THE DESIGN OF A TRACKBALL CONTROLLER

David Martindale

Computer Graphics Laboratory
Department of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1

ABSTRACT

This paper describes the design and operation of a general-purpose graphics input device that provides a trackball, three knobs, and twenty-five lighted function buttons. The controller contained within the device is considerably more "intelligent" than those found in most commercially-available graphics input devices. It transmits updates to the host only when necessary, and there is considerable flexibility in defining what "necessary" means.

KEYWORDS: trackball, intelligent controller, microprocessor

1. Background

At the time this project was begun, the University of Waterloo's Computer Graphics Laboratory had only a tablet for graphics input. It was felt that a variety of other input devices should be available to allow more flexibility in the styles of input available to graphics programs. Thus a project to construct a trackball, a set of control knobs, and a set of lighted function buttons and to interface them to the lab's principal computer was undertaken.

The original inspiration for this project came from a trackball/knob/button unit that was constructed at the National Research Council of Canada in 1979 and 1980. This consisted of a trackball with lighted buttons in one housing, and sixteen lighted buttons, eight knobs, and eight lamps in another. The NRC trackball is rather unique in that it has three distinct data axes. It can be rolled in two directions to provide X and Y information, just like any other trackball. However, it is also possible to grasp the ball firmly and rotate it about its vertical axis to provide Z information.

We wished to provide a similar facility at Waterloo while keeping the cost of construction as low as possible. Since the CGL trackball was to be built virtually from scratch, it seemed worthwhile to reconsider NRC's design, rather than simply copying it.

The final design is in fact quite different, partially because of changes in available technology and partially because of a desire to make the final result "better" in some sense than the NRC design and other commercial input devices. Although the design changed a number of times before construction of the hardware began (and once after it was finished), only the current design will be discussed here.

2. Other Devices

First we will examine a number of other graphics input devices, looking at the manner in which they are interfaced to the host processor and how they operate. This will provide a background for understanding the choices made in the design of our trackball.

2.1. NRC Trackball

This set of hardware is connected to its host (a PDP11/55) via a DEC (Digital Equipment Corp.) DR11-C parallel interface. One board of additional circuitry provides interfacing between the DR11-C and the outside world.

The lamp in each lighted button is controlled by a bit in one of two 16-bit output registers. Changing the state of one lamp requires rewriting an entire 16-bit register, and these registers cannot be read back to determine what was last written to them.

Whenever any of the input devices (trackball, knob, or switch) changes state, the state of all of the input devices is latched (saved) in three 16-bit registers and an interrupt request is made to the host via the DR11-C. The host's interrupt routine must then obtain these three 16-bit values by performing three successive reads of the DR11-C, and decide what has changed by examining the bit patterns in these words. Of course, this means that a specially-written device driver must be added to the host's operating system to handle these interrupts.

Decoding movement of the trackball and knobs is very simple. Both of these types of input transducers generate electrical signals in the form of quadrature square waves. These signals are decoded to generate pulses indicating that a particular input is "increasing" or "decreasing".

Whenever one of these devices moves one unit of distance or angle in either direction, the pulse that results sets a bit in a register indicating either "up" or "down" motion. This starts the sequence that interrupts the host. Thus the host is interrupted for every unit of movement of the trackball or knobs, and the interrupt handler simply increments or decrements a variable recording the position of the appropriate device at each interrupt.

Since the circuitry indicates only that movement has taken place in a particular direction, not how many pulses have occurred since the host last read the interface's status, this scheme depends on the host servicing the interrupt request generated by any one pulse before the trackball or knob has moved another unit of distance and generated another pulse. If the host CPU does not handle the interrupt soon enough, any pulses that followed the first are effectively lost; the host will record at most one unit of movement for each interrupt. The result is that a portion of the distance that the transducer has moved is simply lost. Also, when movement is rapid a substantial portion of the available CPU time can be spent servicing these interrupts.

On the other hand, with a properly-designed driver, it would often be possible to design the applications program so that it is normally asleep, waking up only when an input event occurs. Thus, no CPU time would be consumed when the user is doing nothing.

When a button is pressed, the contacts in its mechanical switch may open and close several times within a few milliseconds; this is referred to as "contact bounce". To produce reliable operation of software using the buttons, these bounces must be filtered out so that the button press is presented as a single event to the host. In this controller, the circuitry is constructed so that the state of the switches are latched and an interrupt requested from the host only when a switch has been closed continuously for about 35ms.

There is only one copy of this debouncing circuitry, triggered by any one switch being closed, and as long as at least one switch remains closed the circuitry is insensitive to further switch closures. Thus each button must be released before another may be pressed. Also, the circuitry makes no attempt to notify the host when a button has been released. Together, these two properties mean that the only actions that can be controlled by the buttons are single events that take place when a button is first pressed. There is no way that a button can cause an action to be performed as long as the button is held down, or for several buttons to be used at once independently.

The custom-built interface circuitry was constructed on a wire-wrap board that was intended to plug into a UNIBUS† backplane next to a DR11-C or quad wide QBUS backplane next to a DRV11. This means that the device as constructed can be used only on processors that possess a UNIBUS or QBUS (currently, this means members of the PDP-11 and VAX families only). With a bit more effort, this circuitry could be housed in its own box with its own power supply, allowing it to be connected to any host computer equipped with a 16-bit parallel interface similar to the DR11-C. However, this parallel interface would likely cost about $1000 per host and even with such an interface installed on each host it would still be moderately difficult to move the trackball from one host to another.

To be fair to the people at NRC, I should point out that their trackball was deliberately designed to do as little as possible in hardware, leaving more work to be performed in the host's driver software. This was done to minimize the amount of hardware debugging needed to get the system working in the first place, as well as to minimize the number of decisions about how the system should operate that are wired into the hardware. (Design changes are easier to make if they require no hardware changes.) Given the goal of minimal hardware, this is actually quite a good design.

2.2. Summagraphics Tablet

The Summagraphics Bit Pad One tablet is available with three types of host interface: RS232 serial, IEEE 488 bus, and eight-bit parallel binary. Of these, the RS232 serial interface is the best choice for use in the lab. It allows the tablet to be immediately connected to any available computer, since extra serial communication ports are usually available on a multi-user computer system. No additional interface hardware need be purchased, nor need any new software be added to the operating system to handle I/O to the device. Also, the tablet can easily be switched from one host to another by a simple RS232 switchbox.

The controller for this tablet contains a microprocessor (an Intel 8035, a member of the "8048 family") supported by a small amount of random logic circuitry. Summagraphics has done a fairly good job of exploiting the flexibility that becomes available when a microprocessor is performing most functions. The user can choose from three operating modes, two output data formats, and eight sampling rates. All these selections can be controlled by the host, simply by sending a one-byte command to the tablet via its RS232 interface.

The position data transmitted to the host is always the absolute position of the puck on the tablet.

In one of the available output modes, position information is sent to the host as ordinary decimal ASCII numbers separated by commas. This format should be readable by almost any language's I/O library, and since the only ASCII characters used are the printable characters plus carriage return and line feed, there should be no problem getting them past almost any conceivable front end processor or network that may be between the tablet and the application program. There is also a "binary" output format that transmits the same amount of information in less than half the number of bytes. This is a more efficient format to use if the hardware, operating system, and programming language being used allow it.

The controller provides three operating modes for the tablet: a single coordinate pair may be transmitted whenever a button on the puck is depressed, a stream of samples may be transmitted whenever a button is depressed, or a stream of samples may be transmitted whenever the puck is sufficiently close to the tablet for its location to be determined. The last of these modes turns out to be the most useful, since most programs want to maintain a tracker on the screen that moves as the puck moves, regardless of whether or not a button is depressed.

In a typical session involving the tablet, the user may spend long periods of time thinking, or using input devices other than the tablet, yet the applications program often wants to know immediately if the puck has been moved. Thus the host must examine tablet coordinates continuously looking for any change, yet the effort is mostly wasted because they usually have not changed. The applications program really does not want to see any data from the tablet unless the position of the puck has changed. In one heavily-used CGL application (the Paint program [Beach82]), a programmable communications processor was set up to perform just this sort of filtering on the data coming from the tablet, relieving the host processor of this burden and improving the responsiveness of the program to user input. However, the tablet controller itself contains a microprocessor, and if the applications program wishes to be notified only when something changes at the tablet, there is no reason the controller should not be performing this filtering of the data on its own and transmitting only changes to the host.

2.3. MPS Input Devices

The Evans and Sutherland Multi Picture System (MPS) has several input devices available that provide the same kind of input functions that we want the trackball to provide. There are different styles of lighted function buttons and input knobs, as well as a joystick and a tablet.

The knobs and joystick are constructed using analog potentiometers to sense position. A voltage is applied across the

† UNIBUS, QBUS, PDP, and VAX are trademarks of Digital Equipment Corporation.

potentiometer and its wiper presents an output voltage that depends on the position of the potentiometer's output shaft. This voltage is then converted to numeric form by an analog-to-digital converter. This method allows many inputs to be provided inexpensively, but has several disadvantages. Analog circuitry can drift slowly with time and may require periodic readjustment to keep it operating accurately. Potentiometers are prone to producing noise in their output when they are rotated due to dirt on the resistance element, and a dirty potentiometer can produce an incorrect output even when it is not moving. Because mechanical contact with the resistance element is required, it is subject to wear (although its useful life may be quite long).

The MPS contains a specialized device processor that can be programmed via a command table to scan the input devices regularly looking for changes. It can be asked to generate an interrupt when particular bits of a register have changed state (corresponding to a change in the position of a switch, for example) or when the value of a number in a register has changed by more than a certain threshold. It is quite general.

However, this processor appears to be fairly difficult for the user to program since special tables must be built and transferred into MPS memory. When the system is first powered on, it is completely passive - there is no default set of command tables that the user can make use of immediately without having to understand how to create them himself.

2.4. Optical Mouse

Next we will take a look at the Mouse Systems Corporation M-1 mouse. Although this device was not available at the time our trackball was designed and thus had no influence on its design, it is nonetheless interesting to compare them.

Data is transmitted to the host over an ordinary RS232 serial connection as packets of five asynchronous characters. The data sent indicates which switches are down and the incremental change in the position of the mouse since the previous update. The switches are debounced by the mouse's internal software. Position increments are sent as 2's complement eight-bit bytes without parity. There is no alternate output representation that uses only printable characters or provides "decimal" output. Thus the host must pass all bytes received to the program reading the data without stripping the "parity" bit and without treating any characters specially. This restriction will prevent the mouse from being used with some hosts.

All control of the mouse is done by physically setting internal DIP switches, together with the external switches that affect some of the diagnostics. The host has no programmable control over the mouse whatsoever.

The mouse is "intelligent" enough to transmit updates to the host only when there has been a change: that is, when a switch is pressed, or released, or the mouse has moved. A complete packet of five bytes is always sent when an update is transmitted. Additionally, there is a "noiseless" mode that prevents an update from being transmitted when the position has changed by positive one count. This avoids the continuous stream of updates that could otherwise take place when the mouse is not moving but is sitting directly over a position boundary on the mouse pad. No further control of when updates are sent is possible.

The mouse's documentation says "Be aware that the counters are NOT snap-shotted since the microprocessor doesn't have a lot of memory. Hence, only at the end of a motion will the coordinates be strictly correct." [MSC83] This seems to indicate that it simply transmits instantaneous delta X and delta Y values alternately, instead of sampling both X and Y changes simultaneously and then transmitting the saved values. Thus, since any pair of X and Y values transmitted are not samples from the same point in time, the path that the mouse followed cannot easily be calculated. Also, since every fifth byte transmitted indicates the positions of the switches, the X and Y sampling intervals are not evenly spaced in time, further complicating any attempt to calculate the mouse's actual path.

3. Design Issues and Decisions

Our trackball controller is designed to be interfaced to a timesharing host computer rather than a dedicated workstation. In order to minimize the load on the host, an emphasis was placed on performing as much processing of raw data as is feasible in the controller. Also, flexibility in interfacing to several hosts was considered important. It should be noted that if the controller had been intended for use in a different environment, the design criteria would have changed and the resulting design would have been different.

This controller is intended to be a one-of-a-kind research project, not a finished commercial product. Thus wherever it was possible to add a feature that *might* prove useful to someone, or to provide several overlapping methods of doing something, the more complex and flexible route was usually chosen. This means that some features that turn out not to be very useful in practice have been included; such is the nature of experimentation.

Unfortunately, increased complexity means that the functioning of the final product will be more difficult to understand than if its design had been kept simple. Since the anticipated users are all researchers, this is expected to be less of a problem than if this were a commercial product.

3.1. Handling of Position Information

It was decided that the controller would simply count pulses received from the trackball and the knobs' shaft encoders and report this count directly as the amount of motion. Any scaling necessary to convert this into the actual physical distance moved by the trackball surface or angular displacement of the knobs would be done by the host. It was felt that in most applications the host would be scaling the data anyway by some constant chosen to give the ball or knob an appropriate "feel", and any host will have better facilities for performing multiplication than the processor chosen for this controller. Also, this means that different trackballs may be attached to the controller without any changes to its firmware.

There are two reasonable ways to send position information to the host: absolute position relative to some reference point, or incremental change from the previously-reported position. Absolute position is very appropriate for a device such as a tablet, whose fixed-size digitizing surface provides a natural reference frame. However, for a trackball or knob, an artificial reference frame would have to be defined and maintained by the controller, since the motion of these devices has no natural limit nor origin. This seemed unnatural and too restrictive, so it was decided that the position updates sent to the host would always indicate the incremental change in position. If the application program wants to deal with an absolute position within some sort of fixed reference frame, it can easily convert the incremental data to this form itself.

Transmitting absolute position does have the advantage that each update is independent of all others, so that the loss of an occasional position update matters little to an applications program that cares only about current position of the input device. With incremental information, the loss of one update represents the permanent loss of that movement, and so some sort of mechanism is needed to prevent the host from losing information if it is too busy to read the incoming data for a short period.

3.2. Host Interface

For the reasons summarized in the description of the Summagraphics tablet above, it was decided to make the physical connection to the host via a standard RS232 serial line.

To avoid loss of data when the host is busy, it was decided that the controller should support XON/XOFF flow control. Later, it was decided to allow the RS232 CTS (Clear to Send) signal to provide hardware flow control as well.

3.3. Data Format

It was decided that the default input and output format used by the controller should be usable in as wide a variety of situations as possible. Data is transmitted using only 7-bit ASCII codes. Thus the parity bit has no significance and can safely be discarded by whatever hardware and software processes the characters. No control characters are used, to avoid the possibility of hardware or operating system software interpreting some characters specially.

A numeric datum appearing in the input or output is represented as a decimal ASCII number. A leading minus sign indicates a negative value. This ensures that the value can be read by the widest possible variety of languages, as well as ensuring that the command and data character streams can be read by humans with a minimum of effort.

Also, upper and lower case letters are equivalent in input commands, and only lower case letters are generated in the output data. This ensures that the controller is still usable with systems that only understand single-case letters.

A postfix notation is used for commands and their parameters. The parameters appear first, followed by the single-letter command code. All parameters are numeric, and the set of characters that can appear in the parameter list is disjoint from the set of characters used for the command codes. Thus, each character is immediately identifiable as being part of a parameter list or a command, and a command can be executed immediately when its command code arrives since its parameters (if any) have already been received at that point.

Given the very general data format described above, it was felt that provision should be made to provide one or more additional formats that made better use of the available communications bandwidth to the host by transmitting the most common information in fewer bytes, at the expense of generating codes that some hosts may not be able to handle. However, such an output format has not yet been implemented.

3.4. Intelligence

A significant problem with many graphics input devices is that they require the host to expend a fair amount of CPU time looking at data from the devices even when the user is doing nothing. On a dedicated single-user system, this may waste CPU time that could otherwise be spent performing some background activity, but at least it doesn't hurt any user's response time. However, on a multi-user time-sharing system, a resource that is consumed by one user is something that could have been utilized by another user.

"Interactive" programs that connect to a user on a terminal (editors, debuggers, etc.) typically require short bursts of CPU time followed by much longer pauses waiting for the next input. If a time-sharing system's scheduler is set up to give high priority to programs exhibiting this behaviour, users will feel that the system is fairly responsive; this is good. But if a graphics program that spends a significant amount of time continuously handling data from its input devices is run on such a system, it will eventually receive quite low priority for use of the CPU because it is rarely suspended. The user of this program may see quite poor response if anything else is going on in the system. If, instead, the host continues to give high priority to the interactive program that is reading input from these devices, other users on the system suffer.

What is needed is a graphics input device intelligent enough to send data to the host when the user is actually doing something with it, but which sends nothing at all when the user is just thinking or eating his lunch, much as a terminal only sends information when the user is actually typing on it. Unfortunately, with continuous-motion devices such as a trackball or knob, it is not clear how often changes in position should be reported while the device is in continuous motion; each application program may have a different idea of what is appropriate. The algorithm that decides when to report a change to the host should be flexible enough to allow the application program to receive just the data it wants just as often as it wants.

Finally, the controller should also support polled and continuous-reporting modes of operation, in case the more sophisticated algorithm is inadequate for some purpose and these very simple modes can be used to obtain the desired result with some additional work on the part of the host.

3.5. User-Friendliness

There are a number of features of any piece of software or hardware that do not affect how well or poorly it performs its intended purpose, but have a great effect on how much the user is inconvenienced while working with it. Since this project was built entirely from scratch, it was felt that there was no excuse for not doing a reasonable job of providing a "friendly" interface to the outside world. With this in mind, the following decisions were made.

It should be possible to read back from the host any parameter, mode, or state internal to the trackball that can be set on command of the host. When the data is read back, it should be in the form of the command that the host would need to send to the controller to set these parameters to their current values. Thus the user need remember only one syntax for both commands and responses, and one set of command codes.

Commands sent to the controller should be carefully checked, so that errors on the user's part are reported to him instead of being ignored or causing strange behaviour.

When an error occurs, an error message in English explaining what was wrong should be returned to the host for display to the user. There is no reason the user should have to look up a funny-numbered error code in a list somewhere. Also, the user should be able to request that the controller re-send the last error message sent, in case his software managed to throw it away without displaying it to him.

The controller should notify the host any time it has been reset, so that the host can re-initialize any internal data structures that contain information about the internal state of the controller.

3.6. Software Design.

Since this is a research project, it is not expected that the first version of the software in the controller will be perfect, or even close to it. The software must be designed so that it can be understood and changed by someone other than its author. To make this possible, it was written in PL/M, a moderately high level language that provides structured data and pointers. [Intel78a]

There must be enough comments in the code at appropriate places for someone other than the author to have a reasonable chance of understanding the code and making major modifications to it without breaking the remaining code. In fact, about one third of the source files are comments.

There is no assembly code whatsoever in the software. This decision was reconsidered several times, when it seemed that assembly code would make a large difference in the speed of some routines, but the temptation was eventually resisted in each case. It was felt that the use of assembly code would make the software considerably more difficult to modify or move to another processor, particularly for someone other than the author.

4. Central Algorithms

The heart of the trackball controller's flexibility is in the algorithms that determine when an update of the trackball or knob positions needs to be sent to the host, and in how these updates are queued if they cannot be transmitted immediately. The operation of this part of the controller needs to be well-defined and well-documented, since the user needs to have control over the parameters that control these functions. Here we will discuss the algorithms that were eventually decided on, and their operation, from the user's point of view.

4.1. Update Control

The pulses produced by motion of the trackball or knobs are first counted by 8-bit up/down counters implemented in hardware. These counters are read periodically (normally every 10 ms) and their change since the previous reading is calculated. These changes are then added to 16-bit values in the controller's memory that represent the amount that a particular axis of motion has changed since the host was last informed.

Each time these internal change values are updated, the controller must decide whether the change is significant enough to report to the host.

The controller divides its input into four distinct and independent "sample channels" for purposes of deciding whether an update needs to be sent to the host. Each of the three knobs is handled as its own channel; decisions as to when updates should be sent to the host occur independently of the other knobs. The "amount of change" value used by the update-checking algorithm is simply the absolute value of the knob's movement since the last update.

The three axes of trackball motion, on the other hand, form a single sample channel. Whenever an update is sent, changes in all three axes of motion are reported. The hardware counters for these three axes are all read by the microprocessor as close together in time as possible (normally within a 50-microsecond period), so each trackball position update represents a new "snapshot" of the position of the ball. The "amount of change" value used when checking the necessity of an update is the maximum of the absolute value of the changes in the three axes. A better measure would be the Euclidean distance moved, but it was felt that computing this was too expensive, since this computation would have to be performed every 10 ms.

The actual decision of whether an update is due is controlled by four parameters called *absolute threshold*, *absolute delay*, *combination threshold*, and *combination delay*. The effect of these are:

- If the amount of change in this channel is greater than or equal to the *absolute threshold*, then an update is due.
- If the elapsed time since an update was actually generated for this channel is greater than or equal to the *absolute delay*, then an update is due. (Time is measured in clock ticks, normally 10ms apart).
- If the amount of change in this channel is greater than or equal to the *combination threshold* AND the elapsed time is greater than or equal to the *combination delay*, then an update is due.

If any of these parameters is zero, the portion of the decision controlled by it is skipped.

Note that this method is very flexible. The optical mouse's sampling algorithm is roughly equivalent to setting the absolute threshold to 1 or 2 and all the other thresholds to zero to disable the remaining conditions. Thus updates will occur at maximum rate as long as there is movement. On the other hand, the tablet's sampling algorithm can be duplicated by setting the absolute delay to the sample period desired and the other thresholds to zero, thus simply generating an update every fixed period of time.

As a more complex example, consider setting the absolute threshold to 50, the combination threshold to 1, and the combination delay to 10. In this case, the controller would produce one update every 100ms when the trackball or knob was moving slowly, but if the input was changing faster than 50 counts every 100ms then the updates would occur more rapidly. When nothing is changing, no updates will occur.

4.2. Queuing

The algorithm described above which decides when an update is "due" is capable of trying to send updates more often than the serial line to the host is capable of handling them. Also, replies to requests from the host, error messages, button activity, and activity in other sample channels can all compete for the transmission bandwidth needed by a channel. Thus it is necessary to decide what should be done if a sample is due but it cannot be transmitted immediately.

If the trackball or knobs are being used for some purpose wherein the application program really cares only what their current value is, the most desirable behaviour is for the controller to remember that a sample is due on that channel and then, when it becomes that channel's turn to send something, the position transmitted is the current one, rather than whatever it was when the decision to send an update was first made. Thus, the data received by the host is always as up-to-date as possible. For example, if the trackball is being used to control a cursor, the cursor's motion may contain large jumps because updates could not be transmitted for a while, but the cursor will never lag much behind the trackball.

On the other hand, someone may want to take samples at fixed intervals and then compute velocity or acceleration of the trackball or knob. Here, a delayed update would make the computations useless. Instead, the controller should take position information at the moment the decision to send the update is made and save it in a queue for later transmission. Whenever it is that channel's turn to send something, the update information at the head of the queue is transmitted. Thus, as long as the queue doesn't overflow, no updates are lost and a complete history of the motion is eventually sent to the host.

After some deliberation, the following (somewhat complex) method of handling samples was chosen. It has the virtue of providing both of the behaviours described above, at the user's choice, as well as handling queue overflow smoothly.

A queue is provided for each sample channel, and the user may set the maximum length of this queue to any value from zero to some large upper limit. When the decision that an update is needed on a particular sample channel is made, either by the update-controlling algorithm described in section 4.1 or by an explicit request from the host, a flag indicating that this channel has an update due is set. Then an attempt is made to insert a sample of the current position data into the queue belonging to this channel. If there is space in the queue, then the current position data and the time since last update are both set to zero. If there was no space, nothing further is done.

If the *update due* flag is set for a particular channel, it will eventually be given a turn to send an update to the host. When this occurs, an attempt is made to fetch a set of update information from the head of that channel's queue. If the queue is empty, the current position information at that point in time is copied from the location at which the timer interrupt routine maintains it, and the current position and the time since last update are zeroed. In either case, the *update due* flag is cleared if there is no more data in the queue. Finally, the update information obtained is formatted for transmission to the host.

Now, note that with a queue length of zero (the default), the controller's behaviour is exactly as described in the first example (second paragraph) of this section. There is never any space in the queue, so there is no queueing. The update that is transmitted (sometimes immediately, sometimes after a delay) represents the position at the time of transmission. With a non-zero queue length, the queuing behaviour described in the second example is obtained: the position is sampled at the time of the update decision and queued (if necessary) behind other samples.

Note that with this algorithm, the switch from queued to non-queued operation occurs automatically when the queue is full. No movement is ever lost, since the data maintained by the timer interrupt is zeroed only when its previous value is assured eventual transmission to the host.

5. Hardware Overview

A prototype controller has been constructed. This section provides a brief description of its implementation at the hardware level.

The trackball chosen was a used unit originally built by Atari, and is quite large and heavy. The X and Y channel outputs are pseudo-square waves at TTL levels. As yet, nothing has been done to provide the additional shaft encoder and pickup wheel needed to add a Z axis to the ball. (All other hardware and software necessary to handle Z-axis information is in place.)

The "knobs" were implemented using optical shaft encoders that generate quadrature square waves similar to the trackball's outputs. These allow continuous motion in either direction, and avoid glitches that can result from dirt in potentiometers. Also, the circuitry to handle them is virtually identical to that for the trackball.

The controller was built around a STD bus backplane and card cage. [PL81] They were selected primarily because they were already on hand in the MFCF (Math Faculty Computing Facility) hardware lab.

The processor board used was a commercial 8085-based board sold by Pro-Log. It contains sockets for 8Kb of EPROM and 4Kb of RAM, which seemed more than adequate when the project was begun. The 8085 was chosen over other processors for a number of reasons: it is very common, and thus fairly inexpensive. A suitable processor card was available and known to work. Most important, the department's microcomputer lab contained an Intel development system that already had available a PL/M compiler plus an in-circuit emulator for the 8085.

In addition to the commercially-built processor board, the controller contains two circuit boards of additional circuitry, all wire-wrapped by hand.

An Intel 8251A serial interface chip was used to handle the serial port that connects to the host. (Descriptions of the Intel chips mentioned here can be found in [Intel83].) The controller was wired as a DTE (Data Terminal Equipment) with a female connector. This is identical to the wiring of what was until recently the most common type of CRT terminal in the lab, and so it matches the cables that are already installed.

Generation of the 10ms clock interrupts is done by an Intel 8253 programmable timer chip.

One Intel 8255A parallel interface chip is used to connect the processor to the switches and lights in the buttons. The switches in the buttons are arranged in a matrix of 4 rows of 8 switches, although the last row has only 1 switch in it. The matrix is equipped with a full set of isolation diodes so current can flow in only one direction through each switch. This means that the position of each switch can be detected regardless of the positions of any of the other switches. (Without the diodes, closing three switches at the corners of a rectangle in the matrix would cause the switch at the fourth corner to appear closed as well.) The switches are scanned and debounced in software (rather than using a keyboard encoder chip) so that all buttons are completely independent of each other, and button-released as well as button-pressed information is available. Thus the host can keep track of exactly which buttons are being held down at any point in time, allowing chording.

There are six identical sets of up/down counters and associated circuitry that handle the quadrature input signals provided by the trackball and knob shaft encoders. This circuitry determines in which direction the encoder or trackball is moving and causes the counters to increment or decrement appropriately. Two additional 8255A's are used to allow these counters to be read by the microprocessor.

6. Software Overview

6.1. Introduction

For several reasons (elaborated below), the structure of the software was kept as simple as possible. When the software was begun, it was tempting to write a multi-process operating system kernel simply because this provides a friendly environment for writing code such as this, but the temptation was resisted. It was felt that this project was not complex enough to need such a kernel, so simplicity prevailed. (See section 7 for more comments on this.)

The 8085 processor has no addressing modes suitable for accessing variables that are located at an offset from the stack pointer. Reference to such a variable requires the execution of several instructions to compute its address, and there is only one register suitable for doing the required 16-bit addition. The net result of this is that the code generated by PL/M for reentrant procedures (where the local variables must be on a stack) is long and slow.

The 8085 does have much better addressing modes for referencing data at absolute locations in memory, and thus by default PL/M generates non-reentrant code, with local variables (and parameters) stored in fixed locations, to take advantage of its greater speed. Because of this, it was decided to avoid using reentrant code whenever possible.

Another goal was to keep the number of critical sections in the code to a minimum. An unprotected critical section is a latent bug that may not show up until the device has been in use for some time and the author is no longer available to support the software, and these can be among the most difficult bugs to find. Also, the presence of many critical sections, even if they are properly protected, makes the code difficult to modify later.

6.2. Control flow

There is a timer interrupt every 10ms. This causes the hardware counters connected to the trackball and knobs to be read, and the internal position data to be updated. Then each of the four sampling channels is checked to see whether an update needs to be transmitted to the host. Finally, the switches are scanned to see if any of them have changed position.

This timer interrupt is the highest priority interrupt in the system, and care is taken to disable it for as little time as possible to produce the best possible uniformity in the timing of samples.

Serial input and output are also interrupt-driven. The receive interrupt handler simply moves the received character into an input queue (after checking for errors and XON/XOFF). The transmit interrupt handler just moves a character from an output queue to the device. By having serial I/O driven by interrupts, the controller should be able to accept substantial bursts of data from the host without losing characters.

All other functions of the controller are performed by a simple loop that runs continuously, looking for something to do. Each time through the loop, any characters sitting in the input queue are processed. Then, if the number of characters in the output queue is below a "low water mark" threshold, the code tries to find something that is waiting to be sent to the host. If it finds and formats one item of output, it returns to the top of the loop again to check for input; thus no more than one "event" of some sort is put into the output queue for each pass through the low-water-mark check. The order in which the various possible items of queued output is checked determines the priority with which they are transmitted to the host if several are waiting at once. The highest priority goes to button press/release, followed by replies to explicit requests from the host for parameters such as queue lengths and thresholds. Finally, the four sample channels are checked in rotation so that any single one of them cannot consume all of the bandwidth available for transmission to the host.

Error messages are produced by simply dumping them into the output queue regardless of its current length. Since they are generated only within the non-interrupt code in well-controlled ways, it is not possible for an error message to appear in the middle of a piece of formatted output. A special hook in the routine that queues characters for output to the host is used to save the last error message so it can be resent to the host on demand.

The only reentrant routine in the entire system is one that manipulate queues, since it is called from many different places. All other code is executed either exclusively from the non-interrupt level polling loop, or is called only from the timer interrupt or one of the serial I/O interrupt routines, which execute with further interrupts from that source locked out. Thus the vast majority of the code avoids needing to be reentrant.

7. Experience

A prototype has been constructed and has received some use. This process has revealed a number of problems, in both design and implementation, which are listed below. Some of these are major problems, and some are just quibbles. However, they are all things that we would do differently if constructing a new version from scratch, and are thus worth documenting.

The software in the controller is not always performing the task that is most urgent at any particular point in time. During the initial design stage, it was felt that the controller would have CPU time to spare, so this would not be a real problem, but of course this assumption turned out to be wrong. It will require some more thought and study to decide the best way to improve the software in this respect.

One possibility, of course, is to write a very small multiprocess operating system kernel that can handle a few processes each of which has a priority associated with it, and rewrite the controller's software as a set of processes. This would make the scheduling of the CPU both cleaner and more efficient, but it is not clear that performance would actually be better. The increased number of execution contexts that could be active at one time would mean that either more of the controller's routines would have to be made reentrant, slowing down their execution, or that there would need to be multiple copies of the same routine. Also, we believe that at least some of the code in the kernel would have to be written in assembly language. It is quite possible that a bit more tuning of the current implementation would be more effective.

The 8085 turned out to be a poor choice for the processor. The controller simply does not do things rapidly enough, even after a fair bit of effort went into tuning the code. For example, input currently must be limited to 4800 baud because the controller simply cannot handle even short bursts of characters at 9600 baud.

The architecture of the processor itself is the main culprit. The lack of facilities for performing 16-bit arithmetic turned out to be a problem because many numeric quantities in the program must be stored as 16 bits to avoid the likelihood of overflow. Thus the code contains many calls to subroutines for performing 16-bit arithmetic.

The lack of addressing modes capable of referencing data at a fixed offset from a pointer register also causes a great waste of execution time and code space whenever a member of a structure or a variable on the stack must be referenced. Reentrant code is particularly bad in this respect, and worrying about keeping most of the code non-reentrant restricted the form that the software took.

The solution to this, of course, is to use a better processor. The Intel 8088 or Motorola 68008 would probably be good choices, because they is are true 16-bit processors internally. A dialect of PL/M is available for the 8088, so the existing source code could be used with little modification. Using the 68008 would require rewriting the code in another language, but this would be a benefit in the long run (see below). Another, cheaper, choice would be the Motorola 6809. It is an 8-bit processor like the 8085, but much better suited to use with a high-level language. Changing to any processor with more effective computing power than the 8085 has in this application would ease or eliminate the need for modifying the software to make more efficient use of the processor.

PL/M initially looked like a good language to use for this project: while its syntax is sometimes a bit awkward, it had structures, arrays, pointers, modules, manifest constants, and other features that made it look like a good language to write a moderate-sized piece of software in. But as time went on, it became increasingly annoying in many small ways. For example, PL/M has structures, but they may not be nested. The compiler will not accept a constant expression in a place where a constant is needed at compile time. This sometimes prevents using the clearest form of a constant; "**1530**" is much less clear than "**3*255*size(integer)**". The declaration rules are sufficiently strict that it is not possible to create a single header file that contains declarations of all routines that are declared in one file and called in another and then have each source file include this header file. Instead, each source file ends up having a long string of declarations of external procedures at its head, which detracts from the readability of the source code.

The solution to this would be to find a compiler for a better language. C would be a very good choice if an appropriate compiler could be found.

The circuitry that decodes the quadrature signals from the trackball and knobs generates its count pulses at different physical positions in the motion depending on which direction the transducer is moving. Also, this circuitry takes a while to detect that a change of direction has taken place. A result of this is that when the user changes the direction of motion of the trackball or knob, the host "sees" it moving in the original direction for as many as two extra counts. If the user is controlling a cursor, this could be most disconcerting. Also, the knobs cannot be used with position-indicating scales, since moving a knob and then physically returning it to its initial position will leave it a short distance away from its initial position as far as the software is able to tell. To be fair, the NRC trackball suffered from this effect to some extent as well, but the error in their case is less because their circuitry provides only half the resolution of ours (one count per complete cycle of the inputs rather than two).

The best way to correct this is to replace all of the quadrature decoding circuitry with a different design that employs a very simple finite state machine. Without going into details, this would produce a circuit that counts up or down on every transition of either of the quadrature inputs. The 8-bit counter would then be effectively "locked" to the motion of the trackball or knob with no positioning errors. As a side benefit, the resolution of this circuitry would be double that of the current design, producing four counts per cycle of the inputs. We believe that this can be done with about the same number of IC packages as the current design if a single copy of the circuitry for the state machine is time multiplexed between the six sets of inputs.

The physical packaging of the project is not ideal. Currently, the trackball is mounted in one rectangular case while the knobs and lighted buttons are mounted on a sloping cabinet that also contains the microprocessor circuitry and the power supply. The power supply contains a switching regulator which makes audible noise, and the power supply and microprocessor require a cooling fan. Having these in the same cabinet as the buttons and knobs guarantees that they must remain within reach, and thus within hearing distance, of the user. It would have been better to put the noise-making components in a separate cabinet that could be kept further away from the user.

The shaft encoders that provide the knob inputs have very good bearings and are too easily turned by the slightest touch. Some material has been added to rub against their shafts and provide friction, but they are still excessively sensitive. There are different shaft encoders available that have a designed-in turning resistance appropriate for panel-mounted knobs; these probably would have been a better choice.

Some circuit board space could have been saved if the unused sections of the programmable timer chip had been used to generate the clock signals for the 8251A serial chip instead of the separate baud rate generator chip and crystal that are currently used.

There should be programmable character strings that are prepended and appended to each "message" from the controller to the host, to handle a wider variety of host communications protocols. For example, some operating systems might require a carriage return and line feed to follow each message in order for it to be passed on to the user program.

8. Conclusions

We are planning a second design iteration, based on the experiences detailed above. The quadrature decoding circuitry will be redesigned to use a finite-state machine as described above. The performance of the controller will be improved by some combination of software changes or a CPU change. The basic design principles described in section 3 and the algorithms described in section 4 will not be changed, as they appear to be sound. We believe that this design is a significant improvement over existing systems.

9. Acknowledgements

The fabrication of the prototype controller, and the design of the custom-built circuits used in it, was done by the MFCF Hardware Lab under the direction of Kim Martin. Ian! Allen designed the controller's error messages. Prof. Farhad Mavaddat and Art Hills provided access to and assistance with the Intel development system on which the controller's code was compiled and debugged.

We would like to thank the people at the National Research Council who designed the original trackball/button/knob unit that provided the inspiration for this project. They provided details of their implementation and comments that assisted in the early design of this project. We would particularly like to thank Marceli Wein for his comments on various facets of the NRC design and on improvements to this one.

This research was supported in part by the Natural Sciences and Engineering Research Council of Canada, the Provice of Ontario through its BILD Program, the University of Waterloo, and the Computer Graphics Laboratory.

References

[Beach82] R.J. Beach, J.C. Beatty, K.S. Booth, E.L. Fiume, and D.A. Plebon, "The Message is the Medium: Multiprocess Structuring of an Interactive Paint Program", *Computer Graphics*, Vol 16, No 3, August 1982, pp. 277-287.

[Intel78a] Intel Corporation, *PL/M-80 Programming Manual*, 1978.

[Intel83] Intel Corporation, *Microprocessor and Peripheral Handbook*, 1983.

[MSC83] Mouse Systems Corporation, *M-1 Mouse Technical Reference Manual*, 1983.

[PL81] Pro-Log Corporation, *Series 7000 STD Bus Technical Manual and Product Catalog*, 1981.