Forschungsbericht WS 2011/2012

Analyse des Userspace I/O Framework (UIO) für den Einsatz in Embedded Linux-Systemen

Prof. Dr.-Ing. Rainer Bermbach

Einleitung

Betriebssysteme wie Linux benötigen sogenannte Treiberprogramme, mit denen sie auf die Hardware des Rechnersystems (z.B. den Festplattencontroller oder das Display) zugreifen und damit arbeiten können. Auch sog. Embedded Systeme brauchen beim Einsatz von Embedded Linux solche Treiber, die aber im Gegensatz zu PC-Systemen nicht schon im Betriebssystem enthalten, sondern spezifisch zu erstellen sind. Beim Einsatz von FPGA-basierten Systemen für prototypische Anwendungen entstehen immer wieder spezielle Hardwaremodule, für die beim Betrieb unter Linux auch immer spezielle Treiberprogramme entworfen werden müssen.

Die übliche Treiberentwicklung geschieht im sog. Kernelspace, dem eigentlichen Kern des Betriebssystems. Dort ist die Entwicklung aus verschiedenen Gründen ungleich komplizierter und schwieriger als die Programmentwicklung für den normalen Anwenderbereich im sog. Userspace. Für viele Arten von Hardwaremodulen gibt es im Linux-Kernel vorgefertigte Subsysteme (Klassen, z.B. für PCI, USB). Die Treiberentwicklung (im Kernelspace) für solche Module ist überschaubar, weil vieles bei diesen Subsystemen gleich ist und daher schon im Kernel bereitgestellt wird. Spezifische Hardware fällt aber i.a. nicht in eines dieser Subsysteme und benötigt eine komplette, vom Kernel weitgehend ununterstützte Treiberentwicklung. Weiterhin sind die Softwareschnittstellen für benötigte Systemaufrufe (Kernel API) nicht konstant, sondern ändern sich vergleichsweise häufig, oft von Version zu Version des Betriebssystems. Dies macht immer wieder zum Teil umfangreiche Änderungen an Treibern notwendig.

Ein Projekt im WS 2010/2011 beschäftigte sich mit der Einbindung von FPGA-basierter Spezialhardware in Embedded Linux-Systeme. Ein Aspekt war dabei, ein möglichst universelles Vorgehen bei der Einbindung der Hardware bzw. beim Treiberentwurf zu erreichen. Aus den o.g. Gründen ist diese Treiberentwicklung aber immer noch vergleichsweise aufwändig.

Seit Linux 2.6.23 enthält das Betriebssystem GNU/Linux bzw. der Linux-Kernel ein sog. Userspace I/O Framework, kurz UIO [Koch2006, Kunst2007]. Dieses Framework erlaubt es, Linux-Treiber für bestimmte Arten von Hardware fast vollständig im Userspace des Betriebssystems zu erstellen. Dort stehen die üblichen, gewohnten Entwicklungshilfsmittel zur Verfügung, was die Treiberentwicklung im Gegensatz zur normalen im Kernelspace wesentlich vereinfacht. Auch die Schnittstellen für die Systemaufrufe (Kernel API) sollen über Betriebssystemversionen konstant (oder wenigstens „konstanter“) sein, was notwendige Treiberanpassungen einschränkt. Das Userspace I/O Framework verspricht also eine deutliche Reduzierung des sonst üblichen Aufwands bei der Erzeugung der Hardwaretreiber.

Vergleich von konventionellen Treibern mit UIO-Treibern

Bei Treibern muss man unterscheiden zwischen sog. Character Devices und Block oder Stream Devices. Nur für die erste Gruppe ist das UIO Framework geeignet. Standardtreiber für Character Devices werden über die Funktionen *read()* und *write()* angesprochen. Komplexere Behandlungen wickelt die Funktion *ioctl()* ab. Auch das vollständige Interrupt Handling geschieht im Kerneltreiber. Wie Abbildung 1 verdeutlicht, verwendet ein solcher Treiber viele spezifische Kernelfunktionen, deren API sich vergleichsweise häufig ändern kann. Dadurch ist der Pflegeaufwand für einen Hardwaretreiber relativ hoch, da typischerweise solche Treiber wegen ihrer speziellen Hardware nicht in den Kernel selbst mit aufgenommen werden und deshalb auch nicht von der Allgemeinheit aktuell gehalten werden. Außerdem kann der Treiber relativ umfangreich sein, da nicht auf Standardklassen wie bei Standardhardware zurückgegriffen werden kann.

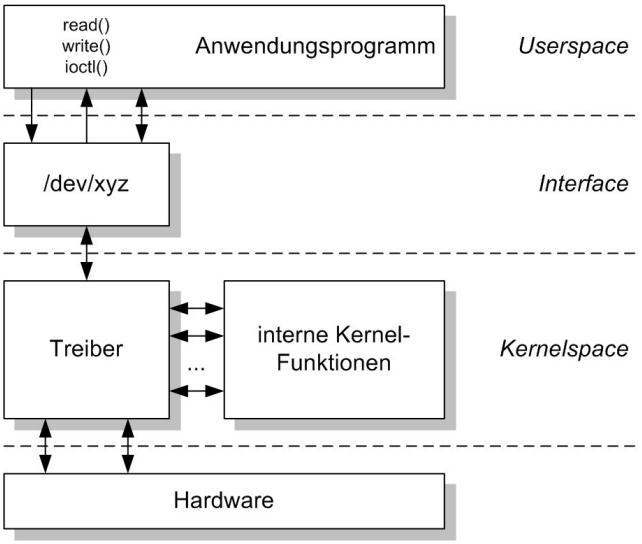

Abb. 1: Aufbau eines konventionellen Treibers

Was macht das Userspace I/O Framework anders (s. Abbildung 2)? Die grundlegende Idee ist, eine allgemeine Abstraktionsschicht über die für Character-Device-Treiber benötigten Kernelfunktionen zu legen. Diese Schicht wird als allgemeiner Teil des Kernels gepflegt. Das Programmierinterface (API) zum Treiber bleibt gleich. Weiterhin erlaubt man nach Anmeldung einen direkten Zugriff auf die Hardware vom Userspace aus. Mit der Funktion mmap() werden Speicherbereiche und Register der Hardware in den Speicherbereich der Anwendung gemappt und damit direkt zugänglich.

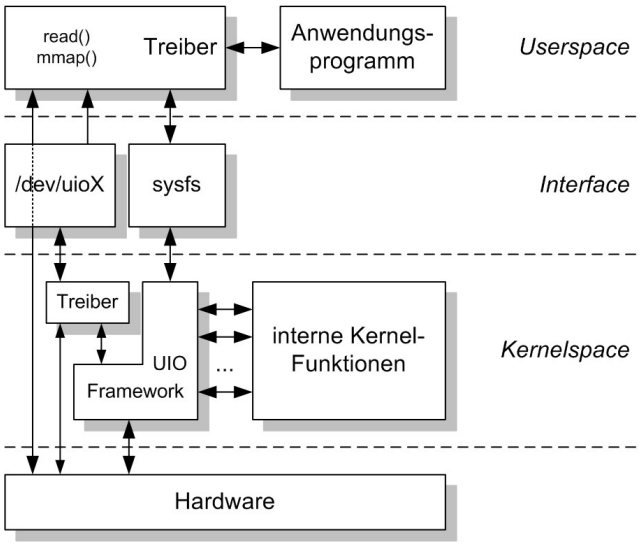

Abb. 2: Struktur eines UIO-Treibers

Im Gegensatz zu anderen Mechanismen garantiert das UIO Framework, dass nur zur spezifischen Hardware gehörige Bereiche dem Userspace-Treiber verfügbar gemacht werden. Der noch im Kernel verbleibende Teil des Treibers reduziert sich auf ein Minimum. Er reagiert beispielsweise auf auftretende Interrupts. Diese können nur im Kernel behandelt werden. Allerdings geschieht dort kaum mehr als die direkte Reaktion auf den Interrupt und seine Bestätigung (Acknowledge). Die restliche Behandlung erfolgt dann auch wieder im Userspace. Will der Userspace-Part auf einen Interrupt warten, so führt er ein blockierendes *read()* aus. Tritt ein Interrupt auf, kommt *read()* mit der laufenden Nummer des Interrupts zurück. Auch ein Polling-Modus (inkl. Timeout) ist mithilfe der Funktion *select()* möglich. Weiterhin legt das UIO Framework auch im virtuellen Filesystem sysfs (vom Userspace zugänglich) verschiedene Verzeichnisse

und Attributdateien an, die z.B. Informationen über den gemappten Speicher u.ä. enthalten. Die Schnittstelle zwischen Userspace-Treiber und Anwendung ist nicht definiert und kann nach Bedarf gestaltet werden.

Struktur von UIO-Treibern

Wie schon beschrieben, benötigt ein UIO-Gerätetreiber einen Part im Userspace und einen im Kernelspace. Der Kerneltreiber ist verhältnismäßig klein und einfach gestaltet. *module_init()* und *module_exit()*, die dem Kernel die Initialisierungsfunktion und die Exit-Funktion bekannt machen, und *MODULE_LICENSE()* sind wie in Standardtreibern vorhanden. Die Initialisierungsfunktion *static void __init uio_kpart_init()* registriert den Treiber im System, während *stativ void __exit uio_kpart_exit()* beim Entladen des Treibers diesen beim System abmeldet. Beide bestehen aus sehr wenigen, einfachen Kernel-Funktionsaufrufen. Die Funktion *static int drv_kpart_remove()* entfernt beim Abmelden des Device die entsprechenden Einträge im sysfs.

Die Funktion static int drv_kpart_probe() legt ihrerseits beim Start die betreffenden Einträge im sysfs an und vergibt die Major- und Minornummer für den Treiber. Die Informationen hierzu gewinnt die dafür aufgerufene Systemfunktion aus der Datenstruktur *struct uio_info kpart_info*, die vorher ebenfalls in *probe()* gefüllt wird. Neben dem Namen des Device und der Version werden Startadresse, Typ und Größe des Speicherbereichs oder I/O-Bereiches des Device eingetragen (bei mehreren Bereichen entsprechend mehrfach). Weiterhin enthält *kpart_info* die Interruptnummer, den Typ (z.B. ob shared) und den Namen der Interruptserviceroutine. Nötige Initialisierungen der eigenen Hardware finden ebenfalls in *probe()* statt. In die Struktur *static struct device_driver uio_driver* trägt man den Device-Namen, die Probe- und die Remove-Funktion ein.

Die Interruptbehandlung muss in der Funktion *static irqreturn_t handler()* erfolgen. Sie überprüft, ob die eigene Hardware überhaupt einen Interrupt ausgelöst hat (sonst return mit IRQ_NONE) und schaltet ihn anschließend aus (Interrupt Acknowledge). Das ist schon alles, was im Kernel an Treiberfunktionalität realisiert werden muss. Da es sich bei typischen Character Devices immer um ähnliche Grundfunktionalitäten handelt, sind die Unterschiede zwischen den Kernel-Parts der Treiber verhältnismäßig gering.

Im User-Part des Gerätetreibers werden zuerst im sysfs die zum Device gehörigen Dateien gelesen. Dadurch erhält man Adresse und Größe des Speicherbereichs und kann diesen anschließend per *mmap()* dem Userspace-Programm zugänglich machen. Dieses kann dann direkt ohne Umwege dort lesen und schreiben (s. Abbildung 3).

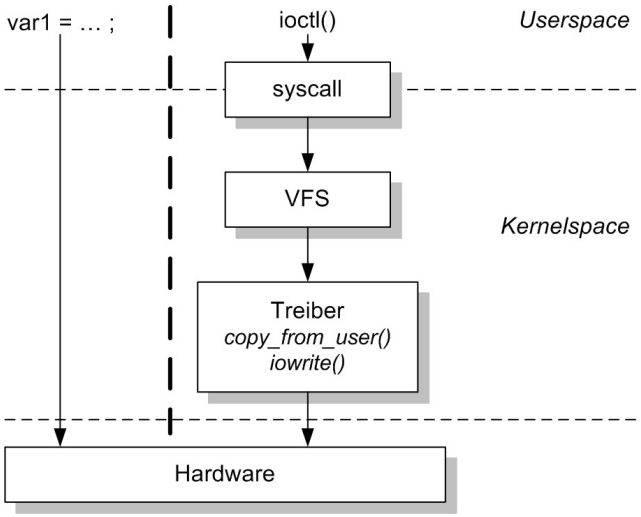

Abb. 3: Hardwarezugriffsmethoden bei UIO- und konventionellen Treibern im Vergleich

Hat die Hardware mehrere Speicher- oder I/O-Bereiche, muss der Vorgang dafür wiederholt werden. Der Rest eines Userspace-Treibers behandelt die spezifische Funktionalität der Hardware und die Kommunikation mit der Anwendung. Selbstverständlich ist auch eine direkte Einbindung der Treiberfunktionalität in der Anwendung machbar. Dadurch entfällt die zeitaufwändige Interprozesskommunikation. Der direkte

Speicherzugriff sorgt für einen sehr effizienten und schnellen Kommunikationspfad mit den Speicherbereichen. Die Systemaufrufe mit ihren umständlichen und zeitraubenden Umkopieraktionen und Kontextwechsel beim konventionellen Zugriff über *ioctl()* entfallen.

Einschränkungen durch das UIO Framework bei Hardware- und Treiberfunktionalitäten

Das Userspace I/O Framework ist primär nur für sog. Character Devices konzipiert. Dies deckt aber den Löwenanteil der spezifischen Hardwaremodule ab, für die Anwender eigene Treiber schreiben müssen. Block und Stream Devices werden andererseits vom Kernel durch entsprechende Klassen bereits gut unterstützt. Es gibt aber Entwicklungen auch für Block Devices eine Art UIO zu realisieren, was aber noch nicht allgemeine Anerkennung gefunden hat.

Was bislang für UIO noch fehlte, war die Möglichkeit, DMA (Direct Memory Access) im Userspace einzusetzen. Dies scheint mit UIO-DMA kürzlich [Kras2012] verbessert worden zu sein, konnte aber noch nicht getestet werden.

Untersuchung des Zeitverhaltens von UIO-Treibern

Wie schon erwähnt geschieht der Zugriff auf Speicher- oder I/O-Bereiche direkt und ist daher sehr schnell. Interessanter ist das Zeitverhalten von UIO-Systemen beim Auftreten von Interrupts.

Hierzu wurden Messungen in konkreten FPGA-basierten Systemen durch geführt [Rod2011]. Entsprechende Hardware löste Interrupts aus. Der Kernel-Part des Treibers setzte direkt bei Behandlung des Interrupts ein Signal. Auf diese Weise kann über die Zeitdifferenz Interruptauslösung – Interruptbehandlung die Latenz bis zum Start der Interruptverarbeitung erfasst werden.

Um die Zeit bis zur Reaktion eines UIO-Treibers im Userspace zu bestimmen und mit konventionellen Treibermethoden vergleichen zu können, entstanden drei verschiedene Beispieltreiber. Der erste ist ein rein Kernel-basierter Treiber. Er setzt ein Signal, sobald das Interruptsignal behandelt war. Ein zweites Signal setzt er im Anschluss, um so eine Vergleichszeit für die anderen Treiberversionen zu erhalten.

Der zweite Treiber ist ein UIO-Treiber. Auch bei ihm setzt der Interruptteil bei der Behandlung des Interrupts ein Signal. Das zweite Signal wird hier durch den Userspace-Teil des Treibers aktiviert, also wenn die Kontrolle nach dem Interrupt (Ende eines blockierenden *read()*) wieder an den Userspace übergeben ist. Der dritte ist ein konventioneller Treiber, der über ioctl() aus dem Userspace das zweite Signal gibt.

Alle Messungen erfolgten einmal ohne besondere Belastung des Systems und einmal mit starker Last (100%). Der Lastfall wurde noch einmal wiederholt mit einer erhöhten Priorität der entsprechenden Userspace-Programmteile.

Die Latenz bis zum Start der eigentlichen Interruptverarbeitung war bei allen drei Versionen erwartungsgemäß ähnlich und lag ohne Last bei etwa 28 – 30µs. Die Last erhöhte diese Zeit auf etwa 34,5µs, wie erwartet unabhängig von Prioritäten im Userspace.

Das zweite Signal erschien beim reinen Kerneltreiber nach durchschnittlich 31,5µs, mit Last nach 36,2µs. Der UIO-Treiber benötigte im Schnitt 142,4µs bzw. 198,0µs unter Last und 196,8µs bei Prioritätserhöhung. Der konventionelle Treiber brauchte 147,4µs, unter Last 191,2µs und 190,1µs mit erhöhter Priorität.

Dies zeigt, dass die beiden Treibervarianten im Userspace durch den Kontextwechsel in den Userspace deutlich länger brauchen, um zu reagieren. Gezielte Messungen des Context Switch zeigten einen Zeitbedarf in der Größenordnung der Differenz zu den Zeiten des Kernel-basierten Treibers. Weiterhin ist den Messungen zu entnehmen, dass es keine wesentlichen Unterschiede in der Reaktionszeit auf einen Interrupt

gibt zwischen UIO- und konventionellen Treibern. Auch unter Last gibt es keine signifikanten Änderungen. Lediglich die Varianz der Zeiten erhöht sich durch die Belastung, was aber leicht nachvollziehbar ist. Eine Kommunikation der Userspace-Programme mit einer Anwendung wurde nicht berücksichtigt. Sie würde die Zeiten natürlich entsprechend erhöhen.

Vergleichende Messungen unter Einsatz des RT-Preempt Patch konnten aus Zeitgründen nicht durchgeführt werden. Die Untersuchungen zu RT Preempt aus dem SS11 lassen aber vermuten, dass der Patch zu höheren Latenzen bis zum Start der eigentlichen Interruptverarbeitung führen wird. Interessant wäre zu untersuchen, ob die Latenzen für die Userspace-Reaktion insbesondere unter Last bei zugewiesener höherer Priorität deutlich niedriger ausfallen als hier gemessen.

Ergebnisse

Das Projekt zeigte, dass das Userspace I/O Framework generell die Treibererstellung von spezifischen Character Devices deutlich vereinfacht. Es wurden daher entsprechende Treiber-Templates für den kleinen Kerneltreiberteil sowie für den Userspace-Teil erstellt, die nur noch geringfügig angepasst werden müssen. Die umfangreichste Adaption betrifft die spezifische Funktionalität der Hardware, die der Treiber natürlich abbilden muss. Die durchgeführten Messungen belegen, dass keine signifikanten Unterschiede bei den Interruptlatenzen im Vergleich zu konventionellen Treibern existieren. Der direkte Hardwarezugriff ist dagegen drastisch schneller. Einschränkungen wie fehlender DMA-Betrieb scheinen durch neuere Entwicklungen beseitigt zu werden. Somit bildet das Userspace I/O Framework eine fast ideale Basis für die Erstellung von Embedded Linux-Treibern insbesondere für prototypische, FPGA-basierte Systeme.

Literatur

Koch2006: Koch, H-J.: *The Userspace I/O Howto*. Kernel.org, 2006. http://www.kernel.org/doc/htmldocs/uio-howto.html

Koch2007: Koch, H-J.: *Eingeklinkt – Linux-Treiber in den Userspace auslagern mit Userspace-I/O*. In: c't Heft 21 (2007), S. 222.

Kras2012: Krasnyansky. M.: *UIO-DMA*. Open-Source.Qualcomm.Com, 2012. https://opensource.qualcomm.com/wiki/UIO-DMA

Kunst2007: Kunst, E-K.; Quade, J.: *Kernel- und Treiberprogrammierung mit dem Kernel 2.6 – Folge 36*. In: Linux-Magazin, Heft 11 (2007).

Rod2011: Rod, O.: Treibererstellung unter Linux mit dem Userspace I/O Framework auf einem FPGA-basierten Prozessorsystem. Diplomarbeit. Wolfenbüttel: Ostfalia Hochschule für angewandete Wissenschaften, 2011.

Kontaktdaten

Ostfalia Hochschule für angewandte Wissenschaften
Fakultät Elektrotechnik
Prof. Dr.-Ing. Rainer Bermbach
Salzdahlumer Straße 46/48
38302 Wolfenbüttel
Telefon: +49 (0)5331 939 42620
E-Mail: r.bermbach@ostfalia.de
Internet: www.ostfalia.de/pws/bermbach