Debugging embedded systems: using a trace buffer to see what went wrong

STUART R BALL

Your program writes values to a trace buffer when it executes significant points in the program code. When something fails, you can look at the trace buffer to see a portion of program history.

When something goes wrong in an embedded system, you usually want to know what happened just before the failure. A trace buffer is a section of memory that provides a way to do that. A trace buffer is typically of binary length (for example, 256, 512, or 1024 locations), with each location holding 1 byte of information. Each byte is an action code. Each action code originates at a key location in your program, where you know from the program sequence that the corresponding specific action has occurred. At each such location, you write an extra program instruction that stores the appropriate action code as the latest entry in the trace buffer. **Listing 1** shows an example trace-buffer routine that your program can call to store a code.

To see the benefits of a trace buffer, consider a hypothetical signal-monitoring system (**Figure 1**). This system monitors a continuous stream of digital data passing between two other systems, and it monitors commands over a serial port from a host PC. It must analyze the data stream for any of three possible patterns of bytes. If it detects any of the three patterns, it sends a message to the PC. The PC can individually mask the patterns, however. When the PC masks a pattern, the monitor still detects that pattern, but it doesn't send a notification to the PC. Assume that the monitor has a regular, 50-msec (20-Hz) interrupt that it uses to measure time-outs and other things.

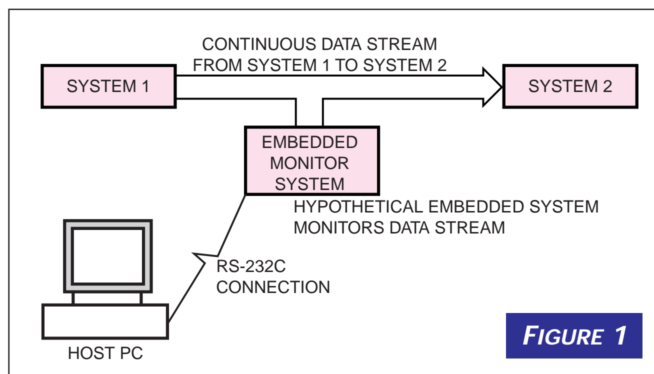
A selection of action codes for this system might look like those in **Table 1**. The last 32 bytes of a trace buffer for this system might look as follows:

64 64 03 04 64 03 04 15 16 17 64 03 05 64 06 08
13 03 04 64 06 07 64 03 64 04 64 64 06 16 07 FF

This article, one of an occasional series on basic debugging techniques, is an adaptation from *Debugging Embedded Microprocessor Systems* by Stuart R Ball. Material reproduced courtesy of Newnes, an imprint of Butterworth-Heinemann, 225 Wildwood Ave, Woburn MA 01801-2041. For more information, check www.bh.com. To order, call 1-800-366-2665.

If the system's trace buffer holds these values after an error occurs, you can analyze the data to find out what went wrong. First, the FF value at the end of line 2 indicates the end of the trace buffer's data. (This terminator is necessary, because, when the buffer is full, it "wraps around" to the beginning.) Three locations before the FF end-of-buffer code is action code 06 (**Table 1**), indicating that the signal-monitoring system detected a sequence containing Pattern 2. Following the 06 is 16, indicating that the system received a command from the host PC to unmask pattern 2. Finally, the code 07 indicates that the system detected the end of pattern 2, but the program code still thought that pattern 2 was masked.

The trace-buffer sequence just described might indicate an error, depending on the system specification. Assume that it is an error and that the signal-monitoring system should have sent a code to the host PC to indicate the presence of pattern 2. According to the trace-buffer sequence, you might suspect one of the following sources of the error:



A hypothetical embedded system that monitors a data stream illustrates the use of a trace buffer for debugging. The data-monitoring system looks for patterns in the data stream.

DEBUGGING EMBEDDED SYSTEMS

- The host PC might be too slow to unmask the pattern and might be sending the command to the monitor system too late.
- The firmware in the monitor system might fail to recognize an unmask command if it occurs simultaneously with the pattern to be unmasked.
- If the monitor system's design should be able to handle unmask commands during pattern detection, then the problem could be a race condition. The trace buffer gives the appearance that the system received all commands in regularly spaced intervals. But the command to unmask pattern 2 could have occurred immediately before detection of the end of pattern 2. Suppose that the monitoring system writes the PC commands to the trace buffer from an interrupt routine that processes the serial inputs. The unmask command might have set a bit somewhere to indicate that pattern 2 is unmasked, but the code might have already tested that bit and decided that the pattern was, in fact, masked. Because the interrupt could have occurred just before writing the 07 code to the trace buffer, this is a plausible explanation.

Expanding the trace buffer

Clearly, you might want to put more information in the trace buffer to track down this hypothetical problem, and you could add commands that provide finer detail. For example, if each of the three patterns that the system is seeking consists of 6 bytes, you might put a value in the table when the system receives each byte. Then, you could see just where in the detection of pattern 2 the system received the unmask command. If the command arrived after the pattern's first or second byte, you might suspect a bug that prevents proper application of an unmask command to a simultaneously detected pattern (the second of the possibilities listed above). On the other hand, if the system received the unmask command just before the last byte of the pattern, you might suspect that the PC is too slow (the first possibility) or that a race condition exists (the third possibility).

If you carry the buffer-expansion approach too far, you fill your trace buffer with so much information that you might overwrite the information you need to see. One way around this problem is to add data, not just action codes, to the buffer. For example, if you have a 16-bit free-running timer, you might grab the count from the timer and store it with each action code. You would thus "time-tag" the values in the buffer, providing some idea of when each action code occurred. For example, if your timer runs at 100 kHz, each count is 10 μsec. If your diagnostic code stores the 16-bit count each time it captures a value, the last few values in your trace buffer might look like this:

64 FF 03 64 12 8B 06 14 23 16 15 14 07 15 15 FF

Now, you have a problem, though. With two FF values in the trace buffer, how do you tell which one marks the end of the buffer? For a moment, note that each 3-byte buffer entry consists of an action code and data. You can then interpret the buffer entries as follows:

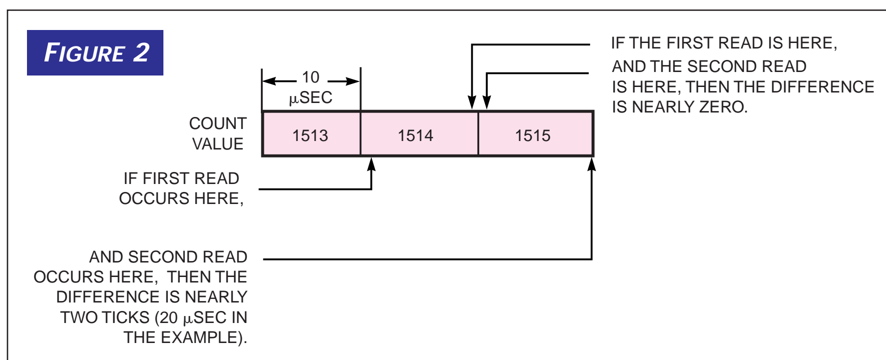
Code 64 (interrupt) at time FF03
Code 64 (interrupt) at time 128B
Code 06 (start of sequence 2) at time 1423
Code 16 (unmask sequence 2) at time 1514
Code 07 (sequence 2 detected, but masked) at time 1515

```
; diagnostic code, writes byte in AL to diagnostic
; buffer, DIAGFIFO.
; DIAGNOSTIC will write a value of 0FFh after
; the requested byte is added to the buffer, to
; mark the end.
; The buffer wraps around, overwriting
; previous data. DIAGNOSTIC saves registers it
; uses (except AL), and disables interrupts.
; Note that DIAGNOSTIC turns interrupts back
; on at the end - don't call from within a routine that
; needs interrupts disabled, unless the CLI/STI is
; removed from DIAGNOSTIC.

; DIAGNOSTIC needs one variable in memory,
; DIAGPOINT, which maintains the current pointer.
; DIAGNOSTIC also must know the start address of
; DIAGFIFO, the trace buffer in RAM.

; DFIFOLEN is a binary value that is ANDED with
; the DIAGPOINT to force a wraparound
; at the end of the buffer. The buffer must be a
; binary length - 256 bytes, 512, etc. DFIFOLEN
; would be 0FF for a 256 byte buffer, 1FF for a
; 512 byte buffer, etc.

diagnostic:
cli                         ; Disable the intr
push si
mov si,diagpoint            ; Current buffer pointer.
mov diagfifo[si],al         ; Store the requested value.
inc si                      ; Incr the pointer,
and si,dfifolen             ; and force a wraparound.
mov diagpoint,si            ; Store the new pointer.
mov diagfifo[si],0ffh       ; Mark the end of the buffer.
pop si
sti                         ; Enable the intr again
retn                        ; All done.
```

TABLE 1—TRACE-BUFFER ACTION CODES

Code	Action
01:	Power-up reset
02:	Host PC signaled that it received a power-up reset
03:	Pattern 1 start detected
04:	Pattern 1 verified, but host PC had it masked
05:	Pattern 1 verified, notification sent to host PC
06:	Pattern 2 start detected
07:	Pattern 2 verified, but host PC had it masked
08:	Pattern 2 verified, notification sent to host PC
09:	Pattern 3 start detected
10:	Pattern 3 verified, but host PC had it masked
11:	Pattern 3 verified, notification sent to host PC
12:	Command to mask pattern 1 received from host PC
13:	Command to mask pattern 2 received from host PC
14:	Command to mask pattern 3 received from host PC
15:	Command to unmask pattern 1 received from host PC
16:	Command to unmask pattern 2 received from host PC
17:	Command to unmask pattern 3 received from host PC
64:	20-msec interrupt service

DEBUGGING EMBEDDED SYSTEMS



Using a counter and a trace buffer is an effective way to time program events, but watch out for timing ambiguity. Measured times vary, because your program reads the counter at different times relative to the counter's incrementing.

Having deciphered the buffer's contents, you still have to determine what the time values mean. Because each 2-byte time code represents the contents of a free-running, 16-bit counter, the time values are a count. In this example, each count represents 10 µsec of resolution. Take the FF03 value as a "zero" timebase; the 128B value obviously occurred after the counter rolled over. So, the difference between 128B and FF03 is 10000 (hex)–FF03+128B, or 1388 (hex), or 5000 decimal. With a 10-µsec timebase, the two interrupts occurred 5000×10 µsec, or 50 msec, apart. This number is unsurprising, given that it's what the interrupt period should be.

The second time duration in this example is easier to compute, because apparently no rollover occurred. The elapsed time is 10 µsec×(1423–128B), or 4.08 msec. Proceeding accordingly with the remaining values, you can rewrite the trace table as follows, including both the absolute time and the time relative to the previous trace point:

	Absolute	Relative
Code 64 (interrupt)	0	0
Code 64 (interrupt)	50 msec	50 msec
Code 06 (start of sequence 2)	54.08 msec	4.08 msec
Code 16 (unmask sequence 2)	56.49 msec	2.41 msec
Code 07 (sequence 2 detected, masked)	56.5 msec	10 µsec

Seeing this time scale, you would probably assume that the problem is a race condition or a slow PC, because the unmask code occurred one count before the "pattern-detected" action code.

Finding the endpoint

Now, look again at the end-of-table problem—determining where your program last wrote into the trace buffer. One solution to the problem is to insert three FF bytes at the end of the table (FF FF FF). Assuming that you have no action codes indicated by FF, you can always find the end of the table by looking for this pattern. The example stores an action code followed by a word of data, so the FF FF FF pattern never occurs except at the end of the table.

The data that you associate with an action code need not be a time tag. You could write the value of a pointer, a value captured from an A/D converter, or any other meaningful number that goes with the action code. The key to making this approach work is to be sure that all action codes write the same amount of data—for example, a code and two additional bytes. Even if some codes need no additional data, write something anyway. Otherwise, it will be nearly impossible in a complex design to determine where the codes are.

Another way around the problem of finding the end of a table in a trace buffer is to use two tables. With this method, one table holds the action codes, and the other table holds the time tag or other data. The data table is twice as large as the action-code table. If each table uses the same pointer (multiplied by 2 for the data table), the pointers are always in sync. An example would then look like:

Code table: 64 64 06 16 07 FF
Data Table: FF03 128B 1423 1514 1515 FFFF

In some cases, you can more easily read two separate tables than one combined table, because the action codes are all grouped together, giving an easy-to-read sequence of events. However, when you need to see the data that corresponds to an action code, you have to switch back and forth between tables, keeping track of which data value goes with which action code.

The obvious drawback of trace buffers is that they take time to execute and require memory to store trace information. Adding time tags or other data takes even more time and memory. In addition, you have to watch for other things when you use a trace buffer.

For example, when you time-tag data using a free-running counter, remember that a rollover (like the one with the first two data points in the earlier example) might be more than one rollover. The count might have rolled over, gone to FFFF, and rolled over again. In most debugging scenarios, you care only that the difference is big, not how big it is. If you need to know the exact time, there are some ways to determine it. One way is to use a counter with more bits—enough that it will never roll over between two events. Another solution, if you are using a regular timer-tick interrupt (such as the 50-msec tick in the example), is to reset the counter at every timer tick. The count is then the time from the last tick. A final method, if your counter can generate interrupts on rollover, is to make the timer rollover interrupt generate a unique action code. Then every rollover is noted in the table.

If you read the time-tagging counter while it is changing, you might get a bogus value. This scenario is more likely to happen if you are using a counter IC that has 16-bit or wider counters but only an 8-bit interface. If the count rolls over from, say, 40FF to 4100, you might read 41FF (if you read the

low byte first) or 4000 (if you read the high byte first). Some counters allow you to "freeze" the count in a latch so you can read it without worrying about this problem. Another solution is to read the count multiple times until you get two that agree.

In the example, the final time value is one count larger than the previous data point (1514 vs 1515). You assume that this value means that the time was one count, or 10 µsec. However, the actual time could have been 0 to 20 µsec. As **Figure 2** shows, if the time that you read for the next-to-last data point (action code 16) occurs just before the timer increments to 1515 and the time read for the last data point (07) occurred just after that increment, the time could be zero (or as close as possible, given software latencies). On the other hand, if the time read for the 16 action code occurs just after the counter increments to 1514 and the time read for the 07 action code occurred just before the counter increments to 1516, then there would be nearly two full counts, or 20 µsec, between the two events. This potential problem can occur not just in time-tagging situations, but also anytime you use a counter to time events.

In the earlier example, a timer interrupt is slower than the data rates and other actions involved. If you have a faster timer, or if events produce action codes infrequently, you might find that your trace buffer contains nothing but timer-tick action codes. In this situation, you might choose not to put timer information in the trace buffer or to provide a variable that you can set during debugging to enable and disable storing of timer action codes.

EDN

Author's biography

Stuart Ball, PE, is an electrical engineer who has spent the last 16 years designing digital, analog, and embedded-µP systems. He is the author of two books, Embedded Microprocessor Systems, Real World Design *and* Debugging Embedded Microprocessor Systems, *both published by Newnes (Woburn, MA). Ball is currently employed at Organon-Teknika (Oklahoma City, OK), a manufacturer of medical instruments.*

VOTE

Please use the Information Retrieval Service card to rate this article (circle one):

High Interest	Medium Interest	Low Interest
578	579	580