Supporting Tcl in Hardware

Scott Thibault

thibault@gmvhdl.com
Green Mountain Computing Systems, Inc.
P.O. Box 275
Richmond, VT 05477

Abstract

The use of high-level languages is known to increase both productivity and software quality, yet assembly language is still popular in the domain of embedded systems. We present a rational for the use of Tcl in embedded systems and for a hardware implementation of Tcl. We describe Tcl on Board!^{TM}, an embedded systems platform based on a Tcl processor. Key components of the Tcl hardware design are presented along with the limitations imposed on the language. An example Web server application is described that demonstrates reasonable performance and a very small memory footprint.

1 Introduction

One of the greatest pressures we face in the embedded systems market is to reduce the product development cycle, *i.e.* shorten time-to-market. While in the past, hardware development has been the dominating factor in the development cycle, software development is increasingly becoming the bottleneck in system design. One of the best approaches to reducing software development time is the use of high-level languages. In this paper, we explore the implementation of an embedded processor for Tcl that gives embedded system programmers access to the benefits of a high-level scripting language without sacrificing memory or performance.

The advantages of using scripting languages for PC applications has been well demonstrated, and their use is widespread. Likewise, there are a number of benefits of using a scripting language, and Tcl in particular, for embedded systems. These include:

- high-level constructs shorten development time,
- the lack of pointers reduces errors,
- automatic memory management reduces errors,
- simple network support shortens development time,
- simple syntax is easy to learn, and
- cross-platform support permits reuse.

Relative to other popular scripting languages, Tcl has some advantages for embedded systems. First, Tcl has the simplest syntax making it very easy to learn. Second, Tcl supports reference-counting memory management as opposed to garbage collection. Garbage collection can be problematic in embedded systems because it exhibits unpredictable timing. Third, Tcl supports easy to use TCP/IP support and extensive string handling capabilities. This is especially desirable in embedded systems with wireless capabilities (*e.g.*, Bluetooth technology) or distributed processors. These systems communicate with other devices and/or the user through PDAs or other remote interfaces. The simplicity of creating network servers and processing strings in Tcl make it an ideal choice[1].

The next section discusses the decision to implement Tcl in hardware rather than porting the standard interpreter to existing embedded processors, or developing a Tcl compiler. The architecture of the processor is described in section 3. Section 4 discusses the compiler and the implications of our approach on language support. The results of this work are presented in section 5 followed by some conclusions in section 6, and future directions in section 7.

[1]Many TCP/IP protocols are based on exchanging command strings.

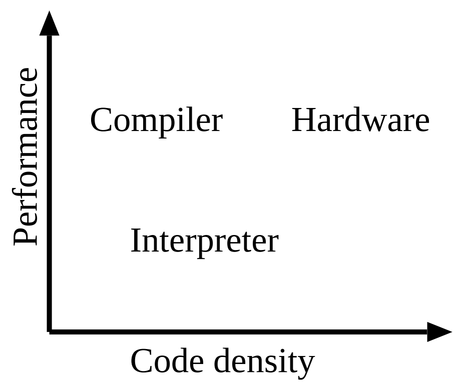

Figure 1: Implementation strategy tradeoff.

2 Motivation

Given that we desire to use a high-level language to improve productivity and code quality, how do we implement that language? There are three major methods of implementation to consider: interpretation, native compilation, or hardware implementation. The correct choice depends on the target application.

When considering the high-performance computing market, implementation in hardware does not make sense. That is because it is easy to compile a language to a general-purpose processor and achieve high performance due to the amount of effort expended in optimizing these processors. However, when we consider the embedded systems market, this argument no longer holds.

The embedded systems market is dominated by 8-bit and 16-bit processors that are not generally designed for high performance. Additionally, these systems are constrained by memory limitations such that code size becomes an important factor. For this reason, we find that assembly language is still widely used in this domain. Figure 1 illustrates the tradeoff involved when considering both performance and code density (*i.e.*, the compactness of a program's representation). The hardware implementation is the only choice that can achieve small programs *and* good performance.

The problem of code size in the compilation approach is clear to see for a dynamic language like Tcl. Consider a simple command such as `expr $a+1`. Using the compiler approach, code must be generated to load the variable `a`, verify that it has a value (*i.e.*, not unset), check the type of the value and call a conversion function if necessary, and check for/invoke any traces on the variable. In contrast, a single instruction could be designed to perform all these tasks using the hardware approach.

A number of processors have been developed for specific languages including LISP [7], Forth [1], and even Java [3]. While Java has some of the features of Tcl (*e.g.*, automatic memory management), it is not as high level as Tcl, requires garbage collection, and the hardware implementations target the high-end processor market (*i.e.*, 32 bit). The other languages that have hardware implementations are not comparable in terms of ease-of-use, networking, and/or popularity.

3 Hardware Implementation

Having narrowed the implementation strategy down to a hardware implementation, there are still various hardware approaches to consider. The first approach would be to implement a pure Tcl interpreter in hardware that loads command strings, parses, and executes them. This approach would be very costly in both chip size and complexity. At the other end of the spectrum, an existing processor could be modified with special instructions that would make it a better target for compilation. This approach can improve both performance and code size. A third approach is to design an instruction set specifically for Tcl that can be easily executed in hardware. Although this approach has higher start-up costs (*i.e.*, processor design) than modifying an existing processor, it provides the highest code density. Due to the memory limitations imposed by embedded systems, our implementation takes this third approach.

3.1 Instruction Set

The instruction set is designed using a simple stack-based architecture. Thus, the basic instruction set includes instructions to push and pop values, and arithmetic instructions that take operands from the stack and push the result. This architecture was chosen for simplicity and compactness[2]. However, the same principles used to support Tcl here would also apply to a register-based architecture.

The principle goal of the instruction set design is to eliminate the performance and code-size overhead re-

[2]Stack architectures are known to have higher code densities than register-based architectures.

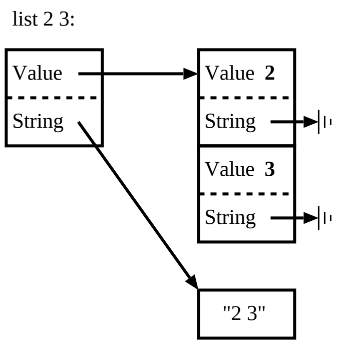

Figure 2: Example representation of the list 2 3 with dual-ported objects.

quired to implement the high-level features of Tcl such as automatic memory management, dynamic typing, and list manipulation. The following sections describe how each of these features is addressed.

3.1.1 Dynamic Typing

The semantics of Tcl are defined in terms of strings. For efficiency, Tcl implementations, including the standard Tcl interpreter, must use other representations internally for storage. It would be unreasonable, for example, to evaluate expressions using only strings. The consequence is that dynamic typing is required.

In a similar manner as the Tcl interpreter, we represent values using dual-ported objects. That is, objects may store one internal representation and/or a string representation. Three internal representations are supported: integer, list, and array. Each object also carries a type that describes the internal representation, or an empty flag to indicate that the internal representation is not in use. The string representation is stored as a pointer, and may be NULL to indicate that the string representation is not available. Figure 2 shows an example of how the list 2 3 might be represented in memory with dual-ported objects. This list has both a structural representation for quick access to the individual elements 2 and 3, as well as a string representation that could be used with `puts` or some other string operation.

Operators calculate only the most convenient representation, and alternative representations are calculated only as they are needed. For example, the `string range` command will return an object with only a string representation, and the `expr` command will return an object with only an integer internal representation. If the `string range` command were used as an operand of the `expr` command, then the string result would be converted to an integer during evaluation of the `expr` command.

We implement dynamic typing in hardware with zero overhead using hardware trapping and special typed push instructions. For example, operations of the `expr` command are pushed on the stack using the `pushInt` instruction. During execution of a `pushInt` instruction, the type of the loaded object is verified to be an integer and a hardware trap is signaled otherwise. The hardware trap invokes a conversion routine that will create an integer representation of the object. The trapping logic operates in parallel to the normal push operation.

3.1.2 Memory Management

Automatic memory management is an important feature of Tcl that increases both programmer productivity and quality. Unlike other languages, Tcl relies on reference counting for automatic memory management. The benefit of reference counting is that memory is freed as soon as it becomes available in a predictable way. This can be very important in an embedded system.

The standard Tcl interpreter allocates all objects from the heap, and maintains a stack of pointers to heap allocated objects. Using this approach, every operation requires an extra memory read to dereference the object pointers in the stack. To eliminate this extra read, our approach stores objects on the stack.

This choice has an important impact on memory management. If copies of objects are stored on the stack, then objects cannot be reference counted. The reason being that objects contain pointers to other objects (in the case of lists and arrays) and/or strings. When a heap-allocated object's reference count reaches zero, the object is no longer needed, but there is no way to know if there is a copy of the object on the stack somewhere that requires access to the object's string or list/array pointers.

The solution is to move the reference counts to the strings, lists, and arrays. When objects are copied, the reference counts of any strings, lists, or arrays pointed to by the object are incremented. Reference counts are decremented likewise when the object is destroyed. Figure 3 illustrates the difference between our implementation and the implementation used by the Tcl interpreter.

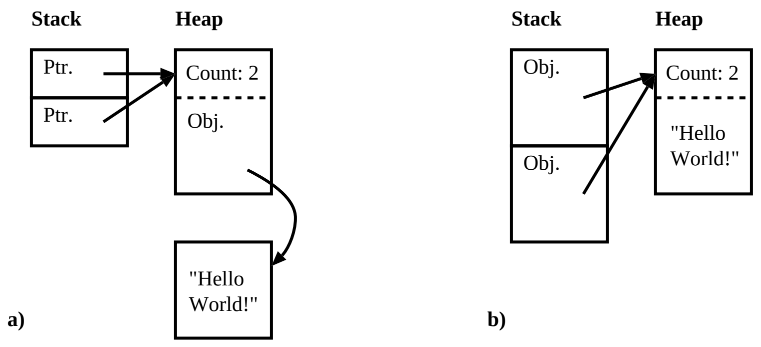

Figure 3: Reference counting example using a) the standard Tcl implementation versus b) our implementation.

Reference counting is implemented in hardware using special instructions. A single instruction examines the type of an object and reads, updates, and writes the object's reference count, if necessary. If the result of the update reduces the count to zero, a hardware trap is generated. The trap invokes a routine to perform deallocation.

For example, to load a global variable, the following instructions might be generated:

```
loadGlobal2 33577
pushObj
incObjRef
incStrRef
```

The `incObjRef` and `incStrRef` instructions in this example cause the reference counts of any referenced objects (lists/arrays or strings respectively) to be incremented. Although the overhead has been greatly reduced, it would better to eliminate the overhead altogether if possible. For example, if the type of the object is known to always be an integer then these two instructions can be eliminated. Through type analysis, the compiler can determine the type of objects in many cases and significantly improve code density by eliminated unnecessary reference counting instructions.

3.1.3 List Manipulation

Since Tcl does not provide user defined types, lists are an essential part of the language. In our implementation, arrays (the only other complex data structure available in Tcl) are also implemented with lists. Consequently, the performance of list manipulation is very important.

Lists are represented internally by a linked list of arrays. An array based representation allows constant-time access to list elements. Using a linked list of arrays rather than a single array helps reduce the time required to perform append operations. Append is implemented by re-sizing the array (which may require copying the whole array) unless the array is large, in which case, a new array would be appended to the linked list.

To maximize list performance, a special purpose instruction is used to access list elements. This instruction inspects only the first array in the list of arrays to compute the address of the list element or signal a trap if the index is out-of-bounds. The trap handler is responsible for traversing the list of arrays to locate elements beyond the first array. A single array in the linked list will be re-sized up to 4095 elements before a new array node is added to the list. Thus, for most list accesses, a single instruction performs bounds checking and address calculation.

3.2 Runtime system

The instruction set provides only the necessary primitives to implement the core Tcl commands in an efficient manner. Some of the core commands are implemented by the compiler (*i.e.*, translated into a series of instructions). The commands implemented by the compiler include `set`, `if`, `while`, `foreach`, `for`, `switch`, `catch`, `error`, `expr`, `incr`, `return`, `break`, and `continue`. The remaining commands are implemented in a runtime library.

Unlike most processors, the Tcl processor does not have

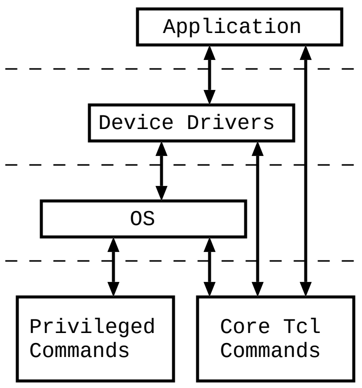

Figure 4: Three tiers of programmability.

an assembler for instruction-level programming. The reason being that the processor is so specialized to Tcl programs that it does not make sense to program in anything else. Thus, the runtime library is written entirely in Tcl.

Operating System (OS) code does, however, required access to certain hardware features such as direct memory access. These features are exposed through the compiler as built-in commands. These low-level commands, together with standard Tcl, are used to implement memory management and the numerous trap handlers described in the previous sections.

In order to support the varying needs of OS code, device driver code, and application code, we define 3 tiers of programmability corresponding to these three levels of code as shown in figure 4. In the OS tier, program code has direct access to memory via pointers and may not be memory safe. For example, the `string length` command can be implemented by the following code:

```
    derefWordPtr [expr [getStr $str] + 1]
```

The `getStr` command extracts from the dual-ported `str` object a pointer to the object's string representation. The `derefWordPtr` is then used to directly read the given address (*i.e.*, the second word of the string where the length is stored).

In the device driver tier, programs do not have direct access to memory, but do require the ability to manage buffers and perform I/O operations with devices. For performance reasons, device drivers must be able to read and write buffers. This is accomplished by offering read and write operations on strings. Figure 5 shows an example procedure taken from the TCP/IP

```
proc ip_fill_header {header dst_hi \
                     dst_lo proto} {
    global ip_id
    global ip_addr0
    global ip_addr1
    global ip_addr2
    global ip_addr3

    swrite header 0 69
    incr ip_id
    swrite_word_big header 4 $ip_id
    swrite header 6 64
    swrite header 8 64
    swrite header 9 $proto
    swrite header 12 $ip_addr0
    swrite header 13 $ip_addr1
    swrite header 14 $ip_addr2
    swrite header 15 $ip_addr3
    swrite_word_big header 16 $dst_hi
    swrite_word_big header 18 $dst_lo
}
```

Figure 5: Example TCP/IP driver code.

driver. This procedure fills in the IP header portion of an IP packet using the `swrite` and `swrite_word_big` commands. These OS commands perform destructive writes on strings in a memory safe way (*i.e.*, the string is automatically expanded if the destination is out-of-bounds). Although memory safe, the `swrite` commands require careful programming because the string is destructively updated, not copied. Bytes and words can be directly read from strings with `sread` commands. Standard Tcl should support an efficient method of accessing string data via the `binary` command, but that is not the case currently.

Finally, in the application tier, program code can only use the standard core Tcl commands. These three tiers of programmability are enforced by the compiler. A compiler option is required to access commands in the device driver tier, and the OS tier is not available to end-users.

4 Language Limitations

As shown in the previous section, the most important features of Tcl (dynamic typing, automatic memory management, and complex data types) can all be implemented efficiently in hardware. There are a number of language features, however, that are troublesome in terms of performance and memory constraints. These difficulties are not specific to the hardware approach, but due to the need to compile. There are several papers that discuss the issues of compiling Tcl [4, 2, 5, 6]. The follow sections summarize some of the main issues.

4.1 Type Analysis

Much of the overhead involved in manipulating Tcl objects can be avoided if the type of the object (*i.e.*, integer, list, string, ...) is known. As discussed in section 3.1.2, for example, reference counting instructions can be completely eliminated if the type of the object is guaranteed to be an integer. It is therefore important to be able to determine the types of as many variables as possible at compile time using type analysis.

There are several Tcl features that make this type analysis difficult. First, a variable name can be computed at runtime. For example, after the command `set $a 1`, the compiler cannot assume anything about any variable types. We do not support runtime computed variable names. This is a reasonable limitation since the same effect can be achieved using arrays.

Second, the `trace` command eliminates almost all possibility of determining variable types because the trace command can modify local variables in the context of the traced operation. The `trace` command is not currently supported, and would be difficult to implement without changing its semantics. Since procedures can access variables with the `upvar` command, there is no real reason that the trace command should be evaluated in the context of the traced operation, but those are the current semantics of Tcl.

Third, the `uplevel` command can also severely inhibit the ability to perform type analysis. The `uplevel` command is also not supported. However, the `upvar` command can be used to achieve the same results. The `upvar` command is supported in the limited case that the first argument is a variable access, and that variable is a parameter of the command. With this limitation, analysis of the effects of `upvar` can be performed at compile time.

Finally, command names can be computed at runtime as in `[$callback $a]`. Since the command being invoked is not known at compile time, there is no way to determine what side effects the command will have. Therefore, the compiler will not be able to assume anything about variable types after the call. We overcome this issue by not permitting commands with side effects to be invoked in this manner. Additionally, the current implementation does not include symbol table information in the program. A special command must be used to create a function pointer variable, which stores the command's address as calculated at compile time. This was a temporary "hack" that will be eliminated in the next version. The `rename` command is also unsupported because it prevents commands from being invoked directly by address.

4.2 Dynamic Evaluation

Of course, `eval` cannot be implemented without a runtime compiler or interpreter. While this is certainly possible, it would likely have an adverse effect on memory requirements. For the same reason, commands that accept scripts as arguments must have constant arguments (*i.e.*, `catch`, `for`, `foreach`, `if`, `proc`, `switch`, and `while`). If dynamic evaluation is supported, it would also degrade the ability to perform type analysis, as described in the previous section, due to indeterminable side effects.

4.3 Miscellaneous Limitations

This section lists the commands that are not particularly difficult to implement, but have not yet been implemented or do not make sense in the given context. The processor does not have a clock at this point, so the `after`, `clock`, and `time` commands are unavailable. MS Windows specific commands are not available. These commands are `dde`, `registry`, and `resource`.

The system does not support any file systems at this time. Consequently, the commands `cd`, `eof`, `exec`, `file`, `fcopy`, `glob`, `load`, `msgcat`, `open`, `pwd`, `seek`, and `tell` are unimplemented. TCP sockets, however, are supported along with the necessary commands for opening, reading, writing and closing sockets.

Unicode is not supported in this version, and thus `encoding` is not implemented. The `exit`, `history`, `interp`, `memory`, and `pid` commands have no useful meaning in our context and are unimplemented.

Namespaces are not currently supported, but will likely be supported in future releases. This includes the commands `namespace`, `package`, and `variable`.

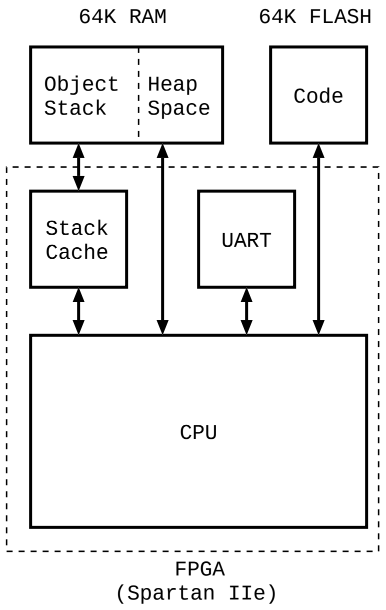

Figure 6: Prototype configuration.

Finally, the `bgerror`, `fblocked`, `flush`, `regexp`, `regsub`, `subst`, `unknown`, and `vwait` commands are unimplemented due to lack of time. All these commands can be implemented and regular expressions have a high priority.

5 Results

We have implemented a working prototype of our Tcl on Board!^{TM} platform. The platform includes a compiler, an instruction-set simulator, various debuggers, runtime libraries and a 16-bit Tcl processor. The following sections describe the prototype implementation and a demonstration Web server application developed for the prototype.

5.1 Prototype Implementation

The configuration of the prototype is illustrated in figure 6. The processor is implemented in VHDL, a hardware description language, and can be realized in any hardware form. A prototype processor has been implemented in programmable logic on a Xilinx Spartan IIe development board. The Tcl processor, including a stack cache but excluding the UART, is approximately 32k logic gates. This size is within the normal range of sizes associated with 16-bit processors as well as Java processors.

The processor operates at 5 MHz on the FPGA, although it is capable of operating up to speeds of 36 MHz[3]. This figure is also comparable to common 8-bit and 16-bit processors. On average, instructions require 3.5 cycles to execute, yielding approximately 10 MIPS at 36 MHz. Of course, a VLSI implementation would be capable of much higher performance.

The 16-bit Tcl processor has an address space of 128k bytes. The prototype has 64k bytes of flash memory for program code, and 64k bytes of RAM for stack, global variable, and heap space.

The runtime libraries currently include the operating system and core Tcl commands, a networking library implementing PPP and TCP/IP, and a serial device driver. The total sizes of these libraries are 38k bytes, 12k bytes, and 1k bytes respectively. The compiler supports function-level linking, which means these sizes do not necessarily reflect the size of complete programs. The instruction set achieves good code density that is about the same as the code density of using C without compiler optimizations.

5.2 Web Server Application

To demonstrate the Tcl on Board!^{TM} platform, we have built a small Web server application. The Web server is about one page of Tcl code (see Appendix A), and implements a simple interface to a hypothetical multi-zone climate control system. The server accepts HTTP requests over a serial port using PPP in order to retrieve and modify the temperature settings for each zone and time of day. Any standard Web browser can be used to communicate with the embedded server.

Using the 5 MHz prototype with a 9600 baud serial connection to the on-board UART, an HTTP request takes 5.25 seconds to complete. 4.71 of those 5.25 seconds is consumed by communication over the serial line. We estimate that a 32 MHz version with a 56k serial port could handle requests in less than a second. The total program size (including all libraries) is just 32k bytes.

[3]The 5 MHz speed is used due to limitations of the flash memory.

6 Conclusion

We have designed and implemented a working embedded systems platform for the Tcl language based on a Tcl processor. In this paper, we have presented an overview of the design including the major hardware optimizations and limitations of the platform. Although the Tcl processor provides the simplicity, productivity, and robustness of a high-level scripting language, the presented results demonstrate that the Tcl processor is similar in performance and size to other embedded processors. These results are achieved through a combination of an optimized Tcl instruction set, compiler optimizations such as type analysis, and novel hardware constructs like hardware reference counting.

The platform is further demonstrated with a working Web server example. The Web server example has a modest footprint of 32k bytes, and operates within an acceptable amount of time for an embedded device. The example also demonstrates the power of Tcl with an implementation requiring only a single page of code.

7 Future Work

The next two immediate objectives for the Tcl on Board! platform are to complete the runtime libraries, and begin optimizing performance and code density. Most notably, the runtime libraries lack support for regular expressions, which will be important for embedded system applications.

There are a number of ways to optimize the platform that have already become obvious. First, the type analysis performed by the compiler can be improved to be able to determine more variable types. For example, the compiler currently makes no attempt to determine the type of procedure parameters. Through global program analysis, it would be possible to determine the types of parameters and optimize the procedure body. Second, the compiler could be augmented with directives that would allow the developer to directly specify a variable's type. We envision that these directives would be used by the OS developers and not by application developers, as this would reduce the productivity advantage of using a scripting language. Finally, the instruction set can be optimized to increase code density by identifying common instruction patterns and implementing them in a single instruction.

A long term direction might be to investigate the possibility of a 32-bit implementation. A 32-bit version would significantly reduce memory I/O. The purpose would not be to compete with other 32-bit processors, but we expect a 32-bit version might be able to compete in the 16-bit market.

References

[1] John Hayes and Susan Lee, "The architecture of the SC32 Forth engine", Journal of Forth Application and Research, v5 n4, 1989.

[2] Brian T. Lewis, "An On-the-fly Bytecode Compiler of Tcl", Proceedings of the 1996 Tcl/Tk Workshop, Monterey, July 1996.

[3] Harlan McGhan and Mike O'Connor, "PicoJava: A Direct Exection Engine for Java Bytecode", IEEE Computer, pp. 22-30, October 1998.

[4] Forest Rouse and Wayne Christopher, "A Typing System for an Optimizing Multiple-Backend Tcl Compiler", Proceedings of the 1997 Tcl/Tk Workshop, Boston, July 1997.

[5] Forest Rouse and Wayne Christopher. "A Tcl To C Compiler", Proceedings of the 1995 Tcl/Tk Workshop, Toronto, July 1995.

[6] Adam Sah and Jon Blow, "TC: A Compiler for the Tcl Language", Proceedings of the 1993 Tcl/Tk Workshop, June 1993.

[7] Guy Steele and Richard Gabriel, "The evolution of Lisp", History of Programming Languages, pages 233-309, ACM Press, 1996.

A Web Server Example

```
proc update_zone {zone no assign} {
    set pos [expr [string first "=" $assign] + 1]
    set val [string range $assign $pos end]
    write_reg [expr ($zone << 1) + $no - 2] $val
}
proc zonehtml {zone no} {
    set data [read_reg [expr ($zone << 1) + $no - 2]]

    set hour24 [expr ($data >> 10) & 31]
    if {$hour24 > 12} {
        set hour12 [expr $hour24 - 12]
    } else {
        set hour12 $hour24
    }
    set quarter [expr ($data >> 8) & 3]
    set min [expr ($quarter << 4) - $quarter]
    set temp [expr $data & 255]
    if {$hour24 == 24 || $hour24 < 12} {
        set ret [format "<TD>%d @ %d:%02d am</TD>" $temp $hour12 $min]
    } else {
        set ret [format "<TD>%d @ %d:%02d pm</TD>" $temp $hour12 $min]
    }
    return $ret
}
proc www_req {handle} {
    set req [split [gets $handle] " ?+"]
    set loc [lindex $req 1]
    puts $handle "<HTML><BODY>"
    switch $loc {
        /current {
            puts $handle "<TABLE border=5><TR><TD>Zone 1</TD><TD>Zone 2</TD></TR>"
            puts $handle [join [list "<TR>" [zonehtml 1 0] [zonehtml 2 0] "</TR>"]]
            puts $handle [join [list "<TR>" [zonehtml 1 1] [zonehtml 2 1] "</TR>"]]
            puts $handle "</TABLE>"
        }
        /update {
            update_zone 1 0 [lindex $req 2]
            update_zone 1 1 [lindex $req 3]
            update_zone 2 0 [lindex $req 4]
            update_zone 2 1 [lindex $req 5]
            puts $handle "Request completed."
        }
    }
    puts $handle "</BODY></HTML>"
    close $handle
}
proc accept {handle ip port} {
    fileevent $handle readable [mk_cmd www_req $handle]
}
proc main {} {
    ppp_attach [init_dev] [list 192 168 0 25]
    socket -server accept 80

    while {1} {
        update
    }
}
```