A Primer on Embedded Systems with a Focus on Year 2000 (Y2K) Issues

Understanding the Millennium Bug

TR-111189

Final Report, August 1998

EPRI Project Manager
J. Weiss

EPRI • 3412 Hillview Avenue, Palo Alto, California 94304 • PO Box 10412, Palo Alto, California 94303 • USA
800.313.3774 • 650.855.2121 • askepri@epri.com • www.epri.com

ORDERING INFORMATION

Requests for copies of this report should be directed to the EPRI Distribution Center, 207 Coggins Drive, P.O. Box 23205, Pleasant Hill, CA 94523, (510) 934-4212.

CITATIONS

This report was prepared by

Applied Microsystems Corporation
5020 148th Ave N.E.
Redmond, WA 98052

Principal Investigator or Author
Arnold S. Berger, Ph.D.

This report describes research sponsored by EPRI and Applied Microsystems Corporation.

The report is a corporate document that should be cited in **the** literature in the following manner:

A Primer on Embedded Systems with a Focus on Year 2000 (Y2K) Issues, EPRI, Palo Alto, CA, 1998. TR-111189

REPORT SUMMARY

This report serves as a primer on embedded systems with a focus on Year 2000 (Y2K) issues. Because embedded systems can be impacted by Y2K issues, a better understanding of embedded systems can help our industry remediate these issues.

Background
The Y2K issue potentially impacts all types of equipment with embedded systems. At times, these systems have been designed by our equipment suppliers. More often, however, they are designed by embedded system companies that specialize in microprocessor design and are then supplied as subsystems to system suppliers. Consequently, often the expertise in how these devices work is outside of industry experience. Even though embedded systems are found in many varied applications, they share many common characteristics. With Y2K issues potentially impacting the microprocessors in embedded systems, it is important to have a better understanding of how these devices work and how date-related issues can impact them. Understanding how dates can be transmitted between systems with real time clocks, and even systems without real time clocks, is essential for testing these systems for Y2K susceptibility.

Objectives
- To develop a better understanding of how embedded systems use date-related information
- To better understand how Y2K issues can impact embedded systems and to develop testing techniques

Approach
EPRI contracted with a vendor experienced in the design of microprocessor-based systems in order to develop an embedded systems primer using current information on typical utility industry applications.

Results
This primer provides an excellent source of information for technicians, engineers, managers, and other personnel assigned to Y2K programs. It addresses embedded system hardware and software design, as well as application programming interfaces (APIs). It also includes a discussion of how dates are transmitted, an example of how

dates and data are transmitted in typical embedded systems, analytical tools, and Y2K analysis strategies.

EPRI Perspective

Y2K issues can affect all microprocessor-based date-related applications. This primer can help provide a basic understanding of embedded systems and the potential impacts of Y2K issues. In a parallel effort, EPRI and the contractor are developing a tool that can be used to determine if embedded systems are exchanging date information across serial communication links.

TR-111189

Interest Categories
Applies to all categories that use microprocessors

Keywords
Year 2000
Embedded systems
Microprocessors

ABSTRACT

Most of the Year 2000 (Y2K) discussion has centered around its impact on mainframe and other back-office software. Very little discussion has been devoted to its impact on embedded systems. While mainframe computers are the database and financial engines of our society, it is the embedded systems that control the operation of our manufacturing base. Embedded systems, defined as systems operating in a real time environment using dedicated microprocessors, traditionally are not under the aegis of the IT organization. Consequently, addressing embedded systems for Y2K is just recently coming to the forefront.

The Y2K issue potentially impacts all types of equipment with embedded systems. In general, these systems were designed by outside vendors or third party system integrators. Consequently, the expertise in how these devices work is many times outside of our experience. Even though embedded systems are found in many varied applications, they share many common characteristics.

With Y2K (date-related issues) potentially impacting the embedded system microprocessor, it is important that we have a better understanding of how these devices work and how date-related issues can impact them. Understanding how dates can be transmitted between systems with real time clocks, and even systems without real time clocks, is essential for testing these systems for Y2K susceptibility.

This primer provides an excellent source of information for technicians, engineers, managers, and other personnel assigned to Y2K programs. The primer consists of a discussion of embedded system hardware and software design, application programming interfaces (APIs), a discussion of how dates are transmitted, an example of how dates and data are transmitted using a flow transmitter as a test case, analytical tools, and Y2K analysis strategies.

CONTENTS

LIST OF FIGURES

1

INTRODUCTION

Unlike most projects, the completion date of the millennium, December 31, 1999, is not going to slip. It is common knowledge that there *may be a* problem surrounding our computer systems' ability to deal with a date that apparently rolls over from 99 to 00. While the financial community is well-aware of the impact of people losing money due to 30 year old COBOL-based computer programs miscalculating the date, for many people, the impacts of the date problem on embedded systems are less easily quantified and understood.

The EPRI-sponsored, Year 2000 (Y2K) Embedded Systems Program with a significant level of participation is an indication that the industry is concerned enough about it to want to learn more. My goal is to try to help you understand the nature of the problem and how it manifests itself within your embedded systems at a more technical level, rather than present a superficial understanding of the source of the Y2K problem. Hopefully, once you have acquired a basic understanding of the nature of these systems, you will be better able to make timely and cost effective decisions about how to deal with potential problems within your particular environment.

At this point you may be wondering how, given the variety of applications that embedded systems are used for, can I cover all of this material in any reasonable depth? Well, I can't tell you about every embedded system that has ever been designed, but I can share with you common characteristics of most of systems that you are likely to have in your own facilities. Hopefully, this knowledge will help you sort out your embedded systems and plan your Y2K testing methods.

Figure 1-1 is an aggregate embedded system showing as many of the variant characteristics as I could envision. While there are a lot of variants, the number is still finite. Understanding how these common threads interact within the system is enough to understand the general behavior of most embedded systems.

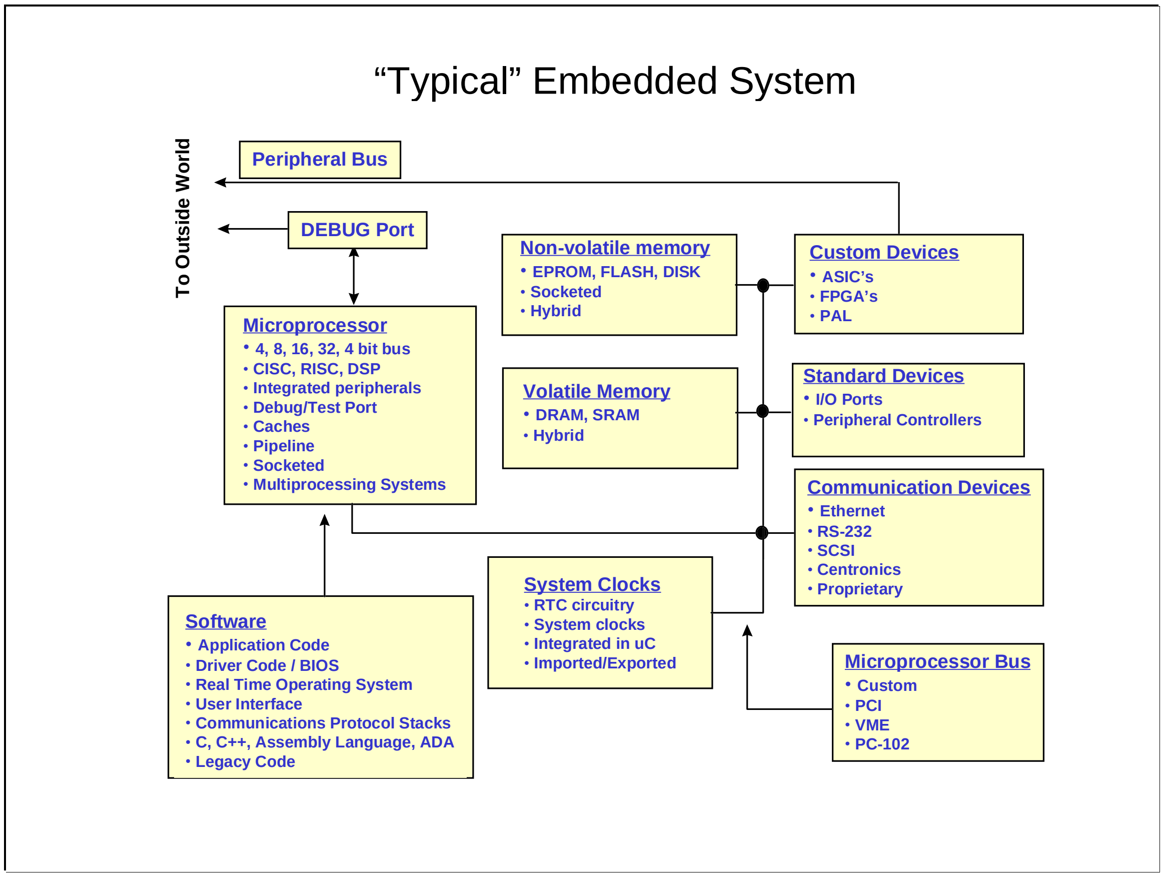


Figure 1-1
Common components of an embedded system

2

EMBEDDED SYSTEMS

What is an embedded system? Simply, it is a device, or combinations of devices[1], that contain ***dedicated computers*** as controllers and calculating elements. A *dedicated computer* is a computer that is programmed to do a single task. The task may be very complex, like controlling an aspect of the flight avionics for a Boeing 777 wide-body aircraft, requiring millions of lines of computer instructions and high-performance microprocessors.

The task might be very simple, perhaps running a lawn sprinkler controller or a dimmer switch on a light. In all cases, the defining factor is that it is not a general-purpose, programmable device, like the personal computer on your desk. That is not to say that a PC can't be an embedded system controller. In fact, it can and often is used in this manner. For example, many numerical control machine tools use whole dedicated computers as embedded control elements. The key is that it does a single controlling task, rather than run Excel, Word, etc.

Most embedded systems are controlled by one or more microprocessors. These integrated circuits have been around since the mid 1970's and have been doubling in complexity every 18 months since that time[2]. The Pentium-class microprocessor that we are all familiar with from our desktop PC is the exception rather than the rule in embedded systems. Most embedded systems are extremely cost-sensitive, so microprocessors in the $1.00 to $25.00 range dominate embedded applications.

In addition, manufacturers tend to drive down the costs of embedded systems by going to devices that have *higher levels of integration* than the microprocessor. These devices are called *microcontrollers*, in recognition of the fact that they dominate the embedded controller application space. In fact, over 90% of the total embedded marketplace is dominated by one device and its derivatives, the 8-bit 8051 microcontroller. Figure 2-1 illustrates the differences between a microprocessor and a microcontroller. Since the *microprocessor* (left side) interfaces with the peripheral devices across external circuits paths (the copper traces on a printed circuit board), it is generally straightforward to

[1] Embedded systems also tend to be hierarchical, with larger systems built upon smaller ones.
[2] This fact, known as Moore's Law (after Gordon Moore, a founder of Intel) has yet to breakdown.

observe data moving through an embedded system that uses a discrete microprocessor and discrete peripheral devices.

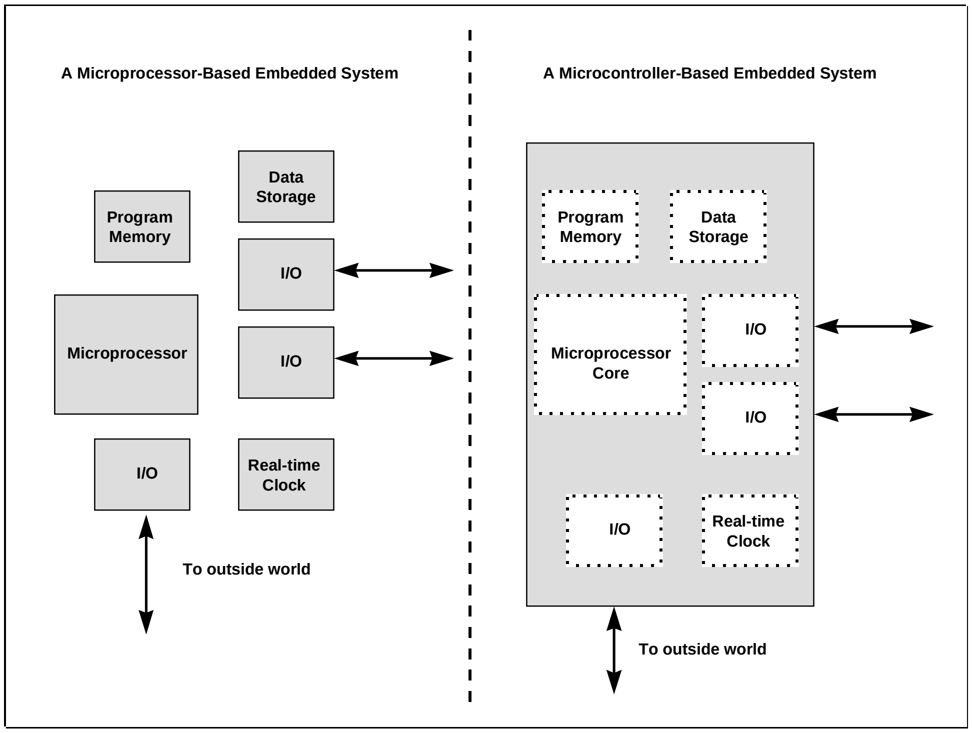


Figure 2-1
An embedded system based upon a microprocessor (left side), and a highly-integrated microcontroller (right side). Note that all peripheral devices and program memory are contained within the microcontroller circuitry and not available to outside visibility

In contrast, the microcontroller buries its Input/Output (I/O) devices on the single silicon die, within its package. This means that observing the system behavior is much more difficult, if not impossible. Specialized tools are often required and even these tools may only provide incomplete data at best.

Finally, in my opinion, the explosion of growth of embedded systems has been due to one property of the microcomputer: the ability to make a decision based upon the state of external events, thus altering its control flow. This is very difficult to do in most electromechanical systems, but pretty simple to do if you use a computer as a decision element. Look at your car if you want a good example of that fact. Exhaust emissions are down to 10 percent of what they were 20 years ago, performance is up, and the carburetor has followed the buggy whip into obscurity. The engine management microcontrollers are the decision making elements in your car’s fuel and ignition systems.

Let's look more deeply into the inner world of the embedded system and try to understand what is going on and why. Figure 2-2 is a block diagram of a "typical" embedded system. The microprocessor is the heart of the system. The rest of the devices exist to provide support and data to the microprocessor. As I mentioned earlier, the microprocessor may be a microcontroller, with some or all of the peripheral devices incorporated on-chip with the *central processing unit* (CPU).

The next functional block of interest is the clock generation and distribution circuit. This is NOT the same as the real-time clock that is causing all the Y2K aggravation. Microprocessors depend upon precisely spaced timing pulses in order to synchronize all of their internal and external operations. The source of these pulses is (confusingly) called the "clock". For those of you constantly upgrading your PC, this is the clock frequency they refer to. Since every operational cycle of the processor depends upon a set number of clock pulses, the faster the clock pulses arrive, the faster the processor can execute its instructions. For example, a fairly typical microprocessor may run at a 20 MHz input clock frequency. This clock is in the form of a square wave with exactly 50 percent duty cycle, oscillating between zero volts (low state) and five volts (high state). Suppose that an average instruction cycle takes four clock cycles to complete. The reciprocal of 20 MHz is 50 nanoseconds[3] (nsec). Therefore, the instruction takes 4 x (50×10^{-9} sec), or 200 nsec, or about five million instructions per second.

Other microprocessor attributes of interest are the width (number of bits) in the address bus and the data bus.[4]

[3] 50 nsec = 50 x 10^{-9} seconds, or 50 billionths of a second.
[4] The term ***Bus*** is used to define a set of grouped signals that are like a spinal cord for the system. The various sub-systems connect to the bus, much like driveways onto a roadway, and allow devices to easily communicate to the microprocessor and to each other.

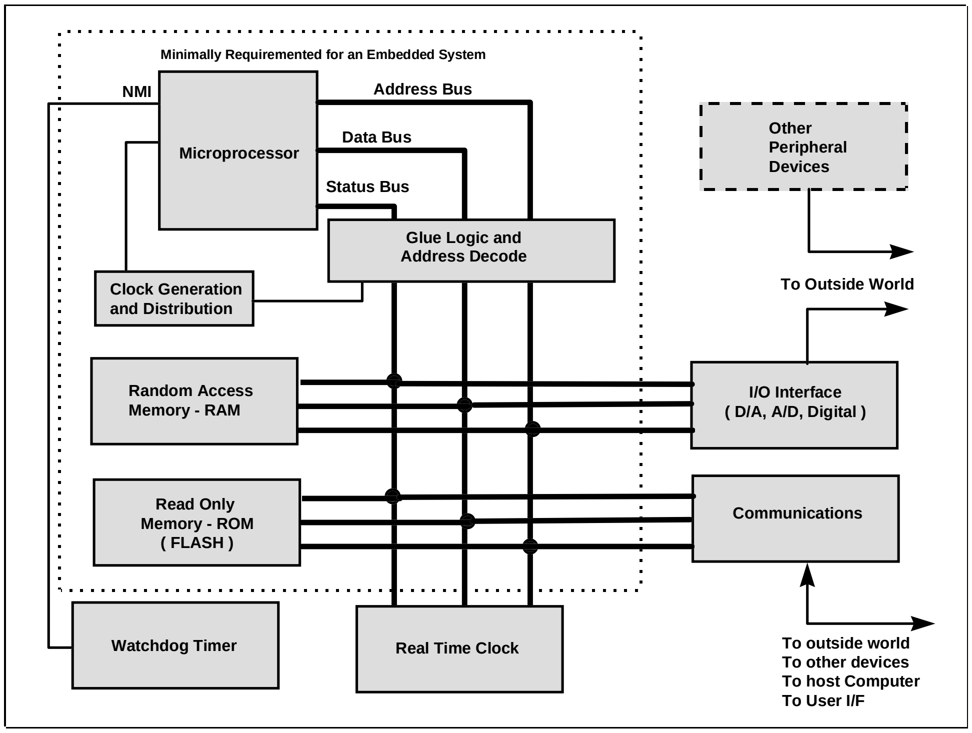

Minimally Requiremented for an Embedded System
NMI
Address Bus
Data Bus
Status Bus
Microprocessor
Other Peripheral Devices
Glue Logic and Address Decode
Clock Generation and Distribution
To Outside World
Random Access Memory - RAM
I/O Interface (D/A, A/D, Digital)
Read Only Memory - ROM (FLASH)
Communications
Watchdog Timer
Real Time Clock
To outside world
To other devices
To host Computer
To User I/F

Figure 2-2
Block Diagram of a "typical" embedded system. The dotted line defines the minimum number of required elements for the system to operate by itself

The width of the address bus, (see Figure 2-2) determines the amount of memory that the processor is capable of addressing. Processors like the 386, 486 and Pentium processors have 32 address bits. This corresponds to 2^{32} distinct physical addresses in the processor's address space, roughly 4.3 billion addresses. Now the typical embedded system has less than ten million distinct address cells (bytes[5]), so most of the address space is taken up by thin air. There are reasons for this, but it is not necessary to review them in this discussion.

The width of the data bus is similarly defined. This bus contains the actual contents of the memory cells. So, if the processor needs to get information stored in memory cell number 1094 (called a ***read*** operation), the data bus would return the data value stored in that cell, 12456, for example.

If the processor wants to store data into a memory cell (called a ***write*** operation), then it sends the data, 12456, to memory cell number 1094. Note that the address bus is one-way only; the address comes out of the processor and is distributed throughout the

[5] A byte represents a group of 8 single address lines, or ***bits***. A single bit is the atomic unit of a digital system. A group of 4 bits taken together is a ***nibble***. 16 bits (two bytes) is a ***word***. 32-bits is a ***long word***. For the sake of confusion, 64-bits is also a long word.

system. The data bus is ***bi-directional***. Data can flow ***from*** the processor or ***to*** the processor on any operational cycle.

The width of the data bus determines how big a number the processor can consume in one cycle. The most common embedded microprocessors in use today have 8-bit wide data busses. This means that they can handle numbers from 0 to 255 (256 total because zero counts as a number), or 2^8. If the system is dealing with negative numbers then the range becomes -128 to +127.

Similarly, a 16-bit data bus can handle numbers ***in one bus cycle*** from 0 to 65,535 or -32,768 to +32,767. Notice that "one bus cycle" is in italics. Even the lowly 8-bit processor can perform calculations with large numbers, just like the larger bit-width processors. However, it will take the 8-bit processor longer because it must perform multiple bus cycles to accomplish equivalent tasks. A good rule of thumb is that doubling the data bus width quadruples the processing speed, while doubling the clock doubles the system speed.

The ***status bus*** is the housekeeping bus. It tells the rest of the system what the processor is doing, and is going to do (read, write, reset, etc.), and it tells the processor what the rest of the system is up to (e.g., an ***interrupt***[6] event). The clock circuit sends clock signals to the rest of the system to keep it synchronized to the processor's internal operation.

The glue logic and address decode block does two jobs. It contains most of the assorted housekeeping hardware that "glues" the system together (hence, the name) and also decides which memory or peripheral device can become active at any point in time. Doing this reduces the number of individual address lines each device must monitor. Without address decoding, every memory device would need to look at all 32-bits and decide when it should respond or remain inactive.

The watchdog timer plays a very important role in critical digital systems. Its job is to monitor the health of the system and force it to shut down or return to a known condition if a fault is detected. The fault might be an imminent power shutdown or a system glitch[7] that forces the processor to deviate from the expected program and start trying to execute non-existent instructions. By whatever method the watchdog timer uses to monitor the system, its function is to force the system back under control in case of a system fault.

[6] The interrupt is a way that a peripheral device can notify the processor that it needs to be serviced, e.g. an A/D converter has data ready to be read, or a communications circuit has data ready to read. Interrupts can be ignored (masked) or must be taken (non-maskable or RESET). An example of a non-maskable interrupt is a signal that the system is losing power and needs to shut down.

[7] Walking across a nylon carpet with leather shoes in Phoenix, AZ and then touching a PC board is a good way to cause a serious system glitch.

Read-only memory, or ROM, is the program storage memory. ROM memory retains its contents even when power is removed from the system. Since the processor can only read from ROM, and not write to it, there is no way for the instruction code to be modified. The processor will always execute the same set of instructions upon power or reset. FLASH memory is a newer form of ROM. Earlier ROM's were either programmed permanently at the factory (mask programmed ROMs) or could be bulk erased by exposing the silicon die to ultraviolet light. The ROM can be reprogrammed using special programming devices, but is not reprogrammable in-circuit.

FLASH memory retains the ROM characteristics of the older memory devices but can be reprogrammed in-circuit. Thus, devices using FLASH memory can be field upgradable, or repaired in the field. Most of the newer BIOS[8] ROM's in modern PC's use FLASH memory for BIOS storage and have utility programs on floppy disk that you can use to upgrade the BIOS ROMs. The caveat is that if you don't reprogram the ROM's properly, and lose the BIOS, then the only recourse is to send it back to the manufacturer. However, devices which are identified as Y2K sensitive AND use FLASH memories are good candidates for field upgrade to remove a Y2K defect.

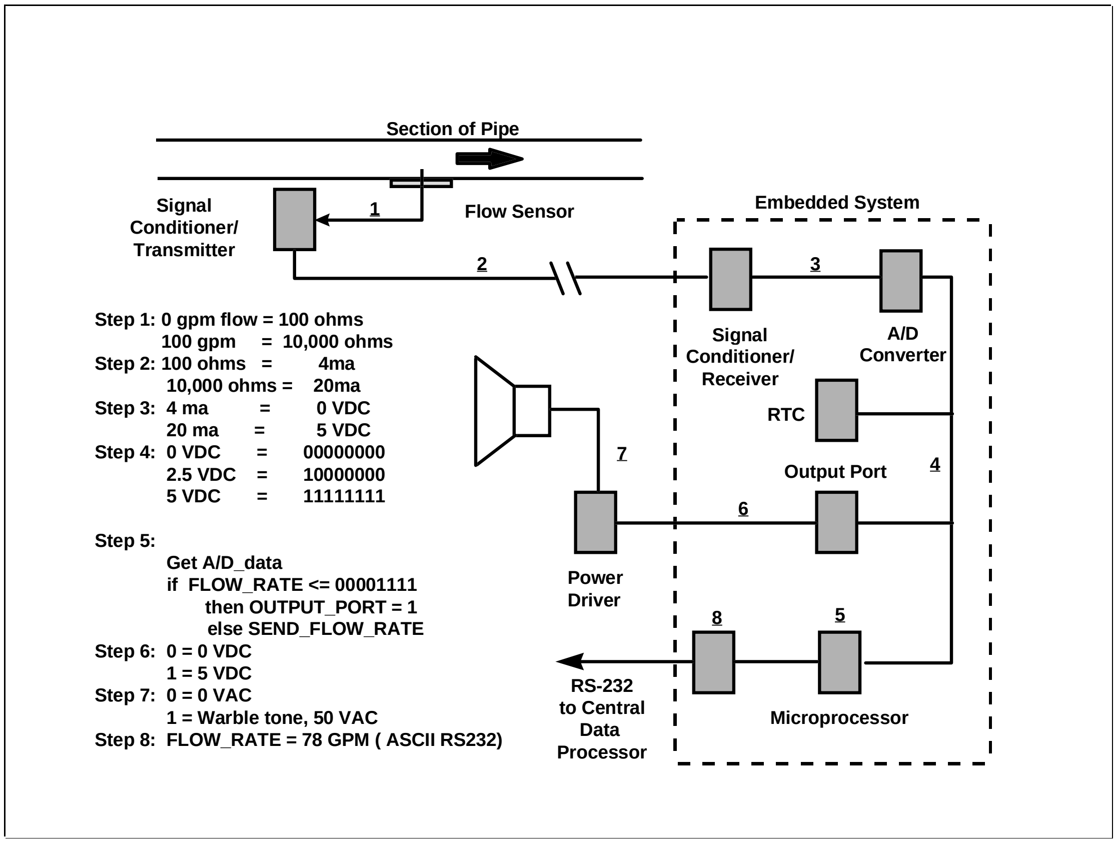
Random access memory or (RAM) memory is volatile memory. It will retain its contents as long as it has power applied to it. If power is removed, even for an instant, there is no guarantee that the contents of the RAM are still intact. Thus, ROM memory is generally positioned so that when the processor first comes alive, either by coming out of RESET, or after power-on, it will begin to executed its instructions from ROM. Typically, it will move initial values for variables into RAM and set up its operating environment before it begins to function as a device. Sometimes you will see ***power on self-tests*** (POSTs) being executed when power is first applied to the device.

Since RAM is faster memory then ROM, embedded systems will often copy the contents of the ROM into RAM and then execute code out of RAM. In this manner, a single 8-bit wide ROM can be used in a system with a 32-bit wide microprocessor.

The microprocessor, clock generation, address decode, glue logic, RAM and ROM are the minimum components necessary to have an embedded system (plus power). This is a rather uninteresting embedded system because, from a user's point of view, it can't do anything useful. To be useful, we need to allow our embedded system to interact with the outside world. The outside world may be a user, through a user interface (keyboard and display, or front panel) as well as an array of input and output devices to connect the embedded system to its environment. Let's consider an analog to digital converter, A / D, as our prototypical circuit. Figure 2-3 is a block diagram of such a system.

[8] BIOS = Basic Input/Output System.

In this example, the flow sensor is an electromechanical transducer that converts a mechanical force (fluid motion) into an electrical signal by changing the resistance of its sensing element. The signal conditioner / transmitter then converts this resistance value into a current that is appropriate for sending over a pair of wires to a receiver. The resistance change of the sensor is the electrical analog of the fluid flow rate and the current sent between the transmitter and receiver is the electrical analog of the resistance of the flow transducer. The receiver converts the current back to a voltage level that is the appropriate input to the **A/D** converter.



```
Step 1: 0 gpm flow = 100 ohms
        100 gpm    =  10,000 ohms
Step 2: 100 ohms   =       4ma
        10,000 ohms =   20ma
Step 3:  4 ma        =       0 VDC
         20 ma       =       5 VDC
Step 4:  0 VDC       =     00000000
         2.5 VDC     =     10000000
         5 VDC       =     11111111

Step 5:
         Get A/D_data
         if  FLOW_RATE <= 00001111
               then OUTPUT_PORT = 1
               else SEND_FLOW_RATE
Step 6:  0 = 0 VDC
         1 = 5 VDC
Step 7:  0 = 0 VAC
         1 = Warble tone, 50 VAC
Step 8:  FLOW_RATE = 78 GPM ( ASCII RS232)
```

Figure 2-3
An embedded system in a physical environment

In this example, the A / D is an 8-bit device. It digitizes the voltage so that a voltage of 0 volts corresponds to a digital value of zero (0000000) and 5 VDC full-scale corresponds to a digital value of 255 (11111111). It is easy to calculate that the resolution of this converter is 5 volts, divided by 256, or about 20 millivolts per step.

Once the A / D converter has digitized the voltage, it signals the microprocessor by generating an *interrupt request*, or an IRQ in computer jargon. The computer reads the A / D value with a special part of the program called the *interrupt service routine.* The interrupt service routine for the A / D may also be called the *A/D driver code.* This terminology is important because it has a lot to do with how embedded software is written, organized, and how Y2K problems may be masked by software.

The embedded processor now must interpret the data value according to the conditions establish in its program. This is called the ***application code***, to distinguish it from the driver code. If the fluid flow rate is within normal bounds, the embedded processor will simply convert the value to its ASCII[9] code equivalent and then write the value to the RS232 communications circuit, or serial data port[10]. In Figure 2-3, this is a flow rate of 78 GPM (78 in binary is 01001110[11]).

In order to send this as an ASCII number, the 8-bit binary number needs to be converted to 2 ASCII digits, or 0100101 (37) followed by 0100110 (38). This is the serial data string that the embedded processor would transmit to the remote location. If you attached a serial data analyzer of some kind to the RS232 port, you would see these two distinct codes go by.

However, the embedded system would probably be programmed to transmit the data as a complete information packet, such as the ***Distributed Network Protocol*** standard, that includes date and time information, header information to identify itself to the remote system, and some sort of preamble to the flow value so that the complete message might be a string of ASCII characters that looks something like:

UNIT_7A3$$12,AUG,98$$14:56:29$$CH1=78CR

Here:

- UNIT_7A3 is the unit's self-identifier that has been pre-programmed into it
- $$ is a delimiter that is used to separate the fields from each other
- 12,AUG,98 is a Y2K sensitive date from the real-time clock chip
- 14:56:29 is the time, also from the chip, note that ":" is also a delimiter
- CH1=78 is the output value (in GPM) of analog channel #1[12]
- CR is a symbol for "Carriage Return" used as an "end of data field" symbol

[9] The *American Standard Code for Information Interchange* (ASCII) defines 127 (seven bit) binary codes for the alphabet, numbers, punctuation and special control characters, such as "carriage return". ASCII is the most widely accepted standard communications code in use today.
[10] This would be built into the processor in the case of a microcontroller.
[11] How I figured this out is the subject of another paper.
[12] In all likelihood, this embedded system can read multiple analog channels

Finally, let's look at what happens if our flow rate is dangerously low[13]. As the code snippet in Figure 2-3 illustrates, the processor takes another path and actuates the alarm signal (steps 6 and 7).

Let's summarize what we've discussed so far. We've looked at the basic components that make up an embedded system (the microprocessor, system busses, clocking system, glue logic, address decoding and peripheral devices) and briefly reviewed their purpose. Next, we put some of these elements together to examine how an embedded system does something useful in the real world. Finally, we saw how an errant date can be generated from inside of our embedded device and transmitted to the rest of the system.

[13] Suppose the pipe carries cooling water to the reactor.

3

DESIGNING EMBEDDED SOFTWARE

In this section we'll examine the process of embedded system design. Understanding the development process is important to understanding how Y2K defects can get introduced into a product as it is being developed. As you will see, the presence or absence of a real-time clock circuit is not a conclusive factor. Also, the tools used in the design of embedded systems are useful in the analysis of embedded systems for Y2K defects.

The design process for embedded systems typically follows several well-defined steps:

1. Specification: The product is conceived and defined by system architects, software team leaders and hardware team leaders.

2. Implementation: Hardware designers and software designers separately develop their pieces of the design. Typically, a development team is composed of seven software developers for each hardware developer.

3. Integration: The prototype hardware and software are brought together and the teams bring the system to life. This phase consists of iterations from integration back to implementation as defects are uncovered in the pieces being integrated.

4. Release: The products are tested for integrity and freedom from defects.

5. Post-release: Often called the "maintenance and upgrade" phase. This is often separate engineering teams that have the task of doing post-release bug fixes and product enhancements. Often, feature-set improvements can be achieved by fine tuning the operational software of the product, thus avoiding a complete redesign of the hardware.

Let's take a look at the software design process. We'll discuss the hardware process later. In most embedded systems, software is developed in layers, often referred to as ***abstraction levels***. This is illustrated in Figure 3-1.

The first level of software that is closest to the hardware is called ***driver level.***
The drivers (also called the BIOS) talk to and control the hardware. Interrupt service routines (ISR's) reside within the driver layer. Let's look at how the software drivers function. Referring back to our A/D example in Figure 2-2, the A/D converter

generates an interrupt when it has completed its conversion of the voltage to an 8-bit binary number. If the processor is accepting interrupts[14], it will automatically alter program flow to its ISR. The ISR saves the current state of the processor (so it can return to what it was doing before it was interrupted) and then gets the value from the A/D and puts it in a memory location. It also sets another signal in memory indicating that there is new data available.

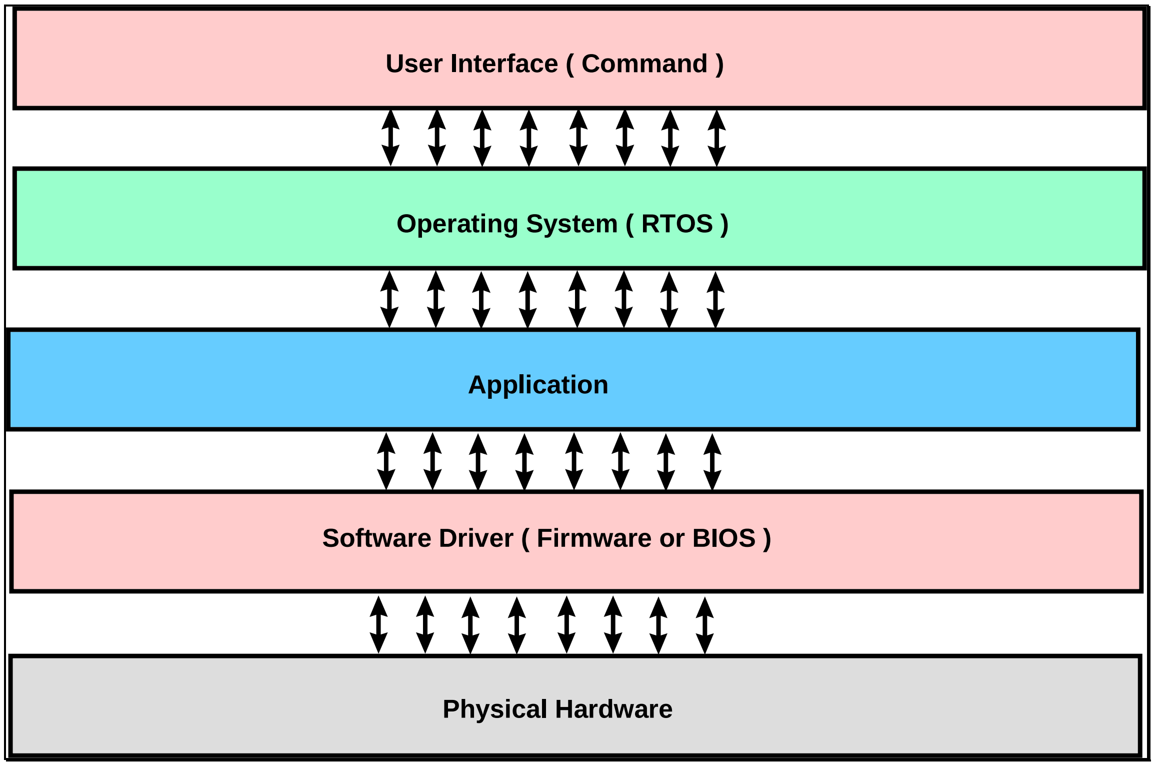

User Interface (Command)
Operating System (RTOS)
Application
Software Driver (Firmware or BIOS)
Physical Hardware

Figure 3-1
Abstraction layers in the software of an embedded system

The software that needs the data is called the application software. This is generally the body of code that gives the embedded system its personality. Since the application software needs to process the data, it will call the ***A/D driver***, or, in other words, send a message to the driver layer requesting data. The A/D driver may do one of several things. It may go to the A/D converter and initiate a conversion, or, if there is already new data available, it may return the location of the data to the application program. This way, the application program operates with the data, instead of a copy of the data. It is the application software that decides if there is a problem with the flow rates and what to do about it.

[14] It may not accept the interrupt if it is already handling a higher priority interrupt

Sitting above the application software is the real time operating system, or RTOS. Not all embedded systems use RTOS's. The more complicated ones do, and simpler ones do not. It is the job of the RTOS to manage multiple applications and system resources all at the same time and do this within the constraints of real-time events occurring more or less randomly.

At the highest level is the user interface. This is where the commands to the device come down, are processed, and spawn new tasks for the processor to carry out. One of the jobs of the RTOS is to manage the initiation of these tasks and to shut them down when they are no longer needed. However, in a simpler system, pushing a front panel button might generate an interrupt and the front panel ISR gets the button-pushed information and then goes back to what it was doing.

The important point is that software layers tend to be developed with defined interfaces between them. This frees designers at the upper layers (called higher abstraction layers) from worrying about implementation details at the lower layers. This has important ramifications for the Y2K problem. Generally, looking down from the higher abstraction layers, the software at the lower level is accessed through an ***application programmer interface,*** or API. The API is designed to be the only access mode that upper layers can have to the functions within the lower layers.

Driver software is not new. The concept of the API is relatively new, although software academics have been advocating the concept for many years. Even driver software can have an API interface assigned to it. The API is the rules that one programmer supplies to another programmer in order for the second programmer to access the functions (software code) in the way that the designer intended. As an example, consider the new programming language called, ***EightBOL***.

The following statement is an API for a function that returns a BOOLEAN value (true or false) depending upon the value that is input to it. Let's have a look:

BOOLEAN SOUND_ALARM (SHORT INT FLOW_RATE)

Recall our fluid flow example from Figure 2-3. The API, above, says that you (the programmer calling this function) must send into the function an integer value (no fractions), representing the rate of fluid flow, between zero and 255.

This API says that you (the programmer calling this function) must send into the function an integer value (no fractions) between zero and 255. The function, when it completes it's execution of its code, will return a TRUE or FALSE value that the programmer can use to determine whether or not to sound the alarm.

The key point here is that you, the programmer cannot see the actual code contained within the function *SOUND_ALARM*. All you see is the interface structure that will allow you to use it. The API is the interface structure. The key point is that you can only access the parts of the function that the developer of the API wants you to see. You do not have *Carte Blanche* to go anywhere in that function, handle internal variables, and generally make a mess of the code.

In theory, there is nothing to stop any software from accessing any functionality of the system. This would allow an application program to directly access the A/D circuit and get the data, bypassing the driver. It makes programming simple, right? Not quite. It makes programming, software quality, and more importantly, software maintenance, a nightmare.

The problem arises because any change made in the system causes side effects to ripple through the entire system. By encapsulating the accesses to the A/D within the A/D driver, changing the A/D isolates any other changes to the driver, not to the entire system. This makes code and product maintenance much easier.

The downside to this is that it allows Y2K problems to be perpetuated, even though Y2K compliant hardware is present in the system. This might mislead an inspector who is trying to determine Y2K compliance by examining the hardware for specific Y2K sensitive devices. Here's the scenario. Suppose that the original real-time clock driver was written for a non-Y2K compliant real-time clock. The driver goes out and reads the various memory cells within the clock circuit that contain the date and time elements. It then formats this information according to the API interface specification and presents it to the higher level software.

Now, suppose that the application code was originally written for a 2-digit year because the original real-time clock (RTC) chip kept track of the year as a 2-digit value. Thus, the driver only needs to reads the year information as two digits, and the application expects the data as a 2-digit year. However, suppose the RTC chip is now Y2K compliant (i.e. a 4-digit year), but the API specification is for the original 2-digit year. The new driver may read the 4-digit year from the RTC correctly, but it will then strip off the century part of the year and present only the 2-digit information in order to comply with the original, 2-digit interface specification. This is the benefit and curse of encapsulating the driver code.

Observing the type of RTC circuit in your device will tell you if the device has a time and date capability, but won't tell you if it is being used properly by the software. However, if it does have a Y2K compliant RTC circuit, then you know that the leap year dates will be properly computed and reported for the year 2000 and thereafter, even though the year may be truncated by the driver.

4

DESIGNING EMBEDDED HARDWARE

There is an issue with the hardware design as well. In our discussion we've been focusing on commercially available devices, such as real-time clock chips (RTCs). However, advances in semiconductor technology have enabled hardware designers to build entire systems on a single silicon die. These are often called ***Systems-on-Chip*** or SOCs. The advantages are obvious: costs go down, reliability, functionality and speed go up. Today, we have the capability of putting four million gates on a single piece of silicon. This is equivalent to all of the circuitry needed to drive a big, high-speed, networked color laser printer in a single package.

The revolution comes from our ability to create ***application specific integrated circuits*** or ASIC's[15] and our ability to create the hardware elements by writing software, instead of creating a schematic diagram. Figure 4-1 is a simple example of this concept.

In Figure 4-1 we have a simple digital logic element, the ***AND*** function (using an AND gate). Both input signals must be true if the output is true. The hardware designer might design this function by choosing a part that contains four 2-input AND gates inside of it, and connecting A, B and C to one of the gates in the package. Such a device is illustrated in the figure.

Actually, the software designer could also implement the logical AND function as software variables, also shown in Figure 4-1. This wouldn't be as fast as the hardware, but logically, it could do the same thing. However, the hardware designer, if the design is an ASIC, could write the Verilog[16] assignment statement for the logical AND gate, but the design tools would translate it into the Integrated Circuit (IC) gate element instead of software instructions for the microprocessor. Once a VHDL design is complete, the design "software" is sent to a silicon manufacturer (often called a foundry) who would convert the VHDL design to a process database that describes the actual logic circuits and interconnections. The foundry would then fabricate the silicon chip according to the description in the designer's VHDL code. Thus, both the hardware designer and the software designer will write a program, but the hardware designer's program gets

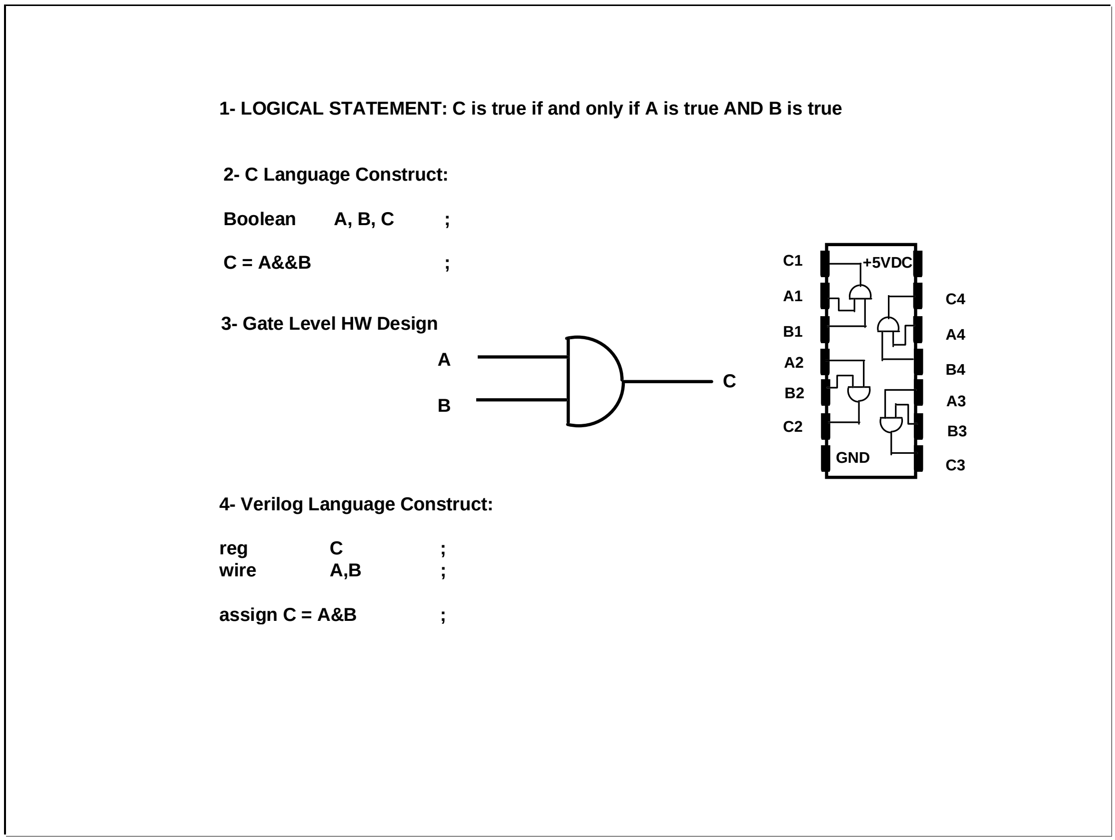
[15] Not to be confused with the running shoes.
[16] ***Verilog*** and ***VHDL*** are two examples of hardware description languages (HDL) used to design integrated circuitry. They are similar in structure to the "C" programming language used by software developers.

translated to instructions for creating the ASIC instead of a series of microprocessor instructions. By the way, where's the microprocessor?

In a SOC, the microprocessor itself may be a series of instructions[17] for the silicon fabricators to create within the SOC. Also, major functional blocks within the SOC can be designed directly by the ASIC designer, or purchased as ***intellectual property*** (IP) from IP suppliers. One IP block that might be purchased could be an RTC circuit element. Or, the hardware designer could create one, since it is only a serial counter with a special algorithm attached to calculate leap years. This is easily described as a software algorithm or hardware algorithm.

The above discussion on ASIC technology has implications for someone who is trying to determine Y2K sensitivity by examining the IC's within an instrument. The inspector may find one or more ASIC devices that are not available as commercial off-the-shelf parts. From this, the inspector may erroneously conclude that there is no RTC circuitry in the product, when, in fact, the circuitry is there, but it is buried within the ASIC.

[17] Typically, it would also be encrypted Verilog code.



1- LOGICAL STATEMENT: C is true if and only if A is true AND B is true

2- C Language Construct:

```
Boolean     A, B, C     ;

C = A&&B                ;
```

3- Gate Level HW Design

4- Verilog Language Construct:

```
reg         C           ;
wire        A,B         ;

assign C = A&B          ;
```

Figure 4-1
Implementation of the logical AND function in hardware and software

One last question that is worth answering is, “What is the difference between an ASIC and a microcontroller?” The simple answer is that a microcontroller is an ASIC that has gone public. The particular core and peripheral devices are generic enough that the device finds applicability in many different design environments.

5

ANALYTICAL TOOLS

Embedded systems have one particular feature that makes debugging[18] them a more specialized process then debugging a program on your PC. This feature is the ***real time*** nature of the operation of an embedded system and its ability to properly interact with the outside world around it. Embedded systems may be roughly divided into two categories: those that are ***time-critical*** and those that are ***time-sensitive***.

A time-sensitive system slows down if some perturbation is introduced, but it continues to function at a lower level of performance. A time-critical system ceases to work if its ability to complete a task within a critical time window is compromised. Real-time embedded systems are most often time-sensitive, but they may be time-critical, depending upon the design and the external dependencies.

An example will help here. Suppose you have a data logger that records a data value every five minutes. If it records the data one minute late, most likely nothing bad will happen. However, suppose you have a fast waveform recorder that has to be able to capture a glitch that may only happen once a month. It can't slow down to the point that it fails to record the waveform of interest. It is a time-critical application.

A debugger is a common program used to debug software. It is a supervisory program that is always in control, even when your application code is executing. In order to see what your program is doing, you enable the debugger and execute one line of your application software at a time, very slowly (single-stepping), as you watch how the program behaves. Under these conditions your software may be running a million times slower then normal. It is easy to imagine embedded systems that cannot be debugged under these conditions, especially when the faults are associated with the interaction of the software and the real time stimuli of the outside world. If you can't use your debugger, what do you do?

The first piece of information that you have at your disposal is the design of your hardware and the design of your software. You know what you wrote and how the

[18] Debugging is the process of finding and removing defects and errors in the hardware and software. These defects are known as ***bugs.*** Legend has it that one of the earliest computing machines, based mostly on relays and vacuum tubes, stopped working because a housefly got caught inside it, and was cooked by a power relay. This was the first computer malfunction traced to a bug. Hence, the name.

hardware and software are supposed to behave. You have access to the software source code listings and software location maps, as well as the schematic diagrams of the hardware. As a competent engineer and member of the embedded system design team you should be able to uncover the bugs in your hardware, software or both.

The most commonly used hardware debug tool is the ***logic analyzer*** (LA). The LA is a passive instrument that connects to the microprocessor busses in such a way that it can record (trace) all of the bus activity of the processor. The tool captures every bit of the processor activity; instruction executions, memory reads and writes, interrupts, etc., up to the limits of its memory. Thus, if you want to see an event of interest, such as why the software is going in one direction at a decision point (branch) when you think it should be going in another direction, you can set the LA to record the events of interest when the processor begins to execute the critical instruction.

The logic analyzer is really nothing more than a very wide circular memory. When it gets to the end of the memory it keeps on going from the beginning. Thus, the last memory address and the first are adjacent to each other. As long as the logic analyzer is turned on, it is always recording. You retain information by deciding when the LA should stop recording new information over the old information. This fact means that you can decide to record information (trigger point) anywhere in time. You can thus record everything up to the trigger point, everything that occurs after the trigger point, or anything in between. In this manner, you may watch all the activity up to the event of interest and then what happens after that.

For embedded system development the premier tool is the ***in-circuit emulator*** (ICE). The microprocessor emulator takes the place of the actual microprocessor in the embedded system. To the system, it looks like the processor, but to the designer, it is a window into the microprocessor. The in-circuit emulator integrates all the functionality of a debugger, logic analyzer, and overlay memory in one package.

Overlay memory is a convenient tool for developing software because it allows you to quickly load and test software without having to try to load it into the EPROM or FLASH memory of the system. Overlay memory is simply RAM memory inside the emulator that is accessed by the emulator instead of the actual memory of the embedded system. By tightly combining the functionality of these three devices into one tool, with a common user interface, the emulator has become the central tool for embedded system development.

The last tool that is worth mentioning is a device called a ***ROM emulator***. It is very similar to the overlay memory in the ICE, but it functions as a stand-alone tool and connects to the ROM socket rather than the microprocessor socket. To the system, it looks like the ROM is plugged in, but, like the emulator, it also connects to the software engineer's computer via an Ethernet connection. It can provide a rapid conduit into the target system for code substitution.

Why did we go through this explanation of the tools used in embedded system development? First, it is useful to understand the constraints imposed by trying to develop software that can only be tested under operational conditions. Second, because these tools tend to be non-intrusive, they may be useful for testing existing embedded systems for Y2K problems without perturbing the operation of the system. We’ll explore that in the final section.

6

Y2K ANALYSIS STRATEGIES

In the previous section we discussed the tools of embedded system development and debug. The Logic Analyzer, In-circuit Emulator, and ROM Emulator are complex and very specialized instruments that are designed to control and observe the behavior of real-time systems, either during the product development phase, or later on, during the product's maintenance and upgrade lifetime. What does this have to do with Y2K? As you'll see in this section on Analysis Strategies, the type of knowledge that you have about the embedded system determines the tools that will be most effective for you to use to gain the insight that you need to make a decision about that device. What kind of decision? Is it benign? That is, is it safe to conclude that it does not generate or use date information. Suppose it does utilize date information in its operation, how will you decide if the information is safe, or this instrument must be replaced or upgraded?

Here's an example. Utilities have many Remote Terminal Units (RTU's) in their inventory of embedded equipment. Applied Microsystems was asked to perform an in-depth analysis of an RTU because it was known to manipulate dates, but a visual inspection of the circuit board did not reveal an identifiable RTC chip. This particular RTU had a Motorola 68020 microprocessor as its controlling element. An in-circuit emulator was used to replace the microprocessor and allow the Applied Engineers to control and observe the low-level behavior of the processor while date information was being manipulated.[19] **This demonstrated the date information was being utilized even though no RTC was present**.

Was the analysis worth the effort involved? I think so because there are many identical RTU's in this utility's inventory. They now know the extent of the Y2K sensitivity of this unit, even though it was manufactured before the date that the vendor identified as being Y2K compliant.

Now that we've raised your level of understanding of your embedded systems so that you can make somewhat informed decisions about how you can eliminate Y2K problems from your own environments before midnight, December 31, 1999, we need to discuss possible strategies that you may want to employ. The key deciding factors are summarized in the Figure 6-1. Here we've recast our "typical" embedded system as a

[19] This analysis was sponsored, in part, by EPRI.

list of characteristics, including the ever-present inputs and outputs (I/O) as well as a set of “views” of the system. These views, ***black box, gray box*** and ***white box*** [20] imply various levels of knowledge about the system under test. Perhaps the single most important differentiation is whether or not you have access to the software source code and the hardware design database. With these sources, you can pretty well determine whether or not a piece of equipment is Y2K sensitive or not.

Is this a practical strategy? Probably not. In the first place, unless you actually designed and built the embedded system, the real manufacturer is not going to give you access to their software or hardware IP. Another reality of life is that for an embedded system of reasonable complexity, trying to decipher software source code[21], even at the source level, can be an expensive and time-consuming task if you were not the original designer. In fact, much has been written about how to create source code to make it understandable and maintainable, but we can’t claim that every software designer follows the guidelines all the time.

[20] This terminology is derived from the software world’s definition of how software is to be tested. Black box testing is analogous to a monkey pounding on the keyboard. Gray box testing implies that you have some understanding of how the software works so you know what to look for to try to stress it and make it fail. White box testing implies that you have full knowledge of the software design and focus your testing efforts on specific software modules and features because you know how to force them to handle exceptions.

[21] Source code is the code that the software engineer actually writes. This is typically written in a high level language such a C, C++, or ADA. This is also the most readable code. The source code is then translated by a ***compiler*** into assembly language, which represents a 1:1 matching of instructions with machine level instructions. Assembly language is much more cryptic and low-level, but is often the actual source code as well. The Assembly language instructions are then passed through an ***assembler*** program which translates instructions written in a pseudo-English form to the actual machine code. The last step in the process is to ***link*** the output of the assembler with other software modules so they form one large, ***executable*** image, or object code. This is the actual blend of instructions and data that form the embedded program. It is the executable code that gets loaded into a ROM.

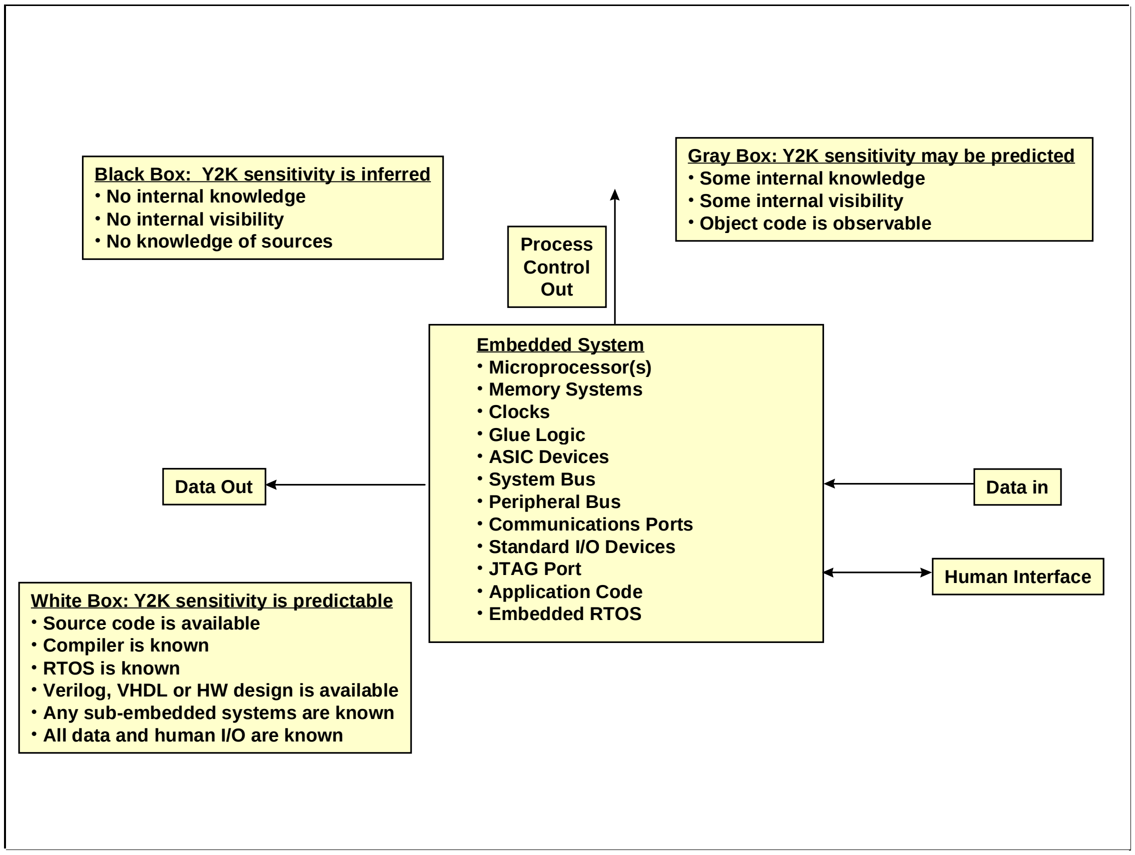
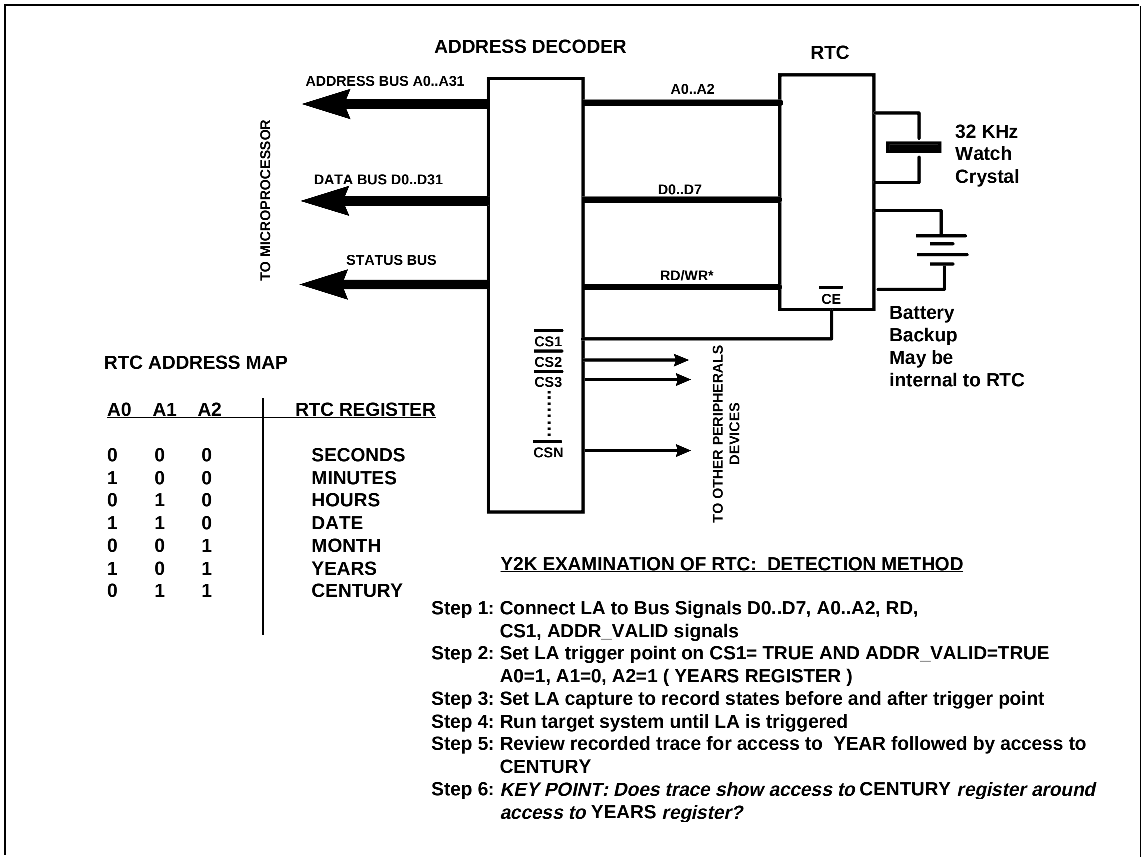

Black Box: Y2K sensitivity is inferred
• No internal knowledge
• No internal visibility
• No knowledge of sources
Process Control Out
Gray Box: Y2K sensitivity may be predicted
• Some internal knowledge
• Some internal visibility
• Object code is observable
Embedded System
• Microprocessor(s)
• Memory Systems
• Clocks
• Glue Logic
• ASIC Devices
• System Bus
• Peripheral Bus
• Communications Ports
• Standard I/O Devices
• JTAG Port
• Application Code
• Embedded RTOS
Data Out
Data in
Human Interface
White Box: Y2K sensitivity is predictable
• Source code is available
• Compiler is known
• RTOS is known
• Verilog, VHDL or HW design is available
• Any sub-embedded systems are known
• All data and human I/O are known

Figure 6-1
Levels of observability into embedded systems

This leaves us with gray box and black box testing. Let's dive into these methods because they are the most likely to be the methods available to you. Gray box testing assumes that you have limited knowledge of the system. Referring back to Figure 6-1, we see that one of the characteristics of gray box testing is access to the internal components, both from the hardware and software perspectives. How do we gain this insight?

Remember the development tools that we discussed in section 4? Although these tools were developed to assist embedded systems designers in their product development tasks, particularly the complex process of hardware and software integration, they also have applicability to probing unknown systems. Remember that one of the key requirements in observing an embedded system operating in real time is that the probe does not perturb the system it's observing. By using an LA or an ICE, a knowledgeable investigator can locate Y2K date sensitivities in the system being probed.

Let's investigate how this might be accomplished. Figure 6-2 shows our embedded system in more detail.



RTC ADDRESS MAP

A0	A1	A2	RTC REGISTER
0	0	0	SECONDS
1	0	0	MINUTES
0	1	0	HOURS
1	1	0	DATE
0	0	1	MONTH
1	0	1	YEARS
0	1	1	CENTURY

Y2K EXAMINATION OF RTC: DETECTION METHOD

Step 1: Connect LA to Bus Signals D0..D7, A0..A2, RD, CS1, ADDR_VALID signals
Step 2: Set LA trigger point on CS1= TRUE AND ADDR_VALID=TRUE A0=1, A1=0, A2=1 (YEARS REGISTER)
Step 3: Set LA capture to record states before and after trigger point
Step 4: Run target system until LA is triggered
Step 5: Review recorded trace for access to YEAR followed by access to CENTURY
Step 6: *KEY POINT: Does trace show access to* CENTURY *register around access to* YEARS *register?*

Figure 6-2
Analyzing a portion of the embedded system for Y2K detection

The portion of the circuit called the ***address decoder*** contains a memory map of the system so that as different address combinations are sent out by the microprocessor, it determines which part of the circuit (RAM, ROM, RTC, etc) should become active and respond. This is a more efficient method than having each individual subsystem decide if it is being addressed.

The address decoder contains the starting address of the RTC. For example, let that be address 04567890. Each of the seven distinct clock registers occupy one address location up from the base address, with the first register, SECONDS, located at the base address. Therefore any address from the microprocessor in the range of 04567890 through 04567896 will cause a memory storage cell of the RTC (register) to send its contents back to the microprocessor.

Our objective is to see if, in fact, the software is accessing the CENTURY data as well as the YEAR data. How do we know that this RTC chip has a CENTURY register? We could have looked it up in a data book, and determined if it had a CENTURY register, but this is more definitive. We'll connect our LA to the A0, A1 and A2 so that we can observe the data coming out of the RTC. Furthermore, we'll filter this information even further by qualifying the trigger value to detect an access to the YEAR register. Why not trigger on the CENTURY register? The reason is if we require that we see a CENTURY access, then by not triggering the analyzer, we haven't conclusively solved the problem, we only know that the LA didn't trigger. By triggering on the YEAR register, we can manually examine the code around the access to the YEAR register and see if the CENTURY register was also accessed. Typically, we would expect to see the software driver go out and read all of the RTC information, not just the year, so not seeing a CENTURY read is reasonably conclusive evidence of a Y2K problem.[22]

What about our ICE? Assuming that we could replace the processor in the system with the ICE probe, then the internal logic analyzer of the ICE would have been able to gather the same information as the external logic analyzer. Also, an ICE has more sophisticated software analysis tools then a logic analyzer, but the ICE is specific to the microprocessor that it is being used with, while the LA is a more generic tool.

Therefore, while it is technically feasible to do this kind of analysis on your embedded devices, ultimately you will have to decide for yourself if its worth the effort. If you have a large number of identical embedded systems, then perhaps the economies of scale will make an in-depth analysis of a few devices worth the effort because you can amortize it over the large number of systems in use.

The above example was straightforward. In real life, things are never that straightforward, and I can't hope to explain all of the subtle "gotcha's" that may occur in the real work. Figure 6-3 is probably a lot closer to what you will encounter in your facilities.

[22] Since LA's are pretty sophisticated instruments we could have set the trigger condition to trigger on an access to YEAR **OR TO** CENTURY. This would have further eliminated any ambiguity.

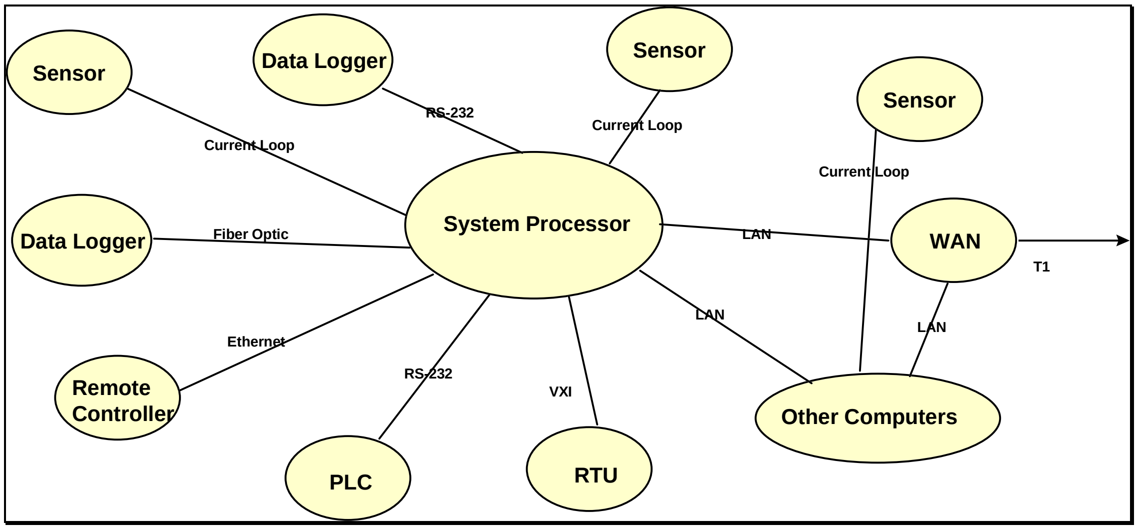


Figure 6-3
A model of a real embedded system

In Figure 6-3 we see many interconnected and interdependent pieces making up our embedded system. We could draw a dotted line around any subgroup of these components and call that an embedded system as well. So what, if anything, can we do about it? Recall that in Figure 6-1, the black box analysis is most likely to be the real situation that you'll be faced with. Let me suggest one approach. We'll call it ***divide and conquer***. While not absolutely foolproof, it's a good starting point method.[23] I could imagine a stand-alone device that generates and digests date information without ever transmitting it to something else, thus making its Y2K sensitivity invisible to a black box scrutiny method.

We'll assume that we have a situation similar to Figure 6-3. Divide the larger system in smaller units and draw the dividing line at the communications interfaces between the various units. Suppose that we're looking at a Remote Terminal Unit (RTU) that concentrates sensor data from several sensors and transmits the data over an RS-232 link to another processor controller in the system. If we can afford to watch all of the data traffic going back and forth over the RS-232 serial line and don't see any dates going by over some suitable period of time, then we can make some informed decisions about what is going on in this particular sub-system. Is it foolproof? No. It is your detailed knowledge of how your facilities operate that will enable you to put the information in its proper context.

[23] There are a number of excellent articles written on Y2K test methods and analysis protocols. My intent is not to devalue these approaches. Rather it is to present a specific example of how a Y2K detection team might actually gather information.

With my disclaimer taken care of, let's move on. Assuming that you are willing to monitor your RS-232 data traffic, how can you hope to detect an errant date flying by in millions of other data characters. There are instruments, called ***protocol analyzers***, that can monitor a serial data stream in a similar manner to a logic analyzer monitoring parallel data streams. The first problem is how to set-up the protocol analyzer to trigger on the date pattern of interest if you're not sure what the date pattern looks like. The second problem is that you need trained personnel to operate the analyzer.

EPRI and Applied Microsystems are working together to create a product to analyze RS-232 serial data streams for date information. Tentatively we're calling it ***the Y2K Sniffer***, and we intend to demonstrate a prototype of the device at the EPRI Y2K Workshop in San Diego, August 24 through 27, 1998.

The Sniffer™ is designed to be used by general plant personnel, rather than highly trained, embedded systems experts. The device is inserted in the serial stream between RS232 devices and automatically configures itself for the data stream. It then watches for ***date-like*** patterns on the data line. Examples of date-like patterns could be :

- spXX/XX/XXXXsp ;where X is a number and sp means space
- XXjunXX ;This one is obvious
- 12-31-99 ;This one is not so obvious

The design challenge is the sophistication of the Sniffer's date filtering algorithms. The goal is to trap every common date pattern and most of the uncommon ones. The Sniffer can also be upgraded in the field, so that new patterns can be loaded into the device as we gain information about the types of date patterns that are in use in real life.[24] It is being designed so that other communications protocols, such as Ethernet, can be monitored as well.

Whenever the Sniffer detects suspicious date patterns it stores them in its own non-volatile memory. The stored patterns can be reviewed by scrolling the display or by saving the contents of the memory to a portable computer for later analysis by trained personnel. Every suspicious date will be date-and-time-stamped by a Y2K compliant real-time clock.

How would you use the Sniffer? The Sniffer will assist you in deciding where to concentrate your energy and resources as you undertake your Y2K investigations. It will uncover those layers and boundaries in your environment where date information is being exchanged. It is a tool that is designed to help you in your Y2K analysis. It cannot substitute for your knowledge of your environment.

[24] For further information about the Y2K Sniffer contact arnieb@amc.com or joeweiss@epri.com

7

SUMMARY AND CONCLUSIONS

Even though embedded systems are found in many varied applications, there are common characteristics that are found in almost all of them. We saw how the elements of an embedded system interact with each other and with the outside world. We looked at the design methods employed in the development of embedded systems and the applicability of these methods to detecting Y2K problems.

Finally, we discussed possible strategies for analyzing embedded systems for Y2K issues and also looked at a new tool being developed specifically for the task of monitoring RS-232 communications between embedded systems and looking for date patterns.

Where do you go from here? For more information, an afternoon spent surfing the World Wide Web would be time well spent. A good place to start, with links to many other sites is the Embedded Software Association's (ESOFTA) website, www.esofta.com. Two papers are featured on the ESOFTA website,

"A Suggested Process To Assist In Identifying Embedded Devices And Systems With A Year 2000 Compliance Problem (33KB, PDF format)" written by Ron Strem and Mike Smith in association with TransAlta Utilities, Calgary, Alberta, Canada.

This brief article is put forward as a starting point for a simple designation standard to help clarify the Y2K compliance investigation and associated communication now in progress. Also see,

"Testing Guidelines" by Kim Smith and John Catterall, Western Power.

Finally, I came upon another interesting article, "Software design for life-critical systems"[25] that discusses the process of designing critical software in more depth than I could cover in this paper. Though not specific to the Y2K problem, the author expands upon several of the software design methods that I only mentioned.

[25] Peter Varhol, Computer Design, Vol. 37, No. 7, July 1998, Pg. 43