BNL - March 7, 1996

The DD User Guide

version 2.5

C.Witzig

Physics 510C
Brookhaven National Laboratory
Upton NY, 11973-5000
e-mail: witzig@bnl.gov

Contents

1 What's New Since Version 2.0? 4

2 Introduction 6

3 The DD System 7

3.1 The FIFO System 7
3.2 Syncronization Issues 8
3.3 Event Distribution 9
3.4 The Garbage Demon 11
3.5 As an Example: E787 Use of the DD System: 11

4 Code Description 13

4.1 Organization of the Code 13
4.2 User Routines 15
4.2.1 `ddu_init` 15
4.2.2 `ddu_attach` 16
4.2.3 `ddu_attached` 16
4.2.4 `ddu_start`, `ddu_stop` 16
4.2.5 `ddu_close` 16
4.2.6 `ddu_set_brc` 17
4.2.7 `ddu_set_dd_wait` 17
4.2.8 `ddu_get_free_buffers` 17
4.2.9 `ddu_set_protection` 17
4.2.10 `ddu_fifo_exist` 18
4.2.11 `ddu_req_fev` 18
4.2.12 `ddu_creq_fev` 18
4.2.13 `ddu_put_fev` 18
4.2.14 `ddu_ins_fev` 18
4.2.15 `ddu_get_fev` 19
4.2.16 `ddu_brc_fev` 19
4.2.17 `ddu_set_brccmd` 20
4.3 System Routines 21
4.3.1 `dds_create()` 21
4.3.2 `dds_close()` 21
4.3.3 `dds_wait4proc()` 21
4.3.4 `dds_wakeup()` 21

4.3.5 dds_flush() 22

5 Example Programs **23**

5.1 A DD Producer Process 23

5.2 A DD Consumer Process 24

5.3 A Monitor Process 26

5.4 A Modifying DD Consumer Process 26

6 Status and Error Reporting **29**

7 Customizing the DD System **30**

7.1 User Entries 30

7.1.1 ddu_link_signal() 30

7.1.2 ddu_def_static_fifos() 30

7.2 User Configurable Event Selection 30

7.3 User Configuration Options 32

7.3.1 DD Configuration 32

7.3.2 Usage of UNIX signals 32

7.3.3 User Changes to the Event Distribution 33

7.3.4 Action upon Detection of a Fatal Error 33

7.3.5 Multiple Copies of the DD System on the Same Host 33

8 Network Connection **34**

8.1 Overview 34

8.2 The dd_server program 34

8.2.1 Server Configuration Files 34

8.2.2 The Scripts 35

8.3 Attaching to a DD System over the Network 35

8.4 The dd_link between DD Systems 36

9 User Interface **37**

9.1 dd_ui User Interface 37

9.2 Process Monitor Interface 37

9.3 Fifo Monitor Interface 38

10 FAQ **40**

1 What's New Since Version 2.0?

The following items are new in version 2.1:

- Various code optimizations with respect to the lousy performance of version 2.0 have been performed.
- `ddu_set_dd_wait` has been introduced. The wait mode while retrieving events is now a property of the process and not the fifo. In particular two processes with different wait modes can read from the same fifo. In addition one can change the wait mode at any time.
- A bug in the fifo monitor programs has been fixed.
- The directory structure has been changed, such that $DD_BIN and $DD_LIB has specific subdirectories depending on the UNIX platform.

The following items are new in version 2.2a:

- `dds_flush` added
- `ddu_creq_fev` added for conditional event request
- `ddu_get_free_buffers()` added. Returns the number of free buffers in the system.

The following items are new in version 2.3:

- The network part has been completely rewritten
- `dd_log` replaces any printf statements to stdout or stderr
- `dd_cleanup`: an example script on how to really cleanup leftover processes

The following items are new in version 2.5:

- `dds_flush()` : fixed bug discovered by M.Leitch
- `dd_link()` : fixed bug in transfer of fifoname.
- `ddu_fifo_exist(char *fifoname`: finds out whether a fifo named fifoname exists

- `ddu_init(char *fifoname struct fifo_mode fmode)` : changed calling sequence to allow to have a fifo with one and only one process attached to it. Replaced the many calling arguments with the following C structure:

```
struct fifo_mode {
            int mode;
            int wait;
            int prescale;
            int suser;
            int *p2ctl;
};
```

  The members of the structures are the same as the old calling arguments of `ddu_init`. `fifo_mode.suser` is a flag to indicate that the fifo is in single user mode (i.e. only one process can be attached to it). The valid values of `fifo_mode.suser` are `FMODE_SINGLE_USER` or `FMODE_MULTI_USER`. The default value is `FMODE_MULTI_USER`.

- `ddu_set_brccmd`:
  Upon a broadcast there is the option to call a function depending on the value of `fev.ctlb2`. `ddu_set_brccmd` sets the pointer to that function.

- `dds_flush(int brc_flag)` :
  Added argument `brc_flag` to determine whether a broadcast should be made at the end of `dds_flush` . This is used in `dd_link` to propagate the flush over the network.

- Created $DD_USR directory as a placeholder in case header files and source code are changed by the user.

- Changed from Tcl7.3 and Tk3.6 to Tcl7.4 and Tk4.0

- Ported to the Alpha (bin/lib subdirectory OSF1).

- Extended `dd_link` to support block transfer.

2 Introduction

The E787 Data Distributor (DD) is a UNIX based event buffer manager, originally developed for the AGS rare kaon experiment E787. It is written entirely in C and currently runs on SGI and HP.

It allows to distribute events to processes in real time in a multiprocessor environment for data reduction, event building, online event analysis and monitoring. The bookkeeping is done with a software implementation of FIFOs (first in, first out) using shared memory and semaphores.

In the following sections we first describe the philosophy of DD system. Its use in the E787 experiment is as an example on how to implement it in HEP experiments.
In the second section we describe the routines that are relevant for the user, followed by a section in which a few typical example programs are presented.
Chapter 5 describes how the user can customize the DD system, followed by a section on the user interface. The network connection of the DD system is under complete revision and its description is (temporarily) omitted.
The last section is a list of FAQ.

3 The DD System

3.1 The FIFO System

The DD system is based on a set of fifos that are created and destroyed in real time. A process attaches to the DD system by creating a fifo from which he will read C structures describing the type and location of an event in shared memory. A process can become a producer by attaching to the `INPUT_FIFO` from where he will read C structures telling him the address of free space to copy an event.

Events are only copied once into the DD shared memory. After that only C structures (called `fev` for fifo event) describing the event (type, size, shared memory id...) are passed between the processes (i.e. written/read into/from the fifos). Once a process reads such a C structure from its fifo, the event "disappears" from the DD system until the process puts it back into the system. This allows modifcation of events (events can be compressed, merged or even deleted and newly created).
In the E787 implementation the fifo event `fev` is the following C structure:

```
typedef structure fifo_entry {
   int shmid;
   int dboff;
   int *p2da;
   int len;
   int ctl[4];
} fifo_entry;
```

where :

`shmid`	the shared memory identifier where the event resides
`dboff`	the offset within the shared memory
`*p2da`	pointer to the start of the event
`len`	length of the event
`ctl[4]`	array of control words (see below).

As it is not desirable that every process receives every event, a set of control words allow the routing of events directly to processes that ask for given event types. E787 uses the level 0 trigger bits as control words.

Processes can attach to the same fifo. In this case the events are shared between the processes. This allows to avoid bottlenecks by distributing the events between processes in a multiprocessor environment.

Fifos can be created and destroyed at any time. This enables e.g. user processes to attach to the DD system, receive a given number of events and then detach from it.

Four exceptions have to be handled: When a process reads an empty fifo, when a process wants to write into a full fifo, when the system runs out of shared memory segments into which to place new events and when "something really bad" happens.

- Reading an empty fifo:
  Every process specifies at initialisation time how the wants to handle this situation. He can either fall asleep until an event arrives or return with an error code or return with an error code and then receive a signal telling him, that an event has arrived.

- Writing into a full fifo:
  This situation is avoided by making the fifos deeper than the maximum number of possible events in the system.

- The system runs out of shared memory segments:
  This situation is identical to reading an empty fifo, whereas now the empty fifo is the `INPUT_FIFO`.

- Something bad happens:
  A demon constantly monitors the DD system and tries to detect and correct error conditions: E.g. if a processes crashes and doesn't detach from the DD system, he will be removed. Alternatively if there is a fatal error (i.e. the disappearance of shared memory segments or semaphores), the system is shut down as gracefully as it can be done.

3.2 Syncronization Issues

There are certain situation in which a syncronization of all processes (or a subset of them) is desired. Typical examples in a HEP experiment are: start a run (but only if all the processes are ready, in particular all the processes that write the data to storage medium), end a run, or distribute a given event to a subset of processes bypassing the fifo system (e.g end of spill signal if the data acquisition has a spill structure).

A set of routines allow to wait until all processes have finished processing the events and are sleeping on semaphores or to wake up processes that are sleeping on a semaphore. In the E787 experiment these routines are used by the main user interface at the beginning and end of a run.

In addition a broadcast mechanism has been developped in order to distribute (broadcast) events. Every process that attaches to the DD system specifies whether he wants to participate in this broadcast mechanism. Two modes are possible: In the first one (SIGNAL mode) the process receives a signal upon which he reads a special fifo, where he finds the broadcast message. This mode is relativly CPU intensive, as a large number of processes can suddenly compete for access to that broadcast fifo. Therefore another mode (USMAIL) has been added in which the broadcast message is simply added into the process's fifo. In this (less reliable) mode the message can be lost if several processes are reading from the same fifo.
E.g. this broadcast mechanism is used in the E787 experiment to distribute the scaler record at the end of every spill to all the processes.

3.3 Event Distribution

As mentioned above processes can specify what kind of events they receive. Every event is given a set of control words when they are injected into the DD system. These words are used to request events of a given type.
In the E787 experiments one of these control words is the level 0 trigger information. The different data loggers (E787 currently writes to 10 different tape drives in parallel) specify which event types they want, thus the different event triggers are written to different tapes.

Currently there are four control words: `ctlw1, ctlw2, ctlb1 and ctlb2`. The first two operates on a equal basis, i.e. a process requests events with say `ctw1 == requested_value`, the other on a bit basis, i.e. a process requests events with - say - second bit of `ctlb1` set. The exact requirements can be specified by the user.

Processes can either request every event, request a prescaled fraction of events of a given type or receive an event only if they ask for it. A process can also request that a given number of events are set apart for him, such that he always gets an event immediately if he asks for it. This feature is also useful if the data taking is characterized by a spill structure (such as at the AGS).

In which order will processes receive events?
The fifos can be imagined as an array (see figure ??) and a process attaching to the DD system will receive by default the first free fifo within that array (note that processes are allowed to attach/detach to/from the DD system at any time).
This behaviour is important if one has consumer processes that modify the event or if one wants the data loggers to log the data before other lower priority / very slow consumers access it.
Fifos can be declared at initialisation time in a given order (so called static fifos). These fifos will be inactive (i.e. no events will be written into them) until a process attaches to them and activates them (user routine `ddu_start()`. These static fifos can be declared in the correct order in the user routine `ddu_def_static_fifos`).

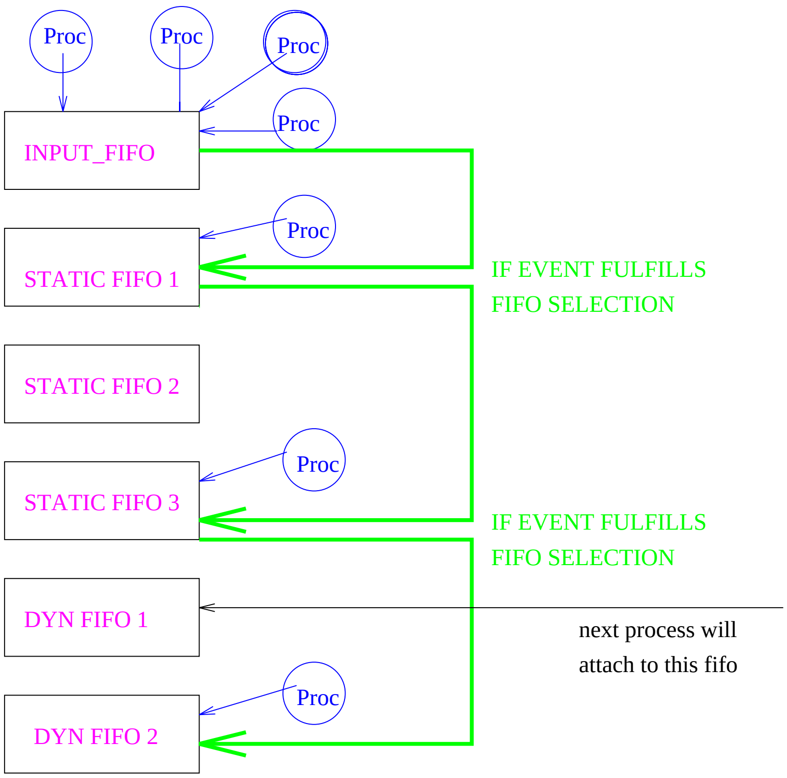

Figure 1: *Schematic view of the event distribution within the fifo system.*

3.4 The Garbage Demon

A special process is launched at initialisation time - the garbage demon. Its main task is to delete the shared memory segments that were allocated to handle events, larger than the default maximum event length. These events are not put back into the `INPUT_FIFO` in order to keep the number of buffers constant over time. As the event is no longer used, they are put into the `GARBAGE_FIFO` to which the garbage demon is attached. This demon has the permission to delete the shared memory segments (note that due to the flexible structure of the DD system, any producing process can create extra large events and thus additional memory segments).
In addition the garbage demon checks in regular intervals whether any processes that were attached to the system suddenly disappeared and then removes such processes from the DD system[1]. The garbage demon also shuts the DD system down if no other processes are attached to it or if he is told to do so.

3.5 As an Example: E787 Use of the DD System:

Figure 2 shows how the DD system is used in the E787 experiment. Up to 16 data reducer (`dr`) receive the individual events from FASTBUS over the SGI VME port. They decode the trigger information of the event, request empty events from the DD system (i.e. read the `INPUT_FIFO`), copy the event into the free shared memory location and inject the event into the DD system. Up to 10 data loggers share the events and write them to tape. These data loggers are syncronized at the beginning and end of a run as well as the end of every spill.
Several monitor programs are attached to the DD system on an optional basis. Typical examples are online event display, processes that monitor the detector performance and the DAQ, online calibration programs and various user programs. Some of these programs are launched at the beginning of every run and exit at the end of a run. Typically 50-60 processes are active at any given time.

The DD system is started and deleted by a control program (`dd_ctl`) which accepts commands from its standard input (pipe to the main DAQ user interface. (Note: The use of this control program is optional. In general the first DD master (here (`dd_ctl`) creates the DD system, the last one (in this example also `dd_ctl`) deletes it.).

[1]e.g. a process running in the debugger encounters an exception - normally a processes makes sure `atexit` that it detaches from the DD system

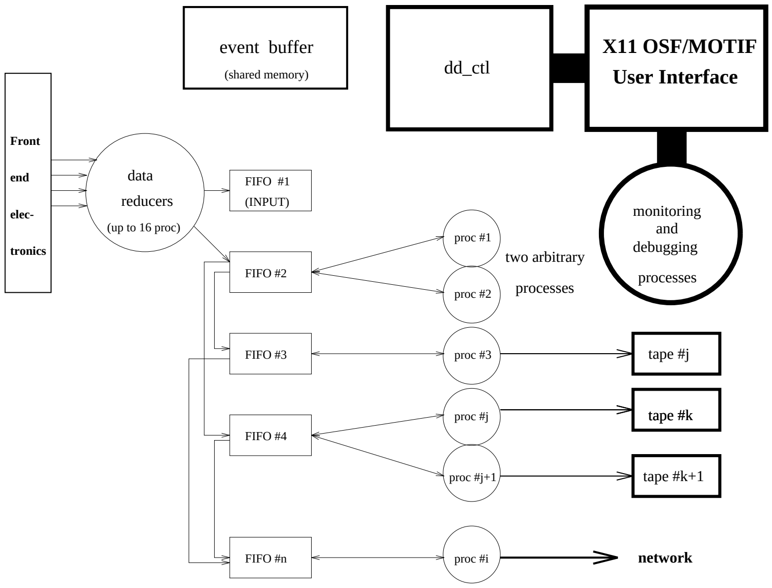

Figure 2: *Overview of the DD system in the E787 experiment.*

4 Code Description

4.1 Organization of the Code

The organization of the code is shown in figure 3. This hierachy is present in every DD routine. It should be noted that the "normal user" calls only user routines (`ddu_xxx`), whereas the system routines (`dds_xxx`) are typically called either by a control program or the main user interface.

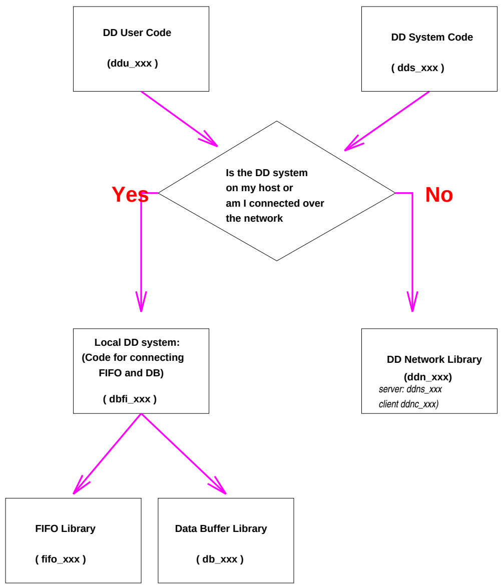

Figure 3: *Overview of the DD code.*

The processes attached to the DD system can be divided into three classes:

- DD masters:
  These are processes that have called at a given time `dds_create()`. This routine creates the DD system (if it doesn't exist) or attaches to it (if it exists).

- DD consumers/producers:
  These are processes that are attached to a DD fifo (i.e. processes that called `ddu_init`).

- Monitor processes:
  These are processes that attached to the DD system, but do not insert or retrieve any events (i.e. they are not attached to a fifo). They typically only called `ddu_attach()` .

Note:
These three classes are not mutually exclusive, e.g. a master can also be a consumer/producer.

4.2 User Routines

Naming convention:

User routines are named as `ddu_xxx` .

4.2.1 `ddu_init`

Usage: int `ddu_init(char *fname, struct fifo_mode fmode );`

Returns: 0 if good status, -1 if bad status.

This routine attaches a process to a fifo, from where it can receive events. If the fifo already exists, it attaches to it. If the event is a producer, the fifo name is "INPUT".

`fname`	Fifoname.
`fmode`	fifo mode (C structure `fifo_mode`)

The C structure `fifo_mode` is defined as follows:

```
struct fifo_mode {
            int mode;
            int wait;
            int prescale;
            int xsuser;
            int *p2ctl;
};
```

with:

`fmode.mode`	`FMODE_ALL` : every event
	`FMODE_COND` : only events that match *p2ctl
	`FMODE_ONREQ` : gets only events if asks for
fmode.wait	`DD_WAIT_SLEEP` : falls asleep if no event available
	`DD_WAIT_ASYNC` : returns with error code if no event available
`fmode.prescale`	prescale factor: gets only every n-th event (default: 1)
`fmode.suser`	FMODE_SINGLE_USER: only one process is allowed to attach to the fifo
	FMODE_MULTI_USER: multiple processes are allowed to attach to the fifo
`*p2ctl`	pointer to the four control words (see below for details).

4.2.2 ddu_attach

Usage: int `ddu_attach()`
Returns: 0 if good status, -1 if bad status.
Attaches to the DD system, but does not create a fifo. This routine is typically called by monitoring processes that want to have access to the shared memory segments, but do not want to access any events.
Returns bad status if the DD system does not exist.

4.2.3 ddu_attached

Usage: int `ddu_attached()`
Returns: 0 if good status, -1 if bad status.

This routine (which in turn is called by every user routine) allows to find out, whether the DD system still exists. Thus a monitor process consists typically of a loop like:

```
int status; status = ddu_attach();
while ( ddu_attached() ) {
/* do something */
}
```

4.2.4 ddu_start, ddu_stop

Usage: int `ddu_start()` , `ddu_stop()`
Returns: 0 if good status, -1 if bad status.
This routine requires that the process is attached to a fifo (i.e. called `ddu_init`). The thus created fifo is in an idle state and has to be actived before events can be obtained. The fifo can be made idle by calling `ddu_stop()`.
Switching back and forth between `ddu_start()`, `ddu_stop()` allows a process to become active/inactive within the DD system without detaching.
NOTE:
While the fifo is NOT active, no events can be retrieved. The INPUT fifo is always active.

4.2.5 ddu_close

Usage: int `ddu_close();`
Returns: 0 if good status, -1 if bad status.
Detaches a process from the DD system. The DD system will continue to exist.

4.2.6 ddu_set_brc

Usage: int `ddu_set_brc(brc_flag);`
Returns: 0 if good status, -1 if bad status.

`brc_flag`	`DD_BRC_OFF`	no broadcast to this process
	`DD_BRC_SIGNAL`	process receives a signal upon broadcast message
	`DD_BRC_USMAIL`	process receives broadcast messages over the less reliable USMAIL service (see section on broadcast mechanism).

4.2.7 ddu_set_dd_wait

Usage: int `ddu_set_dd_wait(dd_wait);`
Returns: 0 if good status, -1 if bad status.

`dd_wait`	`DD_WAIT_SLEEP`	process falls asleep if no event available
	`DD_WAIT_ASYNC`	process returns with nonzero (positive) status word if no event available.

Note that `ddu_set_dd_wait` can be called at any time. This allows a process to switch back and forth between different modes. The default value is as specified in `ddu_init`.

4.2.8 ddu_get_free_buffers

Usage: int `ddu_get_free_buffers();`
Returns: Number of free buffers, -1 if bad status (i.e. the DD system does not exist anymore).

4.2.9 ddu_set_protection

Usage: int `ddu_set_protection(prot_flag);`
Returns: 0 if good status, -1 if bad status.
A demon process (garbage demon) checks in regular intervals, whether there are processes that are clobbering the system (more precisely whether there are fifos that are full but never read out). If such a fifo is found, then the fifo is deactivated and a signal HUP is sent to the process (which probably has already died).
By default the fifo is not protected against this checking mechanism. `ddu_set_protection` allows to change the default behaviour.

`prot_flag`	`FIFO_PROTECTED`	fifo will under no circumstance be removed
	`FIFO_UNPROTECTED`	fifo will be removed (default).

4.2.10 `ddu_fifo_exist`

Usage: `ddu_fifo_exist(char *fifoname)`
Returns 0 if a fifo named *fifoname doesn't exist, -1 if it does.

4.2.11 `ddu_req_fev`

Usage: int `ddu_req_fev(int len, struct fifo_entry *fev)`
Returns: 0 if good status, -1 if bad status, >0 if interrupted system call
Requests a free fifo entry fev pointing to a buffer of minimum size `len`.

4.2.12 `ddu_creq_fev`

Usage: int `ddu_creq_fev(int len, struct fifo_entry *fev, int min_cnt)`
Conditional event request. Same as `ddu_req_fev` but an event is only returned if there are at least more than `min_cnt` free buffers in the DD system. Otherwise +1 is returned.

4.2.13 `ddu_put_fev`

Usage: int `ddu_put_fev(struct fifo_entry fev)`
Returns: 0 if good status, -1 if bad status, >0 if interrupted system call
Puts a fifo entry `fev` back into the DD system. This routine is called by producers to add a new fifo entry or by consumers that are finished handling the event obtained by a previous call to `ddu_get_fev(struct fifo_entry *fev)` .

4.2.14 `ddu_ins_fev`

Usage: int `ddu_put_fev(struct fifo_entry fev)`
Returns: 0 if good status, -1 if bad status, >0 if interrupted system call
Inserts a fifo entry `fev` to the DD system. The fev is inserted at the `INPUT_FIFO`. This routine can be called by producers and consumers. Care has to be taken in the case of a consumer using this routine: As the fev is inserted at the `INPUT_FIFO`, the event might end up at the consumer again (unless the latter modified the `fev.ctl` words).

4.2.15 ddu_get_fev

Usage: int `ddu_get_fev(struct fifo_entry *fev)`
Returns: 0 if good status, -1 if bad status, >0 if interrupted system call
Retrieves a fifo entry `fev` out of the DD buffer. $fev->p2da$ points to the location of the data.

4.2.16 ddu_brc_fev

Usage: int `ddu_brc_fev(struct fifo_entry *fev, int wait)`
Returns: 0 if good status, -1 if bad status.
Arguments: `fifo_entry fev` to be broadcasted, `int wait`
This routine allows to broadcast a fifo_entry to all processes attached to a fifo of the DD system depending on the value of their broadcast flag (as set by `ddu_set_brc`):

`brc_flag`	`DD_BRC_OFF`	no broadcast to this process
	`DD_BRC_USMAIL`	n copies of `fev` are written into the fifos where n is the number of processes attached to the fifo. Note: If n greater 1, then it is not garanteed the every process will get one copy of `fev` (one process might be fast enough to grab two copies!).
	`DD_BRC_SIGNAL`	every process is garanteed to get one copy of `fev`, which are distributed using signals and a special broadcast fifo.
	`DD_BRC_OFF`	process will not be considered in the broadcasts process.

The calling process either returns immediately (`wait == 0`) or waits at most `wait` msec before it times out.
If the calling process is a consumer, it will not receive a `fev` (doesn't send a signal to himself!).

NOTE: Usage of `fev.ctlw2` and `fev.ctlb2`
`fev.ctlw2` and `fev.ctlb2` have a special role in the broadcast command. `fev.ctlw2` is used as a broadcast marker and is not available for the user during this call. `fev.ctlb2` is used for broadcasting commands.
Broadcasting commands: The fev is interpreted as a broadcast command, if `fev.ctlb2` has the following format:

$$fev.ctlb2 == DD\_BRCCMD\_MARKER \mid command;$$

where command is a 16 bit integer. `DD_BRCCMD_MARKER` is defined in `dbfi.h`.
In this case the broadcast is only sent to those processes that ask for broadcast and have their broadcast command for `int cmd` defined, i.e. they previously set the broadcast command to a function by calling `ddu_set_brccmd`. Upon receiving that broadcast command, they call the routine `brc_cmd_fcn`. `ctlw1` and `ctlb1` are the calling arguments for `brc_cmd_fcn` and can be used to pass additional data to the process.

4.2.17 `ddu_set_brccmd`

Usage: `ddu_set_brccmd(int cmd, DD_BRCCMD_FCN brc_cmd_fcn)`
Returns 0 if good status, -1 if bad.
Sets the broadcast command `cmd` to the function `brc_cmd_fcn`. `brc_cmd_fcn` is a pointer to a function of type DD_BRCCMD_FCN as defined in `dd_dcom.h`.
The commands can be defined in `dd_usr_brccmd`.

4.3 System Routines

Naming convention:
User routines are named as `dds_xxx` .

4.3.1 `dds_create()`

Usage: `int dds_create();`
Returns: 0 if good status, -1 if bad status.
This routine creates the DD system if it doesn't exist or attaches to it. The process calling it becomes a DD master process. Several masters are possible. The DD system will be destroyed when all its masters have exited.

4.3.2 `dds_close()`

Usage: `int dds_close(flag);`
Returns: 0 if good status, -1 if bad status.
Argument: `int flag`
This routine is called by a DD master when he detaches from the DD system. If $flag <> 0$ the DD system will be removed. Normally $flag = 0$: In this case the last master to detach will remove the DD system.

4.3.3 `dds_wait4proc()`

Usage: `int dds_wait4proc(int ms);`
Returns: 0 if good status, -1 if bad status.
Argument: `int ms`
This routine waits until all processes attached to DD fifos are sleeping on a semaphore (their fifo is empty). It can be used to synchronize the system, e.g. in a end run events after the trigger has been disabled. Argument `ms` is a timeout value in millisec.

4.3.4 `dds_wakeup()`

Usage: `int dds_wakeup();`
Returns: 0 if good status, -1 if bad status.
Sends a signal to all processes sleeping on semaphores of their empty fifo. This routine can be used to wakeup processes after the trigger has been disabled and no more events are expected.

4.3.5 `dds_flush()`

Usage: `int dds_flush(int brc_flag);`

Returns: 0 if good status, > 0 number of processes still holding events.

Flushes the DD system by:

1. disables input (all input processes will fall asleep until input is enabled)
2. empties all non-empty fifos
3. waits until all processes attached to the DD system are no longer processing events (or timeout). Waits a maximum of `DD_FLUSH_MAXWAIT` ticks, each tick is `DD_FLUSH_WAIT_TICK` seconds long (as defined in `$dd_inc/dd_dcom.h`).
4. enables input again.

One possible use of this routine can be to remove all existing events in the DD system and make sure that all events in the system are new.

If the `int brc_flag` is nonzero, then the flush is followed by a broadcast command DD_BRCCMD_FLUSH (as set in dd_usr_brccmd.h).

5 Example Programs

5.1 A DD Producer Process

The listing below is a producer that first creates the DD system (if it doesn't exist, attaches to the INPUT_FIFO and then injects 100 event into the DD system. If there are no buffers available, the process falls asleep (FMODE_SLEEP) until one has become free.

```
#include <dd_sys.h>
#include <dd_user.h>

main()
{
    int i, j, len, *p;
    int mode, reqcnt, prescale, wait,ctl[4];

    mode = FMODE_ALL;
    wait = DD_WAIT_SLEEP;
    prescale = 1;
    for (i=0;i<4;i++)
        ctl[i] = -1;
    if ( (status = dds_create() ) != 0 ) {
        printf(" dds_create:  status returned %d \n" ,status);
        exit(1);
    }
    if ( (status = ddu_init(''INPUT'',mode,prescale,wait,ctl)) != 0 ) {
        printf("ddu_init status %d \n",status);
        exit(1);
    }
    len = 1000;
    for (i=0; i<100; i++) {
        status = ddu_req_fev(len, &fev);
        if (status == 0){
            p = fev.p2da;
            fev.len = len;
            fev.ctlw1 = i%4+1;
            fev.ctlb1 = rand() & 3;
```

```
            fev.ctlw2 = -1;
            fev.ctlb2 = -1;
            for (j=0; j<len; j++) {
            *p++ = j;
            if ( ( status = ddu_put_fev(fev) ) != 0) {
                printf("ddu_put_fev:  status returned %d \n",status);
                break;
            }
        } else {
            printf("ddu_req_fev:  status returned %d \n",status);
            break;
        }
    }

    ddu_close();
    printf("dd cleaned up\n");
    exit(0);
}
```

5.2 A DD Consumer Process

The following sample program is a consumer that requests all events, which have bit 3 of `fev.ctlb1` set. If no event is available, the consumers returns immediately. Another consumer option is commented out: It receives only events on request, falls asleep when there are no events. In order to minimize the number of unsuccessful tries to get an event, a call to `ddu_req_fev(n_events)` is made. This call makes sure that always at least `n_events` are reserved for this process (unless data taking stopped for one reason or another).

```
#include <dd_user.h>


main(int argc,char **argv)
{
    int *p;
    char fname[100];
    int mode, reqcnt, prescale, wait,ctl[4];
```

```
struct fifo_entry fev;
int go_on;
if (argc < 2) {
    printf("syntax:  argument fifo name needed \n");
    exit(0);
}

strcpy(fname,argv[1]);
mode = FMODE_COND;
wait = DD_WAIT_ASYNC;
/* mode = FMODE_ONREQ; */
/* wait = DD_WAIT_SLEEP; */
prescale = 1;


ctl[0] = -1;
ctl[1] = 4;

ctl[2] = -1;
ctl[3] = -1;
if ( (status = ddu_init(fname,mode,prescale,wait,ctl)) != 0 ) {
    printf("ddu_init:  status returned %d\n",status);
    exit(1);
}

/* if you want to have always at least 5 events */
/* waiting to be picked up by this process */
/* n_events = 5; */
/* ddu_set_reqcnt(n_events); */
ddu_start();

go_on = 1;
while (go_on & & ddu_attached() ) {
    status = ddu_get_fev(&fev);
    if (status == 0){
        printf(``got an event of length %d at address %x \n'',
```

```
                fev.len,fev.p2da);
            if ( (status = ddu_put_fev(fev)) != 0 ) {
                printf("ddu_put_fev:  status returned %d\n",status);
                go_on = 0;
            }
        } else if ( status < 0 ) {
            printf("ddu_get_fev:  status returned %d\n",status);
            go_on = 0;
        } else
            printf(``tried to get an event, but none was available\n'');
    }

    ddu_stop();
    ddu_close();
    exit(0);
}
```

5.3 A Monitor Process

The following sample program is a that stays alive as long as the DD system exists.

```
#include <dd_user.h>


main()
{
    ddu_attach();
    while ( ddu_attached() ) {
        /* monitor something */
    }
    printf(``DD system disappeared \n'');
    ddu_detach();
}
```

5.4 A Modifying DD Consumer Process

The following sample program is a process, that receives every event which has the control word `fev.ctlw1 = 1` set. It then merges two consequetive events into a single event and inject these back into the DD system[2]. Note that the `fev.ctlw1` for the new event has changed in order to avoid turning in circles. (Note: for simplicity no proper error handling is included in the code).

```
#include <dd_user.h>


main(int argc,char **argv)
{
    int len, status, status1, status2, status3
    char fname[100];
    int mode, reqcnt, prescale, wait,ctl[4];
    struct fifo_entry fev1, fev2, fev3;
    int go_on;

    if (argc < 2) {
        printf("syntax:  argument fifo name needed \n");
        exit(0);
    }

    strcpy(fname,argv[1]);
    mode = FMODE_COND;
    wait = DD_WAIT_SLEEP;
    prescale = 1;

    ctl[0] = 1 ;
    ctl[1] = -1;
    ctl[2] = -1;
    ctl[3] = -1;
    if ( (status = ddu_init(fname,mode,prescale,wait,ctl)) != 0 ) {
        printf("ddu_init:  status returned %d\n",status);
        exit(1);
    }

    ddu_start();
```

[2]Clearly not a very intelligent event builder

```
    go_on = 1;
    while (go_on & & ddu_attached() ) {
        status1 = ddu_get_fev(&fev1);
        status2 = ddu_get_fev(&fev2);
        if ( (status1 || status2) == 0) {
            len = fev1.len + fev2.len;
            if ( (status3 = ddu_req_fev(&fev3) ) == 0 ) {
                memcpy(fev3.p2da, fev1.p2da, 4*fev1.len);
                memcpy(fev3.p2da+4*fev1.len, fev2.p2da, 4*fev2.len);
                fev3.len = len;
                fev3.ctlw1 = 2;
                fev3.ctlb1 = -1;
                fev3.ctlw2 = -1;
                fev3.ctlb2 = -1;
                if ( (status = ddu_ins_fev(fev3)) != 0 ) {
                    /* handle error */
                }
                if ( (status = ddu_put_fev(fev2)) != 0 ) {
                    /* handle error */
                }
                if ( (status = ddu_put_fev(fev1)) != 0 ) {
                    /* handle error */
                }
            else {
                /* handle error */
            }
        else {
        /* handle error */
        }
    }
    ddu_stop();
    ddu_close();
    exit(0);
}
```

6 Status and Error Reporting

All DD routines (except the monitor programs) do not print any status or information to `stdout` or `stderr`. They instead write the information into a string and call a routine `dd_log`.

The syntax is `dd_log(int info_id, char *msg)`, where `info_id` is an identifier for the string message. The currently supported `info_id`' s are `DD_INFO` and `DD_ERROR`.

`dd_log` currently simply prints the string to stdout or stderr (depending whether `info_id` is `DD_INFO` or `DD_ERROR`.

This behaviour can of course be changed by simply changing `dd_log`, e.g. popping up a window displaying the information, a feature that might be desirable in a distributed system.

7 Customizing the DD System

7.1 User Entries

Currently two user entries are supported. Both are currently in the file `dd_usercode.c`.

7.1.1 `ddu_link_signal()`

This (by default dummy) routine is called when the DD system defines its signals. It can be used to add user defined signals to the code.

7.1.2 `ddu_def_static_fifos()`

This routine is called after the DD system is initialized. It allows to add static fifos (i.e. fifos that exists regardless whether there are processes attached to them or not). They can be used if a certain order in the fifo chain is desired. The default user routine contains a static fifo TAPE as example (commented out).

7.2 User Configurable Event Selection

As described earlier four control words (`fev.ctlw1, fev.ctlw2, fev.ctlb1, fev.ctlb2`) are used to distribute the events (the first two operate on a equal basis, the other two on a bit basis). The conditions are implemented as a macro and can therefore easily be changed by the user (file $DD_INC/dd_fifo_conditions.h). The relevant listing is given below:

Code fragment for deciding whether to put the event into this fifo

```
else if (p2f->fhdr.fmode == FMODE_COND){
    if ( (p2f->fhdr.fctlw1 != -1) & & FIFO_COND1 )
        this_fifo = i;
    if ( (p2f->fhdr.fctlb1 != -1) & & FIFO_COND2 )
        this_fifo = i;
    if ( (p2f->fhdr.fctlw2 != -1) & & FIFO_COND3 )
        this_fifo = i;
    if ( (p2f->fhdr.fctlb2 != -1) & & FIFO_COND4 )
        this_fifo = i;
}
```

Fifo conditions as implemented in $DD_INC/dd_fifo_conditions.h

```
#define FIFO_COND1 (p2f->fhdr.fctlw1 == fev.ctlw1)
#define FIFO_COND2 (p2f->fhdr.fctlb1 & fev.ctlb1)
#define FIFO_COND3 (p2f->fhdr.fctlw2 == fev.ctlw2)
#define FIFO_COND4 (p2f->fhdr.fctlb2 & fev.ctlb2)
```

7.3 User Configuration Options

7.3.1 DD Configuration

By default the DD system is initialised to a rather large number of fifos and buffers (as defined in the header files `db.h` and `fifo.h`. These values are maximal allowed values. If they turn out to be too small, then the code has to be recompiled.

On the other hand one often wants to have less semaphores and smaller shared memory segments. This can be achieved by pointing the environment variable `DD_CONFIGURATION` to a file specifying the configuration. In this case only the specified number of semaphores and shared memory segments are created. The following variables have to be defined:

`max_fifo_nb`	maximum number of fifos
`fifo_depth`	fifo depth (number of entries per fifo)
`db_maxnbev`	maximum number of event buffers
`db_maxevsize`	maximum length of the event buffers
`db_max_buf_vol`	maximum number of volatile buffers.

- The maximum event size `db_maxevsize` is the event size for the static buffers. If an event is of bigger size than `db_maxevsize`, additional memory is alloced (but not more segments than `db_max_buf_vol`. The system should be set up in a way, such that volatile buffers are only rarely needed.
- The fifo depth should be bigger than the number of buffers (static and volatile).
- The configuration file `$DD_CONFIGURATION` is read as the system is set up. If at later times processes with other configuration values are attaching to the DD system, they will not change the configuration values (an error message is printed).

7.3.2 Usage of UNIX signals

Several UNIX signals are currently used as defined in `$DD_INC/dd_signal_def.h`.

- `BROADCAST_SIGNAL` (default: SIGUSR2)
  This signal is used to broadcast events to the processes (that have their broadcast flag set to `DD_BRC_SIGNAL`).

- `DD_USER_INTERRUPT` (default SIGINT):
  This signal can be used if the user wants to interrupt a consumer process sleeping on a semaphore by hand. If DD_USER_INTERRUPT is set to NULL, then no signal will be linked.

- `DD_SHUTDOWN_INTERRUPT` (default: SIGHUP)
  When the garbage demon notices that the DD system has no more masters attached, it removes it. Before doing so, he sends the `DD_USER_INTERRUPT` to the remaining process (given he has the permission to do so) before removing the shared memory segments and semaphores.
  If the garbage demon receives a SIGHUP, he will shut down the DD system (regardless whether there are still master processes attached to it or not).

7.3.3 User Changes to the Event Distribution

The decision whether an event is put into a fifo, depends upon the control words (see section on fifos). The conditions are macros, that can be changed. They are defined in `$DD_INC/dd_fifo_conditions.h`.

7.3.4 Action upon Detection of a Fatal Error

The user can determine what happens in the case of a fatal error by setting environment variables. If a process encounters a fatal error in the DD code, it executes the routine `dd_error_inv()`.

- If the environment variable `DD_ERROR_GARBAGE_D` is set, then only the garbage demon will execute the error handlers.

- If the environment variable `DD_ERROR_MSG` points to an executable, then a process is execlp. Currently this is a X11 window telling the user to close the DD system.

- If the environment variable `DD_ERROR_SD` is set, then the DD system is closed down automatically.

7.3.5 Multiple Copies of the DD System on the Same Host

If one wants to have multiple independent copies of the DD system on the same host, then one must specify the environment variable `DD_NAME` before `ddu_attach()` or `ddu_init` is called. Currently a process can only be part of one DD system.

8 Network Connection

8.1 Overview

The DD code contains a network section, which allows to retrieve and insert events over the network. The main design goal has been that the interface to the user is the same between local connections and connections over the network. For example the user calls the routine `dds_create` or `ddu_attach()` regardless whether the DD system is locally or remotely.

A client server architecture has been choosen as the actual implementation. A server must be running on each node, where a DD system is running or supposed to run. This server listens on a socket for incoming requests and forks of another server to which the client sends the DD related commands. The client code does not have to be changed in order to run the DD code over the network. TCP/IP has been choosen as the network protocol.

8.2 The `dd_server` program

The DD network server is called `dd_server`. It is mandatory that this server is already running if the DD system is accessed over the network. Several scripts are available that allow to start and stop the servers automatically. Of course the servers can also be started and stopped manually.

8.2.1 Server Configuration Files

There are two main configuration files for using the DD system over the network: (see the examples in the directory `$DD_DATA`)

- `dd_server_configuration_file`
- `dd_configuration_file`

The `dd_server_configuration_file` is a necessary configuration file. In this file the hosts, top DD directory, DD version and the (optional) `dd_configuration_file` are defined. If the hosts have a common disk mounted, the DD top directory and version will be normally the same.

The file `dd_configuration_file` can optionally be specified. It contains a list of

DD systems, their keywords (i.e. `$DD_NAME`), node and optionally a configuration file for the DD systems that specifies the number of buffers, number of fifos, the maximum event size etc etc

8.2.2 The Scripts

`$DD_BIN/start_all_dd_servers`:
Usage: `$DD_BIN/start_all_dd_servers dd_server_configuration_file`
This script scans the file `dd_server_configuration_file`, specified as its argument, and starts the server on every node listed in that file.

`$DD_BIN/stop_all_dd_servers`:
Usage: `$DD_BIN/stop_all_dd_servers dd_server_configuration_file`
This script scans the file `dd_server_configuration_file`, specified as its argument, and stops the server on every node listed in that file.

These calls are done using `rsh` commands. If the (optional) file `dd_configuration_file` is specified in the file `dd_server_configuration_file`, the environment variable for `$DDD_CONFIGURATION_FILE` (distributed DD configuration file) is defined for the server and points to that file. In that case the server will check for every DD system that he is expected to create, whether that particular DD system is declared in that file. If not, the server will not allow that the DD system is created.

8.3 Attaching to a DD System over the Network

There are two ways how a process can attach to a DD system on another host:

- Using `$DD_NAME`:
  If the environment variable `$DD_NAME`, which specifies the name of the DD system to which one wants to attach, has the form `$DD_NAME = dd_name@host_address`, a communication with the `dd_server` on that host is attempted.

- Using `fifo_name`:
  If a process calls `ddu_init` with a FIFO name of the form `fifo_name@host_address`, a connection over the network is attempted.

If both possibilities are attempted, it is demanded that the host_address is the same. Note that user process can also become DD master over the network by calling `dds_create()` and the same distinction between masters and non-masters applies over the network as on local DD systems.

8.4 The dd_link between DD Systems

The program dd_link allows to link two DD systems over the network. Its syntax is:

dd_link from_dd to_dd <attach_mode_file>

where:	from_dd	the DD system from where the events are retrieved,
	to_dd	the DD system to where the events are being sent
	attach_mode_file	an optional filename indicating in which mode dd_link attaches to the from_dd system.

Note: The from_dd, to_dd DD systems have to be defined in the file $DDD_CONFIGURATION_FILE.

It required that dd_link starts runs on the host, where from_dd resides. It creates a fifo named ''->to_dd''. By default it attaches in sleep wait mode and retrieves events ONREQ basis. No event selection is done. This can be changed by specifying the optional parameter <attach_mode_file> (see $DD_DATA/dd_link_conf1 as an example).

Upon succesful connection to the from_dd system (as a master), dd_link starts a server on the to_dd host which attaches to the INPUT_FIFO of the to_dd system (as a master).

dd_link should be considered as an example how one can connect two DD systems. Currently there are hooks for either transfer one event at a time (do_dd_trf or a block mode do_dd_mtrf, in which many events are transfered at a time. The file $DD_INC/dd_link_util.h has the default definitions which can be changed according to everyone's flavor and taste.

9 User Interface

9.1 dd_ui User Interface

This user interface (figure 4 allows to attach to / create the DD system. A child process dd_ctl is forked off and its standard output is attached to the text widget. A pull down menu allows to select commands, a text field at the lower left corner accepts commands directly. The interface is written in Tcl/TK [3]. The executable is $DD_BIN/dd_ui.

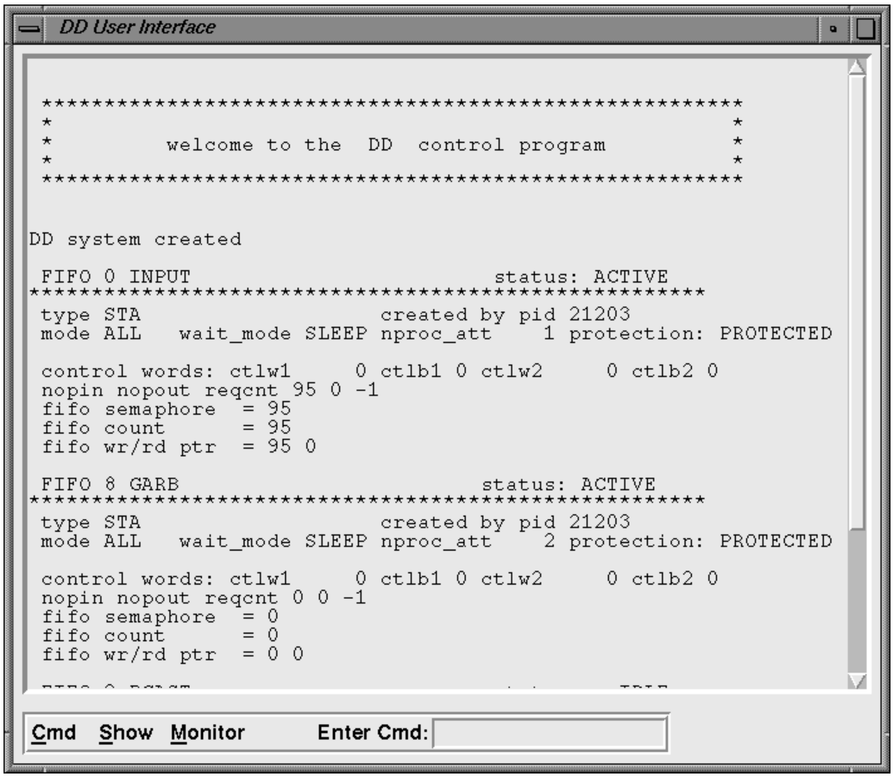

Figure 4: *The* dd_ui *user interface.*

9.2 Process Monitor Interface

This user interface allows to monitor all or selected processes. The main window (figure 5) displays by default all the processes attached to the DD system. The process name, total number of FIFO read/write operations

[3]There is also an older X11/OSF-MOTIF based version available in case Tcl/Tk is not installed ($DD_BIN/dd_ui_color)

and the number of FIFO read/write operations in the last 60 seconds are displayed and continually updated. By pressing the ''select'' button, another window pops up (figure 6) that allows to remove/add processes to the process monitor window. The executable is $DD_BIN/ddp_ui. This window can also be obtained from the dd_ui user interface (pulldown menu monitor).

Figure 5: *The* ddp_ui1 *user interface.*

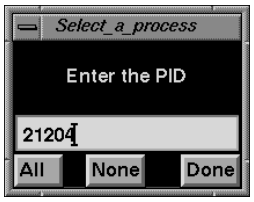

Figure 6: *The* ddp_ui *process selection interface.*

9.3 Fifo Monitor Interface

Two user interfaces for monitoring the FIFOs are currently available. The first one (figure 7) is written in Tcl/Tk and continually displays the FIFOs that are currently in use. Fifos that get created/destroyed will be added/removed. The executable is $DD_BIN/dd_fifo_mon. This window can also be obtained from the dd_ui user interface (pulldown menu monitor).

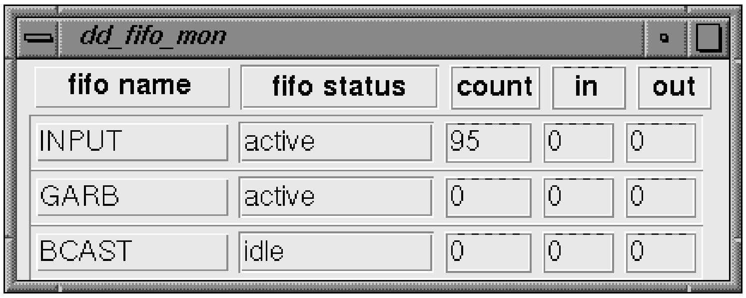

Figure 7: *The fifo monitor user interface.*

Another user interface (figure 8) is a sighly different version and will be replaced in the near future. The executable is $DD_BIN/ddr_ui.

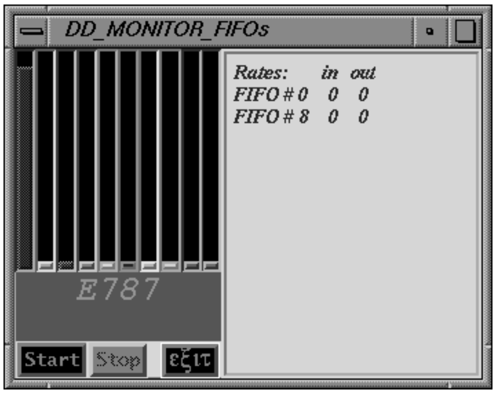

Figure 8: *The fifo monitor user interface.*

10 FAQ

- **How to initialize the DD system?**
  One process calling dds_create().
- **How to remove the DD system?**
  Send a SIGHUP (more precisely $DD_SHUTDOWN_SIGNAL to the garbage demon.
- **How to remove one process from the DD system?** Send a SIGHUP (more precisely $DD_SHUTDOWN_SIGNAL to this process.
- **When does the DD system cease to exist?**
  When the garbage demon sees that there are no more processes attached to it. Optional: When a fatal error is detected or the garbage demon receives a SIGHUP signal.
- **How to be a DD producer?**
  Attach to the INPUT fifo.
- **How to be a DD consumer?**
  Attach to a fifo of your choice.
- **How to attach to the DD system?**
  Call ddu_attach().
- **How does a process know, whether it is attached to the DD system and the latter still exists?**
  Call ddu_attached().
- **How to have multiple DDs running on the same host?**
  Define the environment variable DD_NAME.
- How to attach to a DD system over the network?
  Define DD_NAME as dd_system@node.
- **What happens if a user process dies?**
  Under normal conditions an exiting process calls ddu_close upon exit (C routine atexit). However this doesn't work in all cases (e.g. when running a program crashes while running in certain debuggers). In this case the fifo can be filled with empty buffers. The garbage demon will detect this situation and remove the fifo (if the fifo is not protected - see ddu_set_protection.

- **Why are the shared memory segments and semaphores still present for a few seconds after the last process detached from the DD system**
  `The garbage demon typically waits a few seconds before he closes down the DD system to make sure that it is really not in use anymore.`

- **Which are the most important demonstration programs?**
  `They are all located in $DD_BIN:`

  - `dd_test: A simple test program that attaches to the DD system and accepts commands from the standard input.`
  - `dd_test_p : A simple producer.`
  - `dd_test_c : A simple consumer.`
  - `dd_ui : An X11/OSF motif based user interface to the DD system.`
  - `ddr_ui: A user interface monitoring the fifos.`
  - `ddp_ui: A user interface monitoring the processes.`

- **How to get rid any processes that are still alive?**
  `Call dd_cleanup. Note: you may want to change this script.`

- **Where to complain and report errors?**
  `witzig@bnl.gov`