



# GPIF enträtselt

Das GPIF (General Purpose Interface = Vielzweck-Schnittstelle) ist das Interface nach draußen vom EZUSB FX2 ("Easy USB Effects 2" = Einfaches USB, Effekte 2). Das Handbuch belässt manches im unklaren, und deshalb hier einige erklärende Beispiele.

Also eher "Unverständliche USB-Effekte, in hoher Geschwindigkeit".

## 1. Verständnisprobleme

### 1.1 Besitz von FIFOs

Falsch, nicht Besitz ganzer FIFOs, sondern von einzelnen FIFO-Puffern. Ein Puffer der Quantum-FIFO kann nur im Besitz genau einer Instanz sein:

- der 8051-ZVE
- dem USB-Interface (SIE)
- dem GPIF (oder dem externen FIFO-Master im Slave-FIFO-Betrieb)  
Weil GPIF-Betrieb und Slave-FIFO-Betrieb aus Sicht der FIFOs völlig gleichwertig sind, wird im folgenden nur „GPIF“ zitiert, wenn auch „Slave-FIFO“ gedacht werden kann.

Im folgenden Beispiel gehe ich von OUT-FIFOs aus, d.h. Datentransport vom PC zur Peripherie:

- Am USB-Interface wird ein FIFO-Puffer gefüllt und geht zum GPIF
- Am GPIF wird der FIFO-Puffer geleert und geht leer zum USB zurück

Daraus ergeben sich folgende Konsequenzen:

- Es gibt keinen unmittelbaren Datenstrom vom USB zum GPIF (der Begriff FIFO ist also eher irreführend! Vielleicht ist „Paketdienst“ günstiger. Mit der FIFO-Eigenschaft, dass zuerst abgesendete Pakete auch zuerst ankommen, die Paket-Reihenfolge stets gewahrt bleibt.)
- Doppel-Pufferung ist das Mindeste für sinnvolle (lückenlose) Datenübertragung (deshalb gibt es keine Einfachpufferung)
- Die Implementierung seitens Cypress war so sicherlich einfacher und mit höherer Geschwindigkeit ausführbar (weil ein Puffer so nur an eine Taktquelle angeschlossen werden muss und nicht an zwei asynchronen).

Mit dem Rücksetzen einer FIFO werden die Puffer *nicht* so richtig für leer erklärt, sondern es passiert folgendes:

- OUT-Puffer sind „gefüllt“(!) und im Besitz der 8051-ZVE. Da muss jeder Puffer einzeln zum USB-Besitz überführt werden.
- IN-Puffer sind *leer* und im Besitz des GPIF. So ist alles startklar.

### 1.2 Voll und Leer bei Quantum-FIFOs

Es gibt eine wirklich einfache Festlegung:

- Die gesamte FIFO ist *leer*, wenn alle (zwei bis vier) Puffer leer sind.
- Die gesamte FIFO ist *voll*, wenn alle (zwei bis vier) Puffer gefüllt sind.

Logisch - aber was heißt hier „gefüllt“?

Ein Puffer gilt als „gefüllt“, wenn:

- der Puffer voll ist? (Nein, denkste! Siehe unten!)
- der Puffer *abgeschlossen* wird, egal wie voll. (Auch mit Null Bytes kann ein Puffer abgeschlossen, und damit „voll“ sein. Dazu später.)

Der Abschluss erfolgt wie folgt:

- Auf der USB-Seite *nur* über die Paket-Länge am USB-Paket-Ende (mit Konsequenzen, s.u.)
- Auf der Slave-FIFO-Seite:
  - bei Erreichen eines bestimmten Füllstandes (vorgebar von 8051-ZVE)
  - über eine gesonderte Leitung: PKTEND
  - durch ein Signal der 8051-ZVE

Daraus folgt, dass USB-Puffer immer nur mit Paket-Länge gefüllt werden. Die 512-Byte-Puffer werden also bei USB 1.1 mit einer maximalen Bulk-Länge von 64 Bytes nur schlecht ausgenutzt. Zu lange USB-Pakete werden nicht automatisch auf mehrere Puffer verteilt!

Was die SIE, das serielle USB-Interface, in diesem Falle macht, ist – raten Sie mal! – undokumentiert! Vermutlich wird NAK zurückgeschickt [svw. „Lieber Host, ich habe keinen Platz, versuche es im nächsten Mikroframe noch einmal.“], oder auch keine Antwort [svw. „Lieber Host, ich kann dich nicht hören, versuche es im nächsten Mikroframe noch einmal, aber höchstens dreimal, dann melde Deinem Anwenderprogramm, dass es schief ging.“]. Oder aber, die SIE platziert die Daten irgendwie über Puffer und 8051-XRAM hinweg, und schießt nebenbei die Firmware ab...

Also muss der Puffer stets größer/gleich der maximalen USB-Länge gewählt werden.

Puffer werden stets *abgeschlossen* und gelten dann als *gefüllt*.

**Wichtig (Faustregel):** Leere Puffer sind normalerweise niemals in 8051-ZVE-Besitz, sondern rutschen automatisch zur entsprechenden „Tankstelle“:

- OUT-Puffer rutschen selbständig von (ZVE oder) GPIF zum USB
- IN-Puffer rutschen selbständig von (ZVE oder) USB zum GPIF

Aber es gibt Eingriffsmöglichkeiten.

**Entladestation:** Die ZVE kann in das (Betanken und) Leeren der Puffer eingreifen:

- OUT-Puffer:
  - generieren (und zum GPIF ausgeben)
  - modifizieren (auch die Länge) und zum GPIF weiterreichen
  - verarbeiten (und danach zurück zum USB)
  - verschlucken (sofort zurück zum USB)
- IN-Puffer:
  - generieren (und zum USB geben)

- modifizieren (auch die Länge) und zum USB weiterreichen
- verarbeiten (und danach zurück zum GPIF)
- verschlucken (sofort zurück zum GPIF)

„Verarbeiten“ und „Verschlucken“ sind praktisch gleiche Vorgänge; ob die Daten angefasst werden oder nicht ist für den Puffer ohne Belang. Nebenbei, der Zugriff auf die Puffer durch die drei Instanzen ist grundweg verschieden:

Instanz	Zugriff
ZVE	<i>wahlfrei</i> (adressierbar)
USB	<i>en bloc</i> (USB ist zwar seriell, aber ohne Belang! U.a. wegen CRC)
GPIF	<i>sequenziell</i> (Byte für Byte oder Wort für Wort)

Tabelle 1: Zuordnungen

### Die Sichtweise von GPIF auf Ausgabe-Puffer:

- Die FIFO-Puffer sind vom Fall-Through-Typ (sozusagen Durchfall-FIFOs). Das bedeutet, dass am Ausgang eines nicht-leeren Puffers das erste Byte/Wort „herausschaut“, ohne Taktung. Ist der Puffer leer, rutscht automatisch der nächste Puffer nach. Sind schließlich alle Puffer leer, ist der Ausgang ungültig.  
Das Gegenteil sind sog. *non-fall-through-FIFOs*. Sie benötigen einen Lese-Takt für das erste Byte/Wort.
- Der Puffer-Wechsel scheint „unendlich schnell“ vonstatten zu gehen.
- Im Sinne der Quantum-FIFOs gibt es keine leeren OUT-Puffer für das GPIF, da solche umgehend zur USB-Domäne zurückgehen sollten (aber wo steht das?).

**Null-Byte-Pakete:** Sie sind lästig und sollten vermieden werden. Ein Windows-Host weigert sich, Null-Byte-Bulk-Pakete zu senden; bei Isochron geht es womöglich doch.

**Ungerade-Bytezahl-Pakete:** Was ein 16-bit-GPIF in dieser Situation macht ist noch völlig ungeklärt und nirgends dokumentiert! Vermutlich wird in diesem Fall ein ungültiges High-Byte ausgegeben. Also: vermeiden!!

Bei IN-Paketen kann dies nicht passieren; außer wenn man den automatischen Puffer-Abschluss auf einen ungeraden Füllstand gelegt hat. (Was das GPIF dann macht, ist undokumentiert!!)

## 1.3 Interrupt-Spagetti

Was Cypress vorführt, ist alles andere als lehrreich. Bedenken Sie folgenden Satz, der für die Mikrocontrollerprogrammierung fundamental ist:

Wenn ein Problem ohne Interrupts lösbar ist, dann ist das bereits ein Grund dagegen.

Sie können Interruptanforderungen (IRQs) sehr einfach in einer Abfrageschleife abtesten und darauf reagieren: Ihr Programm wird nicht nur schneller, es entfallen alle Race Conditions, und es ist einfach debugbar.

In der Hauptschleife der 8051-Firmware werten Sie einfach den Inhalt von USBIRQ (für Endpoint EP0) sowie die OUTxCS / INxCS für die übrigen Endpoints aus und verzweigen entsprechend. Nicht anders machen es die Cypress-Beispiele, nur umständlicher.

Sieht so aus als hätte irgendein Chef bei Cypress befohlen, dass ihre Beispiele unbedingt den ach-so-

tollen vektorisierten USB-Interrupt verwenden müssen.

## 1.4 Mit oder ohne Bytezähler?

Hm...

## 1.5 Mehrere Ein/Ausgabeströme und das AUTOOUT/AUTOIN-Feature

...sind eigentlich zwei verschiedene Schuhe!

**AutoOut** sorgt dafür, dass vom USB gefüllte Puffer automatisch zum GPIF-Besitz übergehen. (Sie werden zu Slave FIFOs.) FIFO-Statusleitungen (im Slave-FIFO-Betrieb also externe Leitungen) verändern entsprechend dem Füllstand ihren Zustand.

Das GPIF bleibt jedoch weiterhin „ausgeschaltet“! *Ich habe noch keine Funktion hinbekommen, s.u.!*

**AutoIn** sorgt dafür, dass vom GPIF (oder Slave FIFO) gefüllte und *terminierte(!)* Puffer zum USB-Besitz übergehen. Dabei darf der Füllstand eines Puffers nicht die maximale USB-Paketgröße übersteigen; also für Bulk-Pipes im Full-Speed-Modus maximal 64 Bytes!

Was die SIE in diesem Falle tut, ist – es wird langsam langweilig – undokumentiert! Wahrscheinlich ist, sie sendet auf Gedeih und Verderb alles, und erzeugt USB-Babble, lässt auf PC-Seite Speicher überlaufen (je nach Host-Controller), lässt Windows abstürzen, wer weiß? Die SIE weiß ja nichts von Maximallängen!

Wenn mehrere FIFO-Datenströme über das GPIF laufen sollen (bspw. ein OUT- und ein IN-Strom), muss der Programmierer das GPIF bei entsprechendem Bedarf „anstoßen“ (am einfachsten durch Schreiben auf GpifTrig @BB) - es gibt hierfür *keinen* Automatismus!! Und die GPIF-Wellenform muss so gestaltet werden, dass sie auch umgehend endet:

- Entweder muss das Voll- oder Leer-Bit als Endemarkierung aktiviert werden (durch Setzen von Fifo[2468]Flag in EP[2468]GpifPfStop @E6D3); *dann kann kein Ready-Signal zur Beendigung verwendet werden(!), nur zum Drosseln des Datenstroms - dies kann Deadlocks bewirken,*
- oder es muss vor jedem „Anschubsen“ des GPIF der Transaktionszähler GpifTcB0 @E6D1 auf „1“ gesetzt werden - die Wellenform sollte selbsttätig das Leer- bzw. Voll-Flag abtesten.

Wann das GPIF angestoßen werden muss, lässt sich am leichtesten in einer engen Abfrageschleife abtesten:

- Ausgabe-Datenströme über das Abtesten des Leer-Flags in EP(24|68)FifoFlgs @AA
- Eingabe-Datenströme über das Abtesten des Voll-Flags in EP(24|68)FifoFlgs @AA und/oder einer externen Meldeleitung  
(Via AutoIn rutschen die gefüllten Puffer automatisch zum USB-Besitz.)

**Achtung:** Bei Eingabe-Datenströmen muss ein Mechanismus eingebaut werden, dass halb gefüllte (nicht terminierte) Puffer auf der Slave-FIFO-Seite firmwaremäßig terminiert werden, damit diese vom Rechner gelesen werden können! Gut geeignet hierfür ist die NAK-Interruptanforderung (NakIrq @E65B, InPktEnd @E648), die auch ohne Interrupt zyklisch abgefragt werden kann.

## 2. Anwendung

### 2.1 Volle Ausgabegeschwindigkeit (Xilinx-FPGA konfigurieren)

Das GPIF wird zum Großteil zur Kommunikation mit [CPLD](#) oder [FPGA](#) gebraucht. Clever ist es dabei, die Busverdrahtung so zu gestalten, dass die Konfiguration großer RAM-basierter FPGA ebenfalls über dieselbe Verbindung läuft. Damit spart man sich den Konfigurations-PROM und erleichtert sich das Configware-Update, weil dazu einfach nur eine Treiberdatei ausgetauscht werden muss.

Das folgende Bild zeigt außerdem die Möglichkeit, auch die 8051-Firmware per USB laden zu lassen. Packt man [alles in eine Treiberdatei](#) (.SYS), ist ein komplettes Software-Update sehr einfach. Ein derartiges Gerät startet in mehreren Phasen:

- Anstecken: Windows lädt Treiber anhand VID und PID
- Treiber schiebt Firmware in den Mikrocontroller
- Mikrocontroller-Firmware startet, re-numeriert (nur bei Änderung der Default-Konfiguration erforderlich, Windows lädt Treiber noch einmal)
- Treiber schiebt Configware via Mikrocontroller in den FPGA und startet diese
- Normaler Betrieb

Die [Verzögerung durch den Bootprozess](#) liegt bei unter 1 s *ohne* Re-Numerierung, bei 3 s *mit* Re-Numerierung (je nachdem wie schnell Windows Treiber lädt).

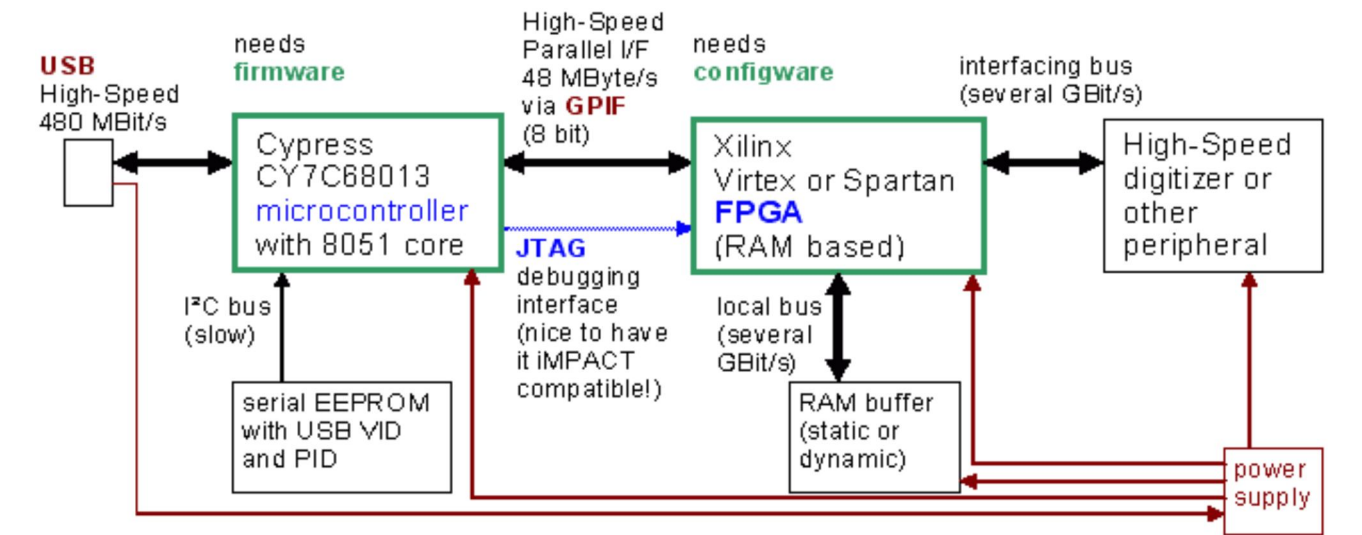


Bild 1: Die günstigste Möglichkeit, einen FPGA mit USB anzuschließen

Ein RAM-Pufferspeicher wird fast immer benötigt, wenn bei hoher USB-Datenrate Unterbrechungen im 100-ms-Bereich überbrückt werden sollen; der FPGA-interne RAM ist dafür viel zu klein oder zu teuer. Für den Mikrocontroller genügt fast immer der kleine 56-polige Gehäusetyp.

Zur Konfiguration eines Virtex-FPGA mit der Methode **Slave SelectMAP** benötigt man folgende Verbindungen, die man auch im nachhinein (also wenn FPGA im regulären Betrieb) als schnelle Übertragungsstrecke nutzen kann:

Cypress CY7C68013A↑↓	Xilinx Virtex II↑↓
IFCLK	CCLK Dedizierter Konfigurationsanschluss
PB0..7	D7..0 Bits stürzen! Wirklich! Bei Virtex ist D0 das MSB! Wird oft überlesen.
CTL0	CS_B
CTL1	RDWR_B

Cypress CY7C68013A↑↓	Xilinx Virtex II↑↓
RDY0	BUSY Wird nur bei komprimierten oder verschlüsselten Konfigurationsdaten gebraucht. Beim Virtex4 ausschließlich zum Rücklesen
PA2 oder irgendein Ausgabepin	PROG_B Dedizierter Konfigurationsanschluss
PA3 oder irgendein Ein/Ausgabepin	DONE Dedizierter Konfigurationsanschluss
PA4 oder irgendein Eingabepin	INIT_B Nicht unbedingt nötig, siehe Text

Mit DONE als Ein/Ausgabe(!) kann man einen definierten Start des FPGA erreichen und benötigt so keine Design-Resetleitung o.ä. Ansonsten könnten Konfigurationsdaten als Eingabedaten für den bereits konfigurierten FPGA fehlinterpretiert werden.

**Hintergrund:** Der Konfigurationsdatenstrom \*.RBT, \*.BIT enthält am Ende einige überflüssige Bits bzw. Bytes, um den Startup auszulösen. Diese könnten, wenn der FPGA das DONE-Pin freigibt, »in den falschen Hals« gelangen.

INIT\_B wird nur benötigt, um:

- das **FPGA-Löschen** definiert **abzuwarten**,
- einen **CRC-Fehler** beim Konfigurieren zu **detektieren**.

Als eine mögliche Handshakeleitung (im konfigurierten Fall) ist INIT\_B auch ein Kandidat für eine RDY<sub>x</sub>-Leitung.

*GPiF-Designer:*

Für „volles Rohr“ benötigt man im GPiF-Designer tatsächlich drei Zustände:

```
// GPiF Waveform 1: FIFOWr
//
// Interval      0          1          2          ...      Idle (7)
//
// AddrMode Same Val  Same Val  Same Val
// DataMode  Activate Activate  NO Data
// NextData  SameData NextData  NextData
// Int Trig   No Int   No Int   No Int
// IF/Wait    IF       IF       IF
//   Term A  EF+1     EF+1     (egal)
//   LFunc   AND      AND      AND
//   Term B  EF+1     EF+1     (egal)
// Branch1   Then 2   Then 2   ThenIdle
// Branch0   Else 1   Else 1   ElseIdle
// Re-Exec    No      Yes      No
// Sngl/CRC   Default Default Default
// CS_B       0       0       1
// RDWR_B     0       0       1
```

Die Einstellung „NextData“ führt zu *Beginn* des Zustandes zum Lese-Takt und damit zum Datenwechsel. Mit geringer GPiF-Taktverzögerung, sofern der FX2 die Taktquelle ist.

*C-Programm:*

Wichtig ist, dass man das Empty-Flag 1 Byte früher braucht! ■ Zeilennummern ■ Umbruch

```
EPxGpifFlgSel=1          // Anhalten bei Empty-Flag, nicht bei TerminalCount
EPxFifoCfg=0x20          // Handarbeit, 8 bit, OEPl (Empty-Flag 1 Byte früher)
```

Man braucht kein **SyncDelay** bei 48 MHz, weil nur 3 Takte Wartezeit erforderlich sind und der C-Compiler genügend (unsinnige) Befehle einstreut.

**Wichtig (Die verflixte Null):** Das GPiF darf nicht aktiviert werden, wenn Null Bytes in den



Ausgabepuffern stehen! Die Zustandsmaschine würde ein ungültiges Byte zum FPGA schaffen.

Bei nur einem Byte läuft die Zustandsmaschine direkt von Zustand 0 (= Fall-Through-Byte zum FPGA schicken) zum Zustand 2 (= FIFO leer lesen, aber FPGA laden beenden [CS\_B=1]). Notfalls kann man sich darauf verlassen, dass der (Windows-)Host kein Null-Byte-Paket schickt.

Ob Bytes im Puffer vorhanden sind, kann nicht mittels EPxCS, EPxFifoFlags oder EPxFifoFlgs abgefragt werden, sobald das EF+1-Feature aktiviert ist. Vermutlich ist EPxFifoBc zu benutzen.

Eine in dieser Hinsicht „eigensichere“ Zustandsmaschine zu konstruieren ist *nicht* möglich, wenn man das EF+1-Feature benutzt.

Im Experiment war es egal, ob IFCLK (Interface-Takt) negiert oder nicht-negiert betrieben wurde (Virtex-II, 48 MHz).

Wie man sich vorstellen kann, sind auch größere FPGAs mit einem Fingerschnipp konfiguriert. Man kann von einer Datenrate von 30 MByte/s (halbe High-Speed-USB-Bandbreite) ausgehen. Serielle Schneckentempo-Konfiguration war gestern.

### 3. Weitergehende Überlegungen

Eine Auslegung der Übertragungsstrecke auf 16 bit (für den regulären FPGA-Betrieb) für mehr Durchsatz erscheint unnötig und als Pinverschwendung, weil die Puffer des FX2 recht klein sind und USB den Flaschenhals darstellt.

Der Verzicht auf den FX2 und Implementation des USB auf dem FPGA selbst bedeutet FPGA-Ressourcenverbrauch und kommt in der Praxis teurer als der zusätzliche Chip. Vom Entwicklungsaufwand ganz zu schweigen. Und den Konfigurations-Flash braucht man in diesem Falle ja auch noch.

Wäre das Xilinx-USB-JTAG-Protokoll (iMPACT) dokumentiert, könnte der FX2 gleich noch die Rolle des „USB Cable“ zum Debuggen übernehmen! Wäre schön, aber das hat noch niemand geknackt...

Genug geträumt! Zurück zur Konfiguration!

#### 3.1 Gedrosselte Ausgabegeschwindigkeit

Die vorhergehende Wellenform wertet BUSY nicht aus. Damit funktioniert sie nicht für komprimierte oder verschlüsselte Bitströme. (Betrifft eigentlich nur Virtex-II. Virtex4 nicht! Der geht immer! Spartan hat eh' keine Verschlüsselung.)

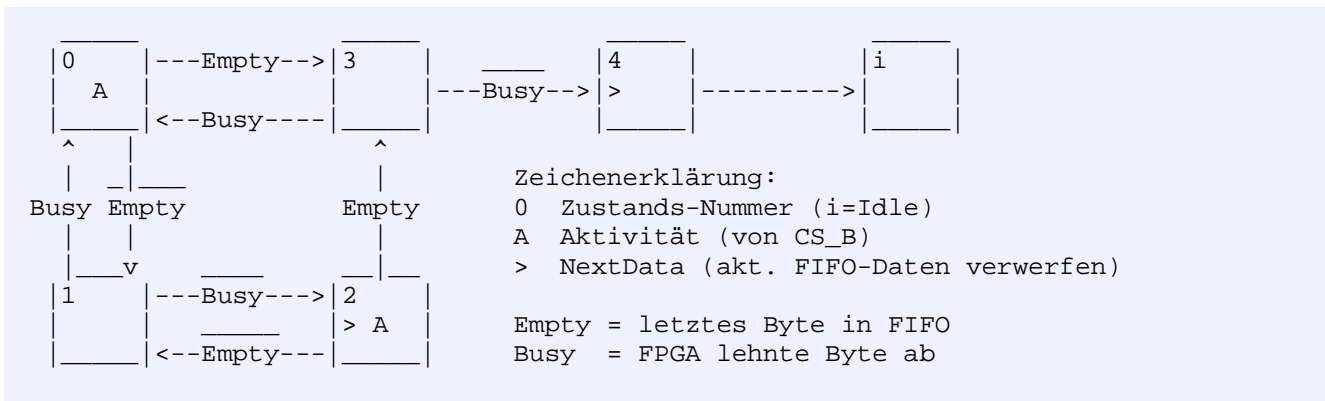
- Man kommt nicht umhin, zwei Takte pro Byte zu verbraten, weil die GPIF-RDYx-Signale durch mindestens ein Flipflop geführt werden und für sofortige Auswertung nicht zur Verfügung stehen.
- Die Konfigurationstaktfrequenz CCLK muss auf 30 MHz abgesenkt werden, um die Vorhaltezeit für BUSY sicherzustellen.
- Flowstate hilft nicht, ebenfalls wegen der BUSY-Verzögerung.

Am Zustandsgrafen habe ich wirklich Stunden zugebracht, und so sieht er aus:

```
// GPIF Waveform 1: FIFOWr
//
// Interval      0          1          2          3          4      Idle (7)
//
// _____
```

```
// AddrMode Same Val Same Val Same Val Same Val Same Val
// DataMode Activate Activate Activate Activate NO Data
// NextData SameData SameData NextData SameData NextData
// Int Trig No Int No Int No Int No Int No Int
// IF/Wait IF IF IF IF IF
// Term A EF+1 BUSY EF+1 BUSY (egal)
// LFunc AND AND AND AND AND
// Term B EF+1 BUSY EF+1 BUSY (egal)
// Branch1 Then 3 Then 0 Then 3 Then 0 ThenIdle
// Branch0 Else 1 Else 2 Else 1 Else 4 ElseIdle
// Re-Exec No No No No No
// Sngl/CRC Default Default Default Default Default
// CS_B 0 1 0 1 1
// RDWR_B 0 0 0 0 0
```

Die Zustandsmaschine als Automatengraf:



C-Programm:

Auch hier wird das Empty-Flag 1 Byte früher gebraucht. Noch wichtiger ist es, die GPIF-Auswertung auf **synchron** zu stellen, damit nur ein Flipflop wirksam wird, nicht zwei.

Es ist ja auch synchron! Es liegt nur eine Takt-Domäne vor.

Auch mit Drosselung sind größere FPGAs mit einem Fingerschnipp konfiguriert. Die theoretische Datenrate von 15 MByte/s (halbe IFCLK-Taktfrequenz) wird wegen diverser Windows-Aussetzer nicht erreicht. Trotzdem: affenschnell.

## 3.2 Volle Lesegeschwindigkeit

Zu schreiben

## 4. Dokumentationsfehler und Bugs

- Das SAS-Bit im Register GPIFREADYCFG muss 1 sein für **synchron**, nicht 0. Fehler auf „Technical Reference Manual“, Seiten 15-100 und 15-101.

```
EPxGpifFlgSel=1; // Anhalten bei Empty-Flag, nicht bei TerminalCount
EPxFifoCfg|=0x20; // Handarbeit, 8 bit, OEP1 (Empty-Flag 1 Byte früher)
GpifReadyCfg|=0x40; // SAS=1, synchron, ein RDYx-Flipflop
```

- Böser Bug:** Eine Reihe Register im XRAM-Bereich (E6xx - nicht alle) sind *nicht* via XAutoDat[12] (Autopointer) ansprechbar! Man muss wohl via DPTR gehen.
- Möglicherweise wichtig:** Niemals darf man Null-Byte-Puffer an ein GPIF schicken, welches mit OEP1 {siehe EP[2468]FifoCfg (@E618)} arbeitet! Deshalb sollte die ZVE die Puffer vor dem Durchreichen daraufhin abprüfen!
- Die Register EP[2468]CS (@E6A3), EP[2468]FifoFlags (@E6A7) und EP(24|68)FifoFlgs (@AB)




widerspiegeln offenbar das Empty- und Full-Flag entsprechend der Einstellung in EP[2468]FifoCfg (@E618). Bspw. ist „empty“ gesetzt, wenn noch/nur ein Byte in der FIFO steckt.

### Lösung:

- INFM1 bzw. OEP1 in EP[2468]FifoCfg (@E618) nicht verwenden, statt dessen das Programmable Flag verwenden,
- EP[2468]FifoBc[HL] abfragen für Füllstand
- **Bin ich blöd?** Das **AutoOut**-Feature in EP[2468]FifoCfg (@E618) habe ich niemals zum Laufen bekommen; immer kommt Murks heraus. **AutoIn** hingegen funktioniert prima.
- **Wann wird abgetastet?** Bei Dateneingabe via GPIF werden die Daten am **Anfang** des „active“-Zustandes abgetastet. Ready-Signale werden je nach Stellung des SAS-Bits (GpifReadyCfg) quasi am **Anfang** des Entscheidungs-Zustandes abgetastet („synchron“), oder sogar **einen Zustand eher** („asynchron“).

## 5. Siehe auch

- [FX2-Referenzdesign als Eigenbau](#) mit CY7C68013-128 in Eagle4
- [Ultraschall-Recorder 1 MSa/s 16 bit 2 Kanäle](#)
- [USB in a nutshell](#)
-  [USB Made Simple](#)
- [Fragen und Antworten rund um USB](#)

[Vorhergehener Artikel \(DownloadFirmware.htm\)](#) [Nächster Artikel \(Hitachi-Code.htm\)](#) [Übergeordnetes Verzeichnis \(Mikrocontroller\)](#)

✉ [Henrik Haftmann](#), erstellt: 10. Juni 2011 — letzte Änderung: 23. Oktober 2020

