Building an ADSL Modem Evaluation and Demonstration Platform with Tcl/Tk

Todd Copeland and David Karoly
Legerity Incorporated

{todd.copeland,david.karoly}@legerity.com

Abstract

This paper describes how we used Tcl/Tk, [incr Tcl] and BLT to incrementally develop software for controlling a complex telecommunication chipset despite a tight schedule and evolving requirements. We describe a few interesting Tcl programming constructs that were key to this accomplishment, and how Tk and BLT made it possible for us to quickly deliver an impressive GUI.

1 Introduction

Our company was developing a chipset intended to complete the central office side of multiple telephone circuits carrying digital data integrated with traditional voice. The chipset's data portion would employ Asymmetric Digital Subscriber Line (ADSL) technology. Using sophisticated modulation and encoding schemes, ADSL conveys digital data in the frequency spectrum above the voice band. In this way, ADSL enhances traditional phone lines to also transport digital data at rates up to approximately 9M bits/second.

In an actual application, a microcontroller orchestrates the operation of the chipset. In previous work with voice-only chipsets, we found it advantageous during development to replace the microcontroller with re-configurable hardware controlled by PC-based software. This software provides Tcl scripting support, enabling engineers to program and control the chipset for testing purposes without the overhead of also developing an embedded system.

We augmented our voice-only software, WinACIF [1], to support the new integrated voice-data project. We developed new code that would execute in the same interpreter as WinACIF to control the Data Digital Signal Processor (DDSP). This would allow our engineers to write scripts to control both the voice and data operations.

This project demanded greater flexibility and faster development from our software design. Since the DDSP feature set, including the control interface, could be changed by loading a different program into its on-chip processor, our support software also had to be re-configurable. Whereas previously our deadline corresponded with the arrival of the first silicon versions of the designs, in this project the designers used hardware-based emulation for their design verification. This meant that we would have to deliver our support tools incrementally. At any point we needed to provide enough functionality to support the next feature the designers had ready to test in the lab.

2 Inverting the partition

In the development of WinACIF, we coded only the GUI in Tcl/Tk. The bulk of the program was developed as a C-coded Tcl extension. This mass of C has proven inflexible and difficult to maintain. Given the demands of this project, it was obvious we would not succeed if the bulk of our code was in C.

Therefore, we decided to invert the organization: nearly all of the code would be written as object-oriented code using [incr Tcl]. C would be used only where absolutely necessary to communicate with the underlying hardware. We hoped that scripting would allow faster development and facilitate end user customization of the system. Using objects, we hoped, would help us write cleaner, more understandable code. We were not disappointed.

3 Binding to DDSP events

One of the [incr Tcl] classes we created is a mechanism for binding callback code to events the DDSP generates. In an actual system application, the DDSP asserts an interrupt line to the external management controller to signal an event occurred. The controller then responds by reading a global primary interrupt status register, and possibly a channel-specific secondary interrupt register, to determine the nature of the event and then take the appropriate action.

Limitations of our lab hardware prevented an interrupt-driven implementation, so instead our Ddsp object includes a polling facility. We used the classic Tcl technique [2] of after callbacks that re-schedule themselves. Here are the Ddsp object methods:

```
body Ddsp::PollStart {} {
    set _pending_poll [::after $interval \
            [code $this PollHandler]]
}

body Ddsp::PollStop {} {
    ::after cancel $_pending_poll
}

body Ddsp::PollHandler {} {
    ReadPriInt
    PollStart
}

body Ddsp::ReadPriInt {} {
    set intrList [reg_rd CIF_EXT_PRI_INT]
    foreach bitname $intrList  {
        $_eventMngr trigger <$bitname>
        switch $bitname {
            "Line_0_Int" {ReadSecInt 0}
            ...
            }
        }
    }
```

In the code above, notice that if a bit is set in the interrupt register, the code calls the trigger method in the event manager passing it the name of the event ($bitname). The event manager maintains an array of lists keyed by event names. Each item in the lists is a complete script. When the trigger method is called, the event manager uses **uplevel** to execute the scripts associated with the event.

Our event manager has a bind method modeled after Tk's **bind**. The following binds a script to the DDSP's counter update event.

```
    ddsp bind <Counter_Update> {
        +puts "Got a counter update!"
    }
```

We use this bind facility within our application to manage event callbacks that update the GUI. We also make the bind facility available to our users so that they can easily extend the application without modifying the core code. This, however, presents the danger that users might remove a critical binding and corrupt the application. By implementing the binding facility as an [incr Tcl] class it is easy to compose two object instances within the Ddsp object. So that user scripts may use bind, we provide a bind method in the Ddsp object which acts as a thin wrapper around one of the event managers. The event manager used by the application remains safely hidden inside the Ddsp object. We gave the application binding priority by triggering the application bindings first.

4 A pliable interface

As mentioned earlier, the meaning and addresses of the registers comprising the control interface of the DDSP are changeable via the program loaded into the DDSP. Therefore, we defined the control interface in a file external to our program. This de-coupled our core code from changes in the register definitions and allowed the DDSP developers to make many changes themselves. The file also defines the mapping of parameter names and types to the physical addresses of the registers. This abstraction guarantees that changes to the physical addresses do not affect our application or user scripts.

We organized the file as **set** commands. The Ddsp constructor simply sources the file, thereby initializing private arrays in the object. There was no need to write code to read and parse the file.

Here is a sample of what's in that file:

```
    set _regInfo(CIF_EXT_PRI_INT) {
        1 05 R {M8 {Self_Test 7 ... Line_0_Int 0}
        {} {} {} {}}
    }

    set _regInfo(Requested_Line_State) {
        1 {0018 0090 0108 0180} R/W {E8 {idle 01
        activating 02 showtime 03} {} {} {}}
    }

    set _regInfo(C_Max_Attainable_Rate) {
        2 {0042 00BA 0132 01AA} R {U16.0 {}
        1.0 0 65535 Kb/s}
    }
```

The array keys correspond to the names of registers, and the values are lists. Each list contains information that describes the register's attributes: its address, read or write status, data format, minimum, maximum and units, among other things. Formats we support include bit masks, hex, enumerations, unsigned integers, signed integers, and fixed-point fractional.

5 Storing code as data

An interesting requirement of the DDSP interface is that the number of bytes read by some commands is not known until the result is returned. The result of such a command includes two fields that indicate the range of data that follows. In our configuration file these commands are defined like this:

```
    set _cmdRspInfo(CMD_READ_ATUC_SNR_TABLE) {
        1 {first frmtList}
        1 {last frmtList}
        {$first $last} {SNR frmtList}
    }
```

Our code refers to the above fields as follows:

```
    set _cmdRspInfo(cmdname) {
        repList grpList
         ...
    }
```

GrpList defines a parameter that consists of a single value or a vector of values as indicated by *repList*. By storing a little Tcl syntax in *repList*, $first and $last, and relying on **set** and **subst**, our read algorithm could be written as follows:

```
    foreach {repList grpList} $defList {
        set start [subst [lindex $repList 0]]
        set stop  [subst [lindex $repList 1]]
        if {$stop == "" } {set stop $start}
        for {set j $start} {$j <= $stop} {incr j} {
            foreach {parmName fmtList} $grpList {
                ... read each paramVal here ...
                # Set variable for use later
                set $parmName $parmVal
            }
        }
    }
```

This has the flexibility to handle the case where the range of data is constant as well as the case where the range of data is variable. We **set** the local variable $parmName in case its value is needed to define start or stop for a subsequent parameter.

6 Implementing a Tcl time-out

The DDSP interface enables the Central Office modem to retrieve status information from the modem on the other side of the phone line. This meant that we needed a way to set a reasonable time-out period so that we wouldn't indefinitely wait on the remote modem to respond.

vwait gave us a mechanism to block the flow of execution and **after** let us trip the **vwait** after a specified time-out period.

```
body Ddsp::cmdrsp {cmdMnem args} {
    # ... check args ...

    set _pending_timeout [::after $_timeout
            [code $this CmdRspTimeout]]

    # ... write the command ...

    vwait [scope _response_ready]
    ::after cancel $_pending_timeout

    if {$_response_ready == "TIMEOUT"} {
        # ... handle timeout case ...
    }
}

body Ddsp::CmdRspTimeout {} {
    set _response_ready "TIMEOUT"
}
```

Of course we needed a way to set _response_ready in the typical case where a response arrives as expected. The following line placed in our class constructor guarantees our **vwait** trips as expected.

```
$_ddsp_eventMgr bind <Response_Ready> [code set
        [scope _response_ready] 1]
```

7 Mixing vwait with after events

Callbacks in general, and in our case **after** callbacks, occasionally need to use the cmdrsp method described above. We soon discovered that having more than one event waiting at the **vwait** is problematic. We decided to remedy this by adding to our event manager an after cmdrsp_idle method. Using this method, a user could safely schedule a command to run when the current cmdrsp completes.

```
body EventMgr::after {sequence script} {
    if {$sequence == "cmdrsp_idle"} {
        if {$_cmdrsp_idle == 1} {
            uplevel #0 $script
            return
        }
        # enqueue script to run when idle
        lappend _after_queue($sequence) $script
    }
}
```

We modified the cmdrsp method to signal the event manager when a cmdrsp is in progress and when one completes.

```
body Ddsp::cmdrsp {cmdMnem args} {
    $_user_eventMgr cmdrsp_busy

    # ... check args, setup timeout,
    # write command, and vwait ...

    $_user_eventMgr cmdrsp_idle
}
```

The call to the event managers cmdrsp_idle method causes the next pending callback to execute when control returns to the event loop. This is accomplished with Tcl's **after idle** command.

```
body EventMgr::cmdrsp_idle {} {

    set _cmdrsp_idle 1
    # ... Return if queue is empty ...

    # dequeue and run callback
    set script [lindex
            $_after_queue(cmdrsp_idle) 0]
    set _after_queue(cmdrsp_idle) [lreplace
            $_after_queue(cmdrsp_idle) 0 0]
    ::after idle uplevel #0 $script
}
```

8 Simulating the DDSP chip interface

The nature of the DDSP's interface made testing our application problematic: many of the actions of our application are dependent on DDSP-generated events or data. The limited availability of the hardware emulation platform and the complexity of using the design simulation environment ruled out our using either to debug our software.

To solve this problem we constructed an [incr Tcl] based model of the DDSP Control InterFace (CIF). Methods in our Cif object allow us to generate events and fabricate data for the purpose of testing our software. This strategy allowed us to work independently of the DDSP development team and deliver tested code without disrupting their work in the lab.

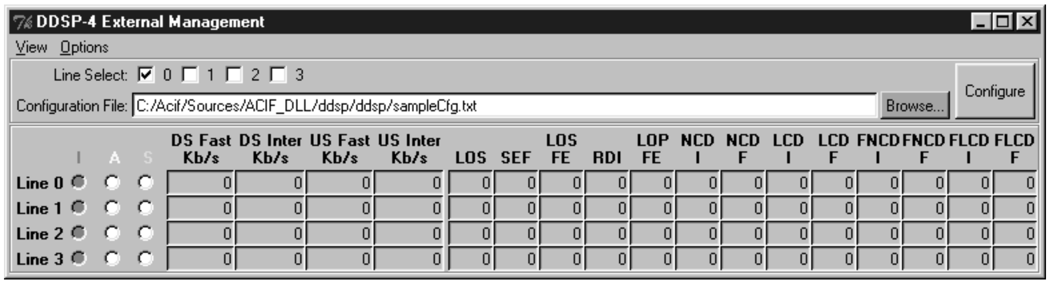
The Cif object composed other objects responsible for simulating various aspects of the DDSP interface. For instance the real CIF would periodically generate a counter value and signal this with an interrupt. Objects containing **after** based schedulers simulated these. These objects were made configurable with [incr Tcl]'s built-in **configure** method so Cif could be used in various testing scenarios.

9 Delivering a compelling demo

Our bind facility combined with Tk and BLT allowed us to easily construct displays that continually update in response to DDSP-generated events. This GUI proved to be a useful tool during chipset development and in demonstrations to our management and development partners.

Our main window, seen in Figure 1, contains a grid of Tk labels that report the type and number of various data transmission errors. The number of occurrences of each type of error is stored directly in the widget's -text option instead of in dedicated variables. When a DDSP error event occurs, code we bound to that event determines which error types occurred and increments the counts in the corresponding widgets. It also changes the foreground color of the updated widgets. To make the update happen, we simply set up the bindings needed to report data defects like this:

```
body DdspWin::CreateBindings {} {
    for {set i 0} {$i < $_numlines} {incr i} {
        $_ddsp bind <Line_${i}_Int-Data_Defect>
                +[code $this Defect Data $i]
    }
}
```


DDSP-4 External Management

View Options

Line Select: ☑ 0 ☐ 1 ☐ 2 ☐ 3

Configuration File: C:/Acif/Sources/ACIF_DLL/ddsp/ddsp/sampleCfg.txt

	I	A	S	DS Fast Kb/s	DS Inter Kb/s	US Fast Kb/s	US Inter Kb/s	LOS	SEF	LOS FE	RDI	LOP FE	NCD I	NCD F	LCD I	LCD F	FNCD I	FNCD F	FLCD I	FLCD F
Line 0	●	○	○	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Line 1	●	○	○	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Line 2	●	○	○	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Line 3	●	○	○	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 1. Main window

Conveniently, the above pattern is used in all the windows that update based on DDSP events. Using our Ddsp object's bind, we are free from the details of communicating with the DDSP.

Other windows can be launched from the main menu. Each window is managed by an object instantiated by the main window object. Figure 2 shows the Bit and Gain window. By linking together three BLT graphs, the engineer may analyze the interrelation between signal-to-noise ratio, bit allocation and attenuation for each modem frequency bin. Figure 3 shows one of the Performance Counter windows. We built this window with BLT's stripchart widget.

Conclusions

Constructing the majority of this application with Tcl and [incr Tcl] rather than C, enabled our success. Had we stayed on the path using mostly C, we simply could not have finished.

Object-oriented programming with [incr Tcl] made this code cleaner and easier to understand than our previous efforts.

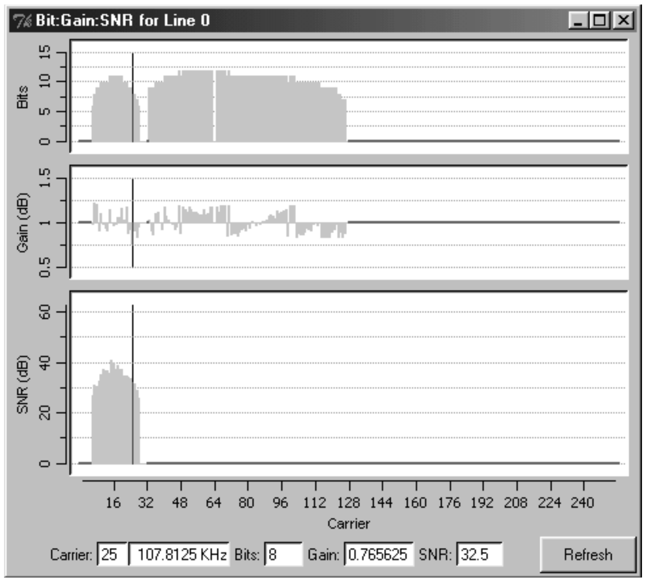
Tcl's `after` and `vwait` commands provide the necessary tools to interface to a chipset intended to operate in an interrupt-driven fashion. By managing application interactions behind the scenes, binding mechanisms allowed us to structure our application in a powerful way. A binding mechanism also gives users a means to extend the application without modifying the core code.

Last, but certainly not least, BLT delivered the graphics capabilities essential to present the complex data to our users in a familiar way.

References

[1] T. Copeland, D. Gardner and D. Karoly. "WinACIF: A Telecom IC Support Tool Using Tcl/Tk", *The Sixth Annual Tcl/Tk Conference Proceedings*, Pages 23-29, San Diego, CA, September 1998. USENIX.

[2] M. Harrison and M. McLennan, "Effective Tcl/Tk Programming: Writing Better Programs with Tcl and Tk," Addison-Wesley, 1997.



Carrier: 25 107.8125 KHz Bits: 8 Gain: 0.765625 SNR: 32.5

Figure 2. Bit and Gain window

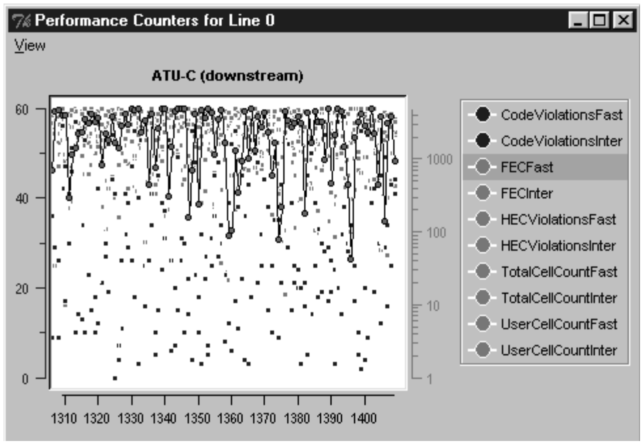


Figure 3. Performance Counter window