

RE: [UPDATED PATCH] EFI support for ia32 kernels

Source: <http://linux.derkeiler.com/Mailing-Lists/Kernel/2003-09/1557.html>

From: Doran, Mark (mark.doran_at_intel.com)

Date: 09/05/03

Date: Thu, 4 Sep 2003 20:52:08 -0700
To: "Linus Torvalds" <torvalds@osdl.org>

On Thu, 4 Sep 2003, Linus Torvalds wrote:

>>

>> *It would be nice. It is especially nice for vendors because they can reuse a single driver image for multiple architectures assuming there is*

>> *an interpreter and EFI support.*

>

>*No. It would be a total nightmare.*

As one of the people responsible for the EFI Specification and our industry enabling efforts around that spec, I'd like to offer some background that I hope will illuminate some of the issues discussed in this thread. This is going to be a bit long...let me apologize in advance for that but I think there's quite a bit of context here and sharing that may help people understand why EFI works the way it does for Option ROMs.

In 1999 when we were first working on the EFI spec in draft form, a number of the OEMs and IHV companies that we talked to told us that an EFI spec without a solution for the "option ROM" problem would not be accepted in the industry.

At that time, I tried to make the case that instead of propagating the problem into the future we should focus on moving the industry to "architectural hardware" that wouldn't even need option ROMs. What I meant by that was add-in cards with common register-level hardware interfaces to allow operating systems code to carry driver and boot loader code that would be able to work across a range of vendors' products. Perhaps the UNDI network card interface that Intel developed would be a good model for a start at this approach as an example; both in terms of how to do it and the level of traction (or lack thereof) one can expect taking this approach.

The trouble with the "architectural hardware" argument proved to be that PCI is already well established and there is a vibrant industry churning

out innovative PCI cards on a regular basis. The idea of a single interface definition for all cards of each of the network, storage or video classes is viewed as simply too limiting and the argument was made to us that to force such a model would be to stifle innovation in peripherals. So effectively the feedback we got on "architectural hardware" was therefore along the lines of "good idea but not practical..."

Faced with that and what amounts to a demand for a solution, we tried to scope the problem. Today's IA-32 Option ROMs are typically 16-bit, IA-32 real mode code, they must live in a magic 128k (192 on some boxes) window below 1MB, and there are no hard and fast rules about what resources on the machine they may or may not touch. The reason the OEM folks asked us to look at solving this issue set in the context of EFI is to try and improve the real nightmares that they face every day. The kind of thing where you plug in an adapter card and suddenly the floppy doesn't work anymore. The kind of thing where you plug in four SCSI controllers and it's highly likely that you can't reach a perfectly good OS install on a drive connected to one of those controllers because the BIOS can't shadow that much ROM code. The kind of thing where ROM code uses I/O reads in lieu of calibrated delays causing controllers to fail on newer, faster systems.

EFI's origins come from the 64-bit side of the house. It was originally conceived in the context of a need for a means to handle programmatic transfer of control from the platform code (BIOS/firmware) to the OS; in other words an abstraction for the platform to support booting shrink-wrap OSes, installed right off the distribution CDs. However, we also worked hard at building a C language binding for those interfaces that would work just as well for IA-32 or even XScale or perhaps even for non-Intel Family processors in fact. The idea being a piece of code written to consume EFI services can compile unmodified and without gratuitous #ifdef's for 32-bit or 64-bit system merely by choice of compiler.

In the context of option ROMs then, this approach would say that you can write a single chunk of C language EFI driver code, your option ROM equivalent. This code can load anywhere in the address space of the machine (EFI uses protected mode, virtual equals physical addressing model), it can use the full address width of the machine for data references and you can compile it for your target machine architecture of choice. So far so good.

However there are some other practical deployment issues that add-in cards bring to bear that we also had to address.

These cards have a habit of traveling from machine to machine. Customers have a reasonable expectation that cards just work when you move them from one system and plug them in to another. Since the receiving system motherboard might have no knowledge of the card you just added, how does it present devices connected to that card as

candidates for booting?? Motherboards cannot reasonably carry code for every device a customer might choose to plug in. Addressing that problem is what the Option ROM does for you.

We also tried to advocate for having the drivers/Op ROM images be separately distributed. Some of the IHVs liked that: just ship a floppy with the card, much cheaper than putting NVRAM memory on card. The OEM folks however point out that the floppy gets lost and now the card is useless to the customer...support calls ensue. Thus the code needs to travel as part of the card.

If a card can travel from system to system, that also means it can cross processor architecture boundaries too – there are Itanium Family and IA-32 family machines with PCI slots that are electrically compatible and the expectation is that the cards work equally well in both system types. For the Option ROM content though that presents a dilemma – what do you carry in the ROM?? Native compiled IA-32 code and also native compiled Itanium family code perhaps. Well that works, the PCI spec says a ROM container can have multiple images; we take advantage of that now to build cards that carry a 16-bit conventional ROM and an EFI driver together and there are also Forth images out there for SPARC and Power systems.

As a practical matter carrying multiple instruction set versions of the same code gets expensive in FLASH memory terms. Consider an EFI compiled driver for IA-32 as the index, size: one unit. With code size expansion, an Itanium compiled driver is going to be three to four times that size. Total ROM container requirement: one unit for the legacy ROM image plus one for an EFI IA-32 driver plus three to four units for an Itanium compiled driver image; to make the card "just work" when you plug it into a variety of systems is starting to require a lot of FLASH on the card. More than the IHVs were willing to countenance in most cases for cost reasons.

EFI Byte Code was born of this challenge. Its goals are pretty straightforward: architecture neutral image, small foot print in the add-in card ROM container and of course small footprint in the motherboard which will have to carry an interpreter. We also insisted that the C source for a driver should be the same regardless of whether you build it for a native machine instruction set or EBC.

We did some other things with EBC's definition too; like not including direct I/O instructions. That may sound odd for an environment specifically designed for I/O devices but if you think about it, it's the motherboard code that knows what is and is not "safe" to do by way of I/O more than the device itself that could find itself in pretty much any old machine design. This we believe will significantly improve the reliability of ROM code...it relieves the add-in card Op ROM writer of any attempts to guess and assume what the I/O environment is that the card will encounter out in the field.

You may ask why we didn't just use an existing definition as opposed to making a new one. We did actually spend quite a bit of time on that very question. Most alternatives would have significantly swelled the ROM container size requirement or the motherboard support overhead requirement or had licensing, IP or other impediments to deployment into the wider industry that we had no practical means to resolve. With specific reference to why we chose not to use the IA-32 instruction set for this purpose, it was all about the size of an interpreter for that instruction set. To provide compatibility 100% for the universe of real mode option ROM binaries out there would require a comprehensive treatment of a very rich instruction set architecture. We could see no practical way to persuade OEMs building systems using processors other than IA-32 to carry along that much interpreter code in their motherboard ROM.

Consider the model of Alpha and FX32 as an example; FX32 would be impractical to carry on the motherboard outside the scope of a running OS. At one point we did have an EFI draft that included a processor binding for Alpha at Compaq's request. That material isn't in the final spec for reasons that don't really relate to EFI. Nevertheless, making an Option ROM solution that could plausibly work on multiple CPU architectures (including ones from outside the IA family) and before there is an OS on the box to support an expansive interpreter loomed large in our thinking at the time. [By the by, we remain open to adding other CPU bindings into the EFI spec should anyone approach us with such a proposal in hand.]

By contrast EBC requires a small interpreter with no libraries (roughly 18k uncompressed total on IA-32 for example) and the average add-in card ROM image size is 1.5 units relative to native IA-32 code. And keep in mind that using byte code for this purpose is in widespread, long time use on other CPU architectures so we felt the technique in general was viable based on industry experience with it. Yes, it's a compromise but the best balance point we have been able find to date.

I agree that the compiler back end for EBC will be used for small chunks of code and relatively few of them at that. That compiler and its back end will by definition end up with less code-mileage on it, if you will. I can only say that Intel is supporting the compiler as a commercial product and we stand behind it just as much as we do the native IA-32 and Itanium compilers. Find a bug, let us know – we'll fix it. We run the same tests on the EBC compiler as we do on the native compilers and a few more besides that do EBC torture exercises. The compiler has been in testing for more than a year and in release for nearly than long now. At any rate, feedback we've received so far doesn't seem to indicate stability problems in the compiler; if your experience varies from that please let me know – I'd like to help fix it for you! Incidentally, nothing prevents someone from retargeting GCC for this application.

It is with no small trepidation, given the assembled company, that I turn to the question of Open Source as it relates to EFI and Option ROM

code. However...

There is nothing about the definition of the EFI spec or the driver model associated that prevents vendors from making add-in card drivers and presenting them in Open Source form to the community. In fact we've specifically included the ability to "late bind" a driver into a system that speaks EFI. In practice that late binding means that code that uses EFI services and that is GPL code can be used on systems that also include EFI code that is not open source.

The decision on whether to make any given driver Open Source or not therefore lies with the creator of that code. In the case of ROM content for an add-in card that will usually be the IHV that makes the card.

Now, we observe that the high-end add-in card makers often preserve their intellectual property behind proprietary code (via binary drivers) and/or "object models" that they implement in the ROM. In today's lexicon that means an INT13 service for a SCSI card or an INT10 service for a video card. Even for Linux OS-present drivers I understand that some open source drivers for such cards don't actually touch the metal directly for all operations they perform – they use some abstraction between the driver and the actual hardware, something that is carried around in the ROM. I suspect that this paradigm is one that will continue for some time to come. Any change in this approach will have to be worked with the vendors who feel commercial pressure to protect their IP with these kinds of mechanisms.

The EFI spec itself is published with a simple copyright statement and not one of Intel's "colored" NDA covers. The sample code that you can download from our web site is free to you and comes with what amounts to a patent license grant so that you can implement EFI, perhaps using our code in derivative fashion, without royalty or other concern. We also have support tools for folks building EFI code, like Option ROM drivers, that is distributed under the FreeBSD license.

Thanks for reading along this far and please let me know if you have any follow up questions or comments.

Cheers,

Mark.

--

Mark Doran
Principal Engineer
Intel Corp., DuPont, WA

-

To unsubscribe from this list: send the line "unsubscribe linux-kernel" in the body of a message to majordomo@vger.kernel.org
More majordomo info at <http://vger.kernel.org/majordomo-info.html>
Please read the FAQ at <http://www.tux.org/lkml/>