`eventio` – ein maschinen-unabhängiges, hierarchisches Datenformat und seine Software-Schnittstelle *

K. Bernlöhr

22. März 2005

Inhaltsverzeichnis

*Originaltitel: *Entwurf eines Datenformats für CRT*. In dieser Ausgabe (ab März 1998) ist die Beschreibung CRT-spezifischer Datenstrukturen nicht enthalten. Die allgemeine Beschreibung wurde vom Original (Stand: 17. August 1994) übernommen und für neu hinzugekommene Datentypen und Funktionen ergänzt.

1 Kriterien des Entwurfs

Wegen der unterschiedlichen Architektur der Rechner mit denen CRT-Daten aufgenommen werden (Motorola 680x0 unter OS-9) und der Rechner, unter denen die Daten ausgewertet werden können sollen (DEC Station mit MIPS-Prozessor unter Ultrix, VAX unter VMS, IBM RS/6000 unter AIX, und wer will auch Intel 8086/80286/i386/i486 unter MS-DOS oder einem anderen Betriebssystem, evtl. auch Atari ST/TT oder Apple MacIntosh), ist die erste Bedingung an ein CRT-Datenformat, dass es ‚rechnertransparent' ist. Daten die auf einem Rechner geschrieben wurden, sollten also auf jedem anderen Rechner der aufgeführten Typen wieder gelesen werden können.

Natürlich soll die Ein- und Ausgabe auch möglichst effizient sein. Um damit nicht schon von vorneherein auf einen bestimmten Prozessortyp festgelegt zu sein, sollte das CRT-Datenformat mehrere interne Formate und deren automatische Umwandlung im Bedarfsfall unterstützen. Da sich die internen Datenformate dieser Rechner außer dem Gleitkomma-Format der VAXen nur durch die Byte-Reihenfolge (0-1-2-3 bzw. 3-2-1-0) unterscheiden, genügt es, wenn die beim Schreiben verwendete Byte-Reihenfolge mit ausgegeben wird, sowie bei VAXen einer Umwandlung von Gleitkomma-Zahlen.

Wenn Ausgangs- und Ziel-Rechner verschiedene interne Byte-Reihenfolge haben, so wird in jedem Fall einem dieser Rechner eine Umwandlung aufgezwungen. Da das CRT-Datenformat nun aber beide Reihenfolgen erlaubt, kann die CPU-intensive Umwandlung auf den Rechner verlagert werden, auf dem diese Aufgabe weniger kritisch ist. Werden die Daten dann für die Offline-Verarbeitung auf einem bestimmten Rechnertyp gehalten, wird von da ab zweckmäßig dessen interne Reihenfolge benutzt, wobei die Daten unter geringen Effizienzverlusten jedoch jederzeit noch von einem Rechner anderen Typs gelesen werden können.

Fehler beim Schreiben, Lesen oder Übertragen der Daten sollten nicht zur Unbrauchbarkeit der gesamten weiteren Daten führen. Daher wird jeweils vor einem Datenblock eine 4 Byte lange Synchronisationsmarke geschrieben. Diese Marke kann beim Lesen in beiden Byte-Reihenfolgen erkannt und damit gleichzeitig die Byte-Reihenfolge der Daten bestimmt werden.

Um möglichst flexibel verschiedenartige Datentypen schreiben zu können bzw. bestimmte Daten aus einer Mischung verschiedenartiger Daten herausfiltern zu können, werden Datenblöcke jeweils mit Typ- und Identifikationsnummern versehen. Der Typ gibt dabei das Schema an, nach dem ein Block aufgebaut ist (und damit im Prinzip mit welcher Funktion er zu ‚lesen' ist), die Identifikationsnummer in der Regel die Herkunft der Daten (z.B. Detektor/Driftkammer/Draht etc.) Um auf nachträgliche Änderungen an einzelnen Typen von Datenblöcken vorbereitet zu sein, sind die Blöcke zusätzlich noch mit einer Versionsnummer ausgestattet. Um Blöcke, an denen man nicht interessiert ist, einfach überspringen zu können, kommt noch eine Längenangabe hinzu.

Um selbst bei der Zusammensetzung von Blöcken bestimmten Typs flexibel zu sein, wird eine hierarchische Struktur eingeführt, die aus einer Schachtelung von Blöcken und Unter-Blöcken besteht, und an deren unterem Ende die ‚elementaren' Datentypen stehen. Wegen der flexiblen Struktur ist es auch eher angemessen von Datenobjekten (*objects*) oder Informationseinheiten (*items*) zu sprechen als von Blöcken, da der Begriff ‚Block' doch eine sehr feste und einfache Struktur impliziert. Solange ein Objekt nur aus Unterobjekten besteht, nicht aber aus elementaren Datentypen, kann dessen Zusammensetzung (Inhaltsverzeichnis) angegeben werden, ohne wissen zu müssen wie einzelne Objekte zusammengesetzt sind. Unterobjekte können gesucht oder übersprungen werden, und selbst eine nachträgliche Löschung aus dem Baum von Objekten ist möglich ohne die Konsistenz der Gesamtstruktur zu gefährden. Nur dem Objekt ganz oben in der Hierarchie, dem ‚Hauptobjekt' bzw. ‚Stammobjekt' (im Baum-Bild is leider oben und unten vertauscht) wird die Synchronisationsmarke vorangestellt. Im Prinzip können sogar verschiedene Hauptobjekte verschiedene Byte-Reihenfolgen haben, was allerdings aus Effizienzgründen die Ausnahme sein sollte.

Aus Effizienzgründen wird auch jedes Objekt beim ‚Schreiben' auf eine 4-Byte-Grenze aufgefüllt. Damit wird angestrebt, dass ein Objekt auch möglichst auf einer solchen Grenze beginnt, und damit möglichst selten Zugriffe auftreten, die Wortgrenzen überschneiden und bei vielen CPU-Typen bzw. Speicher-Architekturen deutlich langsamer ablaufen als auf Wortgrenzen abgestimmte Zugriffe. Das setzt natürlich voraus, dass auch innerhalb der Objekte die Ausrichtung erhalten bleibt. Wenn ein Objekt aus einer Mischung von elementaren und Unterobjekten besteht, ist die optimale Ausrichtung der Unterobjekte nicht garantiert. Außer gewissen Effizienzverlusten sind aber auch bei falscher Ausrichtung alle größeren Nebeneffekte (etwa bei Rechnern, die bei Festkomma- oder Gleitkommazahlen nicht auf ungerade Speicheradressen zugreifen können) durch die Software zu verhindern.

2 Datentypen und Datenstrukturen

2.1 Elementare Datentypen

Als elementare Datentypen gelten Datentypen, wie sie auch in den Programmiersprachen C und FORTRAN als elementare Typen vorkommen. Davon sind implementiert:

Typ	Länge (Bytes)	Beschreibung
Byte	1	Buchstabe oder sehr kurze ganze Zahl
Count[1]	1 to 9	Natürliche Zahl (ohne Vorzeichen)
SCount[2]	1 to 9	Natürliche Zahl (mit Vorzeichen)
Short	2	Kurze ganze Zahl (mit oder ohne Vorzeichen)
Long	4	Lange ganze Zahl (mit oder ohne Vorzeichen)
Real	4	32-Bit-Gleitkommazahl im IEEE-Format
Double	8	64-Bit-Gleitkommazahl im IEEE-Format
String	2+Länge	Zeichenkette mit vorangehender Längenangabe
Long String	4+Länge	Zeichenkette mit vorangehender Längenangabe
Var String[2]	var.+Länge	Zeichenkette mit vorangehender Längenangabe

Daneben werden auf Maschinen mit Unterstützung von 64 Bit großen, ganzen Zahlen, solche als *Long64* Datentyp geschrieben und gelesen. Das schließt Systeme ein, wo diese 64-Bit-Zahlen nur über den Kompiler implementiert sind (z.B. als `long long`). Dabei ist aber zu beachten, dass diese Zahlen dann auf Maschinen ohne 64-Bit-Integer-Unterstützung nicht ohne spezielle Vorkehrungen (z.B. Lesen in zwei 32-Bit-Zahlen) wieder gelesen werden können. Auch bei den Count- und SCount-Elementen können auf Maschinen mit 64-Bit-Unterstützung Werte geschrieben werden, die auf Maschinen ohne 64-Bit-Unterstützung nicht wieder korrekt gelesen werden können.

Die elementaren Daten sind generell ohne explizite Typen- und Längenangaben. Ein lesendes Program muß also 'wissen', wie die Daten zu lesen sind.

Die Synchronisationsmarke wird als *Long* geschrieben und gelesen. Wird die Marke als 0x378A1FD4 statt 0xD41F8A37 gelesen, so muß die Byte-Reihenfolge umgedreht werden. Die Reihenfolge bei *Short* und *Real* wird damit ebenfalls festgelegt. Bei *Strings* wird die Länge im Bereich von 0 bis 32767 der Zeichenkette als Short vorangestellt. Strenggenommen müßten alle Festkommatypen (Byte, Short, Long) nach vorzeichenbehafteten und vorzeichenlosen Typen unterschieden werden.

[1] Seit September 2003.
[2] Seit Dezember 2003.

Solange die Zahlen jeweils wieder in den selben internen Typ eingelesen werden, aus dem sie stammen, sollten durch diese Vereinfachung aber keine Übertragungsfehler auftreten, da alle in Betracht gezogenen CPU-Typen negative Zahlen (bis auf die Byte-Reihenfolge wieder) in der selben Art darstellen.

2.2 Objektstruktur

Die elementaren Daten werden zu Objekten oder Einheiten zusammengefaßt, die wiederum hierarchisch geschachtelt sein können. Ein- und Ausgabe, Festlegung der Byte-Reihenfolge sowie Synchronisation bei Datenfehlern erfolgt auf der jeweils obersten Ebene, dem Hauptobjekt. Die Struktur von Hauptobjekt und Unterobjekten unterscheidet sich nur insofern als dem Hauptobjekt die 4-Byte-Synchronisationsmarke in der einen oder anderen Byte-Reihenfolge vorangestellt ist. Als Synchronisationsmarke dient die 4-Byte-Festkommazahl 0xD41F8A37 (in C-Schreibweise, ist 3558836791 bei Interpretation als ‚unsigned long' bzw. -736130505 als ‚long'). Ansonsten besteht jedes Objekt aus einem Kopf aus drei 4-Byte-Festkommazahlen, die die *Typ/Versions-*, *Identifikations-* und *Längen-* Felder darstellen. Der Rumpf besteht aus dem *Datenfeld*.

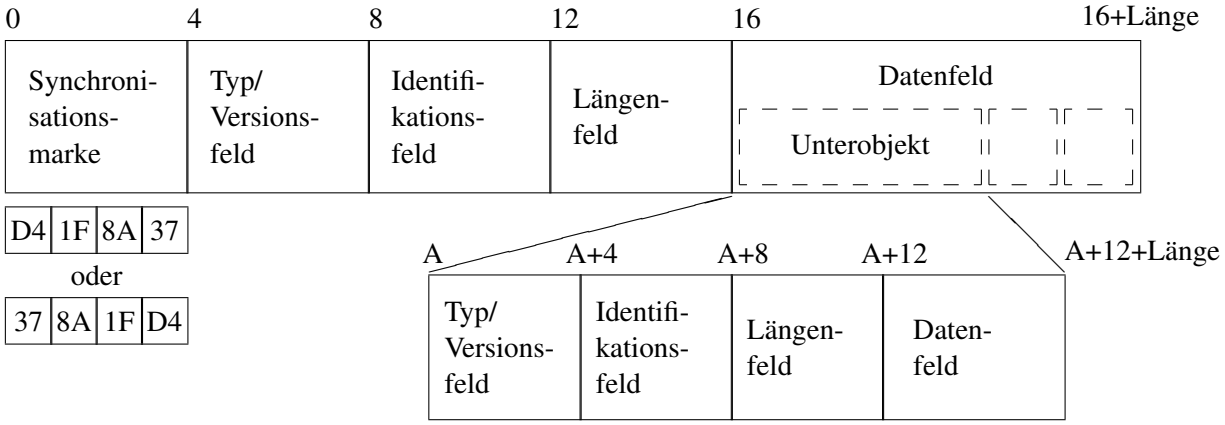

Abbildung 1: Schematische Darstellung eines Hauptobjekts und eines Unterobjekts

Das Typ/Versions-Feld kombiniert eine Typennummer in den Bits 0 bis 15 (Wertebereich 0 bis 65535) und eine Versionsnummer in den Bits 20 bis 31 (Wertebereich 0 bis 4095). Bits 16 bis 19 sind reserviert für künftige Erweiterungen (evtl. einer Erweiterung des Typs auf einen Wertebereich von 0 bis 1048575) und müssen auf 0 gesetzt sein, solange diese Definition nicht geändert wird.

Das Identifikationsfeld soll zur Identifikation dienen, woher die Daten des Objekts stammen (z.B. aus einer Kombination von Detektor-, Kammer- und Drahtnummer oder eine Histogramm-Nummer) und hat daher eine objekt-spezifische Bedeutung. Eine Identifikationsnummer -1 wird als ‚keine Identifikationsnummer angegeben' interpretiert.

Das Längenfeld enthält in Bits 0 bis 29 die Länge des Datenfelds in Bytes, einschließlich 0 bis 3 Füllbytes um am Ende auf eine 4-Byte-Adresse aufzufüllen (Wertebereich 0 bis 1073741823, sollte durch 4 teilbar sein). Bit 30 ist gesetzt, wenn das Objekt nur Unterobjekte enthält, aber keine elementaren Datentypen, also ein Inhaltsverzeichnis angegeben werden kann ohne den spezifischen Aufbau des jeweiligen Typs zu kennen. Bit 31 ist reserviert und muß auf 0 gesetzt sein.

Das Datenfeld kann aus Unterobjekten oder aus elementaren Datentypen bestehen. Besteht es aus elementaren Datentypen (oder einer Kombination von elementaren Datentypen und Unterobjekten), so kann es nur durch eine für den jeweiligen Typ spezifische Funktion ausgepackt werden.

Besteht es dagegen nur aus Unterobjekten (und ist Bit 30 des Längenfelds gesetzt), so kann der Inhalt gelistet werden und nach einzelnen Unterobjekten gesucht werden, ohne den spezifischen Aufbau des Datenfelds zu kennen.

3 Implementierung und Programmierung in C

3.1 Überblick

In der Implementierung des beschriebenen Datenformats werden die Hauptobjekte jeweils komplett im Speicher zusammengebaut bevor sie geschrieben werden bzw. komplett in den Speicher gelesen, bevor sie entpackt werden. Und zwar jeweils genau ein Hauptobjekt in einem Ein-/Ausgabepuffer. Die Funktionen können in fünf Gruppen getrennt werden:

1. Speicherverwaltung der Ein-/Ausgabepuffer,
2. Ein- und Ausgabe, Suche und Überspringen von Hauptobjekten,
3. Verwaltung von Objekten in einem Ein-/Ausgabepuffer,
4. Schreiben/Lesen von elementaren Datentypen in/aus einem Puffer, und
5. Schreiben/Lesen von kompletten Objekten/Objekthierarchien (typ-spezifisch).

Bei der Verwaltung der Ein-/Ausgabepuffer und der Objekte treten zwei Typen von Strukturen auf, von denen im folgenden nur die Elemente gelistet werden, auf die Anwenderfunktionen zugreifen sollten bzw. diese modifizieren sollten:

```
struct _struct_IO_BUFFER
{
   ...
   int input_fileno;
   int output_fileno;
   FILE *input_file;
   FILE *output_file;
   int (*user_function)(unsigned char *, long, int);
   ...
};
typedef struct _struct_IO_BUFFER IO_BUFFER;

struct _struct_IO_ITEM_HEADER
{
   ...
   unsigned long type;
   unsigned version;
   long ident;
   ...
};
typedef struct _struct_IO_ITEM_HEADER IO_ITEM_HEADER;
```

Bei der IO_BUFFER-Struktur stellen die Elemente input_fileno und output_fileno Dateinummern dar, die man beim Öffnen der Ein- bzw. Ausgabedatei mittels open() oder durch Anwendung von fileno() auf eine vorhandene FILE-Struktur erhält, oder aber die Nummer der Standard-Eingabe (0) bzw. -Ausgabe (1) sein kann. Bei Allokierung einer IO_BUFFER-Struktur werden diese Werte auf -1 (nicht vorhanden) gesetzt. Wird nur Eingabe gemacht, braucht auch nur input_fileno gesetzt werden, und entsprechend nur output_fileno bei reiner Ausgabe.

Statt Ein- und Ausgabe über die Basis-Ein-/Ausgabe-Funktionen open, read, write und close können auch die 'Stream' I/O-Funktionen fopen, fread, fwrite und fclose benutzt werden, wenn statt den Elementen input_fileno bzw. output_fileno die Elemente input_file bzw. output_file mit einem Zeiger auf eine entsprechende FILE-Struktur gesetzt werden. Insbesondere bei Daten mit kleinen Blöcken bringt die interne Pufferung der Stream I/O-Funktionen merkliche Leistungsgewinne.

Soll statt Ein- und Ausgabe in Dateien eine andere E/A-Form benutzt werden, so kann dazu der Funktionszeiger user_function auf eine benutzereigene Funktion entsprechenden Typs gesetzt werden, die die Ein- und Ausgabe übernehmen soll.

In der jeweiligen IO_ITEM_HEADER-Struktur sind vor dem Anlegen neuer Objekte jeweils die Typnummer (type), die Versionsnummer (version) und die Identifikationsnummer (ident) zu setzen. Vor dem Suchen und Lesen ist nur die Typnummer anzugeben, die anderen Größen werden anhand der gefundenen Daten gesetzt.

3.2 Verwaltung von Ein-/Ausgabepuffern

```
IO_BUFFER *allocate_io_buffer(size_t length);
```
Damit wird ein E/A-Puffer angelegt, der anfänglich bis zu length Bytes aufnehmen kann, bei Bedarf aber vergrößert wird. Anschließend sind als erstes dort die Dateinummern (oder die Anwenderfunktion) einzutragen.

Beispiel: `iobuf = allocate_io_buffer(20000);`

```
void free_io_buffer(IO_BUFFER *iobuf);
```
Ein E/A-Puffer wird damit wieder freigegeben. Es ist zu beachten, dass die zugehörigen Dateien dadurch nicht geschlossen werden.

Beispiel: `free_io_buffer(iobuf);`

3.3 Ein- und Ausgabefunktionen

```
int write_io_block (IO_BUFFER *iobuf);
```
Nach Fertigstellung eines Hauptobjektes und aller darin enthaltenen Daten, einschließlich Unterobjekten wird der Inhalt des Puffers in die durch iobuf->output_fileno oder iobuf->output_file gekennzeichnete Datei geschrieben, oder, falls dieses nicht gesetzt ist aber iobuf->user_function, wird die Schreibaufgabe der Anwenderfunktion übertragen.

Beispiel:
```
if ( (rc = write_io_block(iobuf)) != 0 )
            return rc;
```

```
int find_io_block (IO_BUFFER *iobuf, IO_ITEM_HEADER *item_header);
```
In der durch iobuf->input_fileno oder iobuf->input_file gekennzeichneten Datei wird nach der Synchronisationsmarke gesucht (normalerweise die nächsten 4 zu lesenden Bytes), der Kopf des gefundenen Hauptobjekts in den E/A-Puffer eingelesen und die beschriebenen Elemente von item_header entsprechend gesetzt.

`int read_io_block (IO_BUFFER *iobuf, IO_ITEM_HEADER *item_header);`
Damit wird das Datenfeld eines zuvor mit `find_io_block` begonnenen Hauptobjekts eingelesen und der weiteren Verarbeitung zugänglich gemacht.

`int skip_io_block (IO_BUFFER *iobuf, IO_ITEM_HEADER *item_header);`
Diese Funktion dient dazu das Datenfeld eines zuvor mit `find_io_block` begonnenen Hauptobjekts zu überspringen, z.B. wenn man an dem betreffenden Typ von Objekt nicht interessiert ist.

Beispiel:

```
IO_ITEM_HEADER item_header;
...
if ( find_io_block(iobuf,&item_header) != 0 )
   printf("Fehler!\n");
else if ( item_header.type == 2540 )
{
   if ( read_io_block(iobuf,&item_header) != 0 )
      printf("Auch ein Fehler!\n");
   ...
}
else
   (void) skip_io_block(iobuf,&item_header);
```

`int list_io_blocks (IO_BUFFER *iobuf);`
Hier handelt es sich bereits um eine einfache Anwendung, die auf den eben beschriebenen Funktionen basiert und eine Liste aller in der Eingabedatei gefundenen Hauptobjekte erstellt und auf der Standard-Ausgabe ausgibt. Ein vollständiges, wenn auch sehr einfaches Anwendungsprogramm für ein Inhaltsverzeichnis der als Standard-Eingabe eingelesenen Daten sieht dann folgendermaßen aus:

```
#include "eventio2.h"
int main()
{
   IO_BUFFER *iobuf;
   if ( (iobuf = allocate_io_buffer(1000)) ==
        (IO_BUFFER *) NULL )
      exit(1);
   iobuf->input_fileno = 0;
   (void) list_io_blocks(iobuf);
   exit(0);
   return 0;
}
```

Wenn die Ein- und Ausgabe nicht in Dateien oder zumindest nicht mit `read` und `write` erfolgen soll, kann nach Allokierung des E/A-Puffers `iobuf` dem Element `iobuf->user_function` die Adresse einer benutzereigenen Funktion zugewiesen werden, die die reine Ein- und Ausgabe erledigt. Die Elemente `iobuf->input_fileno` und `iobuf->output_fileno` dürfen dann nicht gesetzt werden. Die benutzereigene Funktion muß dem folgenden Prototyp entsprechen:

```
int function (unsigned char *buffer, long bytes, int code);
```
Dabei legt `code` die vorzunehmende Handlung (Unterfunktion) fest:

`code=1`**:** Schreiben von `bytes` Bytes ab der Adresse `buffer`.

`code=2`**:** Finden des nächsten E/A-Blocks (Hauptobjekts) und Einlesen des Kopfs dieses Hauptobjekts (16 Bytes) an die mit `buffer` angegebene Adresse.

`code=3`**:** Einlesen des Datenfelds des zuvor gefundenen Hauptobjekts. Dabei ist `bytes` die Länge des Datenfelds in Bytes und `buffer` ist die Adresse des Beginns des Datenfelds (Kopf plus 16 Bytes) im E/A-Puffer.

`code=4`**:** Überspringen des Datenfelds der Länge `bytes` des zuvor gefundenen E/A-Blocks (des Hauptobjekts). `buffer` darf hier nicht verwendet werden, auch nicht als Zwischenspeicher, da zum Überspringen eines großen Blocks nicht soviel Speicher allokiert wird, damit dieser Block auch in den Puffer gelesen werden könnte.

Die benutzereigene Funktion muß bei korrekter Ausführung den Wert 0 zurückgeben, bei Auftreten eines Fehlers (außer Dateiende beim Lesen oder dessen Entsprechung) den Wert -1, und bei Datei- oder Datenende beim Lesen (Unterfunktionen 2–4) den Wert -2.

3.4 Verwaltung der Datenobjekte

Um sicherzustellen, dass die Datenobjekte korrekt verwaltet werden, müssen beim Erstellen die Funktionen `put_item_begin` bzw. `put_item_end` vor bzw. nach dem Schreiben der eigentlichen Daten in den E/A-Puffer aufgerufen werden, also jeweils paarweise und auf das selbe Objekt bezogen. Entsprechend sind beim Entpacken (Lesen aus dem Puffer) die Funktionen `get_item_begin` und `get_item_end` zu verwenden, wobei zum Suchen, Löschen, Zurückgehen und Listen aber noch weitere Funktionen vorhanden sind. Alle Funktionen liefern bei ordnungsgemäßer Ausführung die Zahl 0 zurück, im Falle eines Fehlers aber eine negative Zahl.

```
int put_item_begin (IO_BUFFER *iobuf, IO_ITEM_HEADER *item_header);
int put_item_end   (IO_BUFFER *iobuf, IO_ITEM_HEADER *item_header);
int get_item_begin (IO_BUFFER *iobuf, IO_ITEM_HEADER *item_header);
int get_item_end   (IO_BUFFER *iobuf, IO_ITEM_HEADER *item_header);
int unget_item     (IO_BUFFER *iobuf, IO_ITEM_HEADER *item_header);
int rewind_item    (IO_BUFFER *iobuf, IO_ITEM_HEADER *item_header);
int remove_item    (IO_BUFFER *iobuf, IO_ITEM_HEADER *item_header);
int search_sub_item(IO_BUFFER *iobuf, IO_ITEM_HEADER *item_header,
                    IO_ITEM_HEADER *sub_item_header);
int list_sub_items (IO_BUFFER *iobuf, IO_ITEM_HEADER *item_header,
                    int maxlevel);
```

Beispiele von Funktionen, mit denen jeweils ein Objekt geschrieben bzw. gelesen wird, sind weiter hinten angegeben. Dabei werden nur die `put_item_...` und `get__item_...` Funktionen benutzt. Die `put_item_...` Funktionen sollten nicht mit den anderen Funktionen benutzt werden, d.h. gemischtes Lesen/Suchen und Schreiben/Anhängen/Einfügen werden bisher nicht unterstützt, sind aber beabsichtigt.

Nach einem erfolgreichen Aufruf von `get_item_begin` oder `search_sub_item` steht der Datenzeiger jeweils am Beginn des Datenfelds des jeweiligen Objekts oder Unterobjekts. Diese

Daten können nun gelesen werden oder das Objekt durch `get_item_end` übersprungen werden. In aller Regel sollte zu jedem Aufruf von `put_item_begin` und `get_item_begin`, genau ein entprechender Aufruf von `put_item_end` bzw. `get_item_end` gehören, aber notfalls schließt das Ende eines übergeordneten Objekts auch das Ende all seiner Unterobjekte ein. Mit dem `put_item_end` Aufruf für das Hauptobjekt wird dieses auch in die Ausgabedatei geschrieben.

Schlägt `get_item_begin` fehl, steht der Datenzeiger an der alten Stelle; schlägt dagegen `search_sub_item` fehl, steht er am Ende des Objekts, in dem vergeblich nach einem bestimmten Unterobjekt gesucht wurde. Der Datenzeiger kann dann mit `rewind_item` auf den Beginn des Datenfelds dieses Objekts zurückgesetzt werden. Um gar vor den Beginn des Kopfs eines Objekts zurückzusetzen, kann die Funktion `unget_item` benutzt werden. Der Löschfunktion `remove_item` muß jeweils ein erfolgreicher Aufruf von `get_item_begin` oder `search_sub_item` vorausgehen, aber kein Aufruf von `get_item_end` folgen. `list_sub_items` gibt ähnlich, wie `list_io_blocks` ein Inhaltsverzeichnis der obersten Objektebene der gesamten Daten liefert, ein Verzeichnis der Unterobjekte eines Hauptobjekts oder Unterobjekts (je nachdem wo der Datenzeiger gerade steht). Hat das Argument `maxlevel` einen positiven Wert, werden nur Objekte bis zur entsprechenden Schachtelungstiefe aufgeführt. Nach `list_sub_items` steht der Datenzeiger jeweils am Ende des betreffenden Objekts.

3.5 Verwendung elementarer Datentypen

Für jeden der elementaren Datentypen gibt es jeweils eine Funktion um ein Element in den E/A-Puffer zu schreiben und eine Funktion um ein Element zu lesen. Die Byte-Operationen sind als Makros implementiert. Außer bei Strings gibt es aus Effizienzgründen auch Funktionen, um ganze Vektoren solcher Element zu schreiben bzw. zu lesen. Dabei fällt der Verwaltungsaufwand nur einmal an anstatt für jedes Element. Da bei den Vektor-Funktionen aber keine automatische Umwandlung des Arguments bzw. Rückgabewerts erfolgt, ist hier strikt darauf zu achten, dass jeweils Vektoren des richtigen Typs übergeben werden. Insbesondere bei den Funktionen `put_vector_of_short` und `put_vector_of_int` und deren Umkehrfunktionen ist zu beachten, dass beide für ganzzahlige Werte im Bereich -32768 bis 32767 bzw. 0 bis 65535 bestimmt sind (je nachdem, ob der Wert als *signed* oder *unsigned* Typ aufgefaßt wird), dass aber die Elemente der übergebenen Vektoren auf vielen Rechnern unterschiedliche Länge haben (2 bzw. 4 Bytes).

Wenn die Daten intern in Variablen vorgegebener Länge gespeichert werden sollen, sollten die in ANSI C99 definierten Typen `int8_t`, `uint8_t`, ..., `int64_t`, `uint64_t` verwendet werden, welche nach Einbinden von `io_basic.h` auch für ältere Compiler zur Verfügung stehen sollten. Beachten Sie dass die Funktionen mit 64-Bit-Zahlen nicht auf allen Maschinen zur Verfügung stehen.

Elementarfunktionen:

```
int  put_byte (unsigned char c, IO_BUFFER *iobuf);
int  get_byte (IO_BUFFER *iobuf);
void put_count (uintmax_t i, IO_BUFFER *iobuf);
uintmax_t get_count (IO_BUFFER *iobuf);
void put_scount (intmax_t i, IO_BUFFER *iobuf);
intmax_t get_scount (IO_BUFFER *iobuf);
void put_short (int i, IO_BUFFER *iobuf);
int  get_short (IO_BUFFER *iobuf);
void put_int32(int32_t num, IO_BUFFER *iobuf)
int32_t get_int32(IO_BUFFER *iobuf)
```

```
void put_uint32(uint32_t num, IO_BUFFER *iobuf)
uint32_t get_uint32(IO_BUFFER *iobuf)
void put_long (long l, IO_BUFFER *iobuf);
long get_long (IO_BUFFER *iobuf);
void put_string (char *s, IO_BUFFER *iobuf);
int  get_string (IO_BUFFER *iobuf);
int put_long_string (char *s, IO_BUFFER *iobuf)
int get_long_string (char *s, int nmax, IO_BUFFER *iobuf)
int put_var_string (char *s, IO_BUFFER *iobuf)
int get_var_string (char *s, int nmax, IO_BUFFER *iobuf)
void put_real (double d, IO_BUFFER *iobuf);
double get_real (IO_BUFFER *iobuf);
void put_double (double dnum, IO_BUFFER *iobuf)
double get_double (IO_BUFFER *iobuf)
```

Vektorfunktionen:

```
void put_vector_of_byte (unsigned char *cv, int num, IO_BUFFER *iobuf);
void get_vector_of_byte (unsigned char *cv, int num, IO_BUFFER *iobuf);
void put_vector_of_uint8 (uint8_t *cv, int num, IO_BUFFER *iobuf);
void get_vector_of_uint8 (uint8_t *cv, int num, IO_BUFFER *iobuf);
void put_vector_of_short (short *sv, int num, IO_BUFFER *iobuf);
void get_vector_of_short (short *sv, int num, IO_BUFFER *iobuf);
void put_vector_of_int (int *iv, int num, IO_BUFFER *iobuf);
void get_vector_of_int (int *iv, int num, IO_BUFFER *iobuf);
void put_vector_of_int16 (int16_t *sv, int num, IO_BUFFER *iobuf);
void get_vector_of_int16 (int16_t *sv, int num, IO_BUFFER *iobuf);
void put_vector_of_uint16 (uint16_t *uval, int num, IO_BUFFER *iobuf)
void get_vector_of_uint16 (uint16_t *uval, int num, IO_BUFFER *iobuf)
void put_vector_of_int32 (int32_t *vec, int num, IO_BUFFER *iobuf)
void get_vector_of_int32 (int32_t *vec, int num, IO_BUFFER *iobuf)
void put_vector_of_uint32 (uint32_t *vec, int num, IO_BUFFER *iobuf)
void get_vector_of_uint32 (uint32_t *vec, int num, IO_BUFFER *iobuf)
void put_vector_of_int64 (int64_t *ival, int num, IO_BUFFER *iobuf)
void get_vector_of_int64 (int64_t *ival, int num, IO_BUFFER *iobuf)
void put_vector_of_uint64 (uint64_t *uval, int num, IO_BUFFER *iobuf)
void get_vector_of_uint64 (uint64_t *uval, int num, IO_BUFFER *iobuf)
void put_vector_of_long (long *lv, int num, IO_BUFFER *iobuf);
void get_vector_of_long (long *lv, int num, IO_BUFFER *iobuf);
void put_vector_of_real (double *dv, int num, IO_BUFFER *iobuf);
void get_vector_of_real (double *dv, int num, IO_BUFFER *iobuf);
void put_vector_of_float (float *fvec, int num, IO_BUFFER *iobuf)
void get_vector_of_float (float *fvec, int num, IO_BUFFER *iobuf)
void put_vector_of_double (double *dvec, int num, IO_BUFFER *iobuf)
void get_vector_of_double (double *dvec, int num, IO_BUFFER *iobuf)
```

3.6 Objekttyp-spezifische Funktionen

Als objekttyp-spezifische Funktionen sind für jeden ‚eigenständigen' Objekttyp eine Schreib- und eine Lesefunktion erforderlich (in den E/A-Puffer bzw. aus dem E/A-Puffer heraus). Diese Funktionen sollten die Konsistenz der Argument überprüfen und Fehlerbedingungen bei der Verwaltung von Objekten oder den Ein- und Ausgabepuffern an die aufrufende Funktion weitergeben. Als besonders einfache Beispiele sind nachfolgend die Funktionen zum Lesen und Schreiben der ADC-Nullpunkte vollständig aufgeführt. Bei den komplexeren Funktionen wird in aller Regel ein Zeiger auf eine entsprechende interne Datenstruktur und die Anzahl der zu schreibenden Elemente übergeben. Der Zeiger auf die E/A-Puffer-Struktur ist immer an die objekttyp-spezifische Funktion zu übergeben.

```
#include "initial.h"      /* This file includes others as required. */
#include "eventio2.h"     /* This file includes others as required. */

/* ---------------------- write_offsets ---------------------- */
/**
 *  @short Writes the measured FADC pedestals ('offsets').
 *
 *  The data will usually be the global 'cmpoffset'.
 *
 *  @param  offsets   Vector of FADC pedestals
 *  @param  nchannels No. of channels (no. of 'offsets')
 *  @param  iobuf     The output data iobuf.
 *
 *  @return  0 (O.k.) or -1 (error)
 *
 */

int write_offsets (long *offsets, int nchannels, IO_BUFFER *iobuf)
{
   IO_ITEM_HEADER item_header;

   if ( iobuf == (IO_BUFFER *) NULL )
      return -1;
   if (offsets == (long *) NULL )
   {
      Warning("Attempt to save invalid offset vector");
      return -1;
   }

   item_header.type = 21;               /* Offset data is type 21 */
   item_header.version = 1;             /* Version 1 (preliminary) */
   item_header.ident = get_detector_number();
   put_item_begin(iobuf,&item_header);

   /* Save the time when the data is written although it would */
   /* be better to save the time when the offsets were measured. */
   put_long(time((time_t *) NULL),iobuf);
```

```
   put_short(nchannels,iobuf);          /* No. of FADC channels */
   put_vector_of_long(offsets,nchannels,iobuf);

   return(put_item_end(iobuf,&item_header));
}

/* ------------------------ read_offsets ------------------------ */
/**
 *  @short Reads (previously measured) ADC offset values.
 *
 *  The destination ('offsets' parameter) will usually be the
 *  global 'cmpoffset' data.
 *
 *  @param offsets    The destination of the data being read.
 *  @param nchannels  The expected no. of channels, must match
 *                    the actual no. specified in the input.
 *  @param iobuf      The input data iobuf descriptor.
 *
 *  @return  0 (O.k.),  -1 (error),  or  -2 (e.o.f.)
 *
 */

int read_offsets (long *offsets, int nchannels, IO_BUFFER *iobuf)
{
   IO_ITEM_HEADER item_header;
   int rc;

   if ( offsets == (long *) NULL )
   {
      Warning("Invalid call to read_offsets()");
      return -1;
   }

   if ( iobuf == (IO_BUFFER *) NULL )
      return -1;
   item_header.type = 21;
   if ( (rc = get_item_begin(iobuf,&item_header)) != 0 )
      return(rc);

   if ( item_header.version != 1 )
   {
      Warning("Wrong version no. of offset data to be read");
      return -1;
   }

   (void) get_long(iobuf);    /* Time is ignored, so far. */

   if ( nchannels != get_short(iobuf) )
```

```
   {
      Warning("Wrong no. of channels for reading offsets");
      return -1;
   }

   get_vector_of_long(offsets,nchannels,iobuf);

   return(get_item_end(iobuf,&item_header));
}
```

Die hier besonders ausgezeichneten Kommentare am Beginn der Funktionen sind zur automtischen Erstellung von Dokumentation mit dem Programm `doxygen` bestimmt.

4 Implementierung und Programmierung in C++

(Vorläufig nur englische Fassung hier)

4.1 Introduction

In addition to the C language basic implementation and API, there is an additional application programming interface in C++ which takes care of hiding many of those things the C programmer has to write explicitly. The C++ API also provides a much cleaner and simpler – even though richer and more powerfull – collection of methods to get data from or put data to the internal I/O buffer. Among the added benefits are support for standard library vectors, valarrays and strings.

You should be aware that you may have to link with a different library for the C++ API than for the C API.

4.2 Example code

With the C++ API, a simple program could look like:

```
#include "EventIO.hh"
using namespace eventio;

int main()
{
   int32_t data[2] = { 17, 10815 };
   EventIO iobuf;
   iobuf.OpenOutput("output.file");
   EventIO::Item item(iobuf,"put",99,0,123);
   if ( item.Status() != 0 )
      return 1;
   item.PutInt32(data,2);
   item.Done();
   iobuf.CloseOutput();
   return 0;
}
```

This simple program should write an item of type number 99, version number 0, and ID number 123 to a file. In this case, the item only contains two integers. For a nested item tree, it could look like:

```
#include "EventIO.hh"
using namespace eventio;

int main()
{
   int32_t data1[2] = { 17, 10815 };
   std::string data2("some text");
   EventIO iobuf;
   iobuf.OpenOutput("output.file");
   EventIO::Item item(iobuf,"put",101);
   if ( item.Status() != 0 )
      return 1;

   EventIO::Item item1(item,"put",102);
   item1.PutInt32(data1,2);
   item1.Done();

   EventIO::Item item2(item,"put",103);
   item2.PutString(data2);
   item2.Done();

   item.Done();
   iobuf.CloseOutput();
   return 0;
}
```

4.3 The EventIO and EventIO::Item classes

All of the internals of the `IO_BUFFER` are hidden in the C++ class `EventIO` which, in its constructor and destructor, takes care of the essential initialisation and cleanup. The EventIO class provides methods to open and close input files or functions, locating top-level blocks in the input stream and reading them.

The `EventIO::Item` sub-class is used to get all the items and data into the buffer or get them from the buffer. If the `EventIO::Item` constructor is called with an `EventIO` argument, a top-level item will be created (after finishing any incomplete operations of the buffer). If it is called with a `EventIO::Item` argument, the new item will be a sub-item of the given argument. The whole tree of such items can be created either for `"get"` or for `"put"` operations.

4.4 EventIO methods

The methods provided by the EventIO class can be subdivided into two groups:

1. Methods for connection input/output files or functions. These are mainly the `OpenInput` and `OpenOutput` methods (which take either a file name or a `FILE *` pointer as their argument), the `OpenFunction` method taking a `IO_USER_FUNCTION` argument, as well

as the corresponding `CloseInput`, `CloseOutput`, and `CloseFunction` methods (all without arguments). All of them return 0 on success and -1 or other negative numbers on failure.

2. The `Find()`, `Read()`, `Skip()`, and `List()` methods correspond to the `find_io_block`, `read_io_block`, `list_io_block`, and `list_io_block` C functions. After a `Find()` or `Read()`, you can also use the `ItemType()`, `ItemVersion()`, and `ItemIdent()` methods to inspect the properties of the current top-level item.

For low-level access to C methods, the underlying `IO_BUFFER` can be obtained through the `Buffer()` method.

4.5 Data access methods

For an `EventIO::Item` constructed with a `"get"` argument, data can be retrieved with the `GetXyz(...)` type functions. For an `EventIO::Item` constructed with a `"put"` argument, data can be written to the buffer with the `PutXyz(...)` type functions. In either case, an item should normally be finished by the `Done()` method but will be finished also by the end of its lifetime or by conflicting operations on its parent item, like creating a new `EventIO::Item` item at the same level as the current item or higher up in the hierarchy, or calling the `Done()` method at a higher level. The `Xyz` part here stands for the different data types:

- `Uint8` (Byte in C API)
- `Count`
- `SCount`
- `Int16` (Short in C API)
- `UInt16`
- `Int32` (Long in C API)
- `UInt32`
- `Int64`
- `Uint64`
- `Real`
- `Double`
- `String` (VarString in C API)

There are no methods directly corresponding to the older C data type String and LongString but these are easy to emulate where compatibility with older C code is required (see the `TestIO` program, for example).

All the `GetXyz(...)` and `PutXyz(...)` type functions (with exception of strings) come in a number of flavours:

- for single values,

- for arrays of values,
- for `vector` of values, and
- for `valarray` of values.

With `vector`, `valarray`, as well as with the special `string` methods, the corresponding header files must be included **before** including `EventIO.hh`. In addition, the `vector` and `valarray` methods come in two variants:

- `PutXyz(vec, nelem);`
- `PutXyz(vec);`

The first variant variant is analogous to the array flavour and only writes the contents of the array/vector/valarray. It assumes that the length of it is stored separately. The second variant, not available with arrays, will first put the number of elements to be stored into the buffer (as a `Count` type), and then all of its data elements. It is therefore the same if you write

```
PutCount(vec.size());
PutXyz(vec,vec.size());
```

or

```
PutXyz(vec);
```

The same holds true for the corresponding `GetXyz` methods. The variant without explicit length assumes that the element count precedes the data, the other only reads the given number of data elements. With `vector` and `valarray`, they will be resized to hold all of the data (in the variant without explicit length always, in the other variant only when the data would not fit). In the C API or with the array type flavour, this is not possible. There you always have to provide buffers of suitable length.

Anhang

A Objekt-Formate (Beispiele)

A.1 Histogramm(e)

Objekttyp: 100
Version: 2
Identifikation: Identifikationsnr. des ersten Histograms oder -1

Variable	Typ	Anzahl	Beschreibung
`nhisto`	short	1	Anzahl von Histogrammen
Pro Histogram:			
`type`	byte	1	Histogramm-Art ('I', 'i', 'R', 'r', 'F', 'D')
`title`	string	1	Histogramm-Titel
—	byte	0 oder 1	Füllbyte falls `type` und `title` zusammen ungerade Länge haben.
`ident`	long	1	Histogramm-Identifikationsnummer
`nbins`	short	1	Anzahl der Histogrammintervalle (in x)
`nbins_2d`	short	1	Anzahl der Histogrammintervalle in y oder 0
`entries`	long	1	Anzahl aller Einträge
`tentries`	long	1	Anzahl der Einträge innerhalb der Grenzen
`underflow`	long	1	Anzahl der Einträge unterhalb `lower_limit`
`overflow`	long	1	Anzahl der Einträge oberhalb `upper_limit`
Für Histogramme mit Gleitkomma-Grenzen (Typen 'R', 'r', 'F' und 'D'):			
`lower_limit`	real	1	Untere Histogrammgrenze
`upper_limit`	real	1	Obere Histogrammgrenze
`sum`	real	1	Summe aller eingetragenen Werte und
`tsum`	real	1	das selbe innerhalb der Grenzen
Für Histogramme mit Festkomma-Grenzen (Typen 'I' und 'i'):			
`lower_limit`	long	1	Untere Histogrammgrenze (in x)
`upper_limit`	long	1	Obere Histogrammgrenze (in x)
`sum`	long	1	Summe aller eingetragenen (x-)Werte und
`tsum`	long	1	das selbe innerhalb der Grenzen
Nur für zweidimensionale Histogramme (`nbins_2d > 0`):			
`underflow_2d`	long	1	Einträge unterhalb `lower_limit_2d`
`overflow_2d`	long	1	Einträge oberhalb `upper_limit_2d`
Für Histogramme mit Gleitkomma-Grenzen (Typen 'R', 'r', 'F' und 'D'):			
`lower_limit_2d`	real	1	Untere Histogrammgrenze in y
`upper_limit_2d`	real	1	Obere Histogrammgrenze in y
`sum_2d`	real	1	Nicht benutzt
`tsum_2d`	real	1	Nicht benutzt
Für Histogramme mit Festkomma-Grenzen (Typen 'I' und 'i'):			
`lower_limit_2d`	long	1	Untere Histogrammgrenze in y
`upper_limit_2d`	long	1	Obere Histogrammgrenze in y
`sum_2d`	long	1	Summe aller y-Werte und das selbe
`tsum_2d`	long	1	innerhalb der Grenzen.
Nur für Histogramme mit ganzzahligen Einträgen (Typen 'I', 'i', 'R' und 'r'):			
`counts`	long	(`nbins`)	Histogrammwerte falls `tentries > 0` (1D)
		(`*nbins_2d`)	bzw. 2D Histogram (zeilenweise)
Nur für Histogramme mit gewichteten Einträgen (Typen 'F' und 'D'):			
`content_all`	real	1	Inhalt aus allen Einträgen
`content_inside`	real	1	Inhalt innerhalb Grenzen
`content_outside`	real	8	und in Sektoren außerhalb
`data`	real	(`nbins`)	Histogrammwerte falls `tentries > 0` (1D)
		(`*nbins_2d`)	bzw. 2D Histogram (zeilenweise)

B Interne Strukturen

Bei den internen Datenstrukturen werden hier nur die Elemente aufgeführt, die für den Gebrauch in Anwendungsprogrammen gedacht sind. Zur Terminologie für in 'C' Ungeübte sei noch angemerkt, dass bei einer Deklaration der Form

```
struct _struct_SOMETHING { ... };
typedef struct _struct_SOMETHING SOMETHING;
```

oder kurz

```
typedef struct { ... } SOMETHING;
```

dann im Program eine entsprechende Datenstruktur durch

```
SOMETHING name;
```

definiert wird. Im folgenden wird die kürzere und leichter lesbare Form angegeben, auch wenn meist tatsächlich die längere Form verwendet wird.

B.1 Strukturen für Ein-/Ausgabe und Objektverwaltung

Siehe auch Abschnitt 3.1.

```
typedef struct
{
   ...
   int input_fileno;
   int output_fileno;
   FILE *input_file;
   FILE *output_file;
   int (*user_function)(_struct_IO_BUFFER *, int);
   ...
}  IO_BUFFER;


typedef struct
{
   ...
   unsigned long type;
   unsigned version;
   long ident;
   ...
}  IO_ITEM_HEADER;
```