Bank Address will be incremented each time that the number of counts specified in the Max Count Field are counted. The Max Count Register contains the number of transitions that are to be counted. After this number of events has been counted some action will take place such as incrementing the parameter RAM location pointed to by the Bank Address. This mode provides the capability to count the required number of events and stop. At this time a Link signal may or may not be generated depending upon how the Host Sequence Field is programmed. Another function would be to continue to count pulses indefinitely. Each time the Max Count value is reached, an interrupt flag can be set and a Link signal generated, if desired. For the purposes of this program, the Host Sequence Field will specify Single Shot Action with Link. The Link signal will be directed to channel 2 which will be programmed in the Output Compare mode.

A channel programmed in the Output Compare mode can receive but not generate Links. When channel 2 receives a Link signal from channel 1, channel 2 will respond by outputting a square wave. Thus, channel 2 must be programmed in the following manner. The Channel Control field is programmed with $8F which will direct the associated timer pin, TP2, to be an output and toggle on matching the TCR1 timer. The RATIO field is set to $80 which will result in a 50% multiplier of the value in the RAM location pointed to by REF_ADDR2. The resultant value will be the period of the square wave high time and also of the low time. After all of the appropriate registers for channels 0, 1 and 2 are programmed, a Host Service Request of $36 is issued. This starts channel 0 outputting a PWM pulse train. Channel 1 counts the pulses and issues a Link to channel 2. Channel 2 will start issuing a continuous square wave once the Link signal is received.

This example shows how the timers can be used in conjunction with one another. In one case, the output of one timer is connected to the input of another timer. The second timer affects a third timer by use of an internally generated LINK signal. The ability to use the various timer channels in an interconnected manner is a particularly strong feature in making the MC6832 a very powerful controller device.

The purpose of this paper was to show how a new generation microcontroller can be enhanced to give very high performance in terms of timer granularity. Not only this, but the CPU32 provides the computing power of an MC68020. For applications, such as automobile engine control, the MC68332 brings new functionality to control the mechanics of the engine. Because of the intense computing power of the MC68332, a finer granularity in determining when specific events (spark plug firing) should occur. While timer critical functions are calculated, the CPU32 still can be used to make other calculations, perhaps dynamically adjusting fuel/air ratio, at the same time. The MC68332 has been developed from a system design standpoint. This has caused the peripheral and memory interfacing circuitry to be moved onto the chip itself instead of having to be dealt with externally. This improves system reliability and at the same time reducing overall cost. For electronic controller designs requiring sophisticated hardware control along with intensive software computing ability, the MC68332 provides the next generation of control functionality.
Hardware-Assisted Debugging

Larry Mittag
Independent Consultant
24140 Majella Dr.
Ramona, CA 92065
(619)789-4287
CIS:71545,546

Programming for embedded systems has always been complicated by the difficulty of debugging embedded code, due to restricted hardware environments in comparison to most PC or workstation environments. It is possible, though, to leverage debugging efforts by utilizing hardware on the target, as well as tools generally found in a hardware or systems development lab. In this paper, I will describe some techniques for utilizing various types of hardware to help ease the software debugging burden.

DEFINING THE PROBLEM

Debugging can be thought of as a search for information. The primary questions to be answered include determining what the code is doing at any given, how it got to a certain point, and how long it takes to perform an operation or set of operations. The basic difficulty in answering these questions with embedded systems is that there is normally less I/O capability available. Couple this with tight timing constraints, and the result can be a nightmare.

Attempts to utilize limited I/O capabilities such as serial ports for debugging purposes are at best difficult. At worst, a sort of Heisenberg Uncertainty Principle occurs, where the attempts to characterize the behaviour of the system has distorted that behaviour to the point where the information obtained may be useless. This is especially apparent when debugging real-time behaviour.

The goal is therefore to use nonintrusive measures to obtain information about the system. Any software utilized for this purpose that uses the target system resources will inevitably perturb the system, so we will explore other means.

DEBUGGING AIDS ON THE TARGET HARDWARE

It is remarkable how much information can be obtained if the target system itself is designed to facilitate software debugging. Addition of some LED's and a register to control them provides a low overhead (in terms of added code) means to determine the health of embedded code. The most obvious use of this is a "heartbeat" indication, allowing rapid determination of whether a board has died.

An example of this is the front panel that used to be found on minicomputers and microcomputers, with LED's continuously showing the state of the address and data lines. An experienced operator could tell at a glance whether the system was operating normally, as well as getting a good feel for how heavily loaded it was, since they often ran a certain pattern when the CPU was in its idle loop.

An LED display can help in postmortem situations, also. It is downright rude when a system goes down with no indication as to the cause of death. An error code displayed by an LED register might be just the message from beyond the grave needed to solve the problem.

It is possible to build more sophisticated debugging aids onto target systems, but it is generally not worth it. Beyond the additional cost, such systems are often more prone to failure than the hardware itself. A better approach is to have the hardware engineer spend time bulletproofing the core hardware, and then add some simple debugging aids such as LED's to assist software developers. In general, simpler is better.

Some more esoteric debugging aids may come "free" though, since they are beginning to be integrated into popular microprocessors. The debug registers in the Intel i386 are a good example, allowing breakpoints and some other emulate functions to be implemented without the need for additional hardware. Another noteworthy addition is the software watchdog timer on the Motorola 683XX line, which automatically causes an interrupt if a certain register is not accessed at regular intervals. This can be used to restart fault-tolerant systems or to trigger a postmortem dump which may provide clues as to the cause of death.

The advantage to debugging aids built onto the target hardware are that they are always available. They can be used to debug during system development, as well as after the system is in the field. The disadvantage is that they may add to the cost and complexity of the system being developed and they often do not provide the enough information to solve tough problems. This is when use of external debugging hardware is appropriate.
Hardware-Assisted Debugging
Larry Mittag
Independent Consultant
24140 Majella Dr.
Ramona, CA 92065
(619)789-4287
CIS:71545,546

Programming for embedded systems has always been complicated by the difficulty of debugging embedded code, due to restricted hardware environments in comparison to most PC or workstation environments. It is possible, though, to leverage debugging efforts by utilizing hardware on the target, as well as tools generally found in a hardware or systems development lab. In this paper, I will describe some techniques for utilizing various types of hardware to help ease the software debugging burden.

DEFINING THE PROBLEM

Debugging can be thought of as a search for information. The primary questions to be answered include determining what the code is doing at any given, how it got to a certain point, and how long it takes to perform an operation or set of operations. The basic difficulty in answering these questions with embedded systems is that there is normally less I/O capability available. Couple this with tight timing constraints, and the result can be a nightmare.

Attempts to utilize limited I/O capabilities such as serial ports for debugging purposes are at best difficult. At worst, a sort of Heisenberg Uncertainty Principle occurs, where the attempts to characterize the behaviour of the system has distorted that behaviour to the point where the information obtained may be useless. This is especially apparent when debugging real-time behaviour.

The goal is therefore to use nonintrusive measures to obtain information about the system. Any software utilized for this purpose that uses the target system resources will inevitably perturb the system, so we will explore other means.

DEBUGGING AIDS ON THE TARGET HARDWARE

It is remarkable how much information can be obtained if the target system itself is designed to facilitate software debugging. Addition of some LED's and a register to control them provides a low overhead (in terms of added code) means to determine the health of embedded code. The most obvious use of this is a "heartbeat" indication, allowing rapid determination of whether a board has died.

An example of this is the front panel that used to be found on minicomputers and microcomputers, with LED's continuously showing the state of the address and data lines. An experienced operator could tell at a glance whether the system was operating normally, as well as getting a good feel for how heavily loaded it was, since they often ran a certain pattern when the CPU was in its idle loop.

An LED display can help in postmortem situations, also. It is downright rude when a system goes down with no indication as to the cause of death. An error code displayed by an LED register might be just the message from beyond the grave needed to solve the problem.

It is possible to build more sophisticated debugging aids onto target systems, but it is generally not worth it. Beyond the additional cost, such systems are often more prone to failure than the hardware itself. A better approach is to have the hardware engineer spend time bulletproofing the core hardware, and then add some simple debugging aids such as LED's to assist software developers. In general, simpler is better.

Some more esoteric debugging aids may come "free" though, since they are beginning to be integrated into popular microprocessors. The debug registers in the Intel i386 are a good example, allowing breakpoints and some other emulator functions to be implemented without the need for additional hardware. Another noteworthy addition is the software watchdog timer on the Motorola 683XX line, which automatically causes an interrupt if a certain register is not accessed at regular intervals. This can be used to restart fault-tolerant systems or to trigger a postmortem dump which may provide clues as to the cause of death.

The advantage to debugging aids built onto the target hardware are that they are always available. They can be used to debug during system development, as well as after the system is in the field. The disadvantage is that they may add to the cost and complexity of the system being developed and they often do not provide the enough information to solve tough problems. This is when use of external debugging hardware is appropriate.
EMULATORS

The first piece of external equipment most programmers will turn to is an In-Circuit Emulator (ICE). This is logical, since a good ICE will provide an environment similar to a good symbolic debugger, allowing a very high degree of visibility and control of the target code. Using an ICE to single-step through code and examine variables and registers is almost the definition of debugging for many programmers.

Unfortunately, most programmers never look beyond that. An ICE should be thought of more as a set of resources that are temporarily being loaned to the target system, allowing the programmer to do things they couldn't otherwise do. A good example of this was a problem I was recently called in to solve.

A particular target system was experiencing a variety of flakey errors. Code that ran fine on other systems would act differently on this one, up to and including experiencing fatal errors. Any change in the code to add diagnostics to isolate the problem would change the behaviour, often making the problem go away completely.

I suspected hardware problems, but it was difficult to determine exactly what the nature of the problem might be. The hardware engineer who had designed the prototype board was in Taiwan, so consultation was expensive if not impossible. I left instructions that I was to be called when a fatal error occurred on that system, and that once it happened no one was to touch anything until I arrived.

It took several days, but the call did come. The system was getting an illegal instruction trap at a particular address. I used the ICE to examine that address, seeing what was indeed an illegal instruction. The corresponding byte in the load module differed by a single bit. Changing the value to the correct one allowed the code to run correctly.

I took advantage of the extra memory provided by the ICE to load the code twice, once at the correct location and once at a higher base address. I then had the ICE compare the two areas. The bit had been set wrong only in the lower memory area. It turned out that there was a memory error that caused that bit to be overwritten when a location 4096 bytes away was set. If the two happened to be the same, no problem. If they differed, flakiness resulted. I used the ICE to remap that area into ICE memory instead of target, and all problems went away.

The key to full use of an ICE is to understand fully the degree of control it really does provide. ICE memory can be used to supplement a stingy target, allowing the addition of test routines to the code to assist debugging. Watchpoints can be set to determine what part of the code is overwriting sensitive memory areas. It is also a great tool for ad hoc testing of hardware, since values can be poked into registers and the values read back without the need for writing code.

The other necessity for effective ICE use is to understand what it will not do. Vendor claims can lead you to assume that system behaviour under ICE control is exactly as it is when the ICE isn't installed. I tend to treat this as a pleasant surprise when it occurs, but I generally don't count on it. As a rule, I use an ICE for debugging program logic and most hardware problems. Critical timing problems in both hardware and software may call for other tools.

LOGIC ANALYZERS

Many programmers have never used logic analyzers, and may have been less effective as a result. They will tell you basically anything you want to know about the realtime behaviour of a system. The biggest problem with using them is probably filtering the data to get to the information you need.

A good example of this is a problem I ran into on a large Ada project. A critical load module would run fine under the ICE, but would die after about five minutes of running untethered. The error result was a coprocessor protocol violation between the 68020 and the 68881 processors. The address where it died was never the same. The question of the hour was a common one during system integration: was it software or hardware?

Obviously, ICE's couldn't be used to isolate the problem. I hooked up a Software Analysis Workstation, which is a logic analyzer sold by Cadre with a programmer's user interface and examined what was going on.

I set the trace to break on the error and tell me what happened before it. It soon became apparent that the error only occurred on the coprocessor operation following a coprocessor operation that had been interrupted. A close look at the interrupt handling system code showed the assumption that all stack frames were created equal. The bits that signaled the 68020 to restore the coprocessor status registers from the stack frame were being overwritten, losing the protocol from then on. Total time to find and fix the problem: one and a half days.

This is not to say that a logic analyzer is "better" than an ICE, only different. In general, an ICE is better at debugging logic problems, while a logic analyzer is more appropriate for problems related to system timing. The moral is that the
EMULATORS

The first piece of external equipment most programmers will turn to is an In-Circuit Emulator (ICE). This is logical, since a good ICE will provide an environment similar to a good symbolic debugger, allowing a very high degree of visibility and control of the target code. Using an ICE to single-step through code and examine variables and registers is almost the definition of debugging for many programmers.

Unfortunately, most programmers never look beyond that. An ICE should be thought of more as a set of resources that are temporarily being loaned to the target system, allowing the programmer to do things they couldn't otherwise do. A good example of this was a problem I was recently called in to solve.

A particular target system was experiencing a variety of flakey errors. Code that ran fine on other systems would act differently on this one, up to and including experiencing fatal errors. Any change in the code to add diagnostics to isolate the problem would change the behaviour, often making the problem go away completely.

I suspected hardware problems, but it was difficult to determine exactly what the nature of the problem might be. The hardware engineer who had designed the prototype board was in Taiwan, so consultation was expensive if not impossible. I left instructions that I was to be called when a fatal error occurred on that system, and that once it happened no one was to touch anything until I arrived.

It took several days, but the call did come. The system was getting an illegal instruction trap at a particular address. I used the ICE to examine that address, seeing what was indeed an illegal instruction. The corresponding byte in the load module differed by a single bit. Changing the value to the correct one allowed the code to run correctly.

I took advantage of the extra memory provided by the ICE to load the code twice, once at the correct location and once at a higher base address. I then had the ICE compare the two areas. The bit had been set wrong only in the lower memory area. It turned out that there was a memory error that caused that bit to be overwritten when a location 4096 bytes away was set. If the two happened to be the same, no problem. If they differed, flakiness resulted. I used the ICE to remap that area into ICE memory instead of target, and all problems went away.

The key to full use of an ICE is to understand fully the degree of control it really does provide. ICE memory can be used to supplement a stingy target, allowing the addition of test routines to the code to assist debugging. Watchpoints can be set to determine what part of the code is overwritten sensitive memory areas. It is also a great tool for ad hoc testing of hardware, since values can be poked into registers and the values read back without the need for writing code.

The other necessity for effective ICE use is to understand what it will not do. Vendor claims can lead you to assume that system behaviour under ICE control is exactly as it is when the ICE isn't installed. I tend to treat this as a pleasant surprise when it occurs, but I generally don't count on it. As a rule, I use an ICE for debugging program logic and most hardware problems. Critical timing problems in both hardware and software may call for other tools.

LOGIC ANALYZERS

Many programmers have never used logic analyzers, and may have been less effective as a result. They will tell you basically anything you want to know about the realtime behaviour of a system. The biggest problem with using them is probably filtering the data to get to the information you need.

A good example of this is a problem I ran into on a large Ada project. A critical load module would run fine under the ICE, but would die after about five minutes of running untethered. The error result was a coprocessor protocol violation between the 68020 and the 68881 processors. The address where it died was never the same. The question of the hour was a common one during system integration: was it software or hardware?

Obviously, ICE's couldn't be used to isolate the problem. I hooked up a Software Analysis Workstation, which is a logic analyzer sold by Cadre with a programmer's user interface and examined what was going on.

I set the trace to break on the error and tell me what happened before it. It soon became apparent that the error only occurred on the coprocessor operation following a coprocessor operation that had been interrupted. A close look at the interrupt handling system code showed the assumption that all stack frames were created equal. The bits that signaled the 68020 to restore the coprocessor status registers from the stack frame were being overwritten, losing the protocol from then on. Total time to find and fix the problem: one and a half days.

This is not to say that a logic analyzer is "better" than an ICE, only different. In general, an ICE is better at debugging logic problems, while a logic analyzer is more appropriate for problems related to system timing. The moral is that the
correct tool used intelligently is the shortest path between a problem and the solution.

ORGANIZATIONAL CONSIDERATIONS

Debugging is largely a human activity, so any discussion which only covers the technical aspects is incomplete. I have formulated a couple of general rules that have so far stood the test of time. The first of these is:

1. Debug a system as a system.

The industrial revolution taught us that work could be divided, allowing larger systems to be built. Unfortunately, that division is usually made along "functional" lines, leading to separate groups developing hardware and software, often subdivided further into those having responsibility for pieces of the hardware and software. This leads inevitably to finger pointing during system integration, since it is in the best interests of each group to have problems found in the others.

I ran into this problem in a large project. The hardware was horribly late, a fact used by the software group to slip schedules obscenely. When the hardware finally started arriving, it came to light that software was in at least as bad shape. There was no sympathy or support from the hardware group, since they still wore the scars inflicted by software management.

I was called in to work with a group of hardware engineers on a subsystem redesign on this project. I had been working with them for a while, and there was a large degree of mutual respect. I realized how different the atmosphere was when I was discussing the 68020/68881 problem described above with the lead hardware engineer whose group had selected the board. He was convinced it was a hardware problem, while I maintained it was software. I "won" the argument by proving the problem was in code written by my group. He took defeat gracefully.

2. Use tools before you need them.

I have often seen test and debug equipment sitting unused in a corner of the lab while engineers work feverishly to put out fires. The common explanation is that they don't have time to learn to use new equipment when they are working on the latest drop-deadline. Earlier in the project, when they were working on easier problems, they didn't need the help the new equipment would have given them, so the time wasn't allocated then, either.

People under pressure will revert to techniques that have worked for them in the past. The time to introduce new tools is before they are needed, and people should be encouraged strongly to use them, even if the old tools are still working for them. This will give them a wider repertoire to call on when crunch time hits, as it inevitably does.

CONCLUSION

Embedded systems programming is by nature a multidisciplinary activity. The wider scope of knowledge necessary to perform this activity effectively can be a burden, but the reward is in having a range of types of tools available. A programmer that doesn't take full advantage of these tools is tilting at bigger windmills without the benefit of bigger, better lances.
correct tool used intelligently is the shortest path between a problem and the solution.

ORGANIZATIONAL CONSIDERATIONS

Debugging is largely a human activity, so any discussion which only covers the technical aspects is incomplete. I have formulated a couple of general rules that have so far stood the test of time. The first of these is:

1. Debug a system as a system.

The industrial revolution taught us that work could be divided, allowing larger systems to be built. Unfortunately, that division is usually made along "functional" lines, leading to separate groups developing hardware and software, often subdivided further into those having responsibility for pieces of the hardware and software. This leads inevitably to finger pointing during system integration, since it is in the best interests of each group to have problems found in the others.

I ran into this problem in a large project. The hardware was horribly late, a fact used by the software group to slip schedules obscenely. When the hardware finally started arriving, it came to light that software was in at least as bad shape. There was no sympathy or support from the hardware group, since they still wore the scars inflicted by software management.

I was called in to work with a group of hardware engineers on a subsystem redesign on this project. I had been working with them for a while, and there was a large degree of mutual respect. I realized how different the atmosphere was when I was discussing the 68020/68881 problem described above with the lead hardware engineer whose group had selected the board. He was convinced it was a hardware problem, while I maintained it was software. I "won" the argument by proving the problem was in code written by my group. He took defeat gracefully.

2. Use tools before you need them.

I have often seen test and debug equipment sitting unused in a corner of the lab while engineers work feverishly to put out fires. The common explanation is that they don't have time to learn to use new equipment when they are working on the latest drop-deadline. Earlier in the project, when they were working on easier problems, they didn't need the help the new equipment would have given them, so the time wasn't allocated then, either.

People under pressure will revert to techniques that have worked for them in the past. The time to introduce new tools is before they are needed, and people should be encouraged strongly to use them, even if the old tools are still working for them. This will give them a wider repertoire to call on when crunch time hits, as it inevitably does.

CONCLUSION

Embedded systems programming is by nature a multidisciplinary activity. The wider scope of knowledge necessary to perform this activity effectively can be a burden, but the reward is in having a range of types of tools available. A programmer that doesn't take full advantage of these tools is tilting at bigger windmills without the benefit of bigger, better lances.