2004－EVA－8　(14)
2004／3／19

USB制御命令を持つ専用プロセッサの設計とUSBデバイスへの実装による評価

名野響 †　　大山将城 ††　　清水尚彦 †††
† 東海大学大学院工学研究科
†† 東海大学工学部通信工学科
††† 東海大学電子情報学部コミュニケーション工学科

あらまし　一般的な USB デバイスコントローラはステートマシンであり，外部にデバイス制御用コントローラが必要となる．そこで我々は USB プロトコルとデバイスの両方を制御する USB デバイスコントローラを設計した．このコントローラは内部に MPU を持ち，MPU が USB リクエストをハッシュ関数を用いて高速に判別する事で，デバイス制御に多くの時間を割り当てる事が出来る．本稿はハッシュ関数を用いたリクエスト判別を行う USB デバイスコントローラの設計とその性能を評価する．

キーワード　**USB，USB デバイス，デバイスコントローラ，ハッシュ関数**

Development of Dedicated USB Prosessor Instructions for USB Protocol and Estimation of Implementing to a USB Device Controller

Hibiki Nano†　　Masashiro Ohyama††　　Naohiko Shimizu†††
†Graduate School of Engr. Tokai Univ.
††Communication Eng. Dept., School of Engineering , Tokai Univ.
†††Dept. Comm. Engr., School of Information Technology and Electronics. Tokai Univ.

Abstract　Many of USB Device Contorllers are composed with state machine whose control function can't be modified, therefore another controller is required to control the device itself. Our USB Device Controller incorporates a original MPU which has dedicated instructions discriminate USB requests using a hash function, and it can control both USB protocols and devices connected to the Device Controller. In this paper we evaluate the original instructions with FPGA based our device.

Key words　**USB, USB Device, Device Controller , Hash Function**

1　はじめに

市販されている USB デバイスコントローラの多くはステートマシンによって USB プロトコルのみを制御し，USB ホストから受信したデータを整形しデバイス側へ供給する．このようなデバイスコントローラは外部に汎用プロセッサ等のデバイス制御用コントローラが必要である (図 1(a))．

そこで我々は USB プロトコル制御とデバイス制御の両方を行う 1 チップ USB デバイスコントローラを設計し FPGA に実装した．このデバイスコントローラは内部にプロセッサを内蔵し，そのファームェアによって USB プロトコルとデバイス両方の制御が可能である (図 1(b))．

USB ホストはロースピード (LS：1.5Mbps) 又はフルスピード (FS：12Mbps) のバス速度において，1ms のフレームという単位を形成しトランザクション単位でスケジューリングを行う (USB2.0 仕様で追加されたハイスピード (HS：480Mbps) は 125μs のマイクロフレームを使用する)．また，リクエストタイムアウトが存在し，リクエスト発行後デバイスからの応答が 16 ビット時間 (HS：192 ビット時間) 無ければ，USB ホストがそのデバイスを Disable にする．そのため，USB デバイスはフレーム (またはマイクロフレーム) 内で USB プロトコルとデバ

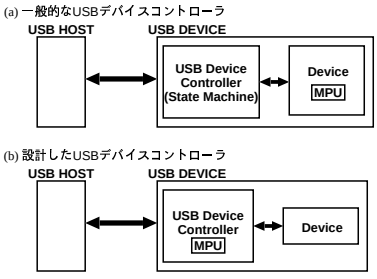

図 1: 一般的な USB デバイスコントローラとの違い

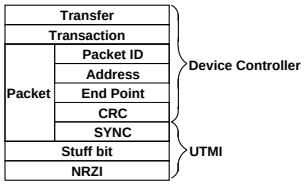

図 2: USB プロトコル

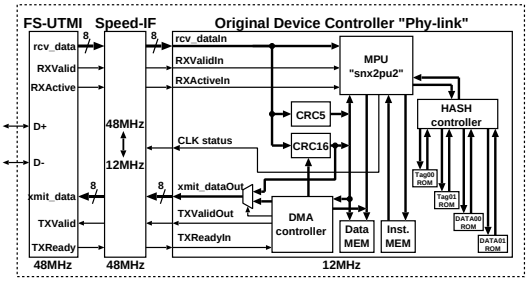

図 3: USB デバイスコントローラ ‘Phy-link’ のブロック図

イスを制御し，タイムアウトを回避して応答しなければならない．

USB プロトコルとデバイスの両方を制御するデバイスコントローラは，USB プロトコル制御に時間がかかるとデバイス制御を行う時間が短くなる問題がある．そこで我々は，USB プロトコル制御に特化した命令セットをデバイスコントローラ内部の MPU に実装し，シンプルなファームウェア設計で USB プロトコル制御を行う．

本稿は各 USB プロトコル制御に特化させた USB 専用プロセッサのハードウェア設計と，それを利用したファームウェア設計を示し，昨年我々が設計した USB デバイスコントローラ ‘IYOYOYO’[4] (オリジナルインターフェースとオリジナル RISC プロセッサによる USB デバイスコントローラ) との比較を行う．また，設計したプロセッサをデバイスコントローラとして FPGA 上に実装し動作確認を行った．

2 USB プロトコル制御

USB プロトコルをバスレベル制御を下位レイヤとして示すと図 2 のようになる．

図 2 に示す通り下位レベルの制御インターフェースとして UTMI[5] を採用し，設計した USB デバイスコントローラは主に転送レベルを制御する．

設計したデバイスコントローラ ‘Phy-link’ のブロック図と特徴を図 3 表 1 に示す．

‘Phy-link’ は USB 制御に特化した命令セットを持つ MPU ‘snx2pu2’ を内蔵する．‘snx2pu2’ のブロック図を図 4 に示す．このプロセッサは 16 ビットの汎用レジスタを 4 つ持ち，3 段パイプラインである．また割り込み線を持ち UTMI のパケット受信により割り込みが発生する．プロセッサ内には汎用レジスタとは別に，デバイスアドレ

表 1: USB デバイスコントローラ ‘Phy-link’ の特徴

モジュール	機能
MPU ‘snx2pu2’	USB プロトコルの転送レベル制御とデバイス制御
HASH controller	ハッシュ関数制御用コントローラ
DMA controller	パケット送信制御用コントローラ
FS-UTMI	フルスピード用 UTMI[5] ・下位レベルの USB プロトコル制御
CRC5/CRC16	USB パケットに含まれる CRC フィールドの算出とエラーチェック
Speed-IF	‘FS-UTMI’ と ‘Phy-link’ のスピード差制御

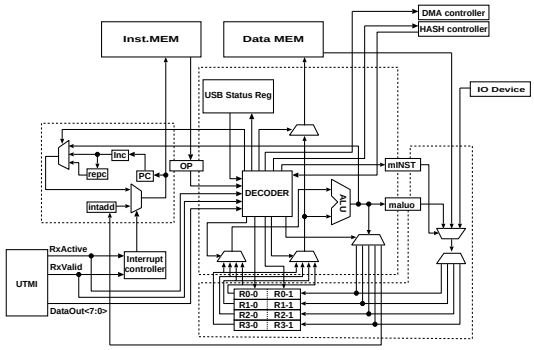

図 4: USB 専用プロセッサ ‘snx2pu2’

スや，受信したパケットの各フィールドを保持する USB ステータスレジスタを持つ．‘snx2pu2’ は ‘FS-UTMI’ を介して USB ホストと通信を行いデバイス制御と USB プロトコルを制御する．

パケットの送信は ‘DMA controller’ で行う．送信データ情報と送信リクエストを ‘DMA controller’ に発行する事で，UTMI プロトコルに従ってパケットデータを ‘FS-UTMI’ へ出力する．

‘snx2pu2’ は転送レベルの制御を行うために，USB ホストが発行するリクエストのターゲット判別とリクエストパターン判別を行う．各判別処理の概要を示す．

- リクエストターゲット判別
  ホストが送信するトークンパケット (図 5) のパケット ID フィールド (PID)，アドレスフィールド (ADDR)，エンドポイントフィールド (EP) から，リクエストのターゲットデバイスと転送タイプを判別する．また，USB プロトコルでタイムアウトが定められているため，16 ビット時間内に判別処理を終了し対応したパケットを送信しなければならない．

- リクエストパターン判別

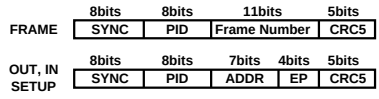

図 5: トークンパケット

表 2: デバイスリクエストとデータフィールドの関係

Request Type	bmRequestType(1Bytes)	bRequest(1Bytes)	wValue(2Bytes)	wIndex(2Bytes)	wLength(2Bytes)
GET_DESCRIPTOR	80	06	DescriptorType & DescriptorIndex	0 or LanguageID	Length
GET_DESCRIPTOR (HID)	81	06	DescriptorType & DescriptorIndex	0 or LanguageID	Length
SET_ADDRESS	00	05	DescriptorAddress	0	0
SET_CONFIGURATION	00	09	ConfigurationNo.	0	0

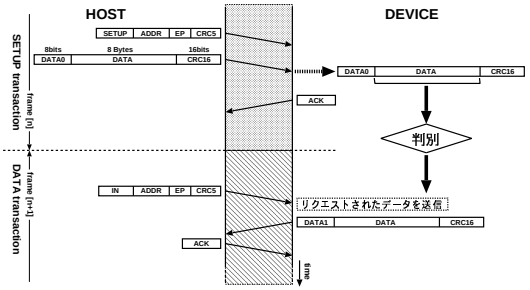

図 6: コントロール転送のリクエストパターン判別

コントロール転送において，ホストが発行するリクエストパターンはセットアップトランザクション内のデータフィールド 8 バイトで決まる (図 6)．リクエストタイプとデータフィールドの関係の一部を表 2 に示す．デバイスコントローラはこのデータフィールドからリクエストパターンを判別し，それに続くトランザクションでリクエストに応じたデータ転送を行う．そのため，このリクエストパターン判別は次のトランザクションが開始される前に終了する必要がある．

3 リクエストパターン判別処理の設計

リクエストのターゲットデバイスと転送タイプの判別にかかる時間は，タイムアウトを回避するためにトークンパケット受信完了後１６ビット時間 (LS，FS) 以内にホストにパケットを送信しなければならない．そこで各フィールドの判別処理ステップを評価し，プロセッサに必要な命令セットを設計した．

リクエストのターゲットデバイスと転送タイプの判別は表 3 に示すトークパケット内の各フィールドから行う．これらの判別は各フィールドパターンが少ないため，比較演算命令を MPU に実装し，マスクを使用した判別処理をファームェアで行う．

しかし，UTMI がバイト整形したパケットデータをデバイスコントローラにプッシュするため，図 7 のように演算しづらいデータとなってしまう．

表 3: リクエストパターン判別に使用するフィールド

フィールド	ビット数	パターン数
PID	8	4(OUT/IN/SOF/SETUP)
ADDR	7	1(ホストがセットしたアドレス (アドレスがセットされるまではデフォルトアドレス [0x00]))
EP	4	Max 15(デバイスが持つエンドポイント数)

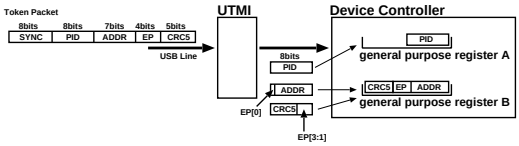

図 7: UTMI のバイトアライン

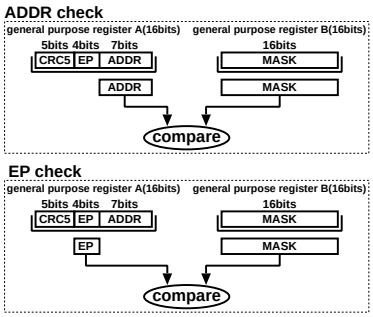

図 8: ADDR・EP フィールド判別命令

そこで比較命令を拡張し，バイト整形されたデータから両フィールドを抜き出し，比較演算する専用命令を実装した (図 8)．

4 USB リクエストパターン判別処理の設計

4.1 判別方法の問題点

USB ホストが発行するデバイスリクエストはトランザクションを複数組み合わせて行う通信であり，図 9 のように最初のトランザクション (a) でリクエストタイプ指定，次のトランザクション (b) 以降でデータ送信リクエストやトランザクション終了リクエストを行う．つまりデバイスがデータ送信を行うデバイスリクエストを受信した場合

1. 図 9(a) で受信したパケットを元に送信データの判別 (USB プロトコル制御)
2. 図 9(b) までに送信データを用意 (デバイス制御)

といった処理をしなければならない．‘1’ の制御に時間がかかると ‘2’ に与えられる時間が減ってしまう．この結果，複雑な処理が必要なデバイスを制御する時間が用意出来ないために，コントローラに接続出来るデバイスの種類が限られる可能性がある．

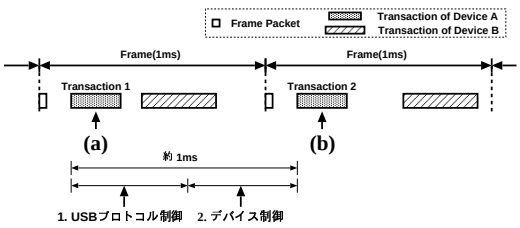

図 9: フレームとデバイスリクエスト

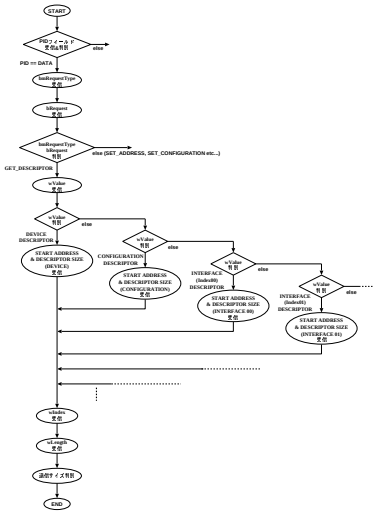

図 10: マスクと比較演算を利用したリクエストパターン判別

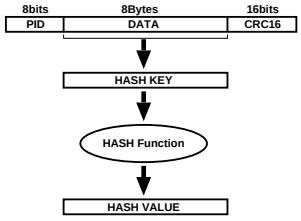

図 11: ハッシュ関数

我々が過去に設計したオリジナル USB デバイスコントローラ [4] は，この判別処理をトークンパケット判別と同様にマスクと比較演算で行った．その際のファームウェア処理を図 10 に示す．図 10 のように判別対象となるデータフィールドサイズが 8 バイトあるため，判別処理が複数回行われマスクの生成と比較演算回数が多くなってしまう事が解る．この結果 [4] はデバイスリクエスト判別に 39 ステップ必要であった．しかもファームウェアで判別しているデバイスリクエストは表 2 に示したパターンのみで，他のクラスやベンダ固有で使用するリクエストには対応していない．それらのリクエストに対応させるためにリクエストパターンを増やした場合，判別に要する命令ステップがさらに増加してしまう．

そこで今回はデバイスリクエスト判別に適したハードウェアを設計した．

4.1.1 ハッシュ関数による判別方法の提案

既述した通りデバイスリクエストはデータフィールド 8 バイトで決定する．そのため，図 11 のようにこの 8 バイトをハッシュ関数の KEY として用いれば判別が可能である．

今回は指定したデータを元にハッシュ検索を行い，リクエストを判別する専用命令 'HAS 命令' を実装した．[4] のファームウェアステップと比較するため，設計した HAS

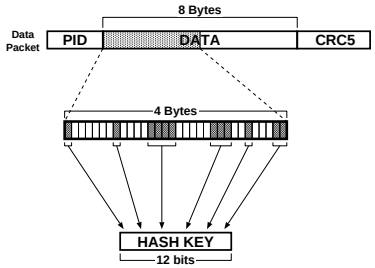

図 12: HASH KEY の生成

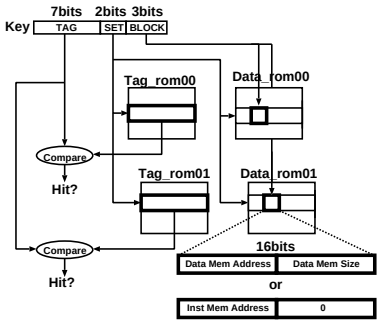

図 13: ハッシュ関数によるリクエスト判別

命令は [4] と同様に表 2 に示すリクエストに対応する．

図 12 のようにハッシュ関数の KEY は，判別対象となる各デバイスリクエストのデータフィールドパターンを比較し，判別に利用可能な 4 バイトから取り出した 12 ビットを設定した (図 12)．また，デバイスリクエストがデータ送信やステータスセットのリクエストであるため，HAS 命令を実行した結果得られる VALUE は，ホストがリクエストした送信データ情報 16 ビット (送信データをセットしてあるメモリアドレスを上位 8 ビット，データサイズを下位 8 ビット) または，リクエストされたステータスセットを行うサブルーチンアドレスとした．得られた VALUE がどちらの属性のデータかを判別するためにサブルーチンアドレスを VALUE に設定するときは下位 8 ビットに 0 をセットする (図 13)．ハッシュの構造は '2 way associative' を採用した．

ファームウェアは必要なデータフィールド 4 バイトを MPU 内部の専用レジスタにセットすると HAS 命令を実行し VALUE を取得する．そして取得した VALUE の属性を判別し，送信データ情報であれば送信データを用意，サブルーチンアドレスであればステータスセットを行う (図 14)．

4.2 HAS 命令の評価

[4] と今回設計したデバスコントローラのデバイスリクエスト判別に必要な命令ステップ数を表 4 に示す．表 4 に示すステップ数は，判別に使用するデータフィールド受信からリクエスト判別完了までにかかる最大値である．

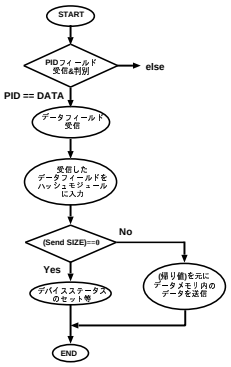

図 14: HAS 命令を利用したファームウェアのデバイスリクエスト判別

表 4: デバイスリクエスト判別にかかる命令ステップの比較

デバイスコントローラ	命令ステップ (データフィールド受信から判別終了までにかかる最大値)
IYOYOYO[4]	39 ステップ
Phy-link	14 ステップ

表 4 のように ‘Phy-link’ の命令ステップは [4] に比べ約 3 分の 1 まで削減出来ている．また [4] が，判別するリクエストパターンの増加に比例して命令ステップも増加するのに対し，ハッシュ関数を使用している ‘Phy-link’ は変化しないという利点がある (但し，ハッシュデータを保存するメモリ容量が増える事になる)．

‘Phy-link’ を実装した際のメモリ容量は表 5 となる．使用したハッシュメモリ容量は表のように 135 バイトで，命令メモリ，データメモリを合計した値の 3.2 %と小容量である．ただし，USB で使用する全てのデバイスリクエスト (標準リクエストや各クラス固有で使用するリクエスト，ベンダリクエストなど) に対応させる場合はこの容量が増える事になる．この問題はハッシュテーブルを最適化し完全ハッシュ関数を設計したり，デバイスコントローラに接続するデバイスに応じて接続するハッシュメモリを変更する事で回避可能である．

5 デバイスへの実装とシミュレーション評価

USB デバイスコントローラ ‘Phy-link’ にゲームパットインターフェースを接続し，USB ゲームパッドを設計した．図 15 のように ‘Phy-link’ にゲームパッド インターフェース内のフラグレジスタと FIFO をメモリマップし，そこから受信したゲームパッドデータを USB ホストに送信する．

表 5: ‘Phy-link’ のメモリ容量

メモリ	サイズ (バイト)
命令メモリ	2048
データメモリ	2048
ハッシュメモリ	135

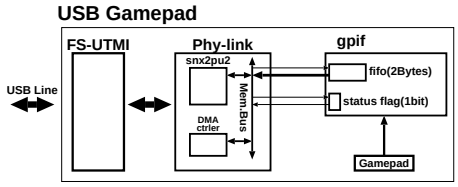

図 15: 設計した USB ゲームパッドのブロック図

表 6: USB ゲームパッドとして実装した ‘Phy-link’ と [4] の比較

	‘Phy-link’	‘IYOYOYO’[4]
Device	ALTERA APEX20KE EP20K160EFC484-3	ALTERA APEX20KE EP20K160EFC484-3
Total logic elements	2,069/6,400	1,645/6,400
ファームウェアステップ数	149	197

USB ゲームパッドとして ALTERA Quartus II で合成した結果を表 6 に示す．表 6 のようにデバイスリクエスト判別処理を改善した事により，[4] に比べファームウェアサイズが約 50 ステップ削減出来ている．

この USB ゲームパッドの動作を verilog シミュレータを使用して確認した (図 16)．シミュレーション結果を表 7 に示す．

表 7 のように設計した USB ゲームパッドはリクエストパケット受信後，13 ビット時間で対応したパケットを送信出来ており，タイムアウト (16 ビット時間) を回避出来ている．また，デバイスリクエストの判別も 137 ビット時間と非常に高速に行えており，次のトランザクションが開始されるまでの約 988.6μs をデバイス制御に割り当て可能である (図 17)．

6 FPGA への実装

シミュレーションにより正常動作を確認した USB ゲームパッドを FPGA に実装した (図 18)．動作確認は各種 Windows2000 と RedHatLinux 9 で行い (図 19)，USB アナライザによって正常なパケットの流れが確認出来た

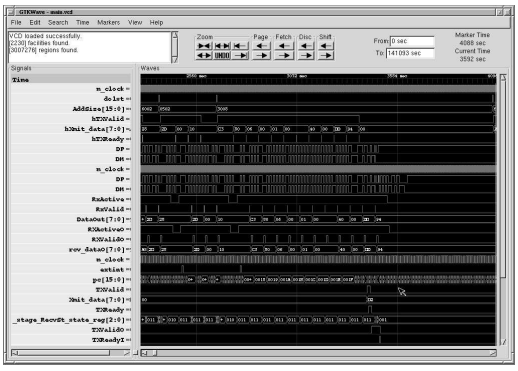

図 16: verilog シミュレータによるシミュレーション

表 7: シミュレーション結果

制御	処理にかかる最大ビット時間
トークンパケット受信からデバイスターゲットを判別し 対応したパケットを送信	13
セットアップパケット受信からデバイスクエスト判別が終了	137

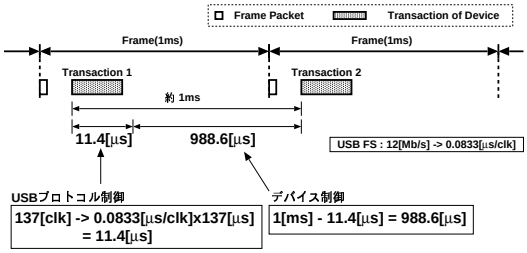

図 17: デバイス制御に割り当て可能な時間

(図 20)．また，USB ハブに複数のデバイスを接続しての正常動作も確認し，デバイスアドレスの判別処理も正常に行われていた．

7 まとめ

ハッシュ関数を利用した命令など USB プロトコル制御に適した命令をハードウェアに実装した事で，リクエストパケット制御に特化した USB デバイスコントローラを設計，実装出来た．また，高速なデバイスリクエスト判別により，デバイス制御に十分な時間を割り当てる事が可能であった．

ただし，ターゲットデバイス判別に比較演算を用いているため，エンドポイントを複数持つデバイスにこのコントローラを実装した場合，判別が間に合わずタイムアウトになる可能性がある．今後はターゲットデバイス判別にもハッシュ関数が適応可能かを検討し，幅広いデバイスに適応可能なデバイスコントローラを設計する．

また，本稿は制御が容易な USB ゲームパッドとして実装したが，今後は他のデバイスを ‘Phy-link’ に接続し，実装，評価を行っていきたい．

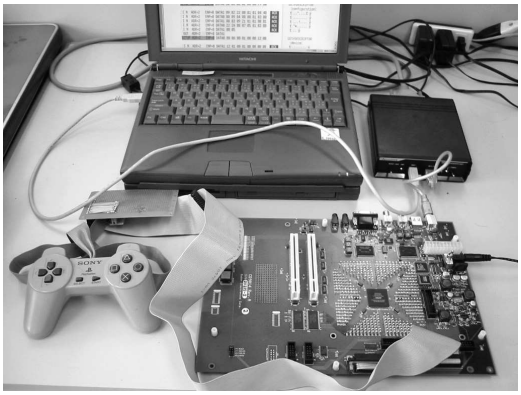

図 18: FPGA への実装

図 19: Windows2000 における正常認識

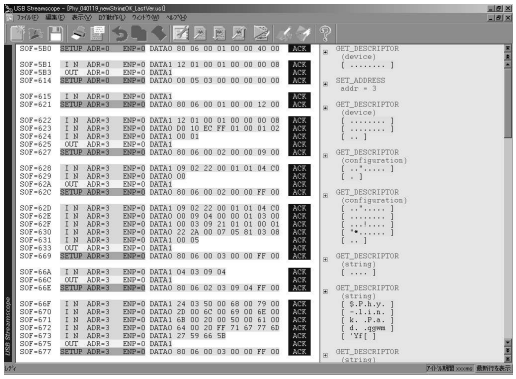

図 20: USB アナライザによる動作確認

参考文献

[1] 『USB ORG』: Universal Serial Bus Specification Revision 2.0. Universal Serial Bus Device Class Definition for Human Interface Device (HID) Firmware Specification Version 1.11. Universal Serial Bus HID Usage Tables. http://www.usb.org/

[2] 山岸誠仁 『USB ハード＆ソフト開発のすべて』 CQ 出版，2001

[3] 名野響，大山将城，近千秋 『USB2.0 コントローラ Phy-link』
第 22 回パルテノン研究会 資料集 pp.117-124，2003

[4] 名野響，神山智章，近千秋，清水尚彦『プロトコル処理の多くをハードウェア化した USB インターフェースと USB デバイスの開発』
第 4 回 組込みシステム技術に関するサマーワークショップ予稿集』 pp.15-19，July 2002

[5] 『UTMI』
http://www.Intel.com/technology/usb/download/