* V22BIS_1200BPS_CONNECTION_ESTABLISHED,
* V22BIS_2400BPS_CONNECTION_ESTABLISHED,
* V22BIS_CONNECTION_LOST,
* V22BIS_DATA_AVAILABLE,
* V22BIS_RETRAINING
*
* pBits -> Pointer to the buffer containing the bits
* received / decoded
*
* NumberSamples - Number of bits received pointed by pBits
*
* Range Issues: None
*
* Special Issues: Instant response to lost carrier to speed lab testing.
/******************************************************************************/
3.2.3 \textbf{InitAnalogRxChannel()}

The sequence:

\begin{verbatim}
pAnalogRxWrite = NULL;
pAnalogRxRead = &AnalogRxBuffer[0];
\end{verbatim}

may appear strange. However, the NULL value is used to keep the ISR from being active with the FIFO.

The FIFO (first in first out) is a special kind of buffer called a circular buffer. When writing is done, the first pointer is used and advanced. When reading is done, the second pointer is used and advanced until it once again equals the first pointer. The pointers wrap around the end of the FIFO, so there is no first location.

This circularity is achieved only with software in this case, although this processor does implement such things in hardware. You will find this software in the assembly language expansions of the beans related to digital filters.
Because location zero is not included in the RAM locations that programs can use, the NULL address can have a special meaning. We take full advantage of this possibility in the following application example.

```c
void InitAnalogRxChannel(void)
{
    int i;
    for (i=0 ; i < RX_BUFFER_SIZE; i++)
        AnalogRxBuffer[i] = 0;
    pAnalogRxWrite = NULL;  // TURN OFF FIFO.. no writes to FIFO if no pointers untill first request comes in
    pAnalogRxRead = &AnalogRxBuffer[0]; // but initialize the input buffer pointer to start.
    PESL(QTIMER_A1, QT_WRITE_LOAD_REG, 0);  // 802x set for 7200 SPS.
}
```

### 3.2.4 InitAnalogTxChannel()

This function is another case of a circular FIFO implemented in C. Again NULL is used to keep the ISR from operating with the FIFO. If it is not using the FIFO, it instead takes the center value of the DAC and uses that. Also, the FIFO is initialized to the center value, which represents zero, or no signal out. The DAC deals with unsigned left-justified 12-bit hexadecimal values. So, 0x8000 means 800 on a scale of 0 to FFF, hexadecimal.

```c
void InitAnalogTxChannel(void)
{
    int i;
}
```
Software Design Details

for (i=0 ; i < TX_BUFFER_SIZE; i++)
    AnalogTxBuffer[i] = 0x8000;  // mid voltage for DAC left just.
pAnalogTxRead = NULL;              // 1st this holds off the ISR reading till first deposit complete
pAnalogTxWrite = &AnalogTxBuffer[0]; // 2nd this allows first deposit to be in first part of fifo.
}

3.2.5 InitPoorMansCodec()

These simply call both of the functions above, InitAnalogRxChannel() and InitAnalogTxChannel().

/**********************************************************************
* Module: InitPoorMansCodec()
* Description: This function initializes the TX and RX FIFO set.
* Returns: None
* Arguments: None
* Range Issues: None
* Special Issues: None
***********************************************************************/
void InitPoorMansCodec(void)
{
    InitAnalogRxChannel(); // init ADC channel FIFO
    InitAnalogTxChannel(); // init DAC channel FIFO
}

3.2.6 ReadAnalogRxData()

The ADC is the origin of this data. It is used to sample the data provided here.

/**********************************************************************
* Module: ReadAnalogRxData()
* Description: This function reads the samples from the AnalogRxBuffer
* and returns them back to the calling function. Will not block
* but may return nothing at all.
* Returns: Number of Samples Read
* Arguments: pRxBuffer - Buffer to store the samples in
*            Size - Number of samples to read
* Range Issues: None
* Special Issues: None
***********************************************************************/
UWord16 ReadAnalogRxData(Word16 * pRxBuffer, UWord16 Size)
Software Design Details

```c
{  UWord16 SamplesRead = 0;
  if ((Size != 0) && pAnalogRxWrite == NULL) pAnalogRxWrite = &AnalogRxBuffer[0]; // allow
ADC ISR to input to FIFO check 0
  else
  {  
    while ((pAnalogRxRead != pAnalogRxWrite) && (SamplesRead < Size)) // while not empty 
      and stuff
      {  
        *pRxBuffer++ = *pAnalogRxRead++;
        SamplesRead++;
        if (pAnalogRxRead >= &AnalogRxBuffer[RX_BUFFER_SIZE])
        {  
          pAnalogRxRead = &AnalogRxBuffer[0];
        }
      }
  }
  return(SamplesRead);
}
```

### 3.2.7 WriteAnalogTxData()

Analog data is written to the DAC. The DAC signal is then fed to a low-pass filter to achieve a baseband signal limited to 4000 hertz. Any glitches produced by the DAC have an energy low enough not to affect the baseband signal. There is no restriction on amplitude, and rounding is used to get maximum resolution. If the FIFO is inactive when this function is called, it will activate the FIFO for the ISR. It does this with the statement:

```c
pAnalogTxRead = &AnalogTxBuffer[TX_BUFFER_SIZE-NUMSAMPLES];
```

The entire function follows:

```c
/*******************************************************************************
* Module: WriteAnalogTxData()
* Description: This function writes the samples to the AnalogTxBuffer.
* For each sample, four samples are actually transmitted.
* The extra three samples are calculated by the DAC
* between the previous sample and the new sample.
* This function also checks for and avoids transmitter overrun errors.
* Returns: Number of Samples Written
* Arguments: pTxBuffer - Buffer containing the samples to be written
* Size - Number of samples to be written
* Range Issues: None
* Special Issues:
* To sync the actual output startup to the first write to the FIFO,
* to minimize handshake delays,
* pAnalogTxRead is initialized to zero when the transmitter is not in use.
*******************************************************************************/
```
Use is initiated here by setting this pointer.

Assumption: write less in one call here than
the size of the buffer.

******************************************************************************/
UWord16 WriteAnalogTxData(Word16 * pTxBuffer, UWord16 Size)
{
    UWord16 SamplesWritten = 0;
    while ((pAnalogTxWrite != pAnalogTxRead) && // fifo not full
            (SamplesWritten < Size)) // samples left to send
    {
        // generate left justified unsigned rounded dac value
        *pAnalogTxWrite++ = ((unsigned int)((*(pTxBuffer++) >> Blue_DAC_Scale)
                        + BLUE_DAC_OFFSET)) << 1;
        SamplesWritten ++;
        if (pAnalogTxWrite >= &AnalogTxBuffer[TX_BUFFER_SIZE])
            pAnalogTxWrite = &AnalogTxBuffer[0]; // wraps the fifo
    }
    // Check to see if this FIFO is being inaugurated.. if this is the first write.
    if ((SamplesWritten !=0 ) && pAnalogTxRead == NULL)
        pAnalogTxRead = &AnalogTxBuffer[TX_BUFFER_SIZE-NUMSAMPLES]; // set back one frame.
    // If you do not set it back one frame, the code will stall when it tries to write
    // and the FIFO is full. This does perhaps increase the response time of the modem
    // including during handshaking, but the tradeoff is that writes to the FIFO will not
    // ever have to wait during V22bis operation. DTMF still relies on the full-check.
    // Turn on the reading of the FIFO by the ISR. ISR is hands off zeros.
    // No need to start if no samples are written yet.
    // Location zero in Ram is not allowed to be allocated.
    // This means a NULL valued pointer can have a special meaning:
    // Not defined yet. This means the sink of this fifo is not yet
    // defined. It will not define until the source is manifest.
    // the source is manifest when the data is written and the condition above
    // is satisfied.
    return(SamplesWritten);
}

3.2.8 ADC_EOS_INT()

Both the DAC and the ADC are serviced in one ISR. This has several advantages:

- The timing is fixed, so that noise can be constant.
- Only one ISR overhead is needed for TX and RX functions.
- FIFOs will have a fixed relationship regarding fullness, with respect to the TX and RX FIFOs.

---

A Low-Cost Soft Modem Using the Freescale Digital Signal Controller MC56F802x/3x Series, Rev. 0
Software Design Details

* or 50.5 μs window. So we are working with signed numbers here.
* Note the new syntax available with CW 8.2 to make it easy to work
* with an array of samples. During this code execution, the ADC
* is waiting for a new trigger from SYNC from timer 3 OFLAG positive
* transition. 1/7200 is 138.8888888 us. 1/8000 is 125 us.
* To save overhead, the DAC is also directed in this ISR.
* The DAC step clock is locked to the ADC sample clock.
***********************************************************************/
// ADC is at F080 on the mc56f8037.
// Result registers begin at C and go to 13 hex on the mc56F802x3x.
volatile int The_ADC_Result1 : 0xF08C ;
volatile int The_ADC_Result2 : 0xF08D ;
volatile int The_ADC_Result3 : 0xF08E ;
volatile int The_ADC_Result4 : 0xF08F ;
volatile int The_ADC_Result5 : 0xF090 ; // this sample is chosen
volatile int The_ADC_Result6 : 0xF091 ;
volatile int The_ADC_Result7 : 0xF092 ;
volatile int The_ADC_Result8 : 0xF093 ;
#pragma interrupt alignsp saveall
void ADC_EOS_INT (void)
/////////////////////////////////ADC///////////////////
{  
PESL(ADC, ADC_CLEAR_STATUS_EOSI, NULL); // Must clear regardless of FIFO or will reenter
  
if ((pAnalogRxWrite != NULL) && (pAnalogRxRead != NULL)) // check if FIFO OFF
  {
*pAnalogRxWrite++ = (Word16)  periphMemRead( (Word16 *) 0xF090);
  // uncomment the assert if debugging is required only.
  // assert (pAnalogRxWrite != pAnalogRxRead); // if so, overrun DSP!! empty after write
  // reduce application cycles.
  if (pAnalogRxWrite >= &AnalogRxBuffer[RX_BUFFER_SIZE]) pAnalogRxWrite =
  &AnalogRxBuffer[0];
  }
}

Note the DAC code. It sets up the DAC to run for four adjusts of the output at a 24400 per second rate.

/////////////////////////////////DAC///////////////////
if ( (pAnalogTxRead != 0) && (pAnalogTxWrite != 0))  // reading from the FIFO is inhibited
by setting pointer zero.
{
  This_Dac_Value = *pAnalogTxRead++ ;
  if (This_Dac_Value > Last_Dac_Value)
  {
    This_Delta_Value = ((This_Dac_Value - Last_Dac_Value) >> 2) & 0xfff0 ;
PESL(DAC0, DAC_DOWN_COUNTING, DAC_DISABLE)
PESL(DAC0, DAC_UP_COUNTING, DAC_ENABLE);
    Last_Dac_Value = Last_Dac_Value + (This_Delta_Value << 2) ;
  }
else
  {
    This_Delta_Value = ((Last_Dac_Value - This_Dac_Value) >> 2) & 0xfff0 ;
PESL(DAC0, DAC_UP_COUNTING, DAC_DISABLE);
PESL(DAC0, DAC_DOWN_COUNTING, DAC_ENABLE);
    Last_Dac_Value = Last_Dac_Value - (This_Delta_Value << 2) ;
  }
DAC_Analog_TX_SetStep ( &This_Delta_Value );
if(pAnalogTxRead >= &AnalogTxBuffer[TX_BUFFER_SIZE])
{  
    pAnalogTxRead = &AnalogTxBuffer[0];  
}

else // DAC data flush.. do not use queue.. just hold center
{
    This_Dac_Value = 0x8000 ;
    if (This_Dac_Value > Last_Dac_Value)
    {
        This_Delta_Value = ((This_Dac_Value - Last_Dac_Value) >> 2) & 0xfff0 ;
        PESL(DAC0, DAC_DOWN_COUNTING, DAC_DISABLE)
        PESL(DAC0, DAC_UP_COUNTING, DAC_ENABLE);
        Last_Dac_Value = Last_Dac_Value + (This_Delta_Value << 2) ;
    }
    else
    {
        This_Delta_Value = ((Last_Dac_Value - This_Dac_Value) >> 2) & 0xfff0 ;
        PESL(DAC0, DAC_UP_COUNTING, DAC_DISABLE);
        PESL(DAC0, DAC_DOWN_COUNTING, DAC_ENABLE);
        Last_Dac_Value = Last_Dac_Value - (This_Delta_Value << 2) ;
    }
    DAC_Analog_TX_SetStep  ( &This_Delta_Value );
}
}

#pragma interrupt reset

3.3 modem.h

There are four little state machines defined here, RING_STATE, AT_OFF_STATES, AT_ON_STATES, and ESCAPEMENT_STATES. The AT_OFF_STATES machine is active when the main program does not have a call established. The various commands that it supports are spelled out in the state names.

AT_ON_STATES machine supports a more limited set of commands for use after escaping to the online command mode from the data state of the modem. The process of escaping from the online data state to the online command mode is achieved with the ESCAPEMENT_STATES machine, which enforces an idle period both before and after the “+++” escape sequence, before entering the online command mode.

There are also the scaling, offsetting, and rounding of the DAC defined here, as well as a way to make the output of the DTMF dialing differ in magnitude from that of the modem code.

    void AT_offline(void) ;
    void AT_online(void) ;

    #define ANS_ON ((int) ring_count_up + 20) // 20 is 2 seconds try to answer during middle of quite no ring period
    typedef enum  {ring_idle, ring_rang, ring_count_up, ring_set_ans_flag=ANS_ON}
        RING_STATE ;

    typedef enum  { AT_off_idle, AT_off_a, AT_off_at, AT_off_ata, AT_off_atd, AT_off_ati,
        AT_off_atq, AT_off_atz,AT_off_plus } AT_OFF_STATES ;

    typedef enum  { AT_on_idle, AT_on_a, AT_on_at, AT_on_ath, AT_on_ato, AT_on_atq,
        AT_on_atz} AT_ON_STATES ;

A Low-Cost Soft Modem Using the Freescale Digital Signal Controller MC56F802x/3x Series, Rev. 0

Freescale Semiconductor
typedef enum { Escape_idle_zero, Escape_pre_idle_1, Escape_p1, Escape_p2, Escape_p3, Escape_post_idle} ESCAPEMENT_STATES;

#define CALIBRATE
#define BLUE_DAC_SCALE_V22bis 0 // was 1. goodboy13 testing full excursion dac
#define BLUE_DAC_SCALE_DTMF 2
    // 2 assumes modem uses all available bits in its signed output.
    // If it comes short in this by n bits, then 2-n may be used.
    // To calibrate this scale, check the value coming out the DAC used
1/4 to 3/4 range.
    // This would be Vcc*1/4 to Vcc*3/4, or 400 to bff hex input to dac.
    // This should be the absolute maximum output, not the average, but
the peak over time.
    // Otherwise peaks will result in nonlinear hits, probably on the local RX.
#define BLUE_DAC_OFFSET 0x4004  // includes rounding .. becomes 8008.
    // The offset is added to the signed output to convert it to an
unsigned output.
    // The final shift is to replace what used to be the sign bit
with the MSB
    // because the DAC will take left justified data.

3.4 Events.c

A bean in Processor Expert has Properties, Methods, and Events. The Events.c file is generated by Processor Expert. Code is added as needed by the application author, in locations flagged for such additions by comments provided by Processor Expert.

/**
** Filename : Events.C
** Project : modem
** Processor : 56F8367
** Beantype : Events
** Version : Driver 01.02
** Compiler : Metrowerks DSP C Compiler
** Date/Time : 4/5/2005, 2:52 PM
** Abstract :
**     This is user's event module.
**     Put your event handler code here.
**     Contents :
**     TwentythSecInt_OnInterrupt - void TwentythSecInt_OnInterrupt(void);
**     TestHarnessDCE_OnError - void TestHarnessDCE_OnError(void);
**     TestHarnessDCE_OnRxChar - void TestHarnessDCE_OnRxChar(void);
**     TestHarnessDCE_OnTxChar - void TestHarnessDCE_OnTxChar(void);
**     TestHarnessDCE_OnFullRxBuf - void TestHarnessDCE_OnFullRxBuf(void);
**     TestHarnessDCE_OnFreeTxBuf - void TestHarnessDCE_OnFreeTxBuf(void);
**     AD1_OnEnd - void AD1_OnEnd(void);
**
** (c) Copyright UNIS, spol. s r.o. 1997-2004
** UNIS, spol. s r.o.
** Jundrovcka 33
** 624 00 Brno
** Czech Republic
** http  : www.processorexpert.com
** mail  : info@processorexpert.com
3.5 TwentythSecInt_OnInterrupt()

This twentieth-second timer is used for mundane timing of events relating to ring detect and the AT command set. It could be supplanted by tapping into another timer that is being used. However, for the sake of clarity, it uses its own bean and its own hardware timer.

This function does two useful things. It operates the ring-detect state machine, designed to trigger an answer exactly between the first ring and the second ring, and it counts up time in the CountTime variable for the AT command set, to reference passing time.
/**
* Event       :  TwentythSecInt_OnInterrupt (module Events)
* Description:
*     When a timer interrupt occurs this event is called (only
*     when the bean is enabled - "Enable" and the events are
*     enabled - "EnableEvent").
* Parameters : None
* Returns     : Nothing
*/

word TimeToggle ; // convert 50MS to 100MS time base

void TwentythSecInt_OnInterrupt(void)
{
    TimeToggle = (word) (1 - TimeToggle);
    if (TimeToggle)
    {
        CountTime++ ;
        RingDetect_GetCounterValue(&ring_pulse_count);
        #ifdef MODEM_DEBUG_RING
        ring_history[ring_index++] = ring_pulse_count ;
        if (ring_index == 100) ring_index = 1;
        #endif
        if (ring_state == ring_idle)
        {
            if (ring_pulse_count >= ring_pulse_min)
            {
                ring_state = ring_rang;
            }
        } else if (ring_state == ring_rang)
        {
            if (ring_pulse_count == 0)
            {
                ring_state = ring_count_up;
            }
        } else if (ring_state >= ring_count_up && ring_state < ring_set_ans_flag)
        {
            ring_state++;
        } else
        {
            ring_state = ring_idle;
        }
        #define  DEBUG_NOP_RING_DETECT // use to diable ring det.
        #ifndef  DEBUG_NOP_RING_DETECT
        is_time_to_shake = TRUE; /* to be cleared by main program or subroutine of it */
    
    
    
}

A Low-Cost Soft Modem Using the Freescale Digital Signal Controller MC56F802x/3x Series, Rev. 0

Freescale Semiconductor
3.5.1 TestHarnessDCE_OnError()

Although they are not expected, errors on the test fixture interface are counted.

```c
/*
**  This event is called when a channel error (not the error
**  returned by a given method) occurs. The errors can be
**  read using <GetError> method.
** Parameters : None
** Returns     : Nothing
**
*/
void TestHarnessDCE_OnError(void)
{
  byte rc ;
  TestHarnessDCEError * ptr ;
  ptr = &the_errors ;
  rc = TestHarnessDCEGetError(ptr);
  //asm(debughlt);
  error_cnt++ ;
}
```

3.5.2 TestHarnessDCE_OnRxChar()

No action is needed by the application program for each character received. Check the bean; it is
configured to use buffers and interrupts, so it does all of that work for you.

```c
/*
**
**
**
**
**
**
*/
```
Software Design Details

** Parameters : None
** Returns : Nothing
** ================================================================
*/
#pragma interrupt called /* Comment this line if the appropriate 'Interrupt preserve
registers' property */
    /* is set to 'yes' (#pragma interrupt saveall is generated before
the ISR) */
void TestHarnessDCE_OnRxChar(void)
{
    /* Write your code here ... */
}
/*

3.5.3 TestHarnessDCE_OnTxChar()

Again, we are using interrupts here. See the bean.

** ===================================================================
** Event       :  TestHarnessDCE_OnTxChar (module Events)
** ===================================================================
** From bean   :  AS1 [AsynchroSerial]
** Description :
**     This event is called after a character is transmitted.
** Parameters  : None
** Returns     : Nothing
** ===================================================================
*/
#pragma interrupt called /* Comment this line if the appropriate 'Interrupt preserve
registers' property */
    /* is set to 'yes' (#pragma interrupt saveall is generated before
the ISR) */
void TestHarnessDCE_OnTxChar(void)
{
    /* Write your code here ... */
}

3.5.4 TestHarnessDCE_OnFullRxBuf()

We have flow control for the PC to modem direction. We trust the PC is faster in the other direction.

/*
** ===================================================================
** Event       :  TestHarnessDCE_OnFullRxBuf (module Events)
** ===================================================================
** From bean   :  AS1 [AsynchroSerial]
** Description :
**     This event is called when the input buffer is full.
** Parameters  : None
** Returns     : Nothing
** ===================================================================
*/
#pragma interrupt called /* Comment this line if the appropriate 'Interrupt preserve
registers' property */
    /* is set to 'yes' (#pragma interrupt saveall is generated before
the ISR) */
3.5.5 TestHarnessDCE_OnFreeTxBuf()

As in the previous section, this is left as a stub because the test harness only needs flow control in one direction. Should the user need bidirectional flow control, these two stubs may be fleshed out.

```c
void TestHarnessDCE_OnFreeTxBuf(void)
{
    //asm(debughlt);
}
```

3.6 v21_processA.c

The V.21 bean was designed to support synchronous mode only. The bean generates a file called v21_process.c. This file was replicated/modified for support of asynchronous character structure and renamed with the suffix A as v21_processA.c.

This new file contains functions v21TxProcessA(), v21RxProcessA(), and getDemodByteA(). The corresponding functions in v21_process.c are not utilized for async in this application: v21TxProcess(), v21RxProcess(), and getDemodByte().

These functions are part of the default application program interface provided by the bean’s methods. In order to support an alternative character structure, this file is duplicated under a related name specialized for the desired character structure: v21_processA.c. Some of the functions in this file were modified to support asynchronous character format over the line. The format includes one start bit, set first, followed by eight data bits, sent least-significant bit first, and then one stop bit, sent last, for a total of ten bits per character. While this is more than the eight bits per character used by sync mode, it does not require sync characters, 0x7e, to be sent. A space bit is just a zero, a stop bit is just a one. Another term for the zero is space; another term for stop bit is mark. Any number of marks, or stop bits, may follow a character, unlike synchronous mode where there is no space between the 8-bit characters. Sync is used for FAX and V.8bis. Most simple applications use async.
For security applications, the character structure may be quite unusual. It may even include reversing the start and stop bits, and the order of the data bits. Such changes can be made by simply changing the programming in this file.

The following sections outline the changes made to support 8n1 async (one start, eight data, one stop).

### 3.6.1 v21TxProcessA()

In this case, the routine v21TxProcessA() is modified to initialize BitCounterTx for the larger character structure of ten bits. It is then checked so that the start bits and stop bits are sent out, as well as the original data bits sent out in the async format.

```c
#ifdef __V21_H
    /**************************************************************************
    * (c) Freescale Semiconductor
    * 2004 All Rights Reserved
    *
    **************************************************************************/  
    * File Name: v21_process.c
    *
    * Description: Includes function V.21 Transmitter and Receiver.
    *
    * Modules Included: v21TxProcessA ()
    *                   v21RxProcessA ()
    *                   v21TxSamplesAmplifyA ()
    *
    * Author: Prasad N. R.
    *
    * Date: 09 Aug 2001
    *
    **************************************************************************/
    
    #include "v21.h"
    #include "v21_api.h"
    #include "arch.h"

    /**************************************************************************
    * Module: v21TxProcessA ()
    *
    * Description: Transmits data bytes using CPFSK modulation. Every
    *              baud will have 24 samples. Outputs start, 8 data, stop.
    *
    * Returns: V21_TX_BUSY - Indicates that Tx. is still transmitting data.
    **************************************************************************/
```
V21_TX_FREE - Indicates that Tx. has transmitted data.

Arguments: pV21 - a pointer to v21_sHandle structure.
            pBytes - a pointer to data bytes to be transmitted.
            NumBytes - Number of data bytes to be transmitted,
                        pointed to by pBytes.

Range Issues: None

Special Issues: 1. pBytes should not be destroyed
                2. if NumBytes is zero, just send mark without status change.

********************************************************************

Result v21TxProcessA (v21_sHandle *pV21, unsigned char *pBytes,
                      UWord16 NumBytes)
{
    bool SatBit = false; /* Variable to remember old saturation bit */
    UWord16 mod_bit;

    mod_bit = 1;           // mark or stop bit
    /* Reset the saturation mode bit */
    SatBit = archGetSetSaturationMode (SatBit);
    if( NumBytes != 0 )
    {
        /* Initialize byte counter */
        if ((pV21->CountStatusTx & V21_BYTE_COUNT_INIT_TX) == V21_BYTE_COUNT_INIT_TX)
        {
            pV21->ByteCounterTx = 0;
            pV21->CountStatusTx = (pV21->CountStatusTx) & V21_BYTE_COUNT_RESET_TX;
        }

        /* Initialize bit counter */
        if ((pV21->CountStatusTx & V21_NEW_BYTE_BIT_TX) == V21_NEW_BYTE_BIT_TX)
        {
            pV21->BitCounterTx = V21_NUM_BITS + 2;  // add one for the start bit, one for the
                                                     // stop bit
            pV21->CountStatusTx = (pV21->CountStatusTx) & V21_NEW_BYTE_BIT_RESET_TX;
        }

        /* Get the bit to be modulated */
        if ( pV21->BitCounterTx == (V21_NUM_BITS + 2))
        {
            mod_bit = 0 ; // start bit.. then lsb..
        }
        else if ( pV21->BitCounterTx == 1)
        {
            mod_bit = 1 ; // finally, stop bit.
        }
        else
        {
            mod_bit = (unsigned int)(pBytes[pV21->ByteCounterTx]) >> (V21_NUM_BITS + 1 -
                               pV21->BitCounterTx);
            mod_bit = mod_bit & V21_GET_BIT_MASK; /* Get LSB */
        }
    }

    /* Get the samples for the bit to be modulated */
Software Design Details

v21Mod (mod_bit, pV21->CpFskSamples, pV21);

/* Amplify the signal to be transmitted unless during quite part of handshake*/
v21TxSamplesAmplifyA (pV21->CpFskSamples, pV21->Gain);

/* Give the samples to the user through Tx callback */
(* (pV21->pTxCallback->pCallback)) (pV21->pTxCallback->pCallbackArg,
pV21->CpFskSamples,
V21_SAMPLES_PER_BAUD);

if (NumBytes != 0 )
{
    /* Decrement the bit counter */
    (pV21->BitCounterTx)--;
    if (pV21->BitCounterTx == 0)
    {
        /* Restore status value */
        pV21->CountStatusTx |= V21_NEW_BYTE_BIT_TX;

        /* Increment the byte counter */
        (pV21->ByteCounterTx)++;
        if (pV21->ByteCounterTx == NumBytes)
        {
            /* Restore status value */
            pV21->CountStatusTx |= V21_BYTE_COUNT_INIT_TX;
            archGetSetSaturationMode (SatBit);
            return (V21_TX_FREE);
        }
    }
}
archGetSetSaturationMode (SatBit);
if (NumBytes !=0)
{
    return (V21_TX_BUSY);
}
else
{
    return (V21_TX_FREE) ;
}

3.6.2 v21RxProcessA()

The function v21RxProcessA() is modified from the original v21RxProcess(). It calls getDemodByteA() instead of getDemodByte(). That is the only change made to this function. The change is required to support the asynchronous mode of the V.21 modem.

/****************************************************************************
 *
 * Module: v21RxProcessA ()
 *
 * Description: Receives samples and demodulates to get back the transmitted characters. The receiver works on a baud of samples (24).
 *
****************************************************************************/
Result v21RxProcessA (v21_sHandle *pV21, Word16 *pSamples, UWord16 NumSamples) {
    Int16 NumBauds;
    UInt16 Clen, tempLen;
    Int16 i, j, k;
    bool SatBit = false; /* Variable to remember old saturation bit */

    /* Reset the saturation mode bit */
    SatBit = archGetSetSaturationMode (SatBit);

    Clen = (unsigned int)pV21->ContextLen; /* Copied here for frequent use */
    if ((Clen + NumSamples) >= V21_SAMPLES_PER_BAUD) {
        /* First complete the context buffer frame */
        tempLen = V21_SAMPLES_PER_BAUD - Clen;
        for (i = 0; i < tempLen; i++) {
            pV21->ContextBufRx[Clen+i] = pSamples[i];
        }

        pV21->p_samples_buf_ptr = pV21->ContextBufRx; /* Set up the pointer */
        v21Rxctrl (pV21); /* Demodulate */

        /* Register first-carrier-detection and first-zero-cross detection */
        if ((pV21->syncFlag == 1) && (pV21->v21_flags.cdbit == 1) &&
            (pV21->first_zero_cross == 1)) {
            pV21->syncFlag = 0; /* Reset sync flag for ever */
            pV21->StatusFlagRx = V21_CD_FLAF_PREV_RX + V21_FIRST_ZERO_CROSS_RX;
        }

        /* We say carrier is lost only when we have detected the carrier before losing it ... */
        if ((pV21->v21_flags.cdbit == 0) &&
            ((pV21->StatusFlagRx & V21_CD_FLAF_PREV_RX) == V21_CD_FLAF_PREV_RX)) {
            archGetSetSaturationMode (SatBit);
            return (V21_RX_CARRIER_LOST);
        }
    }
}

/* Process the demodulated bits */
getDemodByteA (pV21);

/* Next, process the remaining frames */
k = (int)tempLen;
tempLen = NumSamples - tempLen;
NumBauds = (int)tempLen / V21_SAMPLES_PER_BAUD;
tempLen = tempLen - NumBauds * V21_SAMPLES_PER_BAUD;

for (i = 0; i < NumBauds; i++)
{
    /* Process the full-length frames first */
    pV21->p_samples_buf_ptr = &pSamples[k+i*V21_SAMPLES_PER_BAUD];
    v21Rxctrl (pV21); /* Demodulate */

    /* Register first-carrier-detection and first-zero-cross * detection */
    if ((pV21->syncFlag == 1) && (pV21->v21_flags.cdbit == 1) &&
        (pV21->first_zero_cross == 1))
    {
        pV21->syncFlag = 0; /* Reset sync flag for ever */
        pV21->StatusFlagRx = V21_CD_FLAF_PREV_RX + V21_FIRST_ZERO_CROSS_RX;
    }

    /* We say carrier is lost only when we have detected * the carrier before losing it ... */
    if ((pV21->v21_flags.cdbit == 0) &&
        ((pV21->StatusFlagRx & V21_CD_FLAF_PREV_RX) == V21_CD_FLAF_PREV_RX))
    {
        archGetSetSaturationMode (SatBit);
        return (V21_RX_CARRIER_LOST);
    }

    /* Process the demodulated bits */
    getDemodByteA (pV21);
}

/* Store the remaining samples in context buffer */
pV21->ContextLen = (short)tempLen; /* Update context len */
tempLen = (unsigned int)k + i * V21_SAMPLES_PER_BAUD;
i = 0;
for (j = (int)tempLen; j < NumSamples; j++)
{
    pV21->ContextBufRx[i++] = pSamples[j];
}
archGetSetSaturationMode (SatBit);
return (V21_RX_PASS);

else
{
    /* If there are no enough samples to process, save all * the samples into context buffer */
    for (j = 0; j < NumSamples; j++)
    {
        pV21->ContextBufRx[Clen+j] = pSamples[j];
    }
3.6.3 v21TxSamplesAmplifyA()

This function was not modified, but a renamed function was included for possible future modifications. The user-specified gain is important in this case, because the range of the analog signal is limited. It should be adjusted as high as possible, so that the range of the analog signal is completely used. If this is not done, quantization error will increase. Adjustment of the final output level to the –10 dBm range is done in the hardware. This preserves the most information in the signal. This function is only needed for V.21.

```assembly
asm void v21TxSamplesAmplifyA (Word16 *pSamples, Word16 Gain) {
  do  
  #V21_SAMPLES_PER_BAUD,_gain_loop
    move.w  x:(r2),x0                   /* samples' buffer */
    mpyr    x0,y0,a                     /* a = sample x gain */
    move.w  a,x:(r2)+                    /* Store the result in the same buffer. */
  _gain_loop:                               /* endfor */
    rts
  
}
```
3.6.4  getDemodByteA()

This function was modified to support the 10-bit async structure by simply not looking for the sync character used for the synchronous mode. That sync character was 0x7e. After this was done, the user’s application is called on every received bit that follows an initial space condition. It is then up to the user’s callback function to assemble the bits into characters according to the “start, eight data, stop asynchronous” format.

Applications using the sync mode would be FAX and V.8bis.

```c
void getDemodByteA (v21_sHandle *pV21)
{
    Word16 tempBit = 0;

    if (pV21->v21_rx_decision_length == 2)
    {
        /* Get the first demodulated bit out of 2 bits */
        tempBit = (pV21->v21_rxdemod_bits & 0x0003) >> 1;

        /* Give the demodulated bit to the user through Rx callback */
        (*(pV21->pRxCallback->pCallback)) (pV21->pRxCallback->pCallbackArg,
            (char *) &(tempBit), 1);

        /* Recover the second bit. */
        /* First get the 2nd demodulated bit out of 2-bits */
        tempBit = pV21->v21_rxdemod_bits & 0x0001;

        /* Give the second demodulated bit to the user through Rx callback */
        (*(pV21->pRxCallback->pCallback)) (pV21->pRxCallback->pCallbackArg,
```
4 Conclusion - System Performance

The modem is fully capable of error-free operation over average telephone lines in the USA. Modem testing indicates that the modem meets the performance standards required for V.22bis operation. File transfers were easily accomplished without incident.

4.1 Test Setup

4.1.1 Routine File Transfer Testing

For routine file transfer setup, the TAS series 2 was configured with the script as below. This depicts the average USA line:
/ad,s03=1,s07=1,c3/
/exch,bal=1/
/file:cseq=usa1/
/io,i-100,l-230,r-100,t-230/
/ad,i3/
/gd,w17,x00,y16,z00/
/rn,1320,s1/
/nl,q520,c500,m0,x1,y1/
/fs,f+1250,m0,s1/
/pj,10364,f1200,w0,s1/

The Hayes modem was simply configured with factory defaults:
ATZ
For a faster connect, on the Hayes modem, you may use:
ATS37=6
AT\n1
Conclusion - System Performance

AT&C0
This will limit line rate to 2400 Characters per second, turn off the compression and use buffer mode.

4.1.2 Bit Error Rate Testing

The bit error rate test (BERT) run was at 2400 bits per second, V.22bis mode.

White noise was used for the impairment. The contribution of the line to the tests are their phase magnitude responses.

The Hayes equipment was on the A equipment side, the UUT on the B equipment side.

Because the V.22bis contains a built-in scrambler, modem BERT performance was measured as a function of the number of error characters that were received when no characters were being transmitted. A test frame of seven minutes or about one million bits was used for each test. One synchronous bit error results in two asynchronous character errors, because it is seen as a start bit. The idle line condition is marking, or one.

The AGC feature of the TAS series 2 was used to align signal levels in the digital portion of the TAS series 2.

The units used were a Hayes Accura V.92 modem and the UUT (unit under test), which was the MC56F8037EVM with the LCMDC.

The test was done as follows:
/ad,s03=1,s07=1,c3/
/file:cseq=usa1/
/io,i-100,l-230,r-100,t-230/
/ad,i3/
/gd,w17,x17,y16,z16/

Connect. The “gd” command calls for EIA B Standard Gain Characteristics and EIA 2 Standard Group Delay Characteristics on lines to both modems. Further details of these commands are available in the TAS Series 2 Telephone Network Emulator Operations Manual.

Do agc:
/io,a1/
/io,q1/

Hang up.

Connect again.

Count errors.

Add white noise.
Repeat with more and more noise until one bit error in seven minutes. This represents the SNR that the modem can be used with only one bit error in a million, on average.

4.2 Bit Error Rate Test Results

The line impairment was present in both lines, from the central office simulation and to the central office simulation. This resulted in twice the dynamic range requirement imposed by just using one line impairment for one of the legs of the call. The modem operated down to a 16dB SNR with error rate of one bit in a million using the above test configuration on the TAS Series 2.

With no noise, the soft modem operated well down to -51dBm.

The performance of this V.22bis modem in the presence of noise is indicated by the bit error rate test result; the result indicates it will function on the public switched telephone network, as expected.

4.3 Memory Utilization on the MC56F8037

4.3.1 Summary Memory Utilization

```plaintext
# Memory map:

tabular v_addr  p_addr  size     name
0000F000 0000F000 00000000 .x_Peripherals
00000000 00000000 00000004 .p_Interruptsboot
00000000 00000000 00000080 .p_Interrupts
00000080 00000080 00003246 .p_Code
00000001 00000001 00000B1F .x_Data
00000D00 00000D00 00000300 .x_DynMem
000032C6 00000001 000007AF .p_flash_ROM_data
00008000 00008000 00001000 .p_internal_RAM
```

4.3.2 Complete Load Map

```plaintext
# Link map of F_EntryPoint

# .interrupt_vectorsboot

#>00000000 F_vector_addr (linker command file)
00000000 00000004 interrupt_vectorsboot.text F_vectboot(Vectors.c)

# .interrupt_vectors

00000000 00000080 interrupt_vectors.text F_vect(Vectors.c)

# .ApplicationCode
```

A Low-Cost Soft Modem Using the Freescale Digital Signal Controller MC56F802x/3x Series, Rev. 0
Conclusion - System Performance

`#>00000080   F_Pcode_start_addr (linker command file)
00000080  00000057 .text   F_EntryPoint(Cpu.c)
000000D7  00000007 .text   FCpu_Interrupt(Cpu.c)
000000DE  0000010B .text   FPE_low_level_init(Cpu.c)
000001E9  00000473 .text   Fmain(modemblue.c)
0000065C  000001A6 .text   FAT_offline(modemblue.c)
00000802  00000108 .text   FAT_online(modemblue.c)
000009A0  0000000B .text   FCPTDetCallback(modemblue.c)
000009B4  0000001D .text   FInitAnalogRxChannel(modemblue.c)
000009D4  00000019 .text   FInitAnalogTxChannel(modemblue.c)
000009EF  00000005 .text   FInitPoorMansCodec(modemblue.c)
00000A1D  00000108 .text   FAT_offline(modemblue.c)
00000A27  0000003B .text   FADC_EOS_INT(modemblue.c)
00000B33  0000000D .text   FRingDetect_Init(RingDetect.c)
00000B40  00000066 .text   FRingDetect_ResetCounter(RingDetect.c)
00000B46  00000006 .text   FRingDetect_ResetCounter(RingDetect.c)
00000B4A  0000000A .text   FRingDetect_GetCounterValue(RingDetect.c)
00000B50  0000001A .text   FRingDetect_Init(RingDetect.c)
00000B6A  00000044 .text   FBitIO__PutVal(OffHook.c)
00000B7E  00000014 .text   FTELL1_CPTDetCreate(TELL1.c)
00000C0D  00000014 .text   FTELL1_CPTDetInit(TELL1.c)
00000C21  0000000F .text   FTELL1_CPTDetection(TELL1.c)
00000C30  0000002E .text   FTELL1_CPTDetDestroy(TELL1.c)
00000C5E  00000020 .text   FMEM1_Init(MEM1.c)
00000C77  00000014 .text   FmemMallocEM(mem.c)
00000C92  0000001C .text   FmemMallocAlignedEM(mem.c)
00000CAE  00000014 .text   FmemFreeEM(mem.c)
00000CC2  0000001B .text   FmemIsAligned(mem.c)
00000CDD  0000000B .text   FmemProtect(mem.c)
00000CE8  0000005E .text   FMergeFree(mem.c)
00000D46  00000064 .text   FSplitBlock(mem.c)
00000DAA  00000055 .text   FSplitBlockRev(mem.c)
00000E5F  00000054 .text   FmemMalloc(mem.c)
00000F30  00000038 .text   FmemFree(mem.c)
00000FA8  000000AF .text   FmemMalloc(mem.c)`
Conclusion - System Performance

A Low-Cost Soft Modem Using the Freescale Digital Signal Controller MC56F802x/3x Series, Rev. 0

Freescale Semiconductor
Conclusion - System Performance

0001893 00000053 .text FTestHarnessDCE_InterruptTx(TestHarnessDCE.c)
00018E6 00000039 .text FTestHarnessDCE_InterruptError(TestHarnessDCE.c)
000191F 0000001F .text FTestHarnessDCE_Init(TestHarnessDCE.c)
000193E 00000049 rtlib.text F@DummyFn1(process_cpt.asm)
000193E 00000049 rtlib.text FPROCESS_CPT(process_cpt.asm)
000193E 00000049 rtlib.text rtlib.text(process_cpt.asm)
0001987 0000018C rtlib.text F@DummyFn1(cpsi_api.asm)
0001987 00000000 rtlib.text SILENCE_DETECT_CPT(cpsi_api.asm)
0001987 0000018C rtlib.text rtlib.text(cpsi_api.asm)
000198A 00000000 rtlib.text SILENCE_DEBOUNCE(cpsi_api.asm)
00019BE 00000000 rtlib.text FCALLPROGRESS_DETECT_INIT(cpsi_api.asm)
00019EE 00000000 rtlib.text CALLPROGRESS_DETECT(cpsi_api.asm)
00019FA 00000000 rtlib.text CALLPROGRESS_DEBOUNCE(cpsi_api.asm)
0001A03 00000000 rtlib.text no_cpt(cpsi_api.asm)
0001A0F 00000000 rtlib.text cpt_on(cpsi_api.asm)
0001A1D 00000000 rtlib.text noisy_cpt(cpsi_api.asm)
0001A28 00000000 rtlib.text end_cpt(cpsi_api.asm)
0001A36 00000000 rtlib.text cpt_silence(cpsi_api.asm)
0001A43 00000000 rtlib.text noisy_sil(cpsi_api.asm)
0001A4F 00000000 rtlib.text new_cpt(cpsi_api.asm)
0001A58 00000000 rtlib.text check_previous_cpt(cpsi_api.asm)
0001A64 00000000 rtlib.text check_cpt_off(cpsi_api.asm)
0001A71 00000000 rtlib.text reset_cpsi(cpsi_api.asm)
0001A78 00000000 rtlib.text exit_cpt_debounce(cpsi_api.asm)
0001A81 00000000 rtlib.text CALLPROGRESS_DECODE(cpsi_api.asm)
0001AA0 00000000 rtlib.text clear_bursts(cpsi_api.asm)
0001AA5 00000000 rtlib.text check_group2(cpsi_api.asm)
0001AD2 00000000 rtlib.text check_group3(cpsi_api.asm)
0001AEE 00000000 rtlib.text last_on(cpsi_api.asm)
0001AFE 00000000 rtlib.text end_cpt_on(cpsi_api.asm)
0001B06 00000000 rtlib.text not_cpt(cpsi_api.asm)
0001B0D 00000000 rtlib.text exit_decode(cpsi_api.asm)
0001B13 00000044 rtlib.text F@DummyFn1(cpsi_low.asm)
0001B13 00000044 rtlib.text rtlib.text(cpsi_low.asm)
0001B13 00000044 rtlib.text rtlib.text(cpsi_low.asm)
0001B28 00000000 rtlib.text assign1_cpt(cpsi_low.asm)
0001B57 0000003D rtlib.text F@DummyFn1(cpt_api.asm)
0001B57 00000000 rtlib.text PAPI_TONE_DETECT(cpt_api.asm)
0001B57 0000003D rtlib.text rtlib.text(cpt_api.asm)
0001B94 00000042 rtlib.text rtlib.text(cpt_buf.asm)
Conclusion - System Performance

A Low-Cost Soft Modem Using the Freescale Digital Signal Controller MC56F802x/3x Series, Rev. 0

Freescale Semiconductor 89
Conclusion - System Performance

A Low-Cost Soft Modem Using the Freescale Digital Signal Controller MC56F802x/3x Series, Rev. 0
Conclusion - System Performance

00002577 00000000 rttlib.text RX_scr12(v22bis_rx_bchk.asm)
000025CD 00000000 rttlib.text RX_v22dm(v22bis_rx_bchk.asm)
000025F1 00000000 rttlib.text RX_s1end(v22bis_rx_bchk.asm)
000025FD 00000000 rttlib.text RX_wait32bit(v22bis_rx_bchk.asm)
00002619 00000000 rttlib.text RX_waitdm(v22bis_rx_bchk.asm)
00002622 00000000 rttlib.text RX_wait1sec(v22bis_rx_bchk.asm)
00002632 00000000 rttlib.text RX_v22bisdm(v22bis_rx_bchk.asm)
0000266F 00000000 rttlib.text RX_retrain(v22bis_rx_bchk.asm)
00002693 00000000 rttlib.text RX_RETR REP(v22bis_rx_bchk.asm)
0000269E 00000000 rttlib.text RX_RETR A(v22bis_rx_bchk.asm)
000026B8 00000000 rttlib.text RX_NEXT(v22bis_rx_bchk.asm)
000026BA 00000000 rttlib.text ENDRX(v22bis_rx_bchk.asm)
000026BB 00000000 rttlib.text error(v22bis_rx_bchk.asm)
000026DC 00000000 rttlib.text RXDM CD(v22bis_rx_bchk.asm)
000026EA 00000000 rttlib.text RXDMCDON(v22bis_rx_bchk.asm)
0000271A 00000000 rttlib.text RXDMCDOFF(v22bis_rx_bchk.asm)
00002721 00000000 rttlib.text RXDM OFF(v22bis_rx_bchk.asm)
00002733 00000000 rttlib.text CDONOF(v22bis_rx_bchk.asm)
0000273F 0000004A rttlib.text F@DummyFn1(v22bis_rx_bpf.asm)
0000273F 00000000 rttlib.text RXBPF(v22bis_rx_bpf.asm)
0000273F 0000004A rttlib.text rttlib.text(v22bis_rx_bpf.asm)
00002789 00000021 rttlib.text F@DummyFn1(v22bis_rx_car.asm)
00002789 00000000 rttlib.text RXCAR(v22bis_rx_car.asm)
00002789 00000021 rttlib.text rttlib.text(v22bis_rx_car.asm)
000027AA 000000B9 rttlib.text F@DummyFn1(v22bis_rx_cdagc.asm)
000027AA 00000000 rttlib.text RXCDAGC(v22bis_rx_cdagc.asm)
000027AA 000000B9 rttlib.text rttlib.text(v22bis_rx_cdagc.asm)
00002863 0000019 rttlib.text rttlib.text(v22bis_rx_ctrl.asm)
00002863 00000000 rttlib.text RXBAUDPROC(v22bis_rx_ctrl.asm)
00002863 0000019 rttlib.text F@DummyFn1(v22bis_rx_ctrl.asm)
00002866 0000007E rttlib.text rx_no_sti(v22bis_rx_ctrl.asm)
00002870 00000000 rttlib.text rx_next_task(v22bis_rx_ctrl.asm)
0000287C 0000002F rttlib.text F@DummyFn1(v22bis_rx_decim.asm)
0000287C 00000000 rttlib.text RXDECIM(v22bis_rx_decim.asm)
0000287C 0000002F rttlib.text rttlib.text(v22bis_rx_decim.asm)
000028AB 00000049 rttlib.text F@DummyFn1(v22bis_rx_demod.asm)
000028AB 00000000 rttlib.text RXDEMOP(v22bis_rx_demod.asm)
000028AB 00000049 rttlib.text rttlib.text(v22bis_rx_demod.asm)
000028F4 0000007E rttlib.text rttlib.text(v22bis_rx_difdc.asm)
000028F4 00000000 rttlib.text RXDEC4(v22bis_rx_difdc.asm)
Conclusion - System Performance

```assembly
00002C27 00000000 rtlib.text rx_G22D(v22bis_rx_stat.asm)
00002C76 00000000 rtlib.text rx_GBisA(v22bis_rx_stat.asm)
00002C96 00000000 rtlib.text rx_GBisB(v22bis_rx_stat.asm)
00002CA1 00000000 rtlib.text rx_GBisC(v22bis_rx_stat.asm)
00002CE8 00000000 rtlib.text rx_GBisD(v22bis_rx_stat.asm)
00002D17 00000000 rtlib.text rx_GBisE(v22bis_rx_stat.asm)
00002D33 00000000 rtlib.text rx_GBisF(v22bis_rx_stat.asm)
00002D7D 00000000 rtlib.text rx_GBisG(v22bis_rx_stat.asm)
00002D9B 00000000 rtlib.text rx_GRetA(v22bis_rx_stat.asm)
00002DCA 000000F8 rtlib.text F@DummyFn1(v22bis_rx_ton.asm)
00002DCA 00000000 rtlib.text RXUSB1(v22bis_rx_ton.asm)
00002DCA 000000F8 rtlib.text rtlib.text(v22bis_rx_ton.asm)
00002E07 00000000 rtlib.text RXS1(v22bis_rx_ton.asm)
00002EA1 00000000 rtlib.text RXTON(v22bis_rx_ton.asm)
00002EC2 00000091 rtlib.text F@DummyFn1(v22bis_rxmdmini.asm)
00002EC2 00000000 rtlib.text RX_MDM_INIT(v22bis_rxmdmini.asm)
00002EC2 00000091 rtlib.text rtlib.text(v22bis_rxmdmini.asm)
00002F53 0000001E rtlib.text F@DummyFn1(v22bis_tondet.asm)
00002F53 00000000 rtlib.text TONEDETECT(v22bis_tondet.asm)
00002F53 0000001E rtlib.text rtlib.text(v22bis_tondet.asm)
00002F71 00000000 rtlib.text F@DummyFn1(v22bis_rxstub.asm)
00002F71 00000000 rtlib.text rx_stub(v22bis_rxstub.asm)
00002F71 00000000 rtlib.text rtlib.text(v22bis_rxstub.asm)
00002F80 00000000 rtlib.text F@DummyFn1(v22bis_txstub.asm)
00002F80 00000000 rtlib.text tx_stub(v22bis_txstub.asm)
00002FA6 000000CD rtlib.text F@DummyFn1(v22bisapi.asm)
00002FA6 00000000 rtlib.text FINITIALIZE_V22BIS(v22bisapi.asm)
00002FA6 000000CD rtlib.text rtlib.text(v22bisapi.asm)
00002FDB 00000000 rtlib.text FV22BIS_TRANSMIT(v22bisapi.asm)
00002FFC 00000000 rtlib.text FV22BIS_RECEIVE_SAMPLE(v22bisapi.asm)
00003073 00000000 rtlib.text F@DummyFn1(v22_v42d.asm)
00003073 00000000 rtlib.text V42DRV_INIT(v22_v42d.asm)
00003073 00000000 rtlib.text rtlib.text(v22_v42d.asm)
00003086 00000000 rtlib.text FV42_V22DRV_INIT(v22_v42d.asm)
0000308C 00000000 rtlib.text V22_V42DRV(v22_v42d.asm)
000030BB 00000000 rtlib.text V42_V22DRV(v22_v42d.asm)
000030BE 00000000 rtlib.text LAPM_MDM_INIT(v22_v42d.asm)
000030BF 00000000 rtlib.text WRITE_NIBBLE(v22_v42d.asm)
000030C1 00000000 rtlib.text READ_NIBBLE(v22_v42d.asm)
```
Conclusion - System Performance

A Low-Cost Soft Modem Using the Freescale Digital Signal Controller MC56F802x/3x Series, Rev. 0
Conclusion - System Performance

00000066 00000006 rttlib.data rttlib.data(cpt_dc.asm)
0000006C 00000002 .data  FBitIO_portDsc(Cpu.c)
0000006E 00000001 .data  FpModemRxWrt(modemblue.c)
0000006F 00000001 .data  FpModemRxRead(modemblue.c)
00000070 00000000 TX_MEM.data MDMCONFIG(v22bis_gmdmmem.asm)
00000070 00000000 TX_MEM.data FMDMCONFIG(v22bis_gmdmmem.asm)
00000070 0000000E TX_MEM.data TX_MEM.data(v22bis_gmdmmem.asm)
00000072 00000000 TX_MEM.data TX_GAIN(v22bis_gmdmmem.asm)
00000073 00000000 TX_MEM.data MDMSTATUS(v22bis_gmdmmem.asm)
00000073 00000000 TX_MEM.data FMDMSTATUS(v22bis_gmdmmem.asm)
00000074 00000000 TX_MEM.data mode_flg(v22bis_gmdmmem.asm)
00000075 00000000 TX_MEM.data rx_st_id(v22bis_gmdmmem.asm)
00000076 00000000 TX_MEM.data tx_st_id(v22bis_gmdmmem.asm)
00000077 00000000 TX_MEM.data flg_107(v22bis_gmdmmem.asm)
00000078 00000000 TX_MEM.data flg_112(v22bis_gmdmmem.asm)
00000079 00000000 TX_MEM.data flg_109(v22bis_gmdmmem.asm)
0000007A 00000000 TX_MEM.data flg_104(v22bis_gmdmmem.asm)
0000007B 00000000 TX_MEM.data flg_106(v22bis_gmdmmem.asm)
0000007C 00000000 TX_MEM.data loopback(v22bis_gmdmmem.asm)
0000007D 00000000 TX_MEM.data Fretrain_flag(v22bis_gmdmmem.asm)
0000007D 00000000 TX_MEM.data retrain_flag(v22bis_gmdmmem.asm)
0000007E 00000000 TX_MEM.data txI1ctr(v22bis_txmdmmem.asm)
0000007E 00000000 TX_MEM.data TXMEMB(v22bis_txmdmmem.asm)
0000007E 00000047 TX_MEM.data TX_MEM.data(v22bis_txmdmmem.asm)
0000007F 00000000 TX_MEM.data txI2ctr(v22bis_txmdmmem.asm)
00000080 00000000 TX_MEM.data txI3ctr(v22bis_txmdmmem.asm)
00000081 00000000 TX_MEM.data txI4ctr(v22bis_txmdmmem.asm)
00000082 00000000 TX_MEM.data txI51ctr(v22bis_txmdmmem.asm)
00000083 00000000 TX_MEM.data txI52ctr(v22bis_txmdmmem.asm)
00000084 00000000 TX_MEM.data txI61ctr(v22bis_txmdmmem.asm)
00000085 00000000 TX_MEM.data txI62ctr(v22bis_txmdmmem.asm)
00000086 00000000 TX_MEM.data txI72ctr(v22bis_txmdmmem.asm)
00000087 00000000 TX_MEM.data txI82ctr(v22bis_txmdmmem.asm)
00000088 00000000 TX_MEM.data mdm_flg(v22bis_txmdmmem.asm)
00000089 00000000 TX_MEM.data gt_flg(v22bis_txmdmmem.asm)
0000008A 00000000 TX_MEM.data ccitt_flg(v22bis_txmdmmem.asm)
0000008B 00000000 TX_MEM.data tx_ans_flg(v22bis_txmdmmem.asm)
0000008C 00000000 TX_MEM.data tx_rx16(v22bis_txmdmmem.asm)
0000008D 00000000 TX_MEM.data atone_ptr(v22bis_txmdmmem.asm)
0000008E 00000000 TX_MEM.data tx_data(v22bis_txmdmmem.asm)
Conclusion - System Performance

000006A5 00000000 RX_MEM.data BLPG1(v22bis_rxmdmmem.asm)
000006A6 00000000 RX_MEM.data BLPG2(v22bis_rxmdmmem.asm)
000006A7 00000000 RX_MEM.data BOFF(v22bis_rxmdmmem.asm)
000006A8 00000000 RX_MEM.data BHPX1(v22bis_rxmdmmem.asm)
000006A9 00000000 RX_MEM.data BHPY1(v22bis_rxmdmmem.asm)
000006AA 00000000 RX_MEM.data BHPX3(v22bis_rxmdmmem.asm)
000006AB 00000000 RX_MEM.data BHPY3(v22bis_rxmdmmem.asm)
000006AC 00000000 RX_MEM.data BHPE1(v22bis_rxmdmmem.asm)
000006AD 00000000 RX_MEM.data BHPE3(v22bis_rxmdmmem.asm)
000006AE 00000000 RX_MEM.data BACC1(v22bis_rxmdmmem.asm)
000006AF 00000000 RX_MEM.data BACC2(v22bis_rxmdmmem.asm)
000006B0 00000000 RX_MEM.data BLP(v22bis_rxmdmmem.asm)
000006B1 00000000 RX_MEM.data BINTG(v22bis_rxmdmmem.asm)
000006B2 00000000 RX_MEM.data BINTGA(v22bis_rxmdmmem.asm)
000006B3 00000000 RX_MEM.data status(v22bis_rxmdmmem.asm)
000006B4 00000000 RX_MEM.data CARG1(v22bis_rxmdmmem.asm)
000006B5 00000000 RX_MEM.data CARG2(v22bis_rxmdmmem.asm)
000006B6 00000000 RX_MEM.data CARG3(v22bis_rxmdmmem.asm)
000006B7 00000000 RX_MEM.data CARG4(v22bis_rxmdmmem.asm)
000006B8 00000000 RX_MEM.data COFF(v22bis_rxmdmmem.asm)
000006B9 00000000 RX_MEM.data CLP(v22bis_rxmdmmem.asm)
000006BA 00000000 RX_MEM.data RCBUF(v22bis_rxmdmmem.asm)
000006BB 00000000 RX_MEM.data RCBUF_1(v22bis_rxmdmmem.asm)
000006BC 00000000 RX_MEM.data RCBUF_2(v22bis_rxmdmmem.asm)
000006BD 00000000 RX_MEM.data RCBUF_3(v22bis_rxmdmmem.asm)
000006BE 00000000 RX_MEM.data RCBUF_4(v22bis_rxmdmmem.asm)
000006BF 00000000 RX_MEM.data RCBUF_5(v22bis_rxmdmmem.asm)
000006C0 00000000 RX_MEM.data THBUF(v22bis_rxmdmmem.asm)
000006D0 00000000 RX_MEM.data BBUF(v22bis_rxmdmmem.asm)
000006DD 00000000 RX_MEM.data JITTER(v22bis_rxmdmmem.asm)
000006DE 00000000 RX_MEM.data JITG1(v22bis_rxmdmmem.asm)
000006DF 00000000 RX_MEM.data JITG2(v22bis_rxmdmmem.asm)
000006E0 00000000 RX_MEM.data WRPFLG(v22bis_rxmdmmem.asm)
000006E1 00000000 RX_MEM.data ACODE(v22bis_rxmdmmem.asm)
000006E2 00000000 RX_MEM.data EQRT(v22bis_rxmdmmem.asm)
000006E3 00000000 RX_MEM.data EQRT_1(v22bis_rxmdmmem.asm)
000006E4 00000000 RX_MEM.data EQRT_2(v22bis_rxmdmmem.asm)
000006E5 00000000 RX_MEM.data EQRT_3(v22bis_rxmdmmem.asm)
000006E6 00000000 RX_MEM.data EQRT_4(v22bis_rxmdmmem.asm)
000006E7 00000000 RX_MEM.data EQRT_5(v22bis_rxmdmmem.asm)
Conclusion - System Performance

00000779 00000000 RX_MEM.data RN_BITS_BAUD(v22bis_rxmdmmem.asm)
0000077A 00000000 RX_MEM.data TN_BITS_BAUD(v22bis_rxmdmmem.asm)
0000077B 00000000 RX_MEM.data T401_VALUE(v22bis_rxmdmmem.asm)
0000077C 00000000 RX_MEM.data T401B_VALUE(v22bis_rxmdmmem.asm)
0000077D 00000000 RX_MEM.data T403_VALUE(v22bis_rxmdmmem.asm)
0000077E 00000000 RX_MEM.data LASTDP(v22bis_rxmdmmem.asm)
0000077F 00000000 RX_MEM.data WRAP(v22bis_rxmdmmem.asm)
00000780 00000000 RX_MEM.data BBUF PTR(v22bis_rxmdmmem.asm)
00000781 00000000 RX_MEM.data rx_data(v22bis_rxmdmmem.asm)
00000781 00000000 RX_MEM.data Frx_data(v22bis_rxmdmmem.asm)
00000782 00000000 RX_MEM.data NOISE(v22bis_rxmdmmem.asm)
00000783 00000000 RX_MEM.data RETCNT_RM(v22bis_rxmdmmem.asm)
00000784 00000000 RX_MEM.data speed(v22bis_rxmdmmem.asm)
00000785 00000000 RX_MEM.data ICOEFF(v22bis_rxmdmmem.asm)
00000786 00000000 RX_MEM.data BPF_PTR(v22bis_rxmdmmem.asm)
00000787 00000000 RX_MEM.data temp1(v22bis_rxmdmmem.asm)
00000788 00000000 RX_MEM.data temp2(v22bis_rxmdmmem.asm)
00000789 00000000 RX_MEM.data mod_tbl_offset(v22bis_rxmdmmem.asm)
0000078A 00000000 RX_MEM.data TRAINING(v22bis_rxmdmmem.asm)
0000078B 00000000 RX_MEM.data IFBANK(v22bis_rxmdmmem.asm)
0000078C 00000000 RX_MEM.data IBCNT(v22bis_rxmdmmem.asm)
0000078D 00000000 RX_MEM.data TXBD_CNT(v22bis_rxmdmmem.asm)
0000078E 00000000 RX_MEM.data TNSUM(v22bis_rxmdmmem.asm)
0000078F 00000000 RX_MEM.data TNASUM(v22bis_rxmdmmem.asm)
00000790 00000000 RX_MEM.data EQX(v22bis_rxmdmmem.asm)
00000791 00000000 RX_MEM.data EQY(v22bis_rxmdmmem.asm)
00000792 00000000 RX_MEM.data DECX(v22bis_rxmdmmem.asm)
00000793 00000000 RX_MEM.data DECY(v22bis_rxmdmmem.asm)
00000794 00000000 RX_MEM.data RXDATA(v22bis_rxmdmmem.asm)
00000795 00000000 RX_MEM.data DX(v22bis_rxmdmmem.asm)
00000796 00000000 RX_MEM.data DY(v22bis_rxmdmmem.asm)
00000797 00000000 RX_MEM.data RXMEMSIZE(v22bis_rxmdmmem.asm)
00000798 00000000 RX_MEM.data IBPTR_IN(v22bis_rxmdmmem.asm)

#>00000000 _EX_BIT (linker command file)
#>00000001 _NUM IM PARTITIONS (linker command file)
#>00000000 _NUM EM PARTITIONS (linker command file)
Conclusion - System Performance

A Low-Cost Soft Modem Using the Freescale Digital Signal Controller MC56F802x/3x Series, Rev. 0

Freescale Semiconductor

#>000007A8        FmemEXbit (linker command file)
#>000007A9        FmemNumIMpartitions (linker command file)
#>000007AA        FmemNumEMpartitions (linker command file)
#>000007AC        FmemIMpartitionAddr (linker command file)
#>000007AD        FmemIMpartitionSize (linker command file)
#>000007A0        FmemEMpartitionAddr (linker command file)
#>000007AE        FmemEMpartitionSize (linker command file)
#>000007B0        __xRAM_data_end (linker command file)
#>000007AF        __data_size (linker command file)

# .ApplicationData
#>000007B0        F_Xbss_start_addr (linker command file)
#>000007B0        __START_BSS (linker command file)
000007B0 00000001 .bss   FSR_lock(Cpu.c)
000007B1 00000001 .bss   FSR_reg(Cpu.c)
000007B2 00000001 .bss   Fis_time_to_shake(modemblue.c)
000007B3 00000001 .bss   FpAnalogRxWrite(modemblue.c)
000007B4 00000001 .bss   FpAnalogRxRead(modemblue.c)
000007B5 00000001 .bss   FpAnalogTxWrite(modemblue.c)
000007B6 00000001 .bss   FpAnalogTxRead(modemblue.c)
000007B7 00000001 .bss   FPreviousSample(modemblue.c)
000007B8 00000001 .bss   FCumCnt(modemblue.c)
000007B9 00000001 .bss   FLast_Dac_Value(modemblue.c)
000007BA 00000001 .bss   FThis_Delta_Value(modemblue.c)
000007BB 00000001 .bss   FThis_Dac_Value(modemblue.c)
000007BC 00000001 .bss   FBlue_DAC_Scale(modemblue.c)
000007BD 00000001 .bss   Fstate_of_escape(modemblue.c)
000007BE 00000001 .bss   FAT_on_state(modemblue.c)
000007BF 00000001 .bss   FAT_off_state(modemblue.c)
000007BE 00000029 .bss   Fphone_number(modemblue.c)
000007D0 00000001 .bss   Fp_phone_number(modemblue.c)
000007D1 00000001 .bss   Fh_parm(modemblue.c)
000007D6 00000002 .bss   FAC24(modemblue.c)
000007D8 00000002 .bss   FAC12(modemblue.c)
000007DA 00000002 .bss   FAC0(modemblue.c)
000007DC 00000002 .bss   FACa(modemblue.c)
000007DE 00000001 .bss   FLine_Tones(modemblue.c)
000007DF 00000019 .bss   FAnalogTxBuffer(modemblue.c)
000007F8 00000020 .bss   FAnalogRxBuffer(modemblue.c)
00000818 00000010 .bss   FModemRxBuffer(modemblue.c)
Conclusion - System Performance

00000820 00000001 .bss Ftty_in_status(modemblue.c)
00000820 00000064 .bss Fmodem_in(modemblue.c)
00000852 00000064 .bss Ftty_in(modemblue.c)
00000885 00000064 .bss FCodecRxBuffer(modemblue.c)
000008E9 0000000C .bss FCodecTxBuffer(modemblue.c)
000008F5 00000001 .bss FAns_Tone_Detect(modemblue.c)
000008F6 00000001 .bss FAns_Tone_Start(modemblue.c)
000008F7 00000001 .bss FV21_Mode(modemblue.c)
000008F8 00000001 .bss FAT_q_flag(modemblue.c)
000008F9 00000001 .bss FAT_z_flag(modemblue.c)
000008FA 00000001 .bss Fcall_phone_number(modemblue.c)
000008FB 00000001 .bss FCaller_Modem(modemblue.c)
000008FC 00000004 .bss Fv22bis_RXCallback(modemblue.c)
00000900 00000004 .bss Fv22bis_TXCallback(modemblue.c)
00000904 00000001 .bss Frate_negotiated(modemblue.c)
00000905 00000001 .bss Fconnection_lost(modemblue.c)
00000906 00000001 .bss Fv22bis_connection_established(modemblue.c)
00000907 00000001 .bss FbMemInitialized(mem.c)
00000908 00000004 .bss FEmptyExternalMemoryPool(mem.c)
0000090C 00000004 .bss FEmptyInternalMemoryPool(mem.c)
00000910 00000005 .bss FInitialState(mem.c)
00000916 00000028 .bss FExternalMemoryPool(mem.c)
0000093E 00000028 .bss FInternalMemoryPool(mem.c)
00000966 00000001 .bss Fring_state(Events.c)
00000967 00000001 .bss Ferror_cnt(Events.c)
00000968 00000001 .bss Fring_pulse_count(Events.c)
00000969 00000001 .bss FTimeToggle(Events.c)
0000096A 00000001 .bss Fthe_errors(Events.c)
0000096B 00000001 .bss FCountTime(Events.c)
0000096C 00000001 .bss FEnUser(TwentythSecInt.c)
0000096D 00000001 .bss Fv22dibittxcount(v22bisapi.c)
0000096E 00000001 .bss Frxbitcounter(v22bisapi.c)
0000096F 00000001 .bss Fstartbit(v22bisapi.c)
00000970 00000001 .bss FPartialRxByte(v22bisapi.c)
00000971 00000003 .bss FNibbles(v22bisapi.c)
00000974 00000001 .bss FNibbleCount(v22bisapi.c)
00000975 00000001 .bss FNumberBytes(v22bisapi.c)
00000976 00000001 .bss FBytePtr(v22bisapi.c)
00000977 00000001 .bss FByteCount(v22bisapi.c)
00000978 00000001 .bss Fmessageover(v22bisapi.c)
Conclusion - System Performance

0000097A 00000004 .bss  FRXCallback(v22bisapi.c)
0000097E 00000004 .bss  FTXCallback(v22bisapi.c)
00000982 00000001 .bss  FEnUser(WDog1.c)
00000983 0000032 .bss  FOutBuffer(TestHarnessDCE.c)
00000985 00000001 .bss  FOutPtrW(TestHarnessDCE.c)
000009B6 00000001 .bss  FOutPtrR(TestHarnessDCE.c)
000009B7 00000001 .bss  FOutLen(TestHarnessDCE.c)
000009B8 00000032 .bss  FInpBuffer(TestHarnessDCE.c)
000009EA 00000001 .bss  FInpPtrW(TestHarnessDCE.c)
000009EB 00000001 .bss  FInpPtrR(TestHarnessDCE.c)
000009EC 00000001 .bss  FInpLen(TestHarnessDCE.c)
000009ED 00000001 .bss  FErrFlag(TestHarnessDCE.c)
000009EE 00000001 .bss  FSerFlag(TestHarnessDCE.c)
000009EF 0000000E API.bss  API.bss(v22bis_apimem.asm)
000009EF 00000000 API.bss  s_ctr(v22bis_apimem.asm)
000009F0 00000000 API.bss  Tx_Baud_Count(v22bis_apimem.asm)
000009F1 00000000 API.bss  Rx_Baud_Flg(v22bis_apimem.asm)
000009F2 00000000 API.bss  TIME_CNT(v22bis_apimem.asm)
000009F3 00000000 API.bss  TIME_CNTL(v22bis_apimem.asm)
000009F4 00000000 API.bss  TIME_CNTH(v22bis_apimem.asm)
000009F5 00000000 API.bss  in_data_ptr(v22bis_apimem.asm)
000009F6 00000000 API.bss  txrx_status(v22bis_apimem.asm)
000009F7 00000000 API.bss  WordWrFlg(v22bis_apimem.asm)
000009F8 00000000 API.bss  WordRdFlg(v22bis_apimem.asm)
000009F9 00000000 API.bss  StartCompare(v22bis_apimem.asm)
000009FA 00000000 API.bss  SyncWord_mem(v22bis_apimem.asm)
000009FB 00000000 API.bss  Sync_sent_status(v22bis_apimem.asm)
000009FC 00000000 API.bss  SyncWord_rx(v22bis_apimem.asm)

A Low-Cost Soft Modem Using the Freescale Digital Signal Controller MC56F802x/3x Series, Rev. 0
4.4 Core Processor Loading and RTOS

Because the modem uses very little of the resources of the core processor, an RTOS may be used to run several tasks along with the modem task.

4.4.1 Core Processor Load

When the modem is idle, waiting for a call, it consumes almost no MIPS because the hardware is used to count ring pulses without the help of the core. Only when a significant ring is detected is the core interrupted.

The V.22bis (line rate of 2400 characters per second) bean methods are called to receive data for a consumption rate of 6.21 MIPS. The V.22bis bean methods for transmission of data consume 0.94 MIPS. So, the total MIPS for V.22bis is only 7.15. This is just a small fraction of the MIPS available on the 32 MIPS digital signal controller, a little over 20 percent.

The fraction is much less when V.21 is used (line rate of 300 characters per second). It amounts to very little use of the processor.

4.4.2 Use of RTOS to Run Modem Concurrently with Other Tasks

To blend other tasks with the modem, it is advisable to use an RTOS, or real time operating system. A multi-tasking, context-switching RTOS could be used to share the remaining resources with other tasks, such as alarm monitoring or process control. The modem code itself would be one task, because it is written as one thread.

With the RTOS, the modem code would be made into a task which would sleep while waiting for IO to complete. This is what frees up and unlocks the CPU for other tasks.

The key to success here is to make sure that the TX does not have to wait to place data in the FIFO, so that it can be waiting only for RX characters from the ADC. This is the manner in which this project was tuned.
4.5 Peripheral Footprint

The small percentage of peripherals used by the modem is depicted in Figure 52. This also shows how the beans are graphically associated with pins on the device. Pins without such associations are free to use for other tasks. Each bean has a unique graphic icon that is easily associated with the pins in Figure 53.

The total resource usage is summarized in Figure 52. This resource load meter is a convenient feature of CodeWarrior with Processor Expert. As the project is developed, resource usage is easily tracked when beans are added.

![Figure 52. Resource Meter](image-url)
4.6 Conclusions

The soft modem developed here is suitable for incorporation into commercial products requiring communication over the PSTN at speeds up to 2400 bits per second. A traditional telephone codec is not required in the design, resulting in a one chip/one core system capable of a complete mission, including communications functions.

Both V.21 and V.22bis/V.22 are supported. The V.22bis falls back to V.22 when noise dictates.

The modem is easily added to projects developed for the Freescale 56F802x/3x family.
5 Layout and Governmental Certifications

5.1 Design for Performance

The ADC and DAC use analog signals. They should not parallel at close range signals that contain clocks or signals that change often. Strip-mine-type shielding could be considered. We recommend that you refer to Freescale FAQs and application notes relating to optimal use of the ADC. The layout of the EVM may be used as an example, even though it is possible to reduce even further the noise floor. Given the dynamic range available, this is not required to obtain the performance documented herein.

5.2 Design for Agency Approvals

The final step in bringing a product containing a soft modem to market involves obtaining the approval of, and certification by, the various governmental agencies regulating the sale of products that are to be connected to the PSTN. In some countries the same agency that regulates the post office also regulates the modem product industry. In the United States, FCC part 15 and part 68 should be met. Also, UL approval of any electrical appliance is advised.

The government is concerned with several factors:

- **The ability of the equipment to operate in the presence of radio frequency interference.** Governments do not like to receive a lot of complaints about radio frequency interference. The more RFI that your product can be exposed to without faltering, the better the government will like it.

- **The amount of radio frequency interference produced by the product.** Certain frequencies are used by government agencies; these frequencies are especially monitored for compliance. For example, 75 megahertz is allocated to aeronautical radio navigation. If a product broadcasts on that frequency, planes could be in danger.

- **The effect of high voltages from tip and ring (including lightning strikes) on the product’s viability and safety.** In the interest of consumer safety and product merchantability, the product should not be destroyed, and any damage should be limited, when lightning or other high voltage sources come down the tip and ring from the phone pole. The idea is to limit the damage to the parts of the product called the DAA, or data access arrangement, when lightning strikes or power lines become tangled with phone lines. The ring voltage itself is considered high voltage and dangerous.

- **The possibility that the product might become a nuisance by repeatedly calling wrong numbers in the middle of the night to private homes.** With the advent of the FAX machine and computer bulletin board, a disturbing trend began. People were called repeatedly by modems or FAX machines. Some countries require “blacklisting” certain numbers and limiting the number of times other numbers may be called per unit time. To comply with this, firmware must be tested by government agencies or their designated agents.

This last point simply requires software that cannot automatically and repeatedly dial phone numbers. As for the radio frequency issues, they are dealt with by shielding (u metal) and or current loop size minimization. Current-loop minimization is a layout technique in which the open area of a current loop is minimized.
The other points can be dealt with by selecting a DAA, such as the one selected for this design, that meets the standards of multiple governments. Besides that, special layout attention is required for the area of the product’s PC board where this DAA device is mounted.

The layout should minimize the area of current loops, or enclose them in shielding, and isolate the high voltage section of the DAA on a one or two layer section of the board, well fused from the tip and ring, and located at the extreme boundary of the product.

Current loop area is minimized by running the return wire from any circuit next to, above, or below the source wire for the circuit. It is also good to surround these “wires” or traces with ground plane on as many sides as possible.

**NOTE:** This project design had not been examined by government authorities for compliance, but is only used on equipment that simulates the PSTN. Any products developed to be sold to the public will require the respective government agency approvals prior to use on the PSTN of the respective country and before sale.

## 6 References

- SAMS, Understanding Telephone Electronics, Seventh printing 1987.
- ITU-T Recommendation V.22bis, 2400 Bits Per Second Duplex Modem Using The Frequency Division Technique Standardized For Use On The General Switched Telephone Network And On Point-to-Point.
- ITU-T Recommendation V.21, 300 Bits Per Second Duplex Modem Standardized For Use On The General Switched Telephone Network.
- Boonton Electronics, Application Note AN-50 1 Apr. ‘93, Measuring The Peak-to-Average Power Of Digitally Modulated Signals, Charles J. Meyer, Senior Applications Engineer.
- Cermetek Microelectronics, CH1837A/7F/8A Data Access Arrangement Module Data Sheet, V.34bis High Speed DAA Module.

## Appendix A Low-Cost Modem Daughter Card Schematics
A Low-Cost Soft Modem Using the Freescale Digital Signal Controller MC56F802x/3x Series, Rev. 0
A Low-Cost Soft Modem Using the Freescale Digital Signal Controller MC56F802x/3x Series, Rev. 0

Freescale Semiconductor 113

Digital Signal Controller Operation

2100 East Elliot Road
Tempe, Arizona 85284
(512) 895-7215 FAX: (480) 413-2510