USB CAMAC デバイスドライバ内部仕様

- Technical Description of USB-CAMAC Device Driver -

仲吉一男

kazuo.nakayoshi@kek.jp

高エネルギー加速器研究機構

素粒子原子核研究所オンライングループ

目 次

1 Introduction **1**

1.1 目的 . . . 1
1.2 USB の概要 . . . 1
1.3 USB-CAMAC DAQ システムの概要 . . . 2

2 SCSI CAMAC クレート・コントローラ 3929 **3**

2.1 3929 の機能 . . . 3
2.2 3929 における CAMAC シングルアクション . . . 4
2.3 3929 における CAMAC ブロック転送 . . . 5

3 USB-SCSI 変換ケーブル **6**

3.1 LUB-SC . . . 6

4 USB-CAMAC デバイスドライバ **7**

4.1 Linux USB デバイスドライバのフレームワーク . . . 7
4.1.1 USB Request Block(URB) . . . 7
4.1.2 Linux USB コアドライバ関数 . . . 9
4.1.3 usb_driver 構造体 . . . 11
4.2 モジュール関連関数 . . . 11
4.3 USB 関連関数 . . . 12
4.3.1 probe() 関数 . . . 12
4.3.2 disconnect() 関数 . . . 13
4.4 ファイル操作関連関数 . . . 14
4.4.1 open(), close() 関数 . . . 14
4.4.2 ioctl() 関数 . . . 14
4.5 データ転送関連関数 . . . 16
4.5.1 send_cdb() 関数 . . . 16
4.5.2 send_data(), recv_data() 関数 . . . 17
4.5.3 scsi_read(), scsi_write() 関数 . . . 18
4.6 USB-CAMAC デバイスドライバの動作 . . . 20
4.6.1 CAMAC シングルアクション (リード) . . . 20
4.6.2 CAMAC シングルアクション (ライト) . . . 20
4.6.3 CAMAC ブロック転送 (リード) . . . 21
4.6.4 CAMAC ブロック転送 (ライト) . . . 21

5 USB-CAMAC ライブラリ **22**

5.1 USB-CAMAC ライブラリ一覧 . . . 22
5.2 USB-CAMAC ライブラリの詳細 . . . 22

6 性能評価 **23**
6.1 CAMAC シングルアクション 23
6.2 CAMAC ブロック転送 24

謝辞 **25**

参考文献 **26**

1 Introduction

1.1 目的

本システム開発の目的は「データ収集システムにおけるユーザビリティの向上」である. 「ユーザビリティ」とは一般に「使いやすさ」と訳され利用者が「使いやすい」,「わかりやすい」,「操作が簡単」な製品やシステムを高いユーザビリティを持つ製品やシステムということができる.

一般に素粒子・原子核実験のデータ収集 (DAQ) システムは, 各々の実験に特化した特殊なシステムであり, ユーザビリティの高いシステムとはいえない. テストベンチ等で使用する小規模な DAQ システムにおいても, これまでは特定な計算機の専用バスアダプタを使用してデータ収集を行なっていた. 近年 PC/AT 互換機 (PC) の高性能化および低価格化により PC を使用して大規模な DAQ システムの構築が可能となり ISA や PCI バスを VMEbus や CAMAC とのインターフェイスにすることが可能となった. このように PC の標準的なバスを使用することで以前より導入コストは低くなったが専用カードを PC にインストールし, 設定を行なうことは避けて通れなかった.

現在 PC とその周辺機器とのインターフェイスとしてデファクト・スタンダードになりつつある USB を利用することで専用バスカードを必要とせず USB ポートを持つ計算機で DAQ システムの構築が可能となる. また USB の特徴の 1 つであるホットプラグにより PC の電源切断や OS のリスタートなしで必要な時に USB ケーブルを CAMAC 等に接続して使用できるのである.

本内部仕様では Linux における USB CAMAC デバイスドライバの実装方法とその動作について詳細に説明する. 1 節では 開発の目的, USB の概要と USB-CAMAC DAQ システムについて述べる. 2 節では SCSI CAMAC クレート・コントローラの機能とその操作方法について述べる. 3 節では USB-SCSI 変換ケーブルについて説明する. 4 節では Linux における USB デバイスドライバのフレームワークについて述べ, USB-CAMAC デバイスドライバについて実際のプログラムコードを示しながら説明を行なう. 5 節では USB-CAMAC デバイスドライバと対で使用する USB-CAMAC ライブラリについて説明する. 6 節では USB-CAMAC デバイスドライバの性能測定結果を示す.

1.2 USB の概要

USB(Universal Serial Bus) [1–3] は 1994 年に Compaq, Intel, Microsoft, NEC により開発がスタートした. その最初の仕様 USB1.0 は 1996 年 1 月にリリースされ, 最新の USB2.0 は 2000 年 4 月にリリースされた. 現在 USB に 対応したキーボード , マウス, スキャナ等の機器が入手可能である. USB の特徴として次の点が挙げられる.

- USB はひとつのホストを頂点とするツリー構造型のバスである. いかなる種類のコミュニケーションもホストにより開始され, デバイス同士は直接に通信できない.
- 1 台のホストに対して最大 127 台の USB 機器が接続可能.
- USB1.0, USB1.1 仕様では 12Mbps のフルスピードと 1.5Mbps のロースピードの転送速度を選択可能. USB2.0 仕様では上記の 2 つのスピードに加え, 480Mbps のハイスピード転送も可能.
- USB に 対応したキーボード , マウス, スキャナ等の機器は , 電源を入れた状態で USB ケーブルの抜き差しが可能である.
- USB ハブによるデバイスへの電源供給が可能.

1.3 USB-CAMAC DAQ システムの概要

既存の CAMAC クレート・コントローラには, RS232C, GPIB や SCSI を計算機とのインターフェイスにするものはあったが現時点で USB 対応のものは存在しない. そのため既存のハードウエアを利用して USB で CAMAC と計算機を接続することを考えて調査を行なった. その結果 SCSI インターフェイス付き CAMAC クレート・コントローラと USB-SCSI 変換ケーブルを用いて USB-SCSI 経由で CAMAC を制御することにした. 本システムは次の 3 つのハードウエアで構成される (図 1).

- KineticSystems 社 SCSI CAMAC クレート・コントローラ 3929 [4]
- ロジテック 社 USB-SCSI 変換ケーブル LUB-SC [8]
- USB 対応で Intel 系 CPU の PC

図 1 の左から USB ポートを持つ PC, USB と SCSI を接続するための USB-SCSI 変換ケーブル. CAMAC クレートの右端には SCSI CAMAC クレートコントローラがある. SCSI CAMAC クレート・コントローラ 3929 については 2 節で USB-SCSI 変換ケーブル LUB-SC については 3 節で説明する.

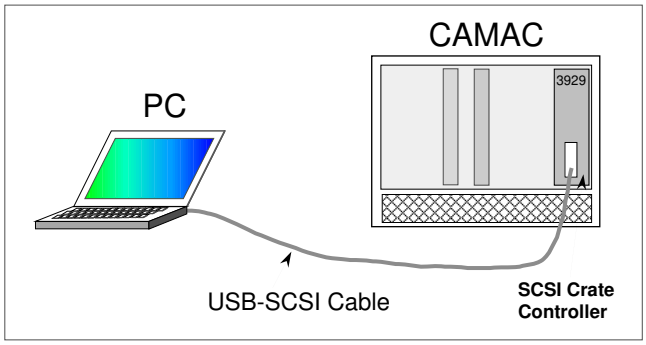

図 1: USB-CAMAC DAQ システムの概念図.

次に本システムのソフトウエア構成について説明する. 以下, ソフトウエア構成を示す.

- Linux Kernel 2.4.0 以上
- UHCI ドライバ (usb-uhci.o)
- USB-CAMAC ドライバ (xcamac.o)
- USB-CAMAC ライブラリ

Linux Kernel 2.4.0 は 2001 年 1 月にリリースされた USB 対応のカーネルである. 今後はこのカーネル 2.4 をベースにした Linux のディストリビューション・パッケージが主流になるはずである. 現在 USB ホストコントローラは UHCI(Universal Host Controller Interface) 系チップセットと OHCI(Open Host Controller Interface) 系チップセットに分類できる. したがってそのデバイスドライバも 2 種類存在する. USB-CAMAC デバイスドライバは今回開発を行なった Linux 用のデバイ

スドライバである. USB-CAMAC ライブラリはそのデバイスドライバのユーザインターフェイスであり CAMAC に慣れているユーザが簡単に操作できるような実装をした. USB-CAMAC デバイスドライバについては 4 節 で USB-CAMAC ライブラリ については 5 節で説明する.

2 SCSI CAMAC クレート・コントローラ 3929

SCSI CAMAC クレート・コントローラ 3929(以下 3929) は, SCSI インターフェイスをもつクレート・コントローラである. 計算機と SCSI で接続するため SCSI インターフェイスをもつ計算機であれば利用できる. 3929 には リスト処理機能のある 3929-Z1B とリスト処理機能のない 3929-Z1A という 2 つのモデルがあり, 開発には 3929-Z1A を使用した. したがって開発したデバイスドライバにはリスト処理機能は実装していない. 以下, デバイスドライバを開発するために必要な 3929 の仕様について説明する [4].

2.1 3929 の機能

ここでは, SCSI コマンドと CAMAC 操作の対応について説明をする. 一般に SCSI では計算機 (イニシエータ) から SCSI 機器 (ターゲット) に対する要求は, Command Descriptor Block(CDB) を SCSI ターゲットへ送ることにより行なわれる [5,6]. 3929 は SCSI のグループ 0 (6 バイトコマンド) とグループ 1(10 バイトコマンド) をサポートしている. 表 1 にそのコマンドとオペレーションコード (OPCODE) を示す.

OPCODE	COMMAND
0x00	TEST UNIT READY
0x03	REQUEST SENSE
0x06	UNBOOK LAM
0x08	RECEIVE
0x09	SINGLE
0x0a	SEND
0x0c	SETUP
0x0d	REGISTER ACCESS
0x0e	RESUME LIST
0x12	INQUIRY
0x1d	SEND DIAGNOSTIC
0x20	EXECUTE LIST
0x22	BLOCK
0x23	LOAD LIST
0xa0	BOOK LAM

表 1: 3929 でサポートしている SCSI コマンドとそのオペレーションコード

2.2 3929 における CAMAC シングルアクション

CAMAC のシングルアクションを実行するためには上述の OPCODE 0x09 の SINGLE コマンドを 3929 へ送る. SINGLE コマンドの CDB の構造を表 2 に示す.

Byte \ Bit	7	6	5	4	3	2	1	0
0	OPCODE = 0x09							
1	LUN = 0				RESERVED = 0			
2	MODE CONTROL							
3	NAF HI							
4	NAF LO							
5	CONTROL BYTE							

表 2: CAMAC シングルアクションに用いる SCSI CDB.

CAMAC ファンクションが CAMAC 制御操作を指定しているなら, SCSI のデータフェイズは起こらない. CAMAC のリードまたはライト操作を指定しているなら, SCSI コマンドフェイズの後に続いてデータ転送が行なわれる. SCSI データフェイズ中に転送されるバイト数は, CDB の MODE CONTROL で指定されたデータ長に依存する. 16 ビット転送は 2 バイトを 24 ビットは 4 バイトを要求する. 3929 により CAMAC 操作の実行が終了した後, ステータス・バイトが送られる. CAMAC 操作がエラーのため中断された場合は CHECK CONDITION が送られる. このエラーは CAMAC Q, X 応答等のパラメタに依存する.

CAMAC シングルアクションを実行する際にデータ長や Q 応答等の設定を行なう MODE CONTROL (表 3) について説明する.

Byte \ Bit	7	6	5	4	3	2	1	0
5	0x00	0x00	0x00	0x00	TM1	WS2	WS1	AD

表 3: SINGLE SCSI コマンドの MODE CONTROL.

- Bits 7 - 4: 0x00 に設定
- Bit 3(TM1): Q-Stop mode なら 0x00 に Q-Ignore mode なら 0x01 に設定する.
- Bit 2,1(WS2, WS1): CAMAC Data Word Size の設定に使用する. 24 ビットなら {WS2, WS1}={0x00, 0x00}. 16 ビットなら {WS2, WS1}={0x00, 0x01}.
- Bit 0: この ビット が 0x00 なら CAMAC X 応答が 0 の場合は, SCSI の CHECK CONDITION 状態となる. この ビット が 0x01 なら CAMAC X 応答が 0 の場合でも SCSI の CHECK CONDITION 状態にはならない.

2.3 3929 における CAMAC ブロック転送

CAMAC の ブロック転送を実行するためには OPCODE 0x22 の BLOCK コマンドを 3929 へ送る. BLOCK コマンドの CDB は表 4 のような構造である.

Byte \ Bit	7	6	5	4	3	2	1	0
0	OPCODE = 0x22							
1	LUN = 0			RESERVED = 0				
2	MODE CONTROL							
3	NAF HIGH							
4	NAF LOW							
5	TRANSFER BYTE COUNT HI							
6	TRANSFER BYTE COUNT MID							
7	TRANSFER BYTE COUNT LO							
8	RESERVED = 0							
9	CONTROL BYTE							

表 4: CAMAC ブロック転送に用いる SCSI CDB.

3929 が CDB を受信後, データフェイズになる. データ転送の方向は CAMAC ファンクションによる. 一旦データ転送が始まると転送量に達するかエラー状態が起こるまで続けられる. 3929 によるブロック転送操作が完了後, ステータスビットが送られる. GOOD または CHECK CONDITION ステータスのいずれかが返される. CHECK CONDITION は CAMAC ブロック転送操作が エラーにより中断されたときに返される. このエラーは CAMAC Q,X 応答などのパラメータに依存する.

BLOCK コマンドの MODE CONTROL は転送モードのタイプ, CAMAC データ長, 他のいくつかのブロック転送に関係するパラメータを指定することに使用される. 表 5 にブロック転送コマンドの MODE CONTROL を示す. データ転送量はブロック転送操作のデータフェイズ中に 3929 から (へ) 転送されるバイト数が指定される.

Byte \ Bit	7	6	5	4	3	2	1	0
9	0x00	0x00	0x01	TM2	TM1	WS2	WS1	AD

表 5: BLOCK SCSI コマンドの MODE CONTROL.

- Bit 7: 0x00 に設定
- Bit 6(FAST): 3929 の FAST データウェイ・アクセス機能 [4] を有効/無効にするために用いる.
- Bit 5:ブロック転送の場合は 1 をセットしなければならない.

- Bit 4, 3(TM2, TM1): Q-Stop mode なら {TM2,TM1}={0x00,0x00}, Q-Ignore mode なら {TM2,TM1}={0x00,0x01}, Q-Repeat mode なら {TM2,TM1}={0x01,0x00}, Q-Scan mode なら {TM2,TM1}={0x01,0x01} に設定する.
- Bit 2, 1(WS2, WS1): CAMAC データ長の設定に使用する. 24 ビットなら {WS2, WS1}={0x00, 0x00}. 16 ビット なら {WS2, WS1}={0x00, 0x01}.
- Bit 0: この ビット が 0x00 なら CAMAC X 応答が 0 の場合は, SCSI の CHECK CONDITION 状態となる. この ビット が 0x01 なら CAMAC X 応答が 0 の場合でも SCSI の CHECK CONDITION 状態にはならない.

3 USB-SCSI 変換ケーブル

現在, 数種類の USB-SCSI 変換ケーブルが入手可能である. SCSI 機器をこのケーブルで接続することにより USB 機器として利用できる. 例えばSCSI インターフェイスを持たない PC でも USB ポートがあれば SCSI スキャナ等が利用できることになる. 今回, ロジテック社の LUB-SC と Adaptec 社の USB-Xchange を購入した. 両者とも Windows や Macintosh 用のデバイスドライバが付属しており Windows98 で問題無く動作することを確認した. Linux 関連雑誌にロジテック社の LUB-SC のデバイスドライバに関連する記事 [7] があり非常に有用であったのでロジテック社の LUB-SC を使用することにした.

3.1 LUB-SC

ロジテック 社の USB-SCSI 変換ケーブル LUB-SC について説明する. その仕様を表 6 に示す.

インターフェイス	USB1.0, Isochronous Tranfer
最大データ転送速度	12Mbps
接続可能 SCSI 機器	MO ユニット, ハードディスク, CD-R/RW
SCSI 機器の最大接続台数	7 台
入力電圧	DC 5V±5%
最大消費電流	500mA

表 6: ロジテック 社の USB-SCSI 変換ケーブル LUB-SC の仕様

その他の情報については ロジテック社のホームページを参照すること [8].

4 USB-CAMAC デバイスドライバ

USB-CAMAC デバイスドライバは CAMAC クレート・コントローラ 3929 に対して USB 経由で SCSI コマンドを発行し CAMAC の制御を行う. 本節では USB-CAMAC デバイスドライバの内部構造とその動作を実際のコードを交えて説明する. はじめに Linux USB デバイスドライバのフレームワークについて説明する. 次に USB-CAMAC デバイスドライバの内部構造について述べる. USB デバイスドライバは通常のキャラクタ型デバイスドライバ [9] とほぼ同じでありモジュールとしてカーネルに組み込むことが可能である. まずモジュール組み込み/削除時に使われるモジュール関連関数について説明する. 次に USB デバイスドライバで特徴的な USB 装置の接続/取り外しに呼ばれる probe() 関数と disconnect() 関数について説明する. また open(), ioctl(), close() 等のファイル操作関数について説明する. 本デバイスドライバで定義した CAMAC データ転送に使用する関数について説明し, 最後に実際の CAMAC 操作でデバイスドライバがどのような動作を行なっているかを説明する.

4.1 Linux USB デバイスドライバのフレームワーク

Linux における USB デバイスドライバは図 2 のような階層構造になっている [10].

- USB コアドライバ
- USB ホストコントローラ・ドライバ
- USB デバイスドライバ

USB コアドライバは USB 制御用の関数群を提供する. USB ホストコントローラ・ドライバは USB ホストで使用されているコントローラチップ用のドライバである. UHCI と OHCI の 2 種類が存在し, そのドライバも 2 つ存在する. USB デバイスドライバは, 各種 USB 装置に対応するドライバであり我々はアプリケーションからそのドライバに対しシステムコールを発行して USB 装置を制御する. 今回開発した USB-CAMAC デバイスドライバ はこの USB デバイスドライバに相当する.

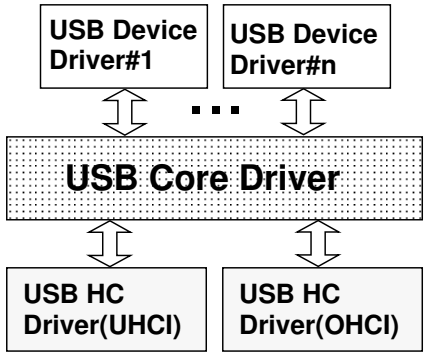

図 2: Linux USB デバイスドライバのフレームワーク概念図 . Linux2.4 カーネルでは USB コアドライバと 2 つのホストコントローラ (HC) ドライバが用意されている. 図の USB Device Driver#1,...,Driver#n はそれぞれ特定のデバイスに対するデバイスドライバである.

4.1.1 USB Request Block(URB)

USB コアドライバで導入されている Linux の USB Request Block(URB) について説明する. これは Windows 環境のドライバにおける URB と同様のものである. URB とは SCSI の CDB のようなものでデバイスドライバは URB を使用して USB データ転送要求を行なう. 以下 linux/usb.h にある URB 構造体を示す.

```
typedef struct urb
{
        spinlock_t lock;               // lock for the URB
        void *hcpriv;                  // private data for host controller
        struct list_head urb_list;     // list pointer to all active urbs
        struct urb *next;              // pointer to next URB
        struct usb_device *dev;        // pointer to associated USB device
        unsigned int pipe;             // pipe information
        int status;                    // returned status
        unsigned int transfer_flags;   // USB_DISABLE_SPD | USB_ISO_ASAP | etc.
        void *transfer_buffer;         // associated data buffer
        int transfer_buffer_length;    // data buffer length
        int actual_length;             // actual data buffer length
        int bandwidth;                 // bandwidth for this transfer request
                                          (INT or ISO)
        unsigned char *setup_packet;   // setup packet (control only)
        //
        int start_frame;               // start frame (iso/irq only)
        int number_of_packets;         // number of packets in this request
                                          (iso)
        int interval;                  // polling interval (irq only)
        int error_count;               // number of errors in this transfer
                                          (iso only)
        int timeout;                   // timeout (in jiffies)
        //
        void *context;                 // context for completion routine
        usb_complete_t complete;       // pointer to completion routine
        //
        iso_packet_descriptor_t iso_frame_desc[0];
} urb_t, *purb_t;
```

次に Linux USB コアドライバで提供されている URB 関連の関数を次に示す.

- urb_t *usb_alloc_urb(int iso_packets)
  この関数で URB をメモリにアロケートする. 返ってくる値はアロケートされた URB へのポインタで, アロケーションに失敗したら 0 が返る. isoframes 引数にはスケジュールしたいアイソクロナス転送のフレーム数を指定する. 制御, バルク, 割り込み転送の場合は 0 である.

- void usb_free_urb(urb_t* urb)
  この関数で指定した URB をメモリから解放する.

- int usb_submit_urb(urb_t *urb)

この関数で指定した URB の転送を行なう.

- int usb_unlink_urb(urb_t *urb)
  この関数で内部リストの URB を削除し, 全てのアロケートされているハードウエアデスクリプタを開放する. usb_unlink_urb() が終了後, usb_free_urb(urb) で URB と関連する可能性のある全てのデータを安全に開放できる.

トランザクションのタイプにより, 以下のような単に URB を作るマクロがある. 一般に全てのマクロは USB デバイスへのポインタ, パイプ転送バッファ, 要求する転送長, 終了ハンドラ, 自分自身のコンテキストが必要である.

- FILL_CONTROL_URB()
- FILL_BULK_URB()
- FILL_INT_URB()
- FILL_CONTROL_URB_TO()
- FILL_BULK_URB_TO()

4.1.2 Linux USB コアドライバ関数

Linux USB コアドライバ は様々な USB デバイスドライバ が使用する USB 関連の関数群を提供する. 以下, 本デバイスドライバで使用した関数について説明する.

- int usb_register(struct usb_driver *new_driver)
  新しい USB デバイスドライバの登録を行なう. 処理が成功すれば 0 が失敗の場合は負の値が返される. 引数 の usb_driver については 4.1.3 で説明する.
  new_driver: usb_driver 構造体.

- void usb_deregister(struct usb_driver *driver)
  すでに登録してある USB デバイスドライバの削除を行なう.
  new_driver: usb_driver 構造体.

- int usb_clear_halt(struct usb_device *dev, int pipe)
  USB エンドポイントが STALL 状態になったとき, この関数でその状態を解除する. STALL 状態とはエンドポイントがエラー等でホストと送受信できない状態を指す.
  dev: usb_device 構造体へのポインタ.
  pipe: 転送に使用するパイプ.

- usb_sndctrlpipe(struct usb_device *dev, unsigned int endpoint)
  引数 endpoint に対する downstream 制御転送用のパイプを作る. これはマクロである.
  dev: usb_device 構造体へのポインタ.
  endpoint: 制御転送を行なうエンドポイント.

- usb_rcvctrlpipe(struct usb_device *dev, unsigned int endpoint)
  引数 endpoint に対する upstream 制御転送用のパイプを作る. これはマクロである.
  dev: usb_device 構造体へのポインタ.
  endpoint: 制御転送を行なうエンドポイント.

- usb_sndbulkpipe(struct usb_device *dev, unsigned int endpoint)
  引数 endpoint に対する downstream バルク転送用のパイプを作る. これはマクロである.
  dev: usb_device 構造体へのポインタ.
  endpoint: バルク転送を行なうエンドポイント.

- usb_rcvbulkpipe(struct usb_device *dev, unsigned int endpoint)
  引数の endpoint に対する upstream バルク転送用のパイプを作る. これはマクロである.
  dev: usb_device 構造体へのポインタ.
  endpoint: バルク転送を行なうエンドポイント.

- int usb_control_msg(struct usb_device *dev, unsigned int pipe, __u8 request, __u8 requesttype, __u16 value, __u16 index, void *data, __u16 size, int timeout)
  制御用 URB を作って送りその終了を待つ. 関数中で FILL_CONTROL_URB() マクロを使って URB 構造体のメンバに値を代入している. コールが成功したら 0 が返る. 失敗すると負の値が返る.
  dev: メッセージを送る usb_device のポインタ.
  pipe: メッセージを送るエンドポイントのパイプ.
  request: リクエスト値.
  requesttype: リクエストタイプ.
  value: リクエストが決める値.
  index: インデックス値.
  data: 転送するデータへのポインタ.
  size: 転送するデータサイズ.
  timeout: タイムアウトまでの時間 (単位は jiffies).

- int usb_bulk_msg(struct usb_device *usb_dev, unsigned int pipe, void *data, int len, int *actual_length, int timeout) バルク転送用 URB を作って送りその終了を待つ. 関数中で FILL_BULK_URB() マクロを使って URB 構造体のメンバに値を代入している. コールが成功したら 0 が返る. 失敗すると負の値が返る.
  dev: メッセージを送る usb_device のポインタ.
  pipe: メッセージを送るエンドポイントのパイプ.
  data: 転送するデータへのポインタ.
  len: 転送を行なうデータのバイト数.
  actual_length: 実際に転送が行なわれたバイト数へのポインタ.
  timeout:タイムアウトまでの時間 (単位は jiffies).

4.1.3 usb_driver 構造体

USB デバイスドライバで使用する重要な構造体のひとつに usb_driver 構造体がある. usb_driver 構造体 は linux/usb.h で定義されており本ドライバでは, 次のように usb_driver を定義している.

```
static struct usb_driver
driver_table = {
        name:         "xcamac",
        probe:        probe_xcamac,
        disconnect:   disconnect_xcamac,
        fops:         &usb_xcamac_fops,
        minor:        USB_XCAMAC_MINOR_BASE,
};
```

また, 上記の usb_xcamac_fops は 次のように定義している. 本ドライバではファイル操作関連の関数は xcamac_open(), xcamac_close(), xcamac_ioctl() を使用する.

```
static struct file_operations
usb_xcamac_fops = {
        owner:           THIS_MODULE,
        ioctl:           xcamac_ioctl,
        open:            xcamac_open,
        release:         xcamac_close,
};
```

4.2 モジュール関連関数

モジュール初期化時によばれる usb_xcamac_module_init() とモジュール削除時によばれる usb_xcamac_module_cleanup() を示す. これは通常の Linux モジュールで用いるものと同様なものである. usb_xcamac_module_init() では, /proc ファイルシステムに本ドライバのエントリを作る. これは/proc/xcamac から現在のドライバのステータスを読むことを可能にする. 次に USB コアドライバで提供されている usb_register() 関数でドライバの登録を行なう.

```
int __init
usb_xcamac_module_init(void)
{
        proc_xcamac_create();
        return usb_register(&driver_table) < 0 ? -1 : 0;
}

void __exit
usb_xcamac_module_cleanup(void)
{
        proc_xcamac_destroy();
        usb_deregister(&driver_table);
}
```

4.3 USB 関連関数

4.3.1 probe() 関数

probe() 関数は USB デバイスが PC に取り付けられたときに USB コアドライバから呼び出される関数である. probe() 関数では次のような処理を行なう.

1. USB デバイスのベンダ ID とプロダクト ID の確認を行なう.
2. デバイスドライバで使用する構造体に必要なメモリ領域を確保する.
3. SET_CONFIGURATION 標準デバイスリクエストでデバイスの configuration を行なう.

```
static void *device_probe(struct usb_device *dev, ...)
{
        ...
        vid = dev->descriptor.idVendor;
        pid = dev->descriptor.idProduct;
        vid_pid = ( vid << 16 ) | pid;

        switch( vid_pid ){
        case VID_PID_LOGITEC: v="LOGITEC"; break;
        default:
                printk("ERR: Unknown Vender ID vid=0x%x pid=0x%x\n", vid, pid);
                return NULL;
        }
        global_xcamac = kmalloc(sizeof(struct xcamac_data), GFP_KERNEL);
        if(global_xcamac == NULL ) {
                printk("no memory!");
                return NULL;
        }
        ...
        if( usb_set_configuration(dev, dev->config[0].bConfigurationValue) ) {
                printk("usb_set_configuration error");
                kfree(global_xcamac);
                global_xcamac = NULL;
                return NULL;
        }
        ...
        return global_xcamac;
}
```

4.3.2 disconnect() 関数

disconnect() 関数は USB デバイスがバスから取り除かれたときに USB コアドライバ から呼び出される関数である. disconnect() 関数では probe() 関数で確保したメモリ領域を解放する.

```
device_disconnect(struct usb_device *dev, void *ptr)
{
        struct xcamac_data *xcamac = (struct xcamac_data *)ptr;

        if(xcamac->is_open) {
        xcamac->is_open = 0;
                xcamac->dev = NULL;
        }
        xcamac->present = 0;

        kfree(global_xcamac);
        global_xcamac = NULL;

        if( probe_count != 1 ){
                printk("illegual probe_count=%d\n", probe_count );
        }

        --probe_count;
}
```

4.4 ファイル操作関連関数

4.4.1 open(), close() 関数

open() 関数は, ユーザプログラムから USB デバイスドライバをオープンするために使用される.

1. デバイスのマイナー番号が正しいか調べる
2. MOD_INC_USE_COUNT マクロを使ってモジュールの利用度を 1 だけ増やす

close() 関数は, ユーザプログラムから USB デバイスドライバをクローズするために使用される. そこでは MOD_DEC_USE_COUNT マクロを使ってモジュールの利用度を 1 だけ減らす.

4.4.2 ioctl() 関数

本ドライバでは ioctl() 関数は, CAMAC の制御およびデータの読み込み/書き込みを行なうために用いる. CAMAC のデータ読み込み, データ書き込みに対応して read() / write() 関数を使うことも考えたが CAMAC のデータ読み込み, データ書き込みのいずれの操作においても, まず SCSI CDB を 3929 に送らなければならない. read() 関数を実装する際, この CDB の書き込み操作の実装をするためのアイディアが思い浮かばなかったので, より柔軟な実装が可能な ioctl() 関数を用いることにした. すなわち ioctl() の引数 cmd でシングルアクションやブロック転送等の CAMAC 動作を指定することにした. また ioctl() の引数 arg として CAMAC 情報を記述した次の構造体 (ブロック転

送とシングルアクションで異なる) を指定することによりカーネル関数 copy_from_user() を使ってユーザ空間からカーネル空間 (ドライバ) へデータを渡したり copy_to_user() を使ってカーネル空間 (ドライバ) からユーザ空間へデータを渡すことができるようにした.

```
struct camac_single{
        unsigned int naf;
        unsigned int data;
        unsigned int qx;
};

struct camac_block{
        unsigned int naf;
        unsigned int mode;
        unsigned int ch;
        void *data;
        unsigned int retlen;
};
```

一般に ioctl() 関数は switch 文から成っており, 引数 cmd に対応したブロックに飛び, そこで処理を行なう. 表 7 に本ドライバでサポートしている ioctl() 関数 のコマンド と対応する CAMAC/SCSI コマンド を示す.

cmd	CAMAC/SCSI COMMAND
INQUIRY	SCSI INQUIRY Command
BLOCKL	CAMAC Long Word Block Transfer
BLOCKW	CAMAC Word Block Transfer
CAMACL	CAMAC Long Word Single Action
CAMACW	CAMAC Word Single Action

表 7: ioctl() の引数 cmd で設定可能な SCSI/CAMAC コマンド

実際のドライバーのコード は次のような構造になっている. 上述のようにコマンド に対応したブロックへ飛び, そこで処理が行なわれるようになっている.

```
xcamac_ioctl( ... ,unsigned int cmd, unsigned long arg)
{
 ...
        switch (cmd) {

                case INQUIRY:
                {
                        ...
                        break;
                }
                case BLOCKL:
                {
                        ...
                        break;
                }
                case BLOCKW:
                {
                        ...
                        break;
                }
                case CAMACW:
               {
                        break;
                }
                ...
        }
...
}
```

4.5 データ転送関連関数

3929 とデータ転送を行なうために使用する関数について説明する.

4.5.1 send_cdb() 関数

send_cdb() 関数は SCSI CDB を 3929 へ送るために使用する関数である. 以下その手順を説明する.

1. USB コアドライバの usb_sndctrlpipe() でコントロール転送用のパイプを取得する.
2. USB コアドライバの usb_control_msg() で CDB を 3929 へ送る.

```
static int
send_cdb(struct file *file, unsigned char *cdb, unsigned short size)
{
        ...
        pipe   = usb_sndctrlpipe(udev, 0);
        ...
        r = usb_control_msg(udev, pipe, request, requesttype,
                     value, index, cdb, size, timeout);
        if( r < 0 ){
                printk("**** send_cdb r = %d\n",r);
                return -1;
        }
        return r;
}
```

4.5.2 send_data(), recv_data() 関数

send_data() 関数は 3929 へデータを送るための関数である. この関数の前に実行される send_cdb() により 3929 は SCSI CDB SCSI のデータフェイズとなり指定されたデータ長のデータ転送が行なわれる. 以下その手順を説明する.

1. USB コアドライバの usb_sndbulkpipe() でバルク転送用のパイプを取得する.
2. USB コアドライバの usb_bulk_msg() でデータを 3929 へ送る.

```
static int
send_data(struct file *file, void *buf, int len)
{
        ...
        udev   = xcamac->dev;
        pipe   = usb_sndbulkpipe(udev, xcamac->ep_out);
        ...
        r = usb_bulk_msg(udev, pipe,(unsigned char *)buf, len,
                                        &ans_len, HZ*10);
        if( r < 0 ){
                printk("send_data ERR=%d\n", r);
        return -1;
        }
 return ans_len;
}
```

recv_data() 関数は 3929 からデータを取得する関数である. この関数の前に実行される send_cdb() により 3929 は SCSI のデータフェイズとなり指定されたデータ長のデータ転送が行なわれる. 以下その手順を説明する.

1. USB コアドライバの usb_sndbulkpipe() でバルク転送用のパイプを取得する.
2. USB コアドライバの usb_bulk_msg() でデータを 3929 から取得する.

```
static int
recv_data(struct file *file, void *buf, int len)
 {
         ...
         udev   = xcamac->dev;
         pipe   = usb_rcvbulkpipe(udev, xcamac->ep_inp);
         ...
         if( (r=usb_bulk_msg(udev, pipe, buf,
                        len, &ans_len, 10*HZ)) >= 0 ){
                 return ans_len;
         }

         return r;
 }
```

4.5.3 scsi_read(), scsi_write() 関数

scsi_read() 関数は, CAMAC からデータの読み込みを行なう関数である. 以下その手順を説明する.

1. USB 制御転送を使用して 3929 に CAMAC データ読み込みに必要な CDB を送る.
2. USB バルク転送を用いてデータを CAMAC から取得する.

```
scsi_read(struct file *file, unsigned char *cdb, unsigned short
        cdb_size, void *buf, int len)
{
 ...
        if( (r = send_cdb(file, cdb, cdb_size)) < 0 ){
                return -EFAULT;
        }
        ...
        if( (r=recv_data(file, buf, cur_len )) < 0 ){
                printk("*** (scsi_read)ERR r = %x\n", r);
        }
...
return len;
}
```

scsi_write() 関数は, CAMAC へデータの書き込みを行なう関数である. 以下その手順を説明する.

1. USB 制御転送を使用して 3929 に CAMAC データ書き込みに必要な CDB を送る.
2. USB バルク転送を用いてデータを CAMAC へ書き込む.

```
scsi_write(struct file *file, unsigned char *cdb, unsigned short
        cdb_size, void *buf, int len)
{
 ...
        if( (r = send_cdb(file, cdb, cdb_size)) < 0 ){
                return -EFAULT;
        }
        ...
        if( (r=send_data(file, buf, cur_len )) < 0 ){
                printk("*** (scsi_read)ERR r = %x\n", r);
        }
...
return len;
}
```

4.6 USB-CAMAC デバイスドライバの動作

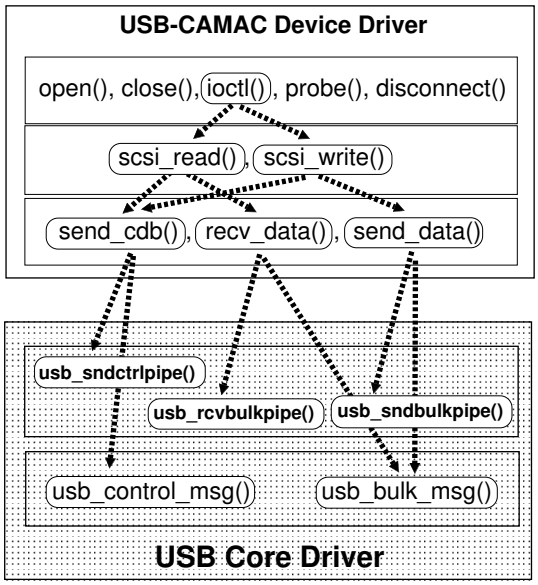

図 3: USB-CAMAC デバイスドライバ内のデータ転送関連の関数と USB コア関数の関係.

USB-CAMAC デバイスドライバのデータ転送関連の関数が USB コアドライバのどのような関数を呼んでいるのかを図式化したものを図 3 に示す. 図から ioctl() システムコールにより scsi_read(), scsi_write() がコールされ, その中で SCSI CDB を 3929 へ送るための関数 send_cdb() がコールされる. send_cdb() からは USB コアドライバの USB 制御転送関連の関数がコールされている. またデータ転送に用いる recv_data(), send_data() からは USB コアドライバのバルク転送関連の関数をコールしていることがわかる.

以下ユーザ・アプリケーションから CAMAC シングルアクションとブロック転送を実行したときのデバイスドライバの動作について説明する.

4.6.1 CAMAC シングルアクション (リード)

USB-CAMAC ライブラリから CAMAC 16 ビットデータの読み込むために camac() 関数をコールした後, デバイスドライバでどのように処理が行なわれるかを説明する. 例として ステーション #2 のスイッチレジスタを読む場合を取り上げる. CAMAC NAF は (n,a,f) = (2,1,0) とする.

1. アプリケーションから USB-CAMAC ライブラリの camacw(naf, &data, &q, &x) をコールする.
2. camacw() は コマンドに CAMACW を指定してドライバの ioctl() 関数をコールする.
3. ioctl() 関数では内部にある CAMACW 処理ブロックへとぶ.
4. CAMACW 処理ブロックでは CAMAC のファンクション (f) に依存した処理を行なう. この場合は f=0 なので CAMAC データ読み込みブロックへとぶ.
5. CAMAC データ読み込みブロックではデータ長を 2 バイトに指定して scsi_read() 関数をコールする.
6. scsi_read() では send_cdb() により SCSI CDB を 3929 へ送る.
7. 次に recv_data() により 3929 からデータを取得する.
8. データおよび Q, X を camacw() の data, q, x に返す.

4.6.2 CAMAC シングルアクション (ライト)

USB-CAMAC ライブラリから CAMAC 16 ビットデータを書き込むために camac() 関数をコールした後, デバイスドライバでどのように処理が行なわれるかを説明する. 例としてステーション #2

のスイッチレジスタにデータを書く場合を取り上げる. CAMAC NAF は (n,a,f) = (2,0,16) とする.

1. アプリケーションから USB-CAMAC ライブラリの camacw(naf, &data, &q, &x) をコールする.
2. camacw() は コマンドに CAMACW を指定してドライバの ioctl() 関数をコールする.
3. ioctl() 関数では内部にある CAMACW 処理ブロックへとぶ.
4. CAMACW 処理ブロックでは CAMAC のファンクション (f) に依存した処理を行なう. この場合は f=16 なので CAMAC データ書き込みブロックへとぶ.
5. CAMAC データ書き込みブロックではデータ長を 2 バイトに指定して scsi_write() 関数をコールする.
6. scsi_write() では send_cdb() により SCSI CDB を 3929 へ送る.
7. 次に send_data() により 3929 にデータを送る.
8. Q, X を camacw() の q, x に返す.

4.6.3 CAMAC ブロック転送 (リード)

USB-CAMAC ライブラリから CAMAC 16 ビットデータをブロック転送で読み込むために cdmaw() 関数をコールした後デバイスドライバでどのように処理が行なわれるかを説明する. 例としてステーション #2 のスイッチレジスタを読む場合を取り上げる. CAMAC NAF は (n,a,f) = (2,1,0) とする. またブロック転送のモードは Q-Stop, Q-Ignore, Q-Repeat, Q-Scan を選択できる.

1. アプリケーションから USB-CAMAC ライブラリの cdmaw(mode, naf, &buf, len, &retlen) をコールする.
2. cdmaw() は コマンドに BLOCKW を指定してドライバの ioctl() 関数をコールする.
3. ioctl() 関数では内部にある BLOCKW 処理ブロックへとぶ.
4. BLOCKW 処理ブロックでは CAMAC のファンクション (f) に依存した処理を行なう. この場合は f=0 なので CAMAC データ読み込みブロックへとぶ.
5. CAMAC データ読み込みブロックではデータ長を (len × 2) バイトに指定して scsi_read() 関数をよぶ.
6. scsi_read() では send_cdb() により SCSI CDB を 3929 へ送る.
7. 3929 から len で指定したデータ長のデータ転送が行なわれ recv_data() により 3929 からデータを取得する.
8. データおよび転送を行なったデータ長を cdmaw() の buf, retlen に返す.

4.6.4 CAMAC ブロック転送 (ライト)

USB-CAMAC ライブラリから CAMAC 16 ビットデータをブロック転送で書き込むために cdmaw() 関数をコールした後デバイスドライバでどのように処理が行なわれるかを説明する. 例としてステーション #2 のスイッチレジスタにデータを書き込む場合を取り上げる. CAMAC NAF は (n,a,f) = (2,0,16) とする. またブロック転送のモードは Q-Stop, Q-Ignore, Q-Repeat, Q-Scan を選択できる.

1. アプリケーションから USB-CAMAC ライブラリの cdmaw(mode, naf, &buf, len,&retlen) をコールする.
2. cdmaw() は コマンドに BLOCKW を指定してドライバの ioctl() 関数をコールする.

3. ioctl() 関数では内部にある BLOCKW 処理ブロックへとぶ.
4. BLOCKW 処理ブロックでは CAMAC ファンクション (f) に依存した処理を行なう. この場合は f=16 なので CAMAC データ書き込みブロックへとぶ.
5. CAMAC データ書き込みブロックではデータ長を (len × 2) バイトに指定して scsi_write() 関数をコールする.
6. scsi_write() では send_cdb() により SCSI CDB を 3929 へ送る.
7. send_data() により 3929 へ len で指定した長さのデータを送る.
8. 転送を行なったデータ長を cdmaw() の retlen に返す.

5 USB-CAMAC ライブラリ

USB-CAMAC ドライバのユーザインタフェイスとして USB-CAMAC ライブラリを作成した. 利用者がこのライブラリを使用することで CAMAC 操作に近い記述でプログラミングが可能となる. 本ライブラリは KEK オンライングループの安 芳次氏が書かれた CAMAC ライブラリ [11] を参考にした.

5.1 USB-CAMAC ライブラリ一覧

ライブラリで使用可能な関数を表 8 示す. このライブラリ関数の詳細な説明は「USB-CAMAC デバイスドライバ外部仕様」を参照.

function	Description
camopen	Open CAMAC device
camcls	Close CAMAC device
naf	Set CAMAC NAF Code
cgenz	Generate CAMAC Initialize
cgenc	Generate CAMAC Clear
cseti	Set CAMAC Inhibit
cremi	Remove CAMAC Inhibit
camac	Execute CAMAC Single Action(24bit)
camacw	Execute CAMAC Single Action(16bit)
cdma	Execute CAMAC Block Transfer(24bit)
cdmaw	Execute CAMAC Block Transfer(16bit)

表 8: USB-CAMAC ライブラリで用意されている CAMAC 関数

5.2 USB-CAMAC ライブラリの詳細

USB-CAMAC ライブラリの詳細を説明する. 例として CAMAC 16 ビット・シングルアクションを実行するための camacw() のコードを次に示す.

```
int camacw(int naf, int *data, int *q, int *x)
{
  struct camac_single cam_sing;

  cam_sing.data = *data;
  cam_sing.naf  = naf;

  if(ioctl(cam_fd, CAMACW, &cam_sing) == -1)
    return -1;

  *q = (cam_sing.qx >> 1);
  *x = (cam_sing.qx && 1);
  *data = cam_sing.data;

  return 0;
}
```

上記コードは, CAMAC シングルアクションで使用する構造体 camac_single のメンバの data と naf に値を代入して CAMACW というコマンドを指定して ioctl() を実行する. ioctl() が正常に終了したら ioctl() から返ってきた値を q, x, data に代入する. その他の cdma 等の関数に関しても同様である.

6 性能評価

USB-CAMAC デバイスドライバの性能を評価するため CAMAC シングルアクションと CAMAC ブロック転送の時間を測定した.

6.1 CAMAC シングルアクション

USB-CAMAC ライブラリを使用して CAMAC の シングルアクションの実行時間を測定した. 実行時間の測定方法は USB-CAMAC ライブラリの CAMAC シングルアクションをコールする前後で gettimeofday() 関数により時間情報を取得し, その差からシングルアクションに要する時間を求めた. 測定は CAMAC Switch Register に対してシングルアクションのリード/ライトを各々 10000 回行なった. その結果を表 9 に示す. 表 9 より CAMAC シングルアクションの速度は, 速くないことがわかる. これは USB のデータ転送速度が遅いのではなく転送を行なうまでのセットアップに時間がかかっていることによる. これは 表 9 の 24 ビット データと 16 ビット データで転送時間が同じであることからもわかる.

CAMAC シングルアクション処理時間のより詳しい情報を得るため, デバイスドライバのソースコードに do_gettimeofday() 関数を埋め込み, scsi_read() 関数内部での処理時間の測定も行なった (表 10). 表より send_cdb() 関数に約 3ms, recv_dat() 関数に約 2ms の処理時間を要していることがわか

	24bit data	16bit data
READ(ms)	5	5
WRITE(ms)	5	5

表 9: CAMAC シングルアクション性能

る. send_cdb() では USB 制御転送が recv_dat() では USB バルク転送がそれぞれ実行されている. 他の転送に比べて制御転送はプロトコル上のオーバヘッドが大きいためこの結果は妥当である.

	Execution Time(ms)
send_cdb()	3
recv_dat()	2

表 10: デバイスドライバ内での処理時間の測定結果.

6.2 CAMAC ブロック転送

CAMAC ブロック転送の性能を測定した. CAMAC Memory Module(8K x 16 ビット) にデータ長を変化させながら転送にかかった時間を測定した. そのデータのグラフを図 4 に示す.

測定データから CAMAC 16 ビット ブロック転送の速度とその Overhead を求めた. その結果を表 11 に示す. 表 11 からわかるようにブロック転送の速度は CAMAC 16 ビット データのリードで 920KB/s と高速であることがわかる. またその Overhead は 5ms である.

	16bit data
READ(KB/s)	919
WRITE(KB/s)	923
Overhead(ms)	5

表 11: CAMAC ブロック転送性能

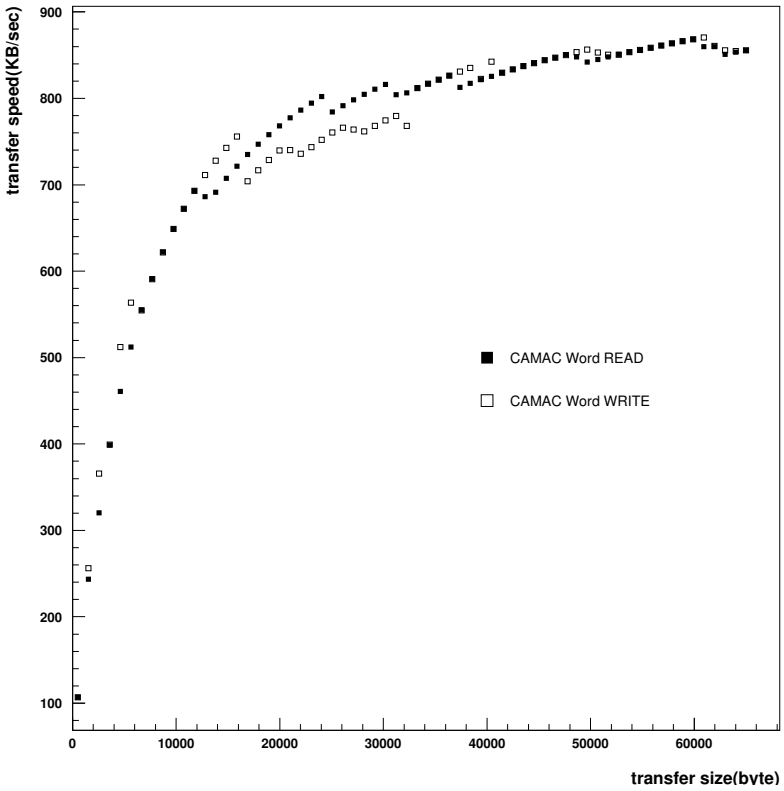

図 4: ブロック転送速度. グラフの横軸はデータ長, 縦軸は転送速度. ■ は CAMAC リード, □ は CAMAC ライトのデータを示している.

謝辞

はじめに今回の技術部専門研修の講師, 安 芳次氏に感謝致します. 充実した講義内容と個別指導により Linux のデバイスドライバ開発およびその周辺の技術・知識を習得することができました. また本研修を計画, 実施して頂いた技術部研修実行委員会委員長の工藤喜久雄氏と委員の皆様に感謝致します. 我々の研修を担当してくださった藤田陽一氏には大変お世話になりました. また今回のデバイスドライバ開発をご支援していただいた藤井啓文氏をはじめとする KEK オンライングループの皆様に感謝致します . 同グループ児玉英世氏の DECstation 用に書かれた 3929 の CAMAC ライブラリのソースコードは大変参考になりました. 藤沢敏喜氏が Linux 関連雑誌に書かれた Linux USB 関連の記事とそのソースコードは本デバイスドライバを開発する上で非常に参考になりました. ここに感謝致します.

参考文献

[1] USB については, 次の USB Implementers Forum のページを参照.
http://www.usb.org/.

[2] John Garney 他, USB Hardware & Software 日本語版, インフォクリエイツ, 1999.

[3] インターフェイス増刊 USB ハード & ソフト開発のすべて, CQ 出版社, 2001.

[4] Model 3929-Z1A SCSI Crate Controller Instruction manual, KineticSystems Corp., 1991.

[5] Small Computer System Interface(日本語訳), 日本規格協会, 1987.

[6] 特集「入出力インターフェイス技法の研究」, インターフェイス 1992 年 5 月号, CQ 出版社.

[7] 藤沢敏喜, 連載「プログラミング工房」, Linux Magazine 2000 年 10-12 月号, アスキー.

[8] USB-SCSI 変換ケーブル LUB-SC については ロジテック 社のホームページを参照.
http://www.logitec.co.jp/.

[9] Alessandro Rubini, Linux デバイスドライバ, オライリージャパン, 1998.

[10] Detlef Fliegl, Programming Guide for Linux USB Device Drivers.
http://usb.cs.tum.edu/usbdoc/

[11] Yoshiji Yasu, Usage Guide of CAMAC Library for UNIX(Version 0.8), 1993.