THE EXECUTION OF A VIDEO EDITING CONTROLLER

by

John D. Barbour

SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

BACHELOR OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1986

Signature of Author _____

Department of Electrical Engineering and Computer Science
May 16, 1986

Certified by _____

Glorianna Davenport
Thesis Supervisor

Accepted by _____

Professor David Adler
Chairman, Undergraduate Thesis Committee

THE EXECUTION OF A VIDEO EDITING CONTROLLER

by

John D. Barbour

Submitted to the Department of Electrical Engineering and Computer Science on May 16, 1986 in partial fulfillment of the requirements for the degree of Bachelor of Science.

Abstract

The Experimental Video Workstation is a design project in computer-aided editing. This paper focuses on the execution software that controls the laser disc players, a record deck, and a video switcher for frame accurate edits. It also discusses the challenges encountered by controlling live machinery with computer software.

Thesis Supervisor: Glorianna Davenport
Title: Lecturer, Film/Video Section, Department of Architecture

Dedication

To Jesus Christ, who got me through MIT alive. And I'd also like to thank all my co-workers on the Experimental Video Workstation for their help on this project, and Sherry Solden for making this stay at MIT a lot more fun.

Table of Contents

List of Figures

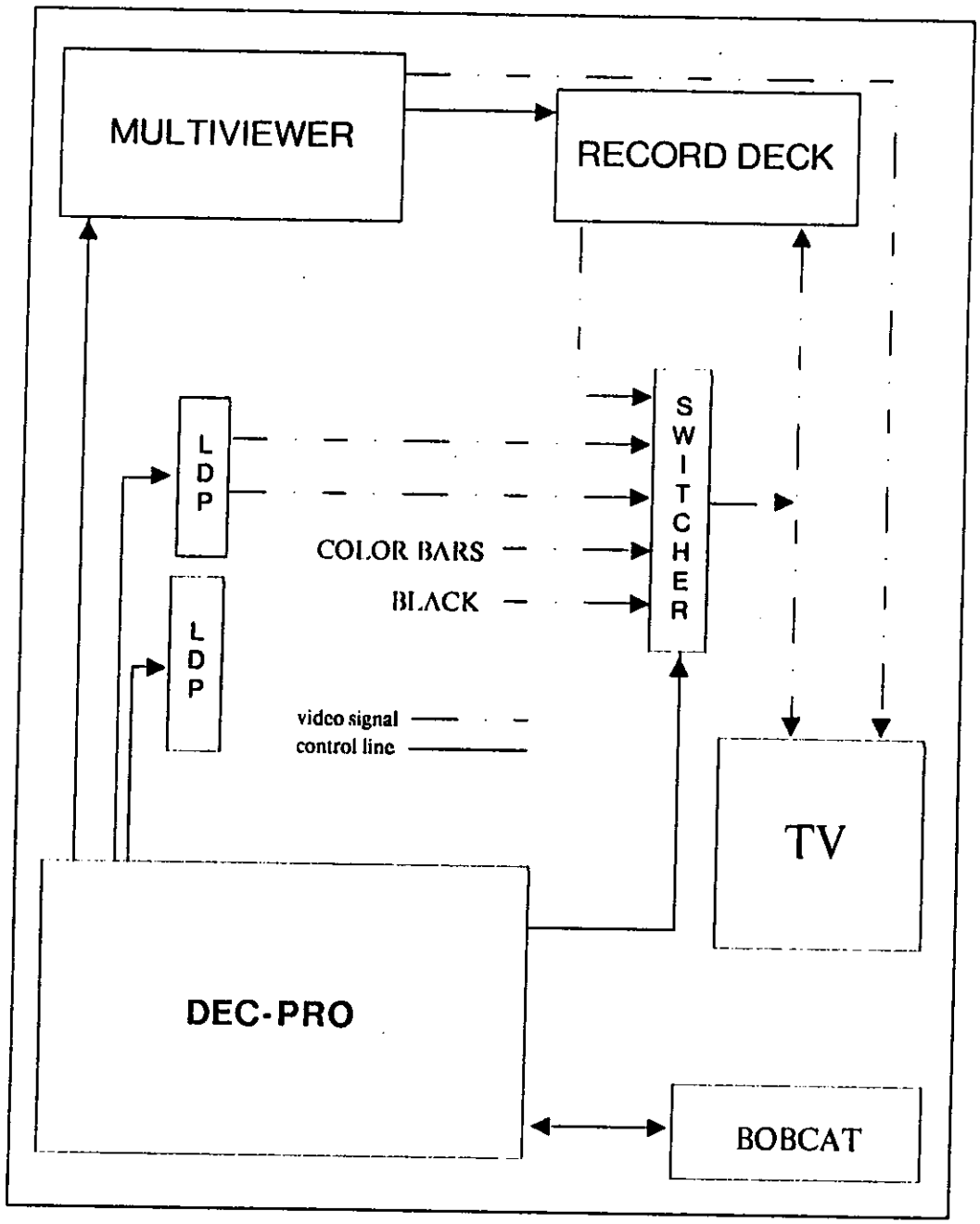
Chapter 1

System Overview

The Experimental Video Workstation is a design project in building a personalized video editing system. This will allow the artist who shoots the film or video footage to edit the material himself in a computer-aided environment. The system is divided into a few basic sections: logging the video footage into a database, creating the EDL (edit decision list), and then executing the EDL to create the final print. The database consists of digitized film or video frames accompanied by any verbal description that the artist wishes to include. Each digitized frame represents a segment of the film or video footage, and this information helps the artist decide what edits to make. With this information, the edit decision list can be constructed and then stored for future use and modification. The latter section is the one of primary concern to this paper. The system is set up with an HP-Bobcat, a Dec-Pro, an Asaca Multiviewer, an Asaca Video Switcher, 2 Sony 1000A, Laser Disc Players (LDP), and a Sony BVU-820 Record Deck. A Multiviewer is a device that allows the editor to view up to thirty-two frames of footage at once. These devices are configured as seen in Figure 1-1. The logging database and the edl manager are run on the Bobcat, while all the machine control and execution software run on the Dec-Pro.

The entire system was set up as a simulation of a future design goal. The software, in this case, was written for the specific machines that were available, but the

Figure 1-1: System Configuration



future goal for the system is to accommodate any arbitrary device. The system was set up in this way because of the availability of the equipment. Much of the equipment was donated to the project, and many of the design decisions were directly affected by the equipment available. The Dec-Pro was used for machine control since it had the required number of serial ports available. The Bobcat would have been selected to handle the entire project, but the hardware for using the serial ports was not available. These machines were chosen primarily for the fact that they ran a UNIX operating system, and this would allow the system to be portable. The language used for the software was C. This is an improvement from the programmer's point of view, since it provides a higher level language rather than the traditional use of assembly language. The code can be easily understood by others, allowing for simplified modifications and improvements.

The laser disc players were used as the input devices for the video material. They provided the benefit of quick, random access to material and a simple serial interface. The record deck was used for recording the final output of the edit as specified by the edl with the discs as input. These devices were also connected to the multiviewer and when practical were controlled through this device. The multiviewer allowed for viewing more than one frame of the video input at any one time, and the video switcher was used for wipes and dissolves to add variety to the editing capabilities.

Chapter 2

Program Input

2.1 Edit Decision List

The execution software was developed to read in an EDL and then perform it on the machines to record a final edit. The purpose of an EDL is to store the decisions made by the editor as a permanent record. This way another recording could be made automatically without having to make the decisions all over again. The repeatability of the EDL is its strong point. With a flexible EDL maintenance program, even small changes can be handled with a minimum of hassle. The idea of storing edit decisions in a file allows them to be portable between systems, which accept the same data format. Other than just having compatible file systems, the information needs to be standard as well. The CMX format is the most common industry standard and therefore, was the reason it was chosen for this system. The format is set up as ASCII characters, and not binary; this avoids problems with byte swapping or different sized integers when changing computer environments. A sample CMX EDL can be seen in Figure 2-1. The EDL contains a list of events specified by an event number and all the specific information about which devices are to be used for input, what type of transition is to be made, and the exact timecodes for the source footage and record points.

Evt	Dev	A/V	Trans	Dur	Source In	Source Out	Record In	Record Out
001	001	B	C		01:28:54:19	01:29:11:28	01:01:00:00	01:01:17:09
001	002	B	D	01:00	01:27:11:15	01:27:14:05	01:01:17:09	01:01:19:29
002	002	B	C		01:28:17:01	01:28:26:07	01:01:19:29	01:01:29:05
003	AUX	B	C		01:26:25:24	01:26:27:12	01:01:29:05	01:01:30:23
004	002	A2V	C		01:26:40:21	01:26:49:29	01:01:30:23	01:01:40:01
005	001	A1	C		01:25:00:09	01:25:21:17	01:01:40:01	01:02:01:09
006	002	A1	C		01:29:34:05	01:29:59:29	01:02:01:09	01:02:27:03
007	001	V	C		01:25:00:09	01:25:21:17	01:01:40:01	01:02:01:09
007	002	V	W000	01:00	01:29:34:05	01:29:59:29	01:02:01:09	01:02:27:03
008	BL	B	C		00:00:00:00	00:00:05:00	01:00:55:00	01:02:30:00
512	001	B	C		00:00:00:00	00:00:00:00	00:00:00:00	00:00:00:00

Figure 2-1: Edit Decision List

2.2 SMPTE Timecode

Timecode is an industry standard set up by SMPTE (Society of Motion Picture and Television Engineers) to index video footage. Each frame of videotape is indexed by a timecode which is expressed in hours, minutes, seconds, and frames. This allows for precise access to each frame of video and audio material. These timecodes are then used in the EDL for the exact specification of frames. This allows the editor to be as accurate as he desires when choosing his edit points. Unfortunately standards never cover everything; video discs have yet another method for accessing individual frames. They are indexed by frame numbers for one-half hour of material, from 1 to 54000. These numbers then have to be transformed into virtual timecodes for use in the EDL. When reading each event from the EDL, it is necessary to convert each timecode into a frame number for accessing an individual frame from a laser disc player, as well as for mathematical computation and comparison with other values.

These methods for indexing the video medium are a mixed blessing. Timecodes have the advantage of offering the editor the information pertaining to how long his production is; but unfortunately, it is very difficult to perform arithmetic operations upon them directly. Conversely, frame numbers offer the flexibility of computation but do not immediately display standard time information pertaining to duration without performing a minor calculation. For total flexibility within this system, each timecode is stored in a structure that breaks apart the components of the input time code and creates a frame number, as well as the character representations of both the timecode and frame number. The other input information from the EDL includes the exact device to be used, the audio and video channels selected for the edit, and the type of transition to be used: cut, wipe, or dissolve. All this information will be needed in the future for smooth operation.

Chapter 3

Making the Edit

The editing process requires the control of asynchronous hardware. This adds the challenge of controlling live machines that are not constrained within the same environment as the software. Since things are moving through time, it is critical that the software be able to keep the system under control at all times.

3.1 Setting it Up

Setting up an edit is not time critical, for the most part. The trouble starts when it is time to perform the edit. The video output of the laser disc players and record deck are hooked up through the multiviewer, but the control lines of the LDPs are hooked directly to the Dec-Pro, this will become important later. All the remote machine commands are accessed through a function called **control**. **Control** accesses the Multiviewer, selects the proper device and then sends the appropriate command to operate that device. **Control** was designed to be called with keyboard commands and the user in mind. It pairs together the machine control with the video output control. This means that whatever machine 1 is controlling, that is also the machine that is producing the video output. A sequence of commands that would play device A and device B would be the following:

This technique, however, does not provide the flexibility that is needed for performing an edit sequence.

```
control(avtr);
control(play);
control(bvtr);
control(play);
```

Figure 3-1: Sample Control Sequence

Through this type of control flow, the switcher is set to the proper channel; and the record deck is set up for the proper audio and video channels. The devices are then prerolled to their respective start positions. This preroll start point is five seconds before the start of the edit desired. This allows time for the devices to get up to speed, accurately spaced and locked to an external synchronization so the edit will be performed properly. Once the devices are cued to the appropriate points, then the program enters the time critical phase of making the edit.

3.2 Synchronization

3.2.1 The Problem

The start of the edit requires synchronizing the LDPs to the record deck. Synchronizing is the process of putting the devices at a constant offset while they are playing. The two in-points in the EDL have an offset that they should be at when the edit starts. That offset is what the program tries to achieve during the preroll time. Once this offset is achieved, the devices are locked and and they will stay at this offset for the remainder of the edit, since both devices are gen-locked -- servo-driven to produce synchronous video. Testing to find out whether or not the two devices are at the proper offset is the semi-complicated part. This is difficult because the machines are moving rapidly through time. Accessing the frame number or timecode from a

device is time consuming and not instantaneous. This is where dealing with the real world becomes a main consideration.

3.2.2 Compromising Perfection

The first sacrifice to be made is that of modularity. Calling functions through **control** is very time consuming. The quickest thing to do is call the machines directly, taking advantage of the fact that the LDPs are hard-wired to the Dec-Pro. At least the delay within **control** is averted. Regrettably, the problem is not all-together avoided; the record deck cannot be directly wired to the Dec-Pro because it requires an RS-422 serial interface instead of an RS-232 interface. The problem is that the RS-422 interface operates at a rate four times that of the RS-232 interface.

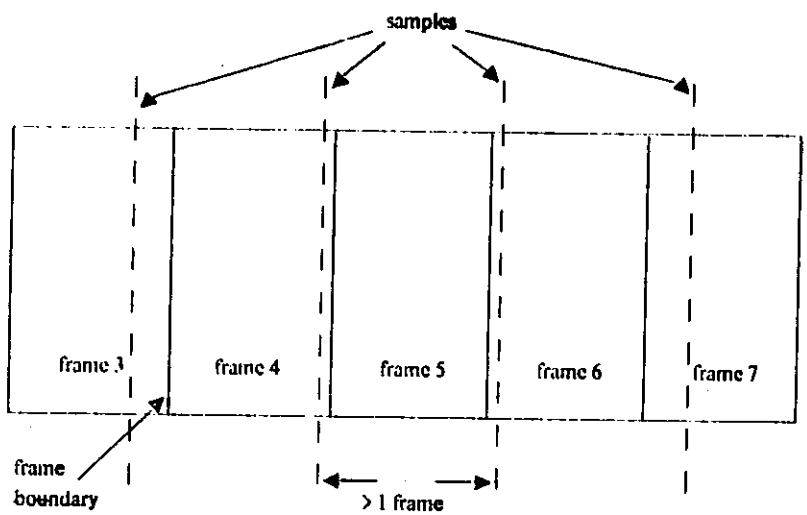


Figure 3-2: How Sampling Rates Affect Accuracy

Although **control** can still be bypassed, the Multiviewer cannot be avoided. It is soon discovered that even this set up cannot keep the system totally under control. The

delay is not anywhere near being negligible. It was determined by running a test loop that accessed frames continually, that every eleventh frame was being skipped. This means that the total cycle time is longer than one frame in duration. Because of this delay, the in-point will be missed, and as a result the edit aborts and tries again. See Figure 3-2.

Another problem encountered because of delay is precise synchronization. If the two read statements cross a frame boundary then the offsets will differ by one extra frame and will register an error, even though the devices are cued and running correctly. Figure 3-3 demonstrates a situation where the devices are synchronized, but Sample 2 registers an error while Sample 1 is correct. A tolerance factor must be added in to allow for the mathematical function to operate properly in view of real constraints, otherwise it will never believe that its job is done and will continually jump around trying to make the offsets match perfectly, even if they already do. This overprecision is a problem when dealing with the real world. Arithmetically one can be exact, but it may not always coincide with life. By trying to be exact, the system jumps around overcompensating. This problem is amplified in laser disc players since they are very responsive. But when the system allows for a slight tolerance, it performs very quickly and still remains accurate.

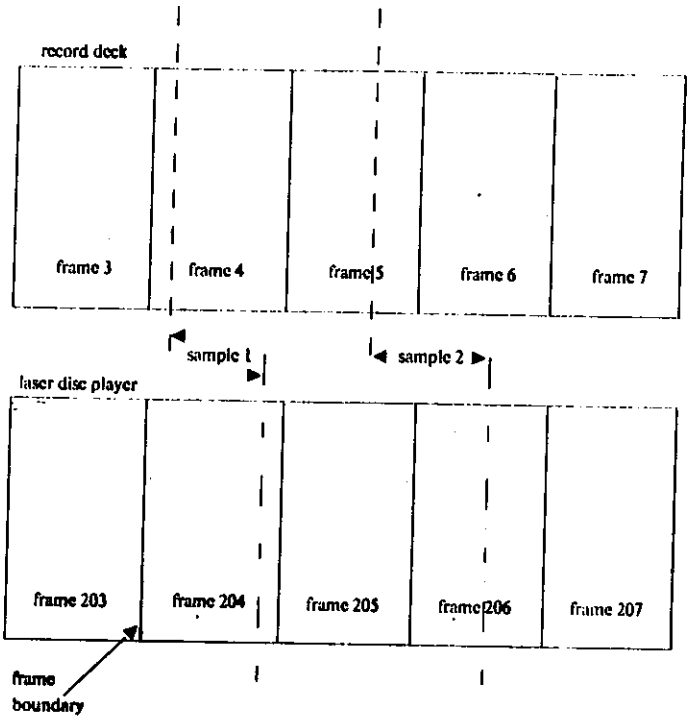


Figure 3-3: Synchronization Error

3.3 The Edit Loop

Once the devices are in sync, the program continually samples the frame numbers looking for the one that begins the edit. The record function is then activated, and the edit takes place. During the edit, the program continues to sample the frame numbers in order to find the end of the edit sequence; and when it finds the out-point, the record function is terminated; and the devices are stopped. The next event is then executed. This basic loop will now be examined more closely.

The edit loop is set up so that the only tasks performed are the reading of frame

numbers, the comparison of such to the in-points and out-points of the sequence, and the activation of only those functions that are absolutely necessary. The loop is designed so that if an inframe is missed or the devices do not synchronize, the edit aborts and the program repeats the event. See Figure 3-2. The edit loop will try each event a maximum of three times before aborting and proceeding on to the next event.

3.4 Dealing with Real Machines

3.4.1 Doing it Simple

The primary design goal in executing the EDL is to be completely frame accurate. A further obstacle to overcome, beyond that of the frame sampling, is the delay associated with controlling real machines. The length of time it takes a machine to respond to a command is again non-zero. This initial delay of start up is dealt with by the synchronization routine. But after that there is still a need to call a function well in advance of when it is desired. How far in advance then becomes the question. An advantage that video has in this case is that it can be viewed and measured even though it is moving at a high rate of speed. Each frame is tagged with a timecode or frame number. This allows for viewing the recorded edit, frame by frame, to determine if it has been recorded correctly. Using the process of trial and error, edits were executed and then inspected to see how much actual delay was involved, thereby producing the appropriate offsets for a variety of functions. In particular, the switcher was an interesting device, since each of sixty-four different wipes had a unique offset. The presence of readable time code proved to be invaluable in allowing for a precise measuring of a medium that sends a frame by every thirty-three milliseconds.

3.4.2 Adding Complexity

Once the system worked for one source, then it was possible for another source to be added. Hooking in the second source was, for the most part, duplicating the framework of the first section; except that it had a few more complications. An additional piece of hardware was being added, the video switcher. This piece of equipment also comes with a whole new set of functions to be incorporated. Not only that but two more general types of transitions must now be dealt with, wipes and dissolves. These functions require a second source to operate in sync with the first source.

3.4.3 Making a Hard Problem Harder

Synchronization is the part that is hardest hit by adding the second device. This means that within that five second preroll, a second device must be synchronized. Since the first synchronization takes about one to two seconds, this leaves one or two seconds before the devices are committed to the edit. The problem with the sampling rate becomes worse. This time three devices are being sampled. Rather than degrading what is already bad enough, each synchronization of a disc is done individually with the record deck. This works well since there is enough time available during the five second preroll segment. The final test before the edit is the most complicated, since three devices are to be sampled. This complication is minimized by the strategic sampling method within each sync loop: the primary device first and then the record deck; and in the second loop, the record deck first and then the secondary device next. And when the final test is made, the order of the respective sampling is as follows: the primary device, the record deck, and the secondary device.

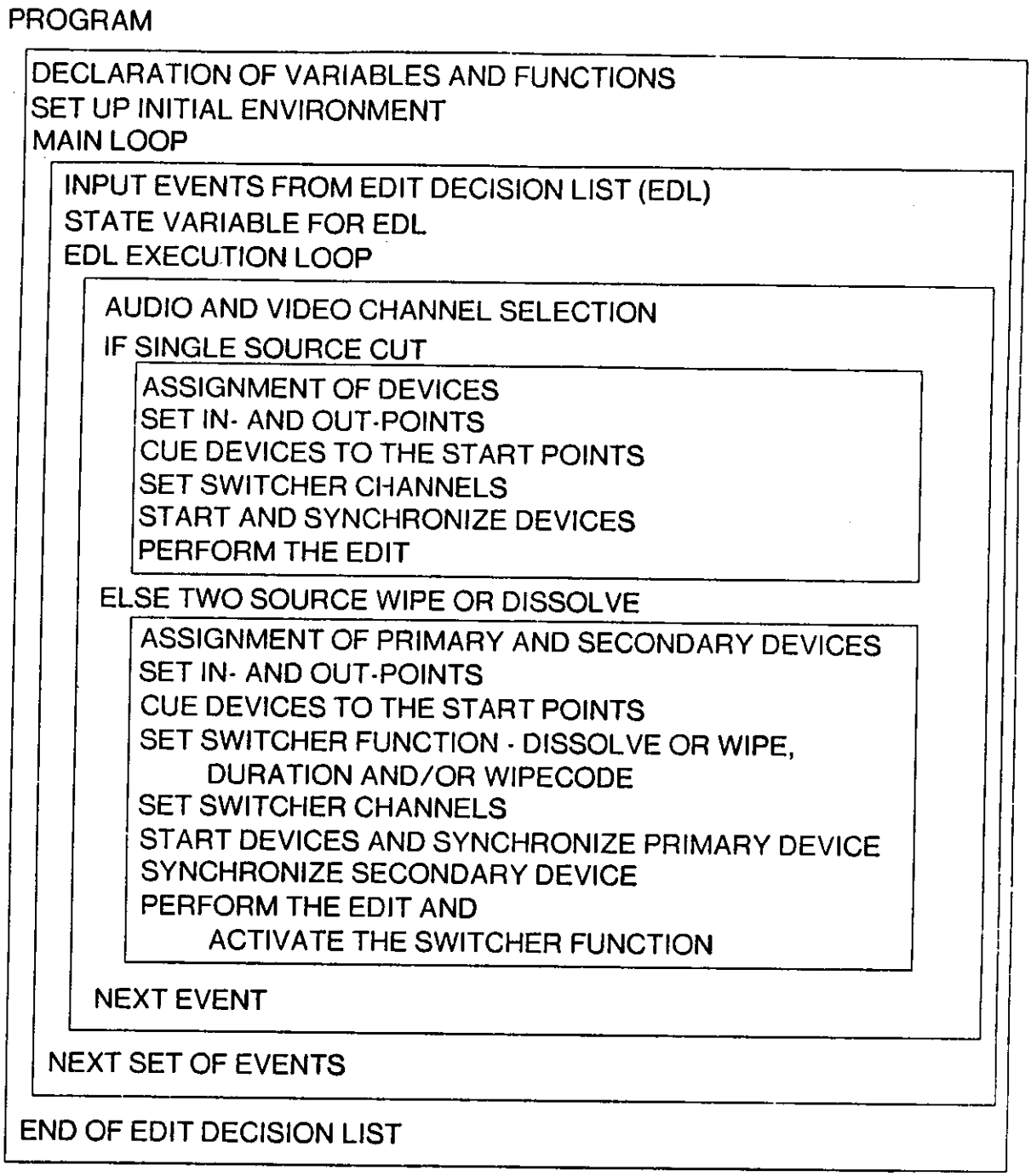
3.5 Dealing with Non-Real Machines

At the other end of the spectrum, the problem of not having any device must also be accounted for. Any direct signal input through the switcher is just as valid as any signal provided by a machine hooked into the system. This system uses two signals -- color bars and black. Another possible origin of just a signal could be a live camera. Commands such as **play** and **stop** suddenly have no meaning; and since the Multiviewer is going to expect an acknowledgement for any command sent, the system will come to a stand still since color bars and black will not return anything. This is why before any source or record device command is sent, a check is made to make sure there is a device available.

3.6 The Program Structure

The program structure consists of some basic loops for controlling each section of the process. The general overview of this code can be seen in Figure 3-4 while the entire program code is included in the Appendix.

Figure 3-4: Structure of Execution Unit



Chapter 4

Conclusion

A major simulation of the Experimental Video Workstation was taken to the National Association of Broadcasters Conference in April 1986. It was displayed with the intention of presenting a design idea for manufacturers of video equipment to consider in the future. As the video industry is heavily inundated with computers, the project has demonstrated its potential as a usable system. It has also elucidated the challenge for applying computer control to the video editing process. The future work of this project rests in providing flexibility to control any arbitrary device used in the editing process.

Appendix

```
/* J. BARBOUR 4-1-86    */
/* FILM/VIDEO MEDIA LAB */
/*
 * This version performs a simple edit within 1 frame at the start of
 * the edit.
 */


#define SCOPE

#include <stdio.h>
#include <fcntl.h>
#include <gened.h>
#include <funcs.h>
#include "global.h"
#include "include/ldpcmds.h"

/*
 * This structure is created to maintain timecodes and all its
 * permutations: frame numbers and character strings.
 */

   struct reftime
   {
      int      hour;
      int      min;
      int      sec;
      int      frame;
      char     tcno[10];
      char     frameno[10];
      long     frm;
   };

/* This structure is created to hold one event line of the edl */

struct line
{
   int      eventno;              /* The event number */
   char     device[4];            /* The source device */
   char     av[4];                /* The audio and video channel select */
   char     shot[5];              /* The type of shot: cut, wipe,
                                   * dissolve */
   struct reftime dur;            /* Duration of wipe or dissolve */
   struct reftime qin;            /* Source in-point */
   struct reftime qout;           /* Source out-point */
   struct reftime rin;            /* Record in-point */
```

```
    struct reftime rout;          /* Record out-point */
};


johnblock(filename)
    char   *filename[20];         /* Filename up to 19 characters long */
{
    int     result;
    int     i, j, k, x, u, y;     /* counters for loops */
    int     a, v;
    char    linebuf[85];          /* safe buffer to read in edl line */
    int     edllen = 79;          /* edl line length determined by input
                                   * file */
    int     fd;
    int     loop, start;          /* variables used to determine what edl
                                   * lines have been executed */
    long    atol();
    long    getframe();           /* get frame number from the laser
                                   * discs */
    long    rvgetframe();         /* get timecode from the record vtr */
    long    tcc();                /* convert timecode to frame number */
    long    convert();            /* convert timecode from input string
                                   * to frame number */
    long    ftotc();              /* convert frame to timecode */
    long    avm();                /* set audio and video modes */

    /* CONTROL CODES */
    /*
     * These control codes are used to send the appropriate characters
     * to the 'control' function
     */
    static TBYTE play[2] = {'c'};
    static TBYTE recdev[2] = {'a'};
    static TBYTE cue[2] = {'n'};
    static TBYTE getin[2] = {'m'};
    static TBYTE getout[2] = {','};
    static TBYTE pause[2] = {'↑'};
    static TBYTE slow[2] = {'b'};
    static TBYTE fast[2] = {'x'};
    static TBYTE stop[2] = {'v'};
    static TBYTE quit[2] = {'q'};
    static TBYTE edit[2] = {9};
    static TBYTE frmdisp[2] = {27, 'P'};
    static TBYTE tcdisp[2] = {27, 's'};
    static TBYTE recpreroll[2] = {'r'};
    static TBYTE setin[2] = {'i'};
    static TBYTE setout[2] = {'o'};
    static TBYTE video[2] = {'u'};
    static TBYTE audio[2] = {'y'};
    static TBYTE setasw[2] = {'$'};

    /* DEVICES */
    static TBYTE rvtr[2] = {'a'};
```

```
static TBYTE avtr[2] = {'s'};
static TBYTE bvtr[2] = {'d'};
static TBYTE aux[2] = {'k'};
static TBYTE blk[2] = {'l'};
/*
 * These are the character strings the input will be compared
 * against
 */
static char dev1[4] = "001";
static char dev2[4] = "002";
static char devaux[4] = "AUX";
static char devax[4] = "AX";
static char devblk[4] = "BLK";
static char devbl[4] = "BL";
/* Used for assignment of devices */
TBYTE   primary[2];
TBYTE   secondary[2];
int     pd;
int     dpd;
int     sd;
int     dsd;

/* variables set up for doing wipes and dissolves */
char    du;
char    wc[4];
int     wipecode;
TBYTE   function[2];
static TBYTE wipe[2] = {']'};
static TBYTE dissolve[2] = {'['};

/* CALCULATION VARIABLES AND CONSTANTS */
/*
 * These are most of the empirical offsets determined for proper
 * machine function
 */
int     evlist = 3;             /* how many events to do at a time */
long    preroll = 150;          /* matching the preroll of record deck*/
long    followon = 40;          /* delay at the end of edit before
                                 * stopping */
long    tolerance = 1;          /* accuracy tolerance, must be at least
                                 * one unless perfect machines */
long    vtroffset = 3;          /* delay for vtr response */
long    wipeoffset = 9;         /* delay for switcher wipe 004 */
long    dissoffset = 9;         /* delay for dissolve */
long    commit = 10;            /* last test for devices in sync */
long    threshhold = 25;        /* last chance to sync devices */
long    absurd = 2000000L;      /* accomodates serial line garbage */

char    funclist[15];           /* same as cmdlist */
long    vtrinfrm;               /* frame mark for activating edit */
long    vtrintc;                /* timecode mark for activating edit */
long    vtroutfrm;              /* mark for ending edit */
long    vtrouttc;               /* mark for ending edit */
```

```
long    errin;              /* error tolerances in and out */
long    errout;
long    aswinfrm;           /* switcher in-point in frame and
                             * timecode */
long    aswintc;

/*
 * These are the actual variables the above offsets are applied to
 * for proper functioning.  For the following assignments  p or pri
 * mean primary device and s or sec mean secondary.  Primary and
 * secondary only refer to source devices not record devices.  In
 * the case of a wipe or dissolve, the secondary device is the
 * second segment or event line in the edl.
 */
long    thresh;
long    commitpt;
long    pfrmdiff;           /* frame difference between record and
                             * source */
long    sfrmdiff;
long    pcurdiff;           /* the difference currently maintained
                             * by devices */
long    scurdiff;

long    prifrm;             /* frame device is at */
long    secfrm;
long    rvfrm;
char    priframeno[10];     /* character representation of frame */
char    secframeno[10];
char    rvtcno[10];
long    rvtc;               /* numeric timecode */
int     try = 0;            /* number of retries */
long    over;

struct line event[15];      /* program can handle 15 events in an
                             * edl */

/*
 * Initial integration of this function in with the upper level
 * program
 */
strcpy(kbdport, KBDPORT);

bufptr = cmdlist;
endbuffer = &(cmdlist[MAXCMDLIST]);
if ((fd = open(filename, ORDONLY)) < 0)
   return (-1);
edl = TRUE;
debug = FALSE;

/* Set up of devices into a consistent state with the program */
control(recdev);
control(stop);
control(play);
```

```
sleep(1000);
control(stop);
/* checks audio and video status and resets them */
if (devtable[RVTR].status & BIT2)
   control(video);

if ((devtable[RVTR].status & BIT0) &&
    !(devtable[RVTR].status & BIT1))
   y = 3;

if (!(devtable[RVTR].status & BIT0) &&
    (devtable[RVTR].status & BIT1))
   y = 2;

if ((devtable[RVTR].status & BIT0) &&
    (devtable[RVTR].status & BIT1))
   y = 1;

for (x = 0; x < y; x++)
   control(audio);

/* Set up main loop */
loop = 0;
start = 0;
/* Start of main loop */
do
{
   loop++;
   /* Start of the Input reading for the first number of events */
   for (i = start + (loop - 1) * evlist; i <= loop * evlist; i++)
   {
      read(fd, linebuf, edllen);
      printf("%s\n", linebuf);

      /*
       * This is the statement which 'parses' the edl line and
       * stuffs it into the structure LINE
       */
      sscanf(linebuf, "%d %s %s %s %2d %*c %2d  \
                       %2d %*c %2d %*c %2d %*c %2d \
                       %2d %*c %2d %*c %2d %*c %2d \
                       %2d %*c %2d %*c %2d %*c %2d \
                       %2d %*c %2d %*c %2d %*c %2d  ",

            &event[i].eventno, &event[i].device,
            &event[i].av, &event[i].shot,
            &event[i].dur.sec, &event[i].dur.frame,
            &event[i].qin.hour, &event[i].qin.min,
            &event[i].qin.sec, &event[i].qin.frame,
            &event[i].qout.hour, &event[i].qout.min,
            &event[i].qout.sec, &event[i].qout.frame,
            &event[i].rin.hour, &event[i].rin.min,
            &event[i].rin.sec, &event[i].rin.frame,
```

```
            &event[i].rout.hour, &event[i].rout.min,
            &event[i].rout.sec, &event[i].rout.frame);

    convert(&event[i].dur);
    convert(&event[i].qin);
    convert(&event[i].qout);
    convert(&event[i].rin);
    convert(&event[i].rout);
    /* marker of event number 512 signalling edl is done */
    if (event[i].eventno == 512)
    {
       i++;
       break;
    }
}                          /* End of Read/Input loop */

start = 1;

/* EDL EXECUTION LOOP */
for (j = ((loop - 1) * evlist); j < (i - 1); j++)
{
   /* calculate threshold points and commit points */
   thresh = event[j].rin.frm - threshhold;
   commitpt = event[j].rin.frm - commit;

   /*
    * Set control to record deck and select audio and video
    * channels
    */
   control(recdev);
   avm(&v, &a, event[j].av);
   if (v == 1)
      control(video);
   for (x = 0; x < a; x++)
      control(audio);

   /* CUT OR WIPE-DISSOLVE */
   /*
    * this determines if the edit is a cut or wipe and dissolve
    * and sends program control to the appropriate section.
    */

   if (strcmp(event[j].eventno, event[j + 1]))
   {

      /* ASSIGN CURRENT DEVICES */
      /* This assigns the primary source to be used in the cut */

      if (!strcmp(event[j].device, dev1))
      {
         strcpy(primary, avtr);
         dpd = LDP1;
         pd = AVTR;
```

```
        } else
        if (!strcmp(event[j].device, dev2))
        {
           strcpy(primary, bvtr);
           dpd = LDP2;
           pd = BVTR;
        } else
           if (!strcmp(event[j].device, devaux) ||
               !strcmp(event[j].device, devax))
        {
           strcpy(primary, aux);
           pd = AUX;
        } else
           if (!strcmp(event[j].device, devblk) ||
               !strcmp(event[j].device, devbl))
        {
           strcpy(primary, blk);
           pd = BLK;
        }
        /* calculate desired frame difference */
        pfrmdiff = event[j].rin.frm - event[j].qin.frm;

        /*
         * This is the point of return if the edit must abort and
         * try again
         */
retry:
        /*
         * SETIN SETOUT .. This sets the real in and out points for
         * the machine control.
         */
        bufptr = cmdlist;
        strcpy(cmdlist, event[j].rin.tcno);
        control(setin);
        vtrinfrm = event[j].rin.frm - vtroffset;
        vtrintc = ftotc(vtrinfrm);
        errin = ftotc(vtrinfrm + tolerance);

        bufptr = cmdlist;
        strcpy(cmdlist, event[j].rout.tcno);
        control(setout);
        vtroutfrm = event[j].rout.frm - vtroffset;
        vtrouttc = ftotc(vtroutfrm);
        errout = ftotc(vtroutfrm + tolerance);

        /* CUE DEVICES */
        /*
         * Set the devices to the appropriate start points for the
         * edit
         */
        if (!strcmp(primary, avtr) || !strcmp(primary, bvtr))
        {
           sprintf(inframe, "%05ld", (event[j].qin.frm - preroll));
```

```
    control(primary);
    control(cue);
 }
 control(recdev);
 control(recpreroll);
 /*
  * holds control until the record deck reaches the preroll
  * point
  */
 while (tcc(rvgetframe(RVTR1, rvtcno))
        != (event[j].rin.frm - preroll));

 /* START DEVICES */
 control(recdev);
 control(play);

 if (!strcmp(primary, avtr) || !strcmp(primary, bvtr))
 {
    control(primary);
    control(play);
 }
 /*
  * TRANFER CONTROL TO RECORD AND SET SWITCHER TO PROPER
  * SOURCE CHANNEL
  */
 control(recdev);

 bufptr = cmdlist;
 if (devtable[ASW].status & BIT0)
    cmdlist[0] = '1';
 else
    cmdlist[0] = '0';
 strcpy(&cmdlist[1], primary);
 control(setasw);

 /* SYNCHRONIZE LDP TO VTR */
 /*
  * This continually gets the frames of both the record and
  * source deck and compares them to the desired frame
  * difference.  If the difference is too great or too small
  * the ldp is sped up or slowed down to match. If the
  * difference is within tolerance, the machines are left to
  * play. This loop is continued until the devices reach the
  * threshhold point.
  */

 if (!strcmp(primary, avtr) || !strcmp(primary, bvtr))
 {
    for (;;)
    {
       prifrm = getframe(devices[pd], priframeno);
       rvtc = rvgetframe(RVTR1, rvtcno);
```

```
        rvfrm = tcc(rvtc);
        pcurdiff = rvfrm - prifrm;

        if (pcurdiff <= (pfrmdiff + tolerance) &&
            pcurdiff >= (pfrmdiff - tolerance))
           send(dpd, FPLAY, 1);
        else
        if (pcurdiff < pfrmdiff)
           send(dpd, FSLOW, 1);
        else
           send(dpd, FFAST, 1);

        if (rvfrm >= thresh)
           break;
     }                  /* End of Sync Loop */

     /*
      * This is the last test for sychronization before the
      * edit
      */
     send(dpd, FPLAY, 1);
     prifrm = getframe(devices[pd], priframeno);
     rvtc = rvgetframe(RVTR1, rvtcno);

     rvfrm = tcc(rvtc);
     pcurdiff = rvfrm - prifrm;
  } else
  {
     /*
      * This is to set up parameters if no physical device is
      * used for a source
      */
     rvtc = rvgetframe(RVTR1, rvtcno);
     rvfrm = tcc(rvtc);
     pcurdiff = pfrmdiff;
  }

  /* COMMIT THE EDIT AND PERFORM IT */
  /* if the devices conform to standards then go ahead */
  if (pcurdiff <= (pfrmdiff + tolerance) &&
      pcurdiff >= (pfrmdiff - tolerance) &&
      rvfrm < commitpt)
  {

     /*
      * This is a loop to look for the in-point. When found,
      * turn on EDIT. Then enter a similar loop to look for
      * the out-point.
      */
     for (;;)
     {
        if ((rvtc = rvgetframe(RVTR1, rvtcno)) == vtrintc)
        {
```

```
                control(edit);
                for (;;)
                {
                   if ((rvtc = rvgetframe(RVTR1, rvtcno)) ==
                       vtrouttc)
                   {
                      control(play);
                      break;
                   } else
                   if (rvtc > errout && rvtc < absurd)
                   {
                      control(play);
                      over = tcc(rvtc) - tcc(vtrouttc);
                      printf("EDIT ERROR--OVERRUN %ld\n", over);
                      break;
                   } else;
                }
                while (tcc(rvgetframe(RVTR1, rvtcno)) <=
                       (event[j].rout.frm + followon));
                break;
             } else
             if (rvtc > errin)
             {
                control(play);
                printf("NOT QUITE\n");
                ++try;
                if (try <= 2)
                   goto retry;
                break;
             } else;
          }
       } else
       {
          printf("RETRY\n");
          ++try;
          if (try <= 2)
             goto retry;
       }                    /* End of the EDIT Loop */

       control(recdev);
       control(stop);
       if (!strcmp(primary, avtr) || !strcmp(primary, bvtr))
       {
          control(primary);
          control(stop);
       }
    }                       /* END OF CUT */
    /*  Start Wipe and Dissolve Loop */
    /*
     * This wipe/dissolve loop is very close to the cut loop in
     * terms of function.  It's main difference is that it is
     * using the functions of the switcher and two source devices
     * are being used.  Those are the only conceptual differences
```

```
 * that are present.  This all leads to a large duplication of
 * code with a few manipulations in the order of events.
 */
else
{
   switch (event[j + 1].shot[0])
   {
   case 'W':            /* set up the switcher wipecode */
   case 'w':
      wc[0] = event[j + 1].shot[1];
      wc[1] = event[j + 1].shot[2];
      wc[2] = event[j + 1].shot[3];
      wipecode = atoi(wc);
   case 'D':
   case 'd':
      /* Assign primary and secondary devices */

      if (!strcmp(event[j].device, dev1))
      {
         strcpy(primary, avtr);
         dpd = LDP1;
         pd = AVTR;
      } else
      if (!strcmp(event[j].device, dev2))
      {
         strcpy(primary, bvtr);
         dpd = LDP2;
         pd = BVTR;
      } else
         if (!strcmp(event[j].device, devaux) ||
             !strcmp(event[j].device, devax))
      {
         strcpy(primary, aux);
         pd = AUX;
      } else
         if (!strcmp(event[j].device, devblk) ||
             !strcmp(event[j].device, devbl))
      {
         strcpy(primary, blk);
         pd = BLK;
      } else;

      if (!strcmp(event[j + 1].device, dev1))
      {
         strcpy(secondary, avtr);
         dsd = LDP1;
         sd = AVTR;
      } else
      if (!strcmp(event[j + 1].device, dev2))
      {
         strcpy(secondary, bvtr);
         dsd = LDP2;
         sd = BVTR;
```

```
            } else
                if (!strcmp(event[j + 1].device, devaux) ||
                    !strcmp(event[j + 1].device, devax))
            {
                strcpy(secondary, aux);
                sd = AUX;
            } else
                if (!strcmp(event[j + 1].device, devblk) ||
                    !strcmp(event[j + 1].device, devbl))
            {
                strcpy(secondary, blk);
                sd = BLK;
            } else;
        }

        /*
         * Set up function list for when the switcher is called
         * during the edit
         */
        switch (event[j + 1].shot[0])
        {
        case 'w':
        case 'W':
            sprintf(funclist, "%02d", wipecode);
            funclist[2] = primary[0];
            funclist[3] = secondary[0];
            funclist[4] = duration(event[j + 1].dur.frm);
            strcpy(function, wipe);
            aswinfrm = event[j + 1].rin.frm - wipeoffset;
            aswintc = ftotc(aswinfrm);
            break;
        case 'd':
        case 'D':
            funclist[0] = primary[0];
            funclist[1] = secondary[0];
            funclist[2] = duration(event[j + 1].dur.frm);
            strcpy(function, dissolve);
            aswinfrm = event[j + 1].rin.frm - dissoffset;
            aswintc = ftotc(aswinfrm);
            break;
        }

        pfrmdiff = event[j].rin.frm - event[j].qin.frm;
        sfrmdiff = event[j + 1].rin.frm - event[j + 1].qin.frm;


wdretry:
        /*
         * SETIN SETOUT
         */
        bufptr = cmdlist;
        strcpy(cmdlist, event[j].rin.tcno);
        control(setin);
```

```
vtrinfrm = event[j].rin.frm - vtroffset;
vtrintc = ftotc(vtrinfrm);
errin = ftotc(vtrinfrm + tolerance);

bufptr = cmdlist;
strcpy(cmdlist, event[j + 1].rout.tcno);
control(setout);
vtroutfrm = event[j + 1].rout.frm - vtroffset;
vtrouttc = ftotc(vtroutfrm);
errout = ftotc(vtroutfrm + tolerance);



/* CUE DEVICES */
if (!strcmp(primary, avtr) || !strcmp(primary, bvtr))
{
   sprintf(inframe, "%05ld", event[j].qin.frm - preroll);
   control(primary);
   control(cue);
}
if (!strcmp(secondary, avtr) || !strcmp(secondary, bvtr))
{
   sprintf(inframe, "%05ld", event[j+1].qin.frm - preroll -
           (event[j].qout.frm - event[j].qin.frm));
   control(secondary);
   control(cue);
}
control(recdev);
control(recpreroll);
while (tcc(rvgetframe(RVTR1, rvtcno)) !=
       (event[j].rin.frm - preroll));

control(recdev);
control(play);

if (!strcmp(primary, avtr) || !strcmp(primary, bvtr))
{
   control(primary);
   control(play);
}
if (!strcmp(secondary, avtr) || !strcmp(secondary, bvtr))
{
   control(secondary);
   control(play);
}
/*
 * TRANSFER CONTROL TO RECORD AND SET SWITCHER
 */
control(recdev);


if (devtable[ASW].status & BIT0)
{
```

```
    bufptr = cmdlist;
    cmdlist[0] = '1';
    strcpy(&cmdlist[1], primary);
    control(setasw);
    bufptr = cmdlist;
    cmdlist[0] = '0';
    strcpy(&cmdlist[1], secondary);
    control(setasw);
} else
{
    bufptr = cmdlist;
    cmdlist[0] = '0';
    strcpy(&cmdlist[1], primary);
    control(setasw);
    bufptr = cmdlist;
    cmdlist[0] = '1';
    strcpy(&cmdlist[1], secondary);
    control(setasw);
}

/* SET WIPE CODE */
bufptr = cmdlist;
strcpy(cmdlist, funclist);

/*
 * SYNCHRONIZE  sync each disc to the record deck
 */

if (!strcmp(primary, avtr) || !strcmp(primary, bvtr))
{
   do
   {
      prifrm = getframe(devices[pd], priframeno);
      rvtc = rvgetframe(RVTR1, rvtcno);

      rvfrm = tcc(rvtc);
      pcurdiff = rvfrm - prifrm;

      if (pcurdiff <= (pfrmdiff + tolerance) &&
          pcurdiff >= (pfrmdiff - tolerance))
      {
         send(dpd, FPLAY, 1);
         if (++k > 3)
            break;
         else;
      } else
      {
         k = 0;
         if (pcurdiff < pfrmdiff)
            send(dpd, FSLOW, 1);
         else
            send(dpd, FFAST, 1);
      }
```

```
    }
    while (rvfrm < thresh);
}
if (!strcmp(secondary, avtr) || !strcmp(secondary, bvtr))
{
    do
    {
       rvtc = rvgetframe(RVTR1, rvtcno);
       secfrm = getframe(devices[sd], secframeno);

       rvfrm = tcc(rvtc);
       scurdiff = rvfrm - secfrm;

       if (scurdiff <= (sfrmdiff + tolerance) &&
           scurdiff >= (sfrmdiff - tolerance))
          send(dsd, FPLAY, 1);
       else
       if (scurdiff < sfrmdiff)
          send(dsd, FSLOW, 1);
       else
          send(dsd, FFAST, 1);
    }
    while (rvfrm < thresh);
}
if (!strcmp(primary, avtr) || !strcmp(primary, bvtr) &&
    !strcmp(secondary, avtr) || !strcmp(secondary, bvtr))
{

   prifrm = getframe(devices[pd], priframeno);
   rvtc = rvgetframe(RVTR1, rvtcno);
   secfrm = getframe(devices[sd], secframeno);

   rvfrm = tcc(rvtc);
   pcurdiff = rvfrm - prifrm;
   scurdiff = rvfrm - secfrm;

} else
   if (!strcmp(primary, avtr) || !strcmp(primary, bvtr) &&
       !strcmp(secondary, aux) || !strcmp(secondary, blk))
{
   prifrm = getframe(devices[pd], priframeno);
   rvtc = rvgetframe(RVTR1, rvtcno);

   rvfrm = tcc(rvtc);
   pcurdiff = rvfrm - prifrm;
   scurdiff = sfrmdiff;

} else
   if (!strcmp(primary, aux) || !strcmp(primary, blk) &&
     !strcmp(secondary, avtr) || !strcmp(secondary, bvtr))
{
   rvtc = rvgetframe(RVTR1, rvtcno);
   secfrm = getframe(devices[sd], secframeno);
```

```
    rvfrm = tcc(rvtc);
    pcurdiff = pfrmdiff;
    scurdiff = rvfrm - secfrm;
} else
    if (!strcmp(primary, aux) || !strcmp(primary, blk) &&
        !strcmp(secondary, aux) || !strcmp(secondary, blk))
{
    rvtc = rvgetframe(RVTR1, rvtcno);
    rvfrm = tcc(rvtc);
    pcurdiff = pfrmdiff;
    scurdiff = sfrmdiff;
}
if (pcurdiff <= (pfrmdiff + tolerance) &&
    pcurdiff >= (pfrmdiff - tolerance) &&
    scurdiff <= (sfrmdiff + tolerance) &&
    scurdiff >= (sfrmdiff - tolerance) &&
    (rvfrm < commitpt))
{

    /* Start the EDIT Loop */
    /*
     * This edit loop is very similar to the one for cuts
     * except that the switcher is involved and must be
     * activated in the middle of the edit
     */

    for (;;)
    {
        if ((rvtc = rvgetframe(RVTR1, rvtcno)) == vtrintc)
        {
            control(edit);
            for (;;)
                if ((rvtc = rvgetframe(RVTR1, rvtcno)) ==
                    aswintc)
                    break;
                else
                if (rvtc > aswintc && rvtc < absurd)
                    goto wdretry;
            control(function);
            for (;;)
            {
                if ((rvtc = rvgetframe(RVTR1, rvtcno)) ==
                    vtrouttc)
                {
                    control(play);
                    break;
                } else
                if (rvtc > errout && rvtc < absurd)
                {
                    control(play);
                    over = tcc(rvtc) - tcc(vtrouttc);
                    printf("EDIT ERROR--OVERRUN %ld\n", over);
                    break;
```

```
                    } else;
                  }
                  while (tcc(rvgetframe(RVTR1, rvtcno)) <=
                        (event[j + 1].rout.frm + followon));
                  break;
               } else
               if (rvtc > errin)
               {
                  control(play);
                  printf("NOT QUITE\n");
                  ++try;
                  if (try <= 2)
                     goto wdretry;
                  break;
               } else;
            }
         } else
         {
            printf("RETRY\n");
            ++try;
            if (try <= 2)
               goto wdretry;
         }

         control(recdev);
         control(stop);
         if (!strcmp(primary, avtr) || !strcmp(primary, bvtr))
         {
            control(primary);
            control(stop);
         }
         if (!strcmp(secondary, avtr) || !strcmp(secondary, bvtr))
         {
            control(secondary);
            control(stop);
         }
         j++;
      }

      /* RESET AV */
      control(recdev);
      if (v == 1)
         control(video);
      for (x = 0; x < (4 - a); x++)
         control(audio);
   }                          /* End of Execution Loop */

   if (j != (loop * evlist))
      start = 0;

}
while (event[j].eventno != 512);      /* End of edl */
```

```
    return (0);
}


convert(time)
    struct reftime *time;

{
    sprintf(time->tcno, "%02d%02d%02d%02d", time->hour, time->min,
            time->sec, time->frame);
    time->frm = ((long) time->min * 60 + (long) time->sec) * 30
       + (long) time->frame + 1;
    sprintf(time->frameno, "%05ld", time->frm);
    return;
}

long
tcc(time)
    long    time;
{
    long    frm;
    long    n;
    long    a = 1000000;
    long    b = 10000;
    long    c = 100;
    long    d = 60;
    long    e = 30;
    long    f = 1;

    time -= a;
    n = time / b;
    time = time - (n * b);
    frm = n * 60;
    n = time / c;
    time = time - (n * c);
    frm = (frm + n) * e + time + f;
    return (frm);
}


int
avm(v, a, str)
    int     *v, *a;
    char    *str;
{

    /* possibles are: 'V', 'B' (V12), 'A1',
                      'A2', 'A1V', 'A2V', 'AA' */

    if (!strcmp(str, "V"))
    {
        *v = 1;
        *a = 0;
```

```
        return;
    }
    if (!strcmp(str, "B"))
    {
        *v = 1;
        *a = 3;
        return;
    }
    if (!strcmp(str, "A1"))
    {
        *v = 0;
        *a = 1;
        return;
    }
    if (!strcmp(str, "A2"))
    {
        *v = 0;
        *a = 2;
        return;
    }
    if (!strcmp(str, "A1V"))
    {
        *v = 1;
        *a = 1;
        return;
    }
    if (!strcmp(str, "A2V"))
    {
        *v = 1;
        *a = 2;
        return;
    }
    if (!strcmp(str, "AA"))
    {
        *v = 0;
        *a = 3;
        return;
    }
    /* will accept 'V12', even though our implementation uses 'B' */
    if (!strcmp(str, "V12"))
    {
        *v = 1;
        *a = 3;
        return;
    }
    printf("getavmode: illegal mode string '%s'; all channels off\n");
    *v = 0;
    *a = 0;
    return -1;

}
```

```
long
ftotc(time)
   long    time;

{
   long    t, frames;
   int     i, digits[3];

   frames = time - 1L;
   digits[2] = frames - ((frames / 30) * 30);
   frames /= 30;
   digits[1] = frames - ((frames / 60) * 60);
   frames /= 60;
   digits[0] = frames - ((frames / 60) * 60);

   t = 0;
   for (i = 0; i < 3; i++)
      t = (t * 100) + digits[i];
   t += 1000000L;

   return (t);
}

duration(dur)
   long    dur;
{
   if (dur == 0)
      return ('0');
   else
   if (dur <= 19L)
      return ('1');
   else
   if (dur <= 23L)
      return ('2');
   else
   if (dur <= 28L)
      return ('3');
   else
   if (dur <= 36L)
      return ('4');
   else
   if (dur <= 53L)
      return ('5');
   else
   if (dur <= 85L)
      return ('6');
   else
      return ('7');
}
```