**Forschungsbericht WS 2010/2011**

# Embedded Linux auf FPGA-basierten Systemen mit freien Prozessor-IPs (3. Teil)

## Unterthema: Entwicklung und Einbindung von Spezialhardware in Embedded Linux

**Prof. Dr.-Ing. Rainer Bermbach**

Der dritte und letzte Teil des Projektes *Embedded Linux auf FPGA-basierten Systemen mit freien Prozessor-IPs* beschäftigte sich mit der Entwicklung von eigenen Hardwaremodulen und ihrer Einbindung und Nutzung in einem Embedded Linux-System. Der erste Part klärte, wie ein passendes Embedded Linux zugeschnitten realisiert werden kann, während der zweite Teil den Linux-Bootprozess auf eingebetteter Hardware untersuchte.

Der enorme Vorzug von FPGA-basierten Embedded Systemen gegenüber regulären liegt in der großen Flexibilität, die die programmierbare Hardwareerstellung (z.B. in VHDL im FPGA ermöglicht. So können die Entwicklung neuer oder die Modifikation bestehender Module das System erweitern und für neuartige Anwendungen nutzbar machen, was bei herkömmlichen Systemen ein komplett neues Hardware Board erfordern würde. Dies ist speziell in prototypischen Anwendungen ein großer Vorteil, da so sowohl Hardware als auch Software vergleichsweise einfach an die Aufgabe angepasst werden können.

Um das Vorgehen bei der Entwicklung und bei der Einbindung von Spezialhardware zu untersuchen, wurde im Projekt eine Reihe von Hardwaremodulen erstellt. Der Schwerpunkt lag dabei weniger auf der Komplexität der Module, als vielmehr auf typischen Interfaces zum steuernden Prozessor. Diese verfügen im Allgemeinen über ein oder mehrere Datenregister (8 - 32 Bit), in denen der Prozessor Daten an das Modul zur Verarbeitung übergibt, und/oder ein oder mehrere Datenregister (8 - 32 Bit), aus denen der Prozessor verarbeitete Daten vom Modul übernimmt. Zusätzlich ist meist noch ein Kontrollregister vorhanden, in dem die Betriebsarten des Moduls gesteuert werden. Ein weiteres Register, das Statusregister, meldet dem Prozessor den Zustand des Moduls, z.B. dass neue Daten zur Abholung bereit stehen oder dass die übergebenen Daten ausgesendet oder verarbeitet sind. Diese Register können in den verschiedenen Systemen entweder im sog. I/O-Bereich als sog. Ports realisiert (I/O mapped I/O) oder auch im regulären Speicherbereich des Prozessors abgebildet werden (Memory mapped I/O).

Als erstes entstand ein einfaches Hardwaremodul, das die auf dem FPGA-Board befindlichen DIP-Switches auslesen sowie die dortigen LEDs ansteuern kann. Das Beispiel eignete sich besonders, da hierfür bereits von Xilinx ein entsprechendes Modul vorhanden ist, das als Referenz dienen kann. Die Hardwareerstellung an sich ist vergleichsweise einfach. Komplizierter ist der Anschluss an den Prozessorbus, hier der *Processor Local Bus* (PLB). Da dies eine immer wiederkehrende Aufgabe ist, hat Xilinx glückli-

cherweise dieses Interfacing durch einen entsprechenden *Wizard* erleichtert. Über eine grafische Oberfläche legt man u.a. fest, über wie viele Register die Anwendung verfügt und mit welchem Businterface das Modul ausgestattet sein soll. Der Wizard liefert letztlich zwei VHDL-Dateien, die eine (Userlogic) für die Realisierung der eigentlichen Hardware mit einem Schreib-/Leseinterface zum gewählten Bus, die andere für das tatsächliche (immer weitgehend gleiche) Interface zum PLB, das die Userlogic instanziiert. In der ersten Datei implementiert man dann die gewünschte Funktionalität (im Beispiel also den Zugriff auf die DIP-Switches bzw. die LEDs). Anschließend kann man das Modul in den allgemeinen Katalog der verfügbaren IPs (eigene und mitgelieferte Hardwaremodule) aufnehmen. Dort ist es dann verfügbar und kann beim Erstellen eines neuen Prozessorsystems gewählt und eingebaut werden.

Beim Einbinden der neuen Hardware wählt man u.a. die Lage im Adressbereich sowie die tatsächlichen Anschlüsse auf dem Board (*.ucf). Diese und weitere Informationen speichert das System u.a. in der Datei xparameters.h und in einer *.h-Datei mit dem Namen des Moduls, auf die bei der Softwareentwicklung zurückgegriffen wird. Über diese sind die Adressen und die Prototypen entsprechender Schreib-/Leseroutinen softwaremäßig verfügbar. Mit Hilfe dieser Schreib-/Leseroutinen kann das Hardwaremodul getestet und in eigenständigen Applikationen eingebunden werden. Die Programme selbst integriert man ebenfalls in der Xilinx-Entwicklungsumgebung und wählt den Speicherbereich aus, in die sie beim Herunterladen der FPGA-Hardwareinformationen geladen werden. Beim Start der Hardware beginnt auch der implementierte Prozessor mit der Abarbeitung des geladenen Programms und bedient z.B. die LEDs – in einem einfachen Beispiel gibt er die an den DIP-Switches eingestellten Werte auf den LEDs aus.

In ähnlicher Weise entstanden auch weitere Hardwaremodule. Ein *Parallel Printer Interface* bedient über eine passende Zustandsmaschine einen Drucker mit Parallelinterface und liefert den Zustand des Druckers (z.B. Busy) auf die Applikationsprogrammebene. Ein *Simple UART* (ein einfacher, asynchroner serieller Sender/Empfänger) empfängt serielle Daten (RS232) und kann solche aussenden. Der UART kann vom Prozessor aus mit entsprechenden Registerzugriffen gelesen und geschrieben werden. Im Nachgang zu diesem Projekt wurden die gesammelten Erfahrungen genutzt, um das Labor Design Digitaler Systeme zu Beginn des Sommersemesters zu aktualisieren. Dort entwickeln die Studierenden jetzt eigene Hardwaremodule mit vergleichbaren Eigenschaften und binden sie in einem FPGA-basierten Prozessorsystem ein, das sie entsprechend programmieren. Das gesamte FPGA-basierte Embedded System empfängt dann serielle Daten und stellt sie über ein Software-FIFO (Ringpuffer) dem Drucker zur Ausgabe zur Verfügung. Kann der Drucker nicht schnell genug ausgeben oder ist angehalten (Busy), kann die Applikationssoftware über den seriellen Sender dem Datenlieferanten mitteilen, dass vorläufig keine neue Daten gesendet werden dürfen (XON-/XOFF-Protokoll).

Ein letztes implementiertes Hardwaremodul realisiert die Umcodierung von dualen Daten in den Gray-Code – der *GrayCoder*. Hier entstanden gezielt auch andere Interfacevarianten. Neben der sonst gewählten Registerimplementation am *Processor Local Bus* wurde ein Modul mit FIFO-Anbindung am PLB realisiert sowie eines mit Anbindung an dem *Fast Simplex Link* (FSL Bus). Letzterer ist eine Art Coprozessorinterface. Die Hardware ist direkt an einem Prozessor-eigenen Register angeschlossen. Da der FSL unidirektional arbeitet, benötigt der GrayCoder zwei solcher Verbindungen. Der *GrayCoder mit FIFO-Interface* verfügt statt der üblichen Datenregister zum Schreiben bzw. Lesen über zwei entsprechende FIFOs. Dem Schreib-FIFO kann der Prozessor ständig Daten übergeben, solange das FIFO nicht voll ist. Aus dem Lese-FIFO kann er kontinuierlich die Ergebnisse abholen, solange

noch welche vorliegen, d.h. das FIFO nicht leer ist. Die Pufferung kann je nach Applikation die Synchronisation des Prozessors mit der Hardware vereinfachen.

Der zweite Teil des Projektes untersuchte die Treiberentwicklung für solche Hardwaremodule für ein Embedded Linux.

Die Anwendung unter Linux greift auf die Hardware über sog. System Calls zu. Hardwaremodule behandelt Linux ähnlich wie Dateien, weshalb auch die gleichen System Calls verwendet werden. Zu diesen Aufrufen gehören

- *open* – Öffnen des Moduls (Anmeldung der Applikation beim Kernel)
- *close* – Schließen des Moduls (Abmeldung), keine Zugriffe mehr möglich
- *read* – Lesezugriff auf die Hardware des Moduls
- *write* – Schreibzugriff auf die Hardware des Moduls
- *ioctl* – spezieller Zugriff auf Funktionen der Hardware

und weitere wie *select*, *fcntl*, *lseek* etc., die aber für die betrachteten Anwendungsfälle nicht benötigt werden. Diese System Calls werden auf Funktionen im Treiber des Moduls umgesetzt und müssen deshalb dort entsprechend implementiert sein.

Um den Treiber im Linux-Kernel zu integrieren, müssen weitere Funktionen vorhanden sein. Entweder wird der Treiber schon mit in den Kernel kompiliert (Built-in-Treiber) oder er wird dynamisch geladen (Modultreiber). In beiden Fällen sind die zwei Funktionen zum Initialisieren bzw. zum Deinitialisieren notwendig, deren Namen dem System mit den beiden Makros *module_init(Name_der_Initialisierungsfunktion)* und *module_exit(Name_der_Funktion_zum Aufraeumen)* im Treiber mitgeteilt werden. (Mit den Schlüsselworten *__init* bzw. *__exit* kann Linux die Ressourcenhandhabung für den Treiber optimieren.) In der Initialisierung muss die Funktion den Treiber beim Kernel registrieren sowie beispielsweise den Speicherplatz des Moduls anfordern.

Die Funktionsnamen zum Öffnen, Schließen, Lesen und Schreiben der Hardware erfährt Linux über eine Struktur *file_operations* im Treiber, deren Name bei der Initialisierung einer Struktur vom Typ *static struct cdev* übergeben wird:

```
struct file_operations Name = {
.owner      = THIS_MODULE,
.open       =
      Name_der_Open_Function,
.release =
      Name_der_Close_Function,
.write      =
      Name_der_Write_Function,
.read       =
      Name_der_Read_Function,
.ioctl      =
      Name_der_Ioctl_Function,
...
};
```

Daneben gibt es noch Makros für die sog. Metainformationen, also z.B. MODULE_LICENSE(“GPL“), was das Lizenzmodell des Treibers kennzeichnet, oder MODULE_AUTHOR() und MODULE_DESCRIPTION(). Im Detail sind noch viele weitere Aspekte zu beachten, um einen funktionsfähigen Treiber für ein Hardwaremodul zu erhalten.

Für alle o.g. Module entstanden passende Treiber und konnten so im Linux-Kernel eingebunden und von Applikationsprogrammseite benutzt werden.

In einem weiteren Schritt erhielten die Hardwaremodule Interruptfähigkeit, d.h. sie können über Prozessorinterruptsignale dem Programm mitteilen, dass z.B. Ergebnisse vorhanden sind oder sich sonstige Änderungen ergeben haben, und so den normalen Programmfluss unterbrechen. Hardwareseitig bekam der Prozessor noch einen Interrupt Controller, der die Verwendung von mehreren Unterbrechungsanforderungen und ihr entsprechendes Handling ermöglicht. In den Hardwaremodulen erzeugt z.B. ein neu berechneter Gray-Code-Wert ein Signal, das als Interruptsignal mit dem Interrupt Controller verbunden wird. Das Kontrollregister der Module verfügt jetzt über zwei zusätzliche Bits. Das eine erlaubt die Interrupterzeugung bzw. unterbindet sie, das andere zeigt einen ausgelösten Interrupt an, damit die Interrupt Service Routine (ISR) auf der Softwareseite überprüfen kann, ob das Modul einen Interrupt erzeugt hat. Im Kontrollregister kann das Interruptbit auch wieder zurückgesetzt werden, damit ein erneut auftretender Interrupt erkannt werden kann.

Der Treiber unter Linux muss nun auch eine Interrupt Service Routine enthalten. Diese erhält die Interruptnummer und die Gerätekennung *(*dev_id)*, die der Hardware bei der Initialisierung zugewiesen wird (*request_irq()*). Für den Fall dass das Linux-System Interrupt Sharing einsetzt, schaut die ISR im Statusregister der Hardware nach, ob sie einen Interrupt verursacht hat. Falls nicht, hat eine andere Hardware, die an dieser Interruptnummer registriert ist einen ausgelöst. Die ISR gibt in diesem Fall *IRQ_NONE* zurück, damit Linux die nächste ISR aufrufen kann, die diese Interruptnummer verwendet. Wenn wirklich die zugehörige Hardware den Interrupt signalisiert, erfolgt jetzt die eigentliche Verarbeitung bzw. Reaktion. Prinzipiell sollte die ISR so kurz wie möglich gehalten werden, da während ihrer Laufzeit andere Interrupts gesperrt sind. Die minimale Aktion, die sie durchführen muss, ist die Quittierung (das Zurücksetzen) des Interrupts in der Hardware und das Setzen eines Flags, um der Anwendung mitzuteilen, dass sich etwas getan hat. Der Rückgabewert der ISR ist in diesem Fall *IRQ_HANDLED*.

Wenn weitere, etwas umfangreichere Aktionen notwendig sind, empfiehlt es sich, für die folgende Verarbeitung eine andere Routine aufzurufen. Hier gibt es zwei grundsätzliche Arten: die im Interruptkontext laufenden Soft-IRQs und die im Kernelkontext („gescheduled“) laufenden Kernel-Threads. Erstere unterscheiden noch einmal in Soft-IRQs im engeren Sinn, in Tasklets und Timer. Die Kernel-Threads kennen die Kernel-Threads im engeren Sinn, die Workqueue und die Event-Workqueue. Bei den im Projekt zum Einsatz kommenden Hardwaremodulen bieten sich – wenn überhaupt nötig – Kernel-Threads an. (Diese Routine wird ebenfalls bei der Treiberinitialisierung mit *request_threaded_irq()* im Kernel angemeldet.) Die eigentliche ISR ruft einen solchen mit dem Rückgabewert *IRQ_WAKE_THREAD* auf. Der Scheduler startet den Thread entsprechend seiner Priorität und dieser beendet seine Arbeit dann mit dem Rückgabewert *IRQ_HANDLED*. Bei der Treiberdeinitialisierung (*release* bzw. *close*) muss der Interrupt mit *free_irq()* wieder freigegeben werden.

Im Rahmen des Projekts entstanden also wie geplant eine Reihe unterschiedlicher FPGA-basierter Hardwaremodule mit zum Teil verschiedenen Interfaces zum Prozessor. Über die damit ermöglichte Ansprache auf Applikationsprogrammebene hinaus wurde untersucht, wie die spezifischen Treiber für diese Module für ein Embedded Linux-System zu gestalten sind. Für sämtliche Hardwaremodule wurden die notwendigen Funktionen in einem passenden Device Driver realisiert und getestet. In einem letzten Schritt erhielten alle Module Interruptfunktionalität, die auch auf Treiberebene implementiert wurde. Die dabei (und in den vorangegangenen Projektteilen) gewonnenen Erkenntnisse ermöglichen es, auch eigene, angepasste Hardware in Embedded Linux-Systemen einzusetzen und so die Vorteile von FPGAs auf der Hardwareseite und den Nutzen eines flexiblen und leistungsfä-

higen Betriebssystems wie Linux speziell in prototypischen Applikationen verbinden zu können. Gegen Ende des Projekts konnten die Erfahrungen bereits in einem kleinen Drittmittelprojekt eingesetzt werden. Nebenbei flossen Ergebnisse auch in die Neugestaltung des Labors Design Digitaler Systeme ein und kommen so den Studierenden in Form aktueller Entwurfsmethodiken zugute.

**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