<BACK

NIALL MURPHY

Principles of User Interface Design

The placement of knobs, buttons, and switches is as essential to the making of a good product as the firmware. This article illustrates the delicate relationship between the two.

There are two fundamental approaches to usability. One is *usability by evaluation*; the other is *usability by principles*. Usability by evaluation involves dissecting a design to find its strong and weak points with a view to making improvements. While this is a necessary part of the validation of any product, it is not the only way to invest in usability. Historically, a lot of usability work is done this way because usability is not considered until someone realizes that the product is hard to use. Nobody notices that it is hard to use until the product, or a sizable portion of it, has been built. The easy part of usability by evaluation is criticizing the current design; the difficult part is deciding what would improve it.

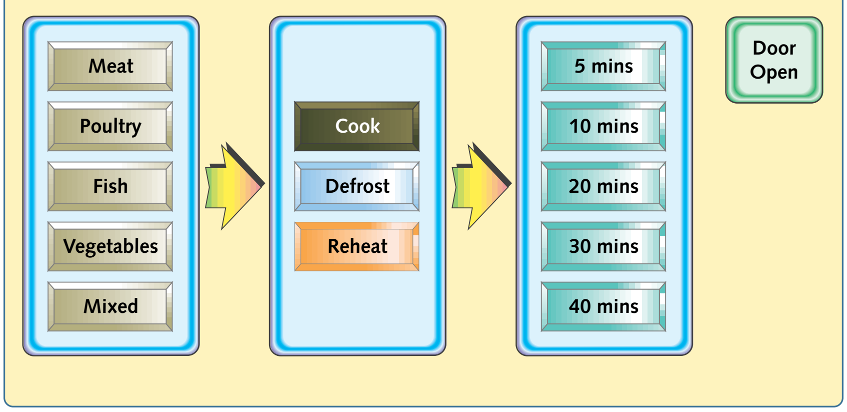
Usability by principles is about deciding ahead of time what usability properties will be desirable on this interface, and what types of people will use it. By naming and defining these principles, you will be equipped with a language that will allow the product's usability features to be discussed and documented more powerfully. It also encourages transfer of usability concepts from one product to another. More importantly it allows you to decide what you want from the interface before you begin designing.

Robustness

We are used to thinking of robustness as a mechanical property. It indicates how well the object tolerates rough use and carelessness. A robust user interface is not necessarily physically strong (though that has other obvious advantages), but it tolerates improper inputs, or makes them impossible. A robust interface not only protects the device from accidental damage due to an incorrect input, but also protects the user and the entities that the device acts upon.

In general, asking for confirmation of an action is a clumsy way to add robustness. Using an embedded interface should be as natural as using a tool from your toolbox.

FIGURE 1 A hypothetical microwave oven front panel



Back in 1982 the Sinclair ZX Spectrum home computer's user manual stated: "Nothing typed at the keyboard can damage this computer." This showed good insight on the writer's part. They realized that they could reduce users' anxiety by reassuring them that they would not destroy their purchase no matter how many silly mistakes they made. It was also a sign of good design of the computer—they made it robust. Many programmers wrote their first programs on that system, without having to worry about any damage a buggy program might cause.

In the desktop world if you request that a file be deleted and the system deletes the file, you would expect nothing else. But what if the initial request was a mistake? A robust interface should not allow an accident like that to occur easily. So the system now prompts users "Do you really want to delete *MyLife'sWork.txt*?" This has made the system more robust, but less usable because now it takes more key strokes to delete a file. A better approach is to allow an undelete command. Now the system is more robust because the file cannot be lost so easily, but the ease of use has not been compromised.

In general, asking for confirmation of an action is a clumsy way to add robustness. Using an embedded interface should be as natural as using a tool from your toolbox. Your hammer does not ask you if you want to hit something, just before impact. If it could, I do not think that many people would consider it a more usable tool. Confirming actions often becomes so automatic that it adds little protection in any case; the user will kit the OK button without fully considering the question asked.

Similarly, error messages, or a beep to indicate an illegal action, may seem like an appropriate response to an invalid action. An automatic teller machine that has been asked for too much money will display a polite message telling you that you are not rich enough. The system is protecting the user from becoming overdrawn. Sometimes you will need to put limits on the user to prevent them setting illegal values. These limits protect the device and the user from user errors. Sometimes the mechanical system will have intrinsic limits that cannot be violated—the top speed of a motor for a conveyer belt may be limited by the circuit driving it. However using an internal mechanical or electrical limit can have the disadvantage that the user may believe that the belt is moving at the entered speed, instead of at the limiting speed. For example, if the belt is physically limited by its motor to a top speed of four feet-per-second, but the user can enter five feet-per-second, the user may make other decisions based on the assumption that the belt is actually moving at five feet-per-second. By limiting the user at the time of input it is possible to give more accurate and meaningful feedback.

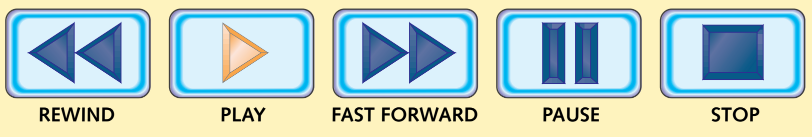
If you have error messages or warnings, be careful how you word them. Putting up a message that says "Illegal action" not only gives the user very little information, it also suggests that he is a criminal! With a little extra effort on the developer's part it is usually possible to provide a context-sensitive message that suggests what is actually wrong, such as "Can't record: no tape inserted."

Now that we have established a protocol for telling the user what he has done wrong, we have improved the product but the interface is becoming less user friendly. Devices that tell you what you can and cannot do are unpleasant to use. In many cases you can get the best of both worlds. On a TV set a user can press an up or down button to change channels. What happens when they reach the highest channel. The TV could beep and flash if the user tries to go any higher, and they will probably realize their mistake. It is much more satisfactory to wrap around to the lowest channel, avoiding the necessity of pointing out a supposed error to the user.

Users find it unpleasant to be told they have made a mistake, so design fewer paths that end in error messages.

Think of the error message as an airbag—it minimizes damage once the accident has happened. You are far better off supplying the user with an anti-lock braking system, which might avoid the accident in the first place. If a device takes numerical input from a keypad, it may be necessary to reject

FIGURE 2 Standard icons for basic VCR control



an input if the value is illegal. If the numeric keypad is replaced with a dial, the dial can simply limit the range at its mechanical limits, avoiding the possibility of an illegal value being entered.

In the last two examples, the system was able to detect that the input was not valid and took alternative action. What about the case where the user performs an action that is a valid input to the system, and then realizes that it was not the appropriate action. The undo command is a popular feature on many desktop applications. The problem with undo is that the user has to figure out how much will get undone. Will a second undo go back further into history, or will it redo the undone command. You cannot always afford to hit the undo button just to find out what will happen.

Sometimes it is possible to provide a set of actions that allow any action taken to be reversed. If you can move a robot arm to the left, then make sure that you can move it to the right just as easily. The effort taken to undo the action is equal to the effort taken to perform the action in the first place—leading to a robust system. A tape recorder (and they do exist) with a fast forward button but no reverse button will be frustrating because the user will have to turn over the tape to undo the action taken when he wound forward too far. The effort to undo the action is greater than the effort taken to cause the action in the first place—leading to a less robust system. Similar situations arise with mode changes. If it takes 10 steps to get from mode A to mode B, but only one key press to get from mode B to mode A, then expect your users to be displeased if they enter mode A by accident. One accidental key press will take 10 actions to rectify. This principle that actions and their opposites should involve similar effort is what Thimbleby calls *commensurate effort*.[1]

One case where you may want to make an exception to this is if mode B is a safer mode than mode A. You may then wish to make it quite difficult to get into the high risk mode, but easy to escape from it. This does not make the system easier to use but it does make it safer.

In an embedded system, reversing some commands is simply impossible when the physical action taken by the device cannot be undone. You cannot unlaunch a rocket! In these cases some confirmation is required. When you implement a confirmation there is a delicate balance between ease of use and certainty of intent. If you make the user type in a complex sequence every time they wish to perform an irreversible action then they are unlikely to do it by accident; however, ease of use will be reduced. If such actions are rare and the cost of an accident is high, the confirmation procedure may be elaborate, such as two operators simultaneously pressing a button that is reserved for this purpose. The buttons could be placed far apart so that one user could not press both. The important thing is to avoid forcing the users to confirm so many details, so many times per day, that they confirm actions automatically without actually reconsidering the consequences.

Consistency

The overall consistency of the design is a property that makes it easier to learn general rules that the user can apply to the whole interface. Consistency in the interface will be reflected in the software and vice versa, so the code can be a good place to detect inconsistency: "I can use this `blinkLight()` function everywhere except this one place. Why? There may be something different about what the user sees as well."

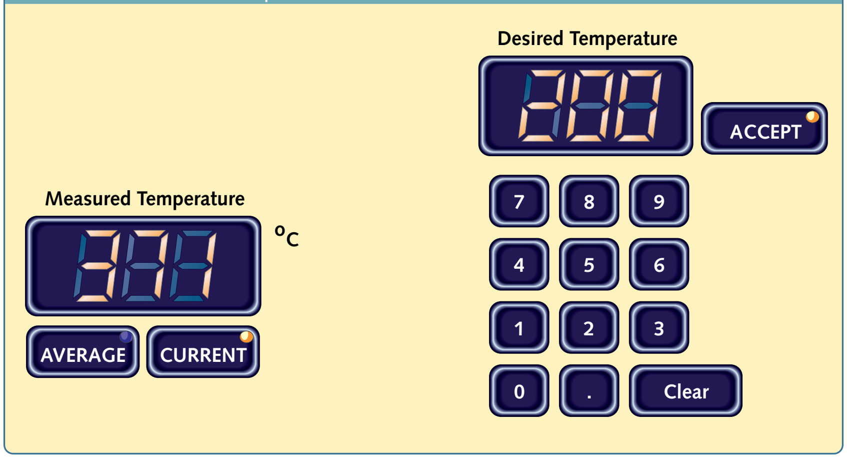
Consistency allows the user to develop general rules about how the interface works. When the user starts to explore a part of the interface that he has not previously used, it is these rules that he will depend on. Do not get lured into lowering your levels of consistency in the more advanced features, assuming that only expert users will be using those features, and that they will figure it out regardless. It is in these advanced features that consistency is of greatest value. The user may only rarely visit the more advanced features, and he will not want to learn them from scratch on each use.

Affordance

Affordance is the property that indicates how obvious a device's function is from its appearance. If I hand you a pair of scissors, you will notice that one end has proportions that fit comfortably in the hand. The other end has sharp edges so you are unlikely to hold it there. The sharp edges rubbing off of each other indicate that it is meant to cut something. The scissors affords holding and cutting.

Some things are obviously easy to use because you can see all of the controls. A tape deck has a door that opens to reveal two spindles that a tape could be placed over. Often the spindles are visible through a glass panel in the door and so are seen even if it does not occur to the observer to open the door. The tape deck has the affordance of something that can hold a tape. Until you examine the controls you do not know if this device is only for copying, or rewinding tapes, or whether it can actually play the tape. If speakers are visible that will be a clue

FIGURE 3 The front panel of a furnace



that it actually plays tapes. The buttons to control the tape may be marked PLAY, REWIND, FAST FORWARD, and STOP giving further information. The observer now knows that the device has three functions, and a way of halting any of them.

Conditioning has taught us that buttons can be pushed and that dials can be turned. But what if the device has a less conventional control. A joystick indicates that it should be grabbed by having grooves to fit fingers on the shaft. If a user approaches a device controlled by a touchscreen, they may not immediately realize that the screen's surface is touch sensitive. Making the on-screen buttons three dimensional will hint to the user that they can be pressed down just like a mechanical button.

If the front panel contains buttons and dials, it will be obvious to the user that the buttons can be pushed and the dials can be turned. But what do those buttons actually do? Labeling is a delicate art. Sometimes a button performs more than one action, and it requires two labels such as a STOP/GO button. The double label looks awkward. POWER does a good job of replacing ON/OFF, but such replacements are not always available. If a dial is graduated, then naming the units may replace the needs to name the function. A dial graduated in degrees Fahrenheit does not need to be marked FURNACE TEMPERATURE, assuming that there are no other temperatures settable on the furnace.

One common fault in industrial design is to place buttons in a regular pattern, in order to make the appearance of the device more symmetrical. They line them up like soldiers, each one looking like the next. This makes sense if each has a similar meaning, such as each one representing a different TV channel. If the buttons perform separate functions try to group them according to function. Use bigger buttons for the more popular functions, and keep the rare-but-nasty functions out of the way. If certain buttons, or a number of choices are likely to be pressed in sequence, arrange them in a left to right ordering, since this is the way people read, and that is the way they will scan a screen or display. See Figure 1 for an example. Arrows are even more useful if the path that we want the user to follow is not as natural as left to right.

Surface area

As the number of features in a device grows, the design is likely to hide many of them. This is often driven by the industrial designers who want a simple form and mechanical engineers who want fewer parts; many of the less frequently used functions end up under a menu, or reusing the buttons already available. Consider a telephone that can access voice mail. If you still have to use the buttons with digits on them, there is no clue by looking at the device that it is capable of accessing voice mail. Do not be fooled into thinking that you are doing the user a favor by making it look like any other telephone that he has used. The phone would be far easier to use if the buttons ENTER VOICE MAIL, NEXT MESSAGE, and DELETE MESSAGE were added. The phone may look more complex, but users not interested in the voice mail options will simply ignore the extraneous buttons.

The front panel of a device is made up of a number of controls and displays. A device with more dials, buttons, and displays has a greater surface area. In this context, the term surface area does not refer to physical size, but the number of controls available and the number of actions that can be performed upon them. The more of the user interface's functionality that is visible to the user, the easier it will be to learn the whole device, and the more obvious it will be what the device is capable of doing. The idea that users will be scared of a device with too many controls is a myth. Most people drive cars with dozens of controls available to the driver.

Users will resent many controls only if there is no organization, and the controls do not provide adequate indication of their purpose. Some of the worst designs are a result of taking a product that has a fundamentally complex interface, and trying to deliver it through a simple front panel. Devices are difficult to use because they have too few buttons, not because they have too many. If your software has to interpret the same button in many different ways, depending on the context, this is a sign that you may be cramming too much control into one component. I call this multiplexing a control. An interface will be less usable if the user has to decide, based on the current context, whether button "2" means exit voice mail, or delete message.

Output can be multiplexed as well. A single LED might indicate power on most of the time, but it can flash to indicate low battery at other times. A single numeric display can show temperature at one stage, and the time of day at another point. The user should have some hint, such as an AM/PM indicator, to show which mode it is in. A second display would increase the surface area, making it a better match for the available functionality. Having twice as many displays would make life easier on the user, not harder, assuming that the two displays are properly labeled.

Compatibility

Three levels of compatibility are at play in an interface. There is compatibility between what the user expects and what the user gets. There is also compatibility between different products of the same type. Finally, there is compatibility between the device and its surroundings and the devices with which it has to cooperate. I will deal with each of these in turn.

A lever-operated press moves down when the arm is raised and up when the arm is lowered. It works this way because of the levering mechanism used. This is good engineering, but bad usability. While the user may learn that the arm moves in a direction opposite to the press, he may revert to the more natural mapping in an emergency, thus causing an accident. Making the actual behavior compatible with the expected behavior can be easier in software where the engineering issues can be hidden. So UP buttons should be above DOWN buttons, and LEFT buttons should be on the left of RIGHT buttons. Similarly, high and low limits should be displayed in the appropriate vertical alignment. If a set of controls are near each other, or have similar coloring, the user will expect them to be related in some way. In general, you do not want to surprise the user.

Compatibility between products is not such a simple issue. The history of products in your market may dictate that certain practices are followed, long after a better way of doing the job has been found. Sometimes arbitrary differences exist in the marketplace and you will have to choose which trend to follow.

Standards bodies that have been so active on the desktop have only made minor inroads into the embedded world. Some attempts have been made to standardize alarm sounds and lights in hospitals, but that is a long way from standardizing the whole interface. ISO 9995 has set a standard for the arrangement of letters on the buttons of a telephone or other numeric keypad, though phones the world over still vary in the placement of letters. Annoyingly, numeric keypads on computers are upside-down when compared to the telephone standard. The symbols on the main buttons of a VCR, shown in Figure 2, have also been standardized, though the rest of the controls on the VCR may vary greatly—even within one product line.

Some standards are not so quickly accepted. ISO 8601 is a standard for date and time formats. It dictates a yyyy-mm-dd format for dates. However, Europe currently uses the dd/mm/yy format and the U.S. uses the mm/dd/yy format. The ISO standard is a more logical format since it starts with the most significant unit and then moves to the successively smaller units as you read it from left to right. This corresponds to normal numbers and telephone numbers (country code followed by area code followed by phone number). The ISO standard also has the distinct advantage that it can be sorted chronologically by simply sorting them alphabetically, since the most significant unit (year) is to the left. While this standard is common in Japan and a few other countries, it is not used by the vast majority of the world—mainly because the other formats are so well established that it is impossible to dislodge them.

If no standard is in place to keep the behavior of competing products in line, then at least try to ensure that all products in a family, or from the same company, are compatible. Compatible mental models and behavior are more important than compatible appearance. Matching interactions contributes more to the ability to transfer knowledge gained from one product to another than matching the color or the company logo.

Compatibility with older products can sometimes stand in the way of a consistent orthogonal interface. The competitors or predecessors of a particular piece of equipment may have set a precedent for the way certain operations are performed. For some common operations you may have to follow the established norm, even if that does not fit in with the way other operations on your interface behave. Engineers often find this frustrating—their elegant design is being soiled by what is seen as an artificial and unfair requirement created by history, rather than being part of the perfect solution to the problem at hand.

When the interaction has to change, it can be advantageous to deliberately change the physical appearance so that the users will not be expecting the same behavior. On an older product we had one red light and one yellow light, which had the words ALARM and CAUTION written on them. For the newer product, the rules for when the lights turned on and when they flashed completely changed because we had to comply with a new regulatory standard. However, the design of the front panel was kept consistent with the old product, keeping the words ALARM and CAUTION and the same color scheme. This caused confusion when users of the older product saw the same alarm names and expected the same meaning. If the names had changed, there would have been a hint to them to expect different behavior.

A product also needs to be compatible with the devices that would tend to be used alongside it. If you are selling components of a rack stereo system, the on/off button for each one should have similar positioning. If you sell TVs and VCRs, the buttons for changing channels should work in a similar fashion.

You must also attempt to be compatible with the surrounding environment. If the device will be used in a noisy environment, very quiet alarm sounds will not be appropriate. On the other hand, a pocket calculator may be used by students in a quiet library, so you do not want it to make loud key-click noises. Sometimes you will not be able to guess the environment in advance. One automatic teller machine lobby that I have used has mirrored walls, similar to an elevator, to prevent customers feeling claustrophobic. Unfortunately, the same mirrored surface makes the user's key presses—including their PIN—visible from almost anywhere in the lobby!

Directed interfaces

Some interfaces strongly suggest a direction. A question-and-answer session provides an interaction where the direction is dictated by the user interface. The user may have the option to break out of the sequence, but the questions themselves suggest to the user that the next appropriate action is to provide an answer. That type of interaction is considered *directed*. An example of a less directed interface is a car dashboard. You start the ignition and any number of things are available for you to do. The sequence of putting it into first gear and pulling out of the parking space into traffic involves lots of options—you could have gone into reverse, turned on the lights and/or wipers, and so on. The interface does not suggest any one path more than any other. This is a non-directed interface.

You would make your interface more or less directed for a number of reasons. A more directed interface suits a device with a single simple goal. It also suits a novice, who will not want to make many decisions. A non-directed interface provides more power to the user who knows how to navigate the device's features. They can go directly to the control that they want without having to follow a predefined sequence. Sometimes an interface will change from one to the other. An automatic teller machine will give you little or no options until you have entered your card and personal identification number. This is as it should be. The machine should not provide any service to a user who attempts to bypass this step. Once the card has been validated, the interface can relax and allow the user a bit more flexibility. The interface is directed enough that the novice does not get lost, but more than one option is available at each step.

Novice users usually prefer directed interfaces with an obvious path; non-directed interfaces are more powerful but more difficult to use.

Multithreading

Some interfaces allow many independent paths through different parts of the interface to be active simultaneously. Consider a hi-fi system with a CD player, tape deck, tuner, and amplifier. The tape deck can rewind while the user is changing channels on the tuner. The amplifier may be instructed to take input from the CD player, so that is the audible source of music. Each piece of equipment is being allowed an independent thread of control. This is analogous to multithreaded software. In most hi-fi systems you would consider the ability to multiplex an accident of the design of independent components that are not capable of communicating with each other. A smarter system might change the amplifier to be directed to the last device that was used, and disable all others. This would avoid accidentally leaving a tape playing when you are listening to the radio. However, this also means that you lose the ability to rewind a tape while listening to the radio—something users may find useful. The conclusion from this investigation of the multithreaded hi-fi: multithreading makes use of the device more challenging, but offers more power to the user.

Multithreading of an interface does not necessarily require the software to be multithreaded. The state of each thread of control can be stored separately, and events from each thread be handled by the same real-time task.

Windowing systems typically have a separate thread of control for each application. Embedded systems have slightly different needs. Many process control devices consist of a settings area and a monitoring area. Allowing each one a thread of control helps reinforce in the user's mind that monitoring and control are two separate activities. Consider the interface to a furnace shown in Figure 3. While the user is in the middle of typing in the new temperature, he may decide that he wants to check the average temperature. He can press the AVERAGE button, and see the display change. The number he was entering on the keypad is still valid. He can return to that activity and accept the new value. If there was a single thread of control, operations on the monitoring side would cancel incomplete actions on the settings side and vice versa.

As an aside, the ACCEPT key in this control panel was deliberately kept as far as possible from the CLEAR key. These keys are often placed next to each other since they have related functions. Unfortunately on many systems, the cost of pressing one instead of the other is quite high. In this case the ACCEPT key was placed beside the figure being accepted to encourage the user to look at the digits they have just entered before committing them.

Modes

Some interfaces require navigating through a number of modes. Sometimes these modes will correspond to a different phase or operation. In such cases the modes will seem natural and necessary to the user. I expect my calculator to behave differently in hexadecimal mode than in decimal mode.

Modes have a bad reputation in the user interface community, and this reputation has been earned by many user interfaces that make terrible use of modes. Text editors like vi, with insert and control modes, can cause havoc when the user starts to type a sentence, only to realize that he has just typed seven different commands, and half of the screen has been indented and changed to upper case. My microwave oven has a grill mode and a microwave mode. I have often walked up to it, inserted some food, and turned it on for five minutes, only to realize that I had the oven in grill mode rather than microwave mode. The problem is not with the modes themselves, which are often quite necessary, but with recognizing which mode the device is in. Sometimes the clue is quite small and easy to overlook. My VCR has a play mode and a record mode. There is never any confusion, because the word PLAY or RECORD appears on a large one-line text display when one of those modes is active. So little other information is being presented that I cannot overlook this visual clue.

The most troublesome modes often are invented for the sole purpose of allowing some of the controls or displays to be multiplexed. This is true for the text editor, which would be far more useful if more buttons were available that were dedicated to the command functions. If you are short of space for extra controls or displays, you may not be able to justify much space for a mode indicator. Now you have two bad features: a mode that does not reflect a different task or role for the user, and a mode that does not make itself obvious. Use modes where necessary, but always make the current mode obvious.

Equal opportunity

Keeping the paths that the user has to follow short and simple is always an advantage. One of the ways to keep the paths short is to use the data already available from the device output. When you select cruise control in a car, you do not have to dial in the speed, since it is already available. The interface takes one of the outputs, the current speed as displayed on the speedometer, and uses it as an input to the cruise control system. A VCR may use a similar mechanism. If the user is setting a timer, the currently selected channel can be assumed as the channel that is used for this setting. The user may have the option of changing it, but guessing defaults in this way can reduce the amount of interactions that the user has to make.

The principle of using a piece of device output as a piece of user input is known as equal opportunity.[2] In windowing systems, the ability to cut text from one window and paste it into another is quite common. Many systems limit the usefulness of this feature by not allowing it for all output. I occasionally get an error message that I wish to e-mail to the system administration team. I am forced to retype the message because my computer will not allow me to copy it and paste it into my e-mail window. If data is already available, do not force the user to enter it again. A number of other examples of equal opportunity are available on my Web site.

Multiple paths

If your interface provides more than one way to perform a function, ask yourself "Is there a reason for each path?" You should ask this question, because the user will. If one way is slow but obvious, and another is quick but only likely to be known by the expert user, then each has a purpose. The quick method is a shortcut. In other cases it may be useful to have a feature available from a number of different modes, to reduce the need for the user to change modes to find this popular feature.

However, if the alternatives are arbitrary, the user may assume that there is some difference in the result—perhaps some side effect that he never noticed while using this feature. This may lead to user discomfort since he believes that there is something about the interface that he does not understand.

Migrating from mechanical controls

Having complete software control over the interface removes the user by one more level from the mechanics of the device itself. If software interprets all user inputs and then generates signals to control the device based on those inputs, you have a fly-by-wire system. This may affect the user in a number of subtle ways. A dial connected to a potentiometer may have had a logarithmic relationship with the voltage output. The software control system may make that relationship linear.

Analog needles showing values are usually damped to prevent oscillations of the needle that would make it difficult to read the value indicated. A real-time graph of the value may show a lot of variability that was previously not visible to the user. The user may believe that the new device is doing a worse job of controlling the process, when in fact it is just doing a far better job of reporting the state of the process.

Having software in the loop will greatly increase the amount and types of information that can be presented to the user. Engineers are inclined towards a more-information-is-always-better philosophy because of the nature of their work. This does not always lead to better interfaces. Users want enough information to solve the problem at hand. They may not be as skilled as a typical engineer at filtering out detailed information that is only of peripheral interest.

When a display is converted from a mechanical indicator to a software-controlled display, it may be tempting to change the type, as well as the quantity, of information presented. Consider the fuel level indicator on a car dashboard. If this is replaced by a small liquid crystal display (LCD) display, a variety of information can be presented to the user in a number of possible approaches, such as expressing remaining fuel as a percentage, or in gallons, or in number of miles remaining before the fuel runs out. In each case we allow the user to form a conceptual model of the fuel tank. Each of these models is different, but once we choose one we must be sure that we can fill in all of the information that the model requires. To convert from gallons to miles remaining, we need a conversion factor. Using the average fuel consumption of the car may not be sufficient if the driver has been sitting in a traffic jam for 45 minutes. What seemed initially a very simple requirement now demands that the fuel monitor also monitor consumption. Now the user has a new rule to learn. The number of miles that he reads from the monitor is only valid if he continues in his current driving pattern. You may think that this is a trivial rule to understand, but it is important to realize that every rule that is created by the interface designer must eventually be learned by the user if he is to make the most effective use of the interface. The rule may be learned from the user manual, if he is one of the few people who would read that section of the car's manual. Otherwise, it is gleaned from observation, since the interface does not explicitly tell him. Such hidden rules can be dangerous. If there are too many of them, the user will constantly find himself surprised by the device's actions and this will lead to mistrust.

Having moved from an analog device to a digital display, many interfaces fall into the trap of giving the user far more precision than they require. There is little value in telling users that they have enough fuel for another 31.7 miles, since they would be foolish to wait until the last 0.7 of a mile before refueling. The vagueness of the needle in this case is an advantage. The needle says "I don't know exactly how much further you can go, and we are not at the panic stage yet, but if you see a gas station, you may as well stop." Since no one can know exactly how many more miles the fuel will last, this information may be more appropriate than 31.7 miles. The more precise value may lure the driver into a false sense of security. A tempting alternative may be to present the driver with a range of values. However, I do not think many drivers would appreciate a display that said 31.7 miles ±10%. Remember, they are not all engineers and may not be exposed to the concept of tolerance every day.

If the tank is low on fuel, the sensible thing to do is fill up at the next opportunity. Needle gauges are notoriously non-linear, but drivers rarely complain. The designer may do well to copy the functionality of the needle. This will be less threatening for the new user because he can relate easily to his past experience with the needle display. An interface controlled from software could control a needle, or use a bar graph on an LCD display, or a bar graph made up of a number of LEDs. Which appearance is chosen is less significant than the functionality attached to it. Users will quickly adapt to a new appearance, but users take longer to adapt to changes in functionality. **esp**

Niall Murphy has been writing software for user interfaces and medical systems for ten years. He is the author of Front Panel: Designing Software for Embedded User Interfaces *(R&D Books). Murphy's training and consulting business is based in Galway, Ireland. He welcomes feedback and can be reached at nmurphy@panelsoft.com.*

References

1. Thimbleby, Harold. *User Interface Design*. New York City: ACM Press, 1990.
2. Equal Opportunity Tutorial available at *www.panelsoft.com/tut_equal*.

Resources

Niall Murphy's on-line list of usability resources is available at *www.panelsoft.com/usable.htm*.