SLAC-PUB-5450
March 1991
(E/I)

The Cheetah Data Management System*

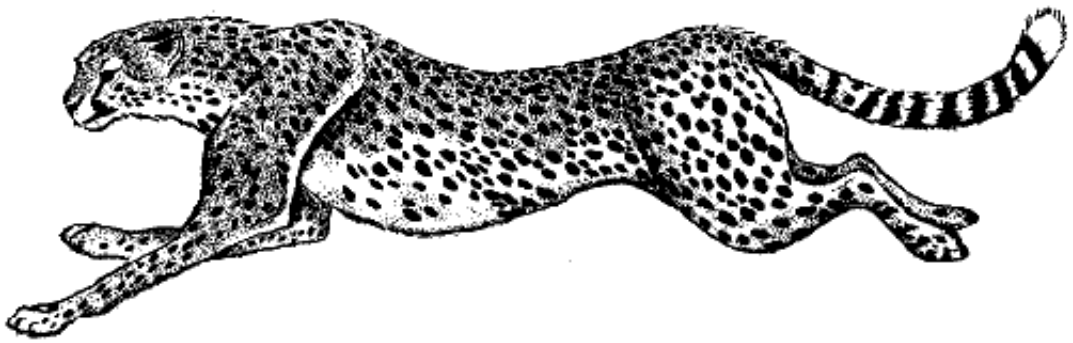
Paul F. Kunz

Stanford Linear Accelerator Center
Stanford University, Stanford, CA 94309-4349, U.S.A.

and

Gary B. Word

Department of Physics
Rutgers University, Piscataway, NJ 08855-0849, U.S A.



ABSTRACT

Cheetah is a data management system based on the C programming language. The premise of Cheetah is that the "banks" of FORTRAN based systems should be "structures" as defined by the C language. Cheetah is a system to manage these structures, while preserving the use of the C language in its native form. For C structures managed by Cheetah, the user can use Cheetah utilities such as reading and writing, in a machine independent form, both binary and text files to disk or over a network. Files written by Cheetah also contain a dictionary describing in detail the data contained in the file. Such information is intended to be used by interactive programs for presenting the contents of the file. Cheetah has been ported to many different operating systems with no operating system dependent switches.

* Work supported by the Department of Energy, contract DE-AC03-76F00515.

Invited talk presented at the INFN Eloisatron Project, 14th Workshop: Data Structures for Particle Physics Experiments, Erice, Italy, 11–18 Nov. 1990.

1. The Goal of the Cheetah System

The basic entities managed by Cheetah are structures as defined in the C programming language. To quote the introductory paragraph in the chapter on structures from Kernighan and Ritchie[1]

> A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling. (Structures are called "records" in some languages, notably Pascal.) [...] Structures help to organize complicated data, particularly in large programs, because in many situations they permit a group of related variables to be treated as a unit instead of as separate entities.

The design of Cheetah is based on the premise that structures of the C language are ideally suited to organize the complicated data within large high energy physics (HEP) programs, such as event reconstruction and Monte Carlo simulation programs. C structures are clearly the equivalent of the "banks" referred to by many HEP developed data management systems. These HEP developed systems are all based on the FORTRAN programming language and thus they all suffer to a greater, or lesser degree, in integration with the compiler, the operating system, and/or the symbolic debugger. The goal of the Cheetah system is to exploit the features of the C language while adding the data management tools needed by HEP such as input and output to storage. Cheetah is designed to allow the code writer to use native C, with only minimal calls to Cheetah functions. Cheetah maintains a symbol table of data structures which is useful to interactive programs, reads and writes machine independent binary and textual data files, and provides for seamless client-server networking.

2. Introduction to C Structures

Consider the structure definition shown at the top of the right hand column of this page. The keyword **`struct`** introduces a structure declaration which is a list of declarations enclosed in braces. The name **`track`** is the *structure tag*. The variables named in a structure are *members* of the structure. A **`struct`** declaration is a way the user can declare a new data type in C.

```
struct track {
    float x[3];
    float p[3];
    float ptot;
    int charge;
};
```

In the Cheetah context, structures are the "banks" of FORTRAN based systems. The structure tag is used by Cheetah as a name for the structure type. Normally, one has a set of structures of the same type, such as multiple Monte Carlo tracks within an event. Thus, one could make a declaration in the following way:

```
struct track **mctrack;
```

The variable **`mctrack`** declared in this way in the Cheetah context is known as a *family array pointer.* The use of the double star (**`**mctrack`**) may be confusing to a novice user of the C language. It says that **`mctrack`** is a pointer to a pointer to a **`track`** structure. Due to an artifact of the C language it can also say that **`mctrack`** is a pointer to an array of pointers to **`track`** structures. Thus an expression like:

```
total_momentum = mctrack[i]->ptot;
```

would set the local variable **`total_momentum`** to the value of the member **`ptot`** of the **`i`**-th structure in the **`mctrack`** family. Cheetah makes extensive use of this artifact. As structures are added via the Cheetah function **`chadd()`**, Cheetah not only allocates the memory space for the structure, but it also allocates space for the array of pointers to the structures. This is the fundamental mode in which Cheetah operates.

For the user, the use of C structures in this way provides a clear and concise access to the data stored in the structures. For the C compiler, the string **`mctrack[i]->ptot`** is a name of a variable and thus can be used like any other variable as part of an expression, as an argument to a function, etc. Yet, to the programmer, this string has great mnemonic value, e.g., access to data by name. The **`ptot`** component of the string is used here, for example, to denote the total momentum of the track. The member name **`ptot`** can be used in many structures without confusion, since it is only part of the complete variable name. We call this "reuse of the name space." Also, the family component of the variable name (**`mctrack`** in the example above) has

strong mnemonic content. Thus, with the C language, we have an environment very different from the FORTRAN one where variables in common blocks generally have to be unique throughout the whole program or where variables in the banks of some HEP developed systems may have lost their mnemonic value altogether.

3. Memory Management with Cheetah

It is often useful to look behind the scenes to understand how the system is working. Figure 1 shows how memory is being allocated when Cheetah is used. In the user code, the variable **`mctrack`** is a pointer (a 4 byte quantity) to an array allocated by Cheetah. This array is a set of pointers to the actually structures which Cheetah allocates. The array is allocated automatically when the user adds a member to a family with the Cheetah **`chadd()`** function.

Internally, Cheetah uses the **`malloc`** function to allocate memory space. It is part of the C standard library of routines. Thus, it is provided by the vendor of the compiler which is most frequently the vendor of the hardware platform. At least on UNIX platforms, the quality and speed of the hardware will be measured by benchmarks consisting of many programs written in C, including the UNIX kernel itself. Thus, one can assume that the vendor has paid a lot of attention to the speed and efficiency of the **`malloc`** function. Also, on many platforms, there are very sophisticated debugging tools associated with **`malloc`**. The Cheetah system thus benefits greatly by making direct use of these standard utilities.

4. Input and Output with Cheetah

Although the C language and its standard utilities provide for the management of and access to data stored in structures, it does not provide for input and output of these structures in a user friendly way similar to that provided by the FORTRAN based HEP data management systems. The Cheetah system has been written specifically to provide these functions in a user friendly manner.

As an input and output system, Cheetah is capable of writing collections of structures to disk and reading them back again. The structures are organized by families, so each read or write operation works on a collection of families called a record. When a new record is read, existing families of the same name as one being read are replaced by the newly read family. At the head of a Cheetah data file, there is a dictionary describing the names of the families in the file and their structure types, the names of the members of the structure types, each member's type and other information so that the structure declaration can be reconstructed when the file is read.

Consider the program in Example 1. This sample code opens a Cheetah data file, reads one record, and writes it out to another file. The function to open a Cheetah file, **`chopen`**, is analogous to the C file open function **`fopen`**. It takes the same two arguments: the name of the file as a string, and the access mode as a second string. It returns a pointer to a Cheetah file structure, known as a Cheetah file pointer, just as **`fopen`** returns a file pointer. The Cheetah file pointer is used as a parameter in calls to all further manipulations on that file.

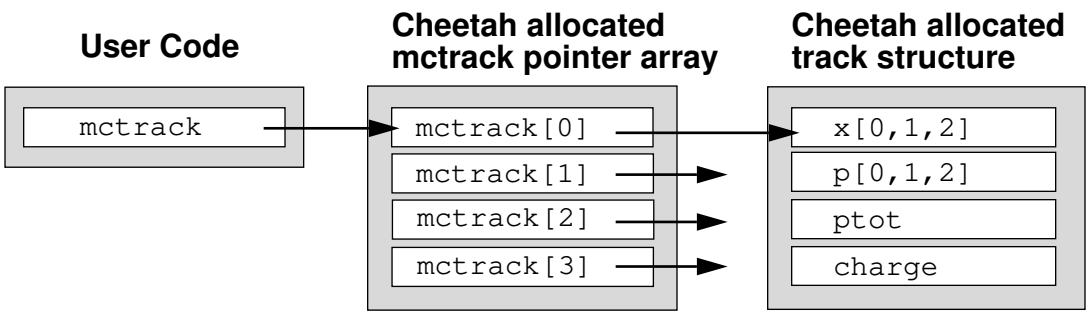


Figure 1. Memory layout with Cheetah system.

```
#include "cheetah.h"
main( ) {
    ch_file *ch_filep_in, *ch_filep_out;
    ch_list *ch_listp;
    int return_code;

    ch_filep_in = chopen("test.chdata","r");
    ch_filep_out = chopen("test2.chdata","w");
    ch_listp = chread(ch_filep_in);
    if (ch_listp != NULL) {
       return_code = chwrite(ch_filep_out,ch_listp);
    }
    exit(0);
}
```

Example 1. Reading and writing Cheetah Files.

The Cheetah read function will read one record from the input file, reconstruct all the structures in that record, and allocate the arrays of pointers to the structures. The Cheetah read function only requires one argument: the Cheetah file pointer. It returns a pointer to a linked list of families or **NULL** if the end of file is reached. The linked list is known as a Cheetah list. It is used as an argument to Cheetah functions that want to work on the record as a whole. For example the **chwrite** function takes two arguments; the Cheetah file pointer for output and a Cheetah list pointer.

```
#include "cheetah.h"
#include "track.h"
/* to find the index of the fastest track in the mctrack family,
 * fastest = fastest_track("mctrack");
 */
int fastest_track(char *track_family;) {

    int fastest;
    int last_track;
    struct track **tracks;

    tracks = chfptrs(track_family); /* find array of pointers */
    last_track = chidlast(track_family); /* index of last track */
    fastest = 0;
    for ( i = 0; i < last_track; i = i + 1 ) {
       if (tracks[i]->ptot > tracks[fastest]->ptot) {
          fastest = i;
       }
    }
    return fastest;
}
```

Example 2. Example of accessing Cheetah managed data.

The average user need not be concerned about the structure of a Cheetah list. Its just a convenient handle used to group a set of families together. In the same way, the user is not concerned about the structure to which a file pointer points. However, for some applications, such as interactive programs like Reason[2], the Cheetah list pointer contains enough information to decipher the contents of the entire record. Cheetah also provides functions to add or delete members of the list.

5. Example of Usage

An example usage of Cheetah is given in Example 1.In this example, a function finds the index of the track with the highest momentum in any family of the **track** structure type. It shows two Cheetah utilities that make working with C structures and the resultant handling of pointers easier. The Cheetah function **chfptrs()** returns the family array pointer of a family which is specified by name. Also, the Cheetah function **chidlast()** returns the index of the last member currently in memory of the given family.

6. Creating Cheetah Structures

So far we've considered examples of code in which the Cheetah dictionary has already been created or is read in at the beginning of a Cheetah file. In this section, we'll show one method of creating a dictionary. An example code from a Monte Carlo particle generator is shown in Example 3. The function **mclund** is set up to be called at different stages. Only the initialization stage concerns us which is the case with **option = INIT**. The structure type **event** is declared to Cheetah with the function call **i_event()** on line 13. This function is generated automatically by the **chgen** command which will be described later. The declaration of **i_event()** is included in the **event.h** file, while the definition is given in a separate source file, as described below.

After the call to **i_event** is made, Cheetah knows about the structure type with structure tag **event**. It knows nothing about any families of type **event** yet. The creation of a family is done with the function call **chfcreat()** as shown on lines 16--18. It takes two arguments.

```
1  #include "cheetah.h"
2  #include "event.h"
3  #include "track.h"
4
5  ch_list *mclund(enum options option) {
6      static ch_list *ch_listp;
7      int irc ;
8
9      switch(option)
10     {
11     case INIT:
12
13        irc = i_event();
14        irc = i_track();
15
16        irc = chfcreat("mcevent","event");
17        irc = chfcreat("mctrack","track");
18        irc = chfcreat("phtrack","track");
19
20        ch_listp = chlcreat("mcevent");
21        ch_listp = chladd(ch_listp,"mctrack");
22        . . .
```

Example 3. Creating new Cheetah families and types.

The first is a name for the family while the second argument is a structure type already known to Cheetah. In this way two families can be created which have the same structure type, as is shown on lines 17 and 18. A list of families is created using **chlcreat**, which takes a family name as it argument, as shown on line 20. Line 21 shows how other families may be added to a **ch_list** using **chladd**, which accepts an existing list pointer and the family name to add to the list.

The **i_event()** and **i_track()** functions, which initialize the Cheetah structures which describe the **event** and **track** structures, are generated using the **chgen** utility. This utility reads a file which contains the C-like descriptions of the **event** and **track** structures and generates a C source file containing these initialization functions. Alternatively, the definition of a Cheetah structure can be built dynamically by a set of Cheetah utilities, a feature often useful to interactive programs, for example.

7. Cheetah Text Files

In addition to the ability to read and write binary files, Cheetah can also read and write text files. The text file contains two sections, the first describes the structures and declares the families that are in the file, the second section contains the individual records. In Example 4, a Cheetah

```
struct runevent { int run, event; };

struct track {
    float x[3];
    int charge;
    float p[3], ptot;
    };

struct track mctrack;/* Declares family mctrack
    of type track.*/
struct runevent header;/* Declares family header
    of type runevent.*/

{       /* Begins a record.*/
    header = 101 1;/* Assigns header[0].*/
    track =
    1. 2. 3. -1 4. 5. 6. 7. ,/* Assigns track[0].*/
    4. 5. 6. +1 1. 2. 3. 4. ,/* Assigns track[1].*/
    1. 2. 3. -1 4. 5. 6. 7. ,/* Assigns track[2].*/
    ;  /* Ends family track.*/
}       /* Ends a record.*/

{       /* Begins a new record.*/
    header = 101 3;/* Assigns header[0].*/
    track =
    1. 2. 3. -1 4. 5. 6. 7. ,/* Assigns track[0].*/
 [2]4. 5. 6. +1 1. 2. 3. 4. ,/* Assigns track[2].*/
       /* track[1] is NULL.*/
    ;  /* Ends family track.*/
}       /* Ends a record.*/
```

Example 4. An example Cheetah text file containing two records.

text file containing two families and two records is shown. This file may be read directly, using **`chread`** as in Example 1, or may be used to create a binary file using the Cheetah importer routine **`chimp.`** Similarly, there exist a Cheetah exporter routine, **`chexp`**, which creates a text file from a binary file.

8. Cheetah Networking

Cheetah has been designed to permit Cheetah records to be transmitted over the network transparently to the end user. The actual transmission of data over the network is effected using Sun's XDR product (eXternal Data Representation). The handshaking between the client and the server is affected using Sun's RPC product (Remote Procedure Calls). These products are generally part of a TCP/IP networking package. To retrieve records from another machine, the end user simply creates a file (called a Cheetah service file) containing the internet address of the remote machine, the name of the remote server, and, if the records are to be read from a file, the name of the file on the remote machine. The remote server may also be designed to create the records internally using, for example, a Monte Carlo generator followed optionally by additional processing, or may send samples of records from a real-time data stream. From the point of view of the end user, the only difference between reading records from the network and reading them directly from a file is the creation of the service file to provide the link to the remote data stream. Of course, writing to a remote server is just as transparent to the end user. Servers have been prototyped which can be used to read Cheetah data files, Jazelle data files, and a slightly modified Adamo text file. A server shell exists with user hooks to allow for users to write other special purpose servers.

Because of the use of XDR, which uses a canonical format for data transmission, the server and the client may exist on computers of differing architecture. To date, networking has been accomplished between a NeXT UNIX computer, an IBM mainframe with VM/CMS computer, and a DEC VAX/VMS computer, with no changes required in Cheetah code.

9. Summary

The C language, with minimal extensions for input and output of data structures as afforded by Cheetah, provides a nice environment in which to accomplish HEP tasks. The C-structures are natural replacements for the "banks" of traditional FORTRAN based data management systems. By letting Cheetah manage the pointers to the structures, the casual user is relieved of this responsibility and, instead, is given a straightforward method to access the structures. Cheetah is also well suited for use by interactive programs, when access to the symbol table describing the data is essential and where dynamic creation of data structures is also desirable. Cheetah has both a binary and a text file format, each of which is machine independent, and can be used to move structures from one machine to another over a network. Although Cheetah was designed to be used by C programs, it can be and has been used from FORTRAN programs in order to make use of its network transparent file structure and its dynamic memory allocation. Cheetah has also attained its goal of portability, having been ported to many different operating systems with no operating system dependent switches. Cheetah does contain switches, however, so that it may compile with either an ANSI C compiler or a traditional K&R C compiler.

10. References

[1] B. W. Kernighan and D. M. Ritchie, *The C Programming Language* (Prentice Hall, Englewood Cliffs, 1978).

[2] W.B. Atwood, Richard Blankenbecler, Paul F. Kunz, Benoit Mours, A. Weir, G. Word, *8th Conf. on Computing in High Energy Physics*, Santa Fe, NM, Apr 9-13, 1990.