Principles of User-Interface Design
User-Interface Design for Embedded Systems

Larry Constantine
Acton, Mass.

Larry Constantine is a pioneer of the structural revolution and the developer of structured design. A consultant and frequent lecturer on CASE and the integration of object-oriented development with traditional structured methods, he also teaches new approaches to managing and organizing design teams and work groups. With Ed Yourdon, he wrote Structured Design, the definitive text on the subject still widely used after 14 years in print. He is a graduate of Sloan School of Management at MIT and has served on the faculties of four major universities.
Effective user interface design for embedded systems starts with recognizing the user interface as important and then putting users at the center of the design and development process. Embedded systems developers need to be aware of established general principles of human-machine interaction as well as the special features and constraints that characterize their particular embedded system applications. A view that bridges hardware and software design issues is needed.

I was sitting with friends around a kitchen table in Sheffield, England, when one of them asked what time it was. I was fresh off the plane and had forgotten to reset my wrist watch, so it took me a few seconds to remember and do the conversion. Before I could reply, someone else glanced across at the microwave and called out, “It’s 88:88.” It is a commentary on our times and a symptom of the sorry state of user interface design in embedded systems that wherever we go in the world we find microwaves and televisions and VCRs flashing 12 midnight or 00:00 or 88:88. These are some of the most ubiquitous embedded systems in the world. They have been through several generations of development and redesign, and competition for customers is keen, yet so few of them seem to make simple functions, like setting the time, simple enough for the average consumer to remember how to do it after a power failure or after moving the appliance to another outlet.

Strangely, as the cost of the equipment goes up, the interfaces seem to get worse. Friends in the medical field complain about the controls for patient monitoring equipment. Lab technicians have to scurry back to manuals to use digital instruments. Building supervisors may use only a fraction of the capabilities of computerized environmental monitoring systems. Then there is my sophisticated fax machine, with its automatic fax/voice switching and its easy-to-use “one-touch” dialing. All I have to do to set one of these “one-touch” dialing keys is (pardon me while I reread the manual) press the [PROGRAM] key, then [4], then [CONFIRM/ENTER], after which it displays an “informative” message:

1. Enter One-Touch

To this I must respond by pressing [CONFIRM/ENTER] again, not [MODE/SELECT] (makes no sense to me either!), before pressing the key I want to reprogram. It goes on and gets worse. This is brutal, unconscionably obscure programming.

It is possible to do these things right, or at least make them so straightforward that you only have to learn once. My car stereo allows me to assign a new station to one of the buttons just by tuning the station, then holding in the desired button until a confirming chirp is heard. This may not quite qualify as intuitive, but it does seem to make sense and is not hard to remember. From the programming standpoint, it is no more difficult to code this elegant approach than the clumsy one in my fax machine. The fax machine cost five times as much as the car stereo, yet it has one of the worst user interfaces imaginable. How did this happen? How could anyone be perverse or dumb enough to design such an interface?

My guess is that nobody is quite that perverse or stupid, certainly not anyone still employed. Probably nobody actually designed the user interface; it was just programmed. This is perhaps the most widespread problem with user interfaces for embedded systems.

In order to get better user interfaces, you have to actually design the user interface. All of us can think of systems that were produced but never really designed. If the user interface is not recognized as a distinct facet of the system requiring special treatment and attention as an integral whole, it is likely to emerge from the development cycle without ever having been planned or designed.

The best way to assure that the user interface will get the focus it deserves is to make it someone’s job (Constantine, 1992). Some person or group should have responsibility for the design of the user interface and accountability for the results. The developers assigned responsibility for the user interface should be supported with training, consultation, and other resources, and should be given the time and authority to do the job. Ideally, one person or group should oversee both the software and hardware aspects of the user interface for embedded systems projects. When this is impossible, at least the hardware people and the software people should be having a dialogue, which means two-way communication in which there is give and take across the boundaries between the hardware and software design domains.

What Is Special about Embedded Systems?

The relationship between hardware and software is a central matter in embedded systems programming. Embedded systems programmers like to think that what they do is special, even unique, in the software development world. Although catalogues of special aspects have been proposed, when all is said and done, embedded systems programming has all the same ingredients as all other computer programming, only the proportions may differ. P. J. Plauger (1989) has argued that what makes embedded and real-time systems development special is a preoccupation with time. Time, timing, speed, sequencing, and synchronization loom large in the minds of designers and programmers. These matters inform all programming, but are given special attention in embedded systems and real-time programming.

When it comes to the user interface, embedded systems also seem to differ from “normal” applications development. After all, embedded systems are embedded in equipment or hardware that almost completely determines or at least constrains the appearance of the interface to the user. The Windows programmer on a PC can arrange radio buttons and slide bars anywhere on the screen and give them whatever look and feel is desired. The embedded
Effective user interface design for embedded systems starts with recognizing the user interface as important and then putting users at the center of the design and development process. Embedded systems developers need to be aware of established general principles of human-machine interaction as well as the special features and constraints that characterize their particular embedded system applications. A view that bridges hardware and software design issues is needed.

I was sitting with friends around a kitchen table in Sheffield, England, when one of them asked what time it was. I was fresh off the plane and had forgotten to reset my wrist watch, so I took me a few seconds to remember and do the conversion. Before I could reply, someone else glanced across at the microwave and called out, “It’s 88:88.” It is a commentary on our times and a symptom of the sorry state of user interface design in embedded systems that wherever we go in the world we find microwaves and televisions and VCRs flashing 12 midnight or 00:00 or 88:88. These are some of the most ubiquitous embedded systems in the world. They have been through several generations of development and redesign, and competition for customers is keen, yet so few of them seem to make simple functions, like setting the time, simple enough for the average consumer to remember how to do it after a power failure or after moving the appliance to another outlet.

Strangely, as the cost of the equipment goes up, the interfaces seem to get worse. Friends in the medical field complain about the controls for patient monitoring equipment. Lab technicians have to scurry back to manuals to use digital instruments. Building supervisors may use only a fraction of the capabilities of computerized environmental monitoring systems.

Then there is my sophisticated fax machine, with its automatic fax/voice switching and its easy-to-use “one-touch” dialing. All I have to do to set one of these “one-touch” dialing keys is (pardon me while I reread the manual) press the [PROGRAM] key, then [CONFIRM/ENTER], after which it displays an “informative” message:

1. Enter One-Touch

1-3 Enter/Select

To this I must respond by pressing [CONFIRM/ENTER] again, not [MODE/SELECT] (makes no sense to me either), before pressing the key I want to reprogram. It goes on and gets worse. This is brutal, unconscionably obscure programming.

It is possible to do these things right, or at least make them so straightforward that you only have to learn once. My car stereo allows me to assign a new station to one of the buttons just by tuning the station, then holding in the desired button until a confirming chirp is heard. This may not quite qualify as intuitive, but it does seem to make sense and is not hard to remember. From the programming standpoint, it is no more difficult to code this elegant approach than the clumsy one in my fax machine. The fax machine cost five times as much as the car stereo, yet it has one of the worst user interfaces imaginable. How did this happen? How could anyone be perverse or dumb enough to design such an interface?

My guess is that nobody is quite that perverse or stupid, certainly not anyone still employed. Probably nobody actually designed the user interface; it was just programmed. This is perhaps the most widespread problem with user interfaces for embedded systems.

In order to get better user interfaces, you have to actually design the user interface. All of us can think of systems that were produced but never really designed. If the user interface is not recognized as a distinct facet of the system requiring special treatment and attention as an integral whole, it is likely to emerge from the development cycle without ever having been planned or designed.

The best way to assure that the user interface will get the focus it deserves is to make it someone’s job (Constantine, 1992). Some person or group should have responsibility for the design of the user interface and accountability for the results. The developers assigned responsibility for the user interface should be supported with training, consultation, and other resources, and should be given the time and authority to do the job. Ideally, one person or group should oversee both the software and hardware aspects of the user interface for embedded systems projects. When this is impossible, at least the hardware people and the software people should be having a dialogue, which means two-way communication in which there is give and take across the boundaries between the hardware and software design domains.

What Is Special about Embedded Systems?

The relationship between hardware and software is a central matter in embedded systems programming. Embedded systems programmers like to think that what they do is special, even unique, in the software development world. Although catalogues of special aspects have been proposed, when all is said and done, embedded systems programming has all the same ingredients as all other computer programming, only the proportions may differ. P. J. Plauger (1989) has argued that what makes embedded and real-time systems development special is a preoccupation with time. Time, timing, speed, sequencing, and synchronization loom large in the minds of designers and programmers. These matters inform all programming, but are given special attention in embedded systems and real-time programming. When it comes to the user interface, embedded systems also seem to differ from “normal” applications development. After all, embedded systems are embedded in equipment or hardware that almost completely determines or at least constrains the appearance of the interface to the user. The Windows programmer on a PC can arrange radio buttons and slide bars anywhere on the screen and give them whatever look and feel is desired. The embedded
systems developer programming a digital oscilloscope is presented with the layout of switches and dials as a given and told to program to it. A telephone keypad has twelve keys and that’s it. The packaging and design people for the instrument manufacturer have determined that black anodized is in, an alphanumeric LCD display is the output, and membrane switches are to be the sole input. Period.

As a rule, human interfaces for embedded systems applications are argued to be more specialized and more fixed. On a standard AT-style extended keyboard, a shifted keystroke can be assigned any meaning, while a rotary volume control is a rotary volume control, whether it is directly functioning as a voltage divider or merely as an input to an embedded program. Right?

Perhaps. But some of the differences between embedded systems user interfaces and the interfaces on the general purpose platform used by the programmer for development are more a matter of degree than absolute differences. In software for general purpose computing platforms, the hardware interface is also a given, but it is apt to be more generic, more intrinsically flexible. If the developer wants an application to sell widely, it had better depend on a mouse and keyboard for input and a monitor for output to the user, perhaps supplemented by audio feedback. And, as standards for graphical user interfaces spread, the use of these devices will probably have to conform to many rules and preferences for look-and-feel.

Furthermore, although the general rule in embedded applications is that the input-output interfaces are specialized to the application to some greater or lesser degree, embedded systems programmers are increasingly presented with hardware interfaces every bit as rich and flexible as on general purpose platforms. The input interface is often an array of buttons, chiclet keys, or membrane switches, in which all positions are completely undistinguished from each other and largely interchangeable. VCRs use the television screen as a display for on-screen programming; lab instruments often have full keyboards; and even LCD panels are getting larger and more flexible. Some embedded system programming even involves conventional full-screen displays, but these are likely to be the exception rather than the rule. Just how these more flexible input-output devices are utilized can often be determined or influenced by the embedded systems programmer.

Another important feature of many embedded systems applications is that the ease and accuracy of use may be even more critical than in other kinds of software. Many such systems are used on an “everyday” basis with a high frequency of interaction by untrained users. Others are critical systems in which the risks of injury, damage, or loss are substantial. Both high usage frequency and critical behavior put a premium on getting the user interface right so that the interface itself does not slow work or generate costly errors. Small details may substantially influence the success of a system. Discount and liquidation catalogs are brimming with consumer products that failed in the market place because of relatively small, annoying features that made them substantially more difficult to use.

Good user interfaces require extraordinary attention to detail. Seemingly unimportant matters can substantially interfere with smooth usage. Requiring an action that goes against how people otherwise think and work introduces a permanent increase in the probability of error in using the system, which can contribute to fatigue and frustration and further compound the difficulties with the system (Constantine, 1990). Even so small a thing as putting up-down scrolling controls side-by-side can be shown to continue to lead to user errors regardless of the thoroughness of training or duration of use.

Some of the transgressions in this area are egregious. Shown below is the layout of the entry keys for a VCR remote control. Two rows of six keys have been allocated to numeric entry for channel numbers. Perhaps the engineers were partially reformulated dualists. The result is that the user must always read the numbers carefully. Even so, something approaching a third of the time the wrong key is pressed, and entering a two-digit number almost guarantees an error. Users of this control usually stick to the channel-up/channel-down buttons on the left, which slows down getting the channel you want. In such cases the software developers would have had to get together with the hardware designers. One could always hope that they could be dissuaded from such an infelicitous layout. (The madness continues in the screwball way the same keys are used to enter dates for programming the VCR or resetting the clock. Instead of something marginally sensible like the days being on keys 1-7, they are here on keys 6-0 plus “SP” and “EP.” And why did some idiot put the tape speed select up here with the channel select anyway? Ah, yes, two left over keys in the two rows of six!)
systems developer programming a digital oscilloscope is presented with the layout of switches and dials as a given and told to program to it. A telephone keypad has twelve keys and that's it. The packaging and design people for the instrument manufacturer have determined that black anodized is in, an alphanumeric LCD display is the output, and membrane switches are to be the sole input. Period.

As a rule, human interfaces for embedded systems applications are argued to be more specialized and more fixed. On a standard AT-style extended keyboard, a shifted keystroke can be assigned any meaning, while a rotary volume control is a rotary volume control, whether it is directly functioning as a voltage divider or merely as an input to an embedded program. Right?

Perhaps. But some of the differences between embedded systems user interfaces and the interfaces on the general purpose platform used by the programmer for development are more a matter of degree than absolute differences. In software for general purpose computing platforms, the hardware interface is also a given, but it is apt to be more generic, more intrinsically flexible. If the developer wants an application to sell widely, it had better depend on a mouse and keyboard for input and a monitor for output to the user, perhaps conventional full-screen displays, but these are likely to be the exception rather than the rule.

Constantine

Both high usage frequency and critical behavior put a premium on getting the user interface right so that the interface itself does not slow work or generate costly errors. Small details can be assigned any meaning, while a rotary volume control is a rotary volume control, whether it is directly functioning as a voltage divider or merely as an input to an embedded program. Right?

Perhaps. But some of the differences between embedded systems user interfaces and the interfaces on the general purpose platform used by the programmer for development are more a matter of degree than absolute differences. In software for general purpose computing platforms, the hardware interface is also a given, but it is apt to be more generic, more intrinsically flexible. If the developer wants an application to sell widely, it had better depend on a mouse and keyboard for input and a monitor for output to the user, perhaps supplemented by audio feedback. And, as standards for graphical user interfaces spread, the use of these devices will probably have to conform to many rules and preferences for look-and-feel.

Furthermore, although the general rule in embedded applications is that the input-output interfaces are specialized to the application to some greater or lesser degree, embedded systems programmers are increasingly presented with hardware interfaces every bit as rich and flexible as on general purpose platforms. The input interface is often an array of buttons, chiclet keys, or membrane switches, in which all positions are completely undistinguished from each other and largely interchangeable. VCRs use the television screen as a display for on-screen programming; lab instruments often have full keyboards; and even LCD panels are getting larger and more flexible. Some embedded system programming even involves conventional full-screen displays, but these are likely to be the exception rather than the rule. Just how these more flexible input-output devices are utilized can often be determined or influenced by the embedded systems programmer.

Another important feature of many embedded systems applications is that the ease and accuracy of use may be even more critical than in other kinds of software. Many such systems are used on an "everyday" basis with a high frequency of interaction by untrained users. Others are critical systems in which the risks of injury, damage, or loss are substantial. Both high usage frequency and critical behavior put a premium on getting the user interface right so that the interface itself does not slow work or generate costly errors. Small details may substantially influence the success of a system. Discount and liquidation catalogs are brimming with consumer products that failed in the market place because of relatively small, annoying features that made them substantially more difficult to use.

Good user interfaces require extraordinary attention to detail. Seemingly unimportant matters can substantially interfere with smooth usage. Requiring an action that goes against how people otherwise think and work introduces a permanent increase in the probability of error in using the system, which can contribute to fatigue and frustration and further compound the difficulties with the system (Constantine, 1990). Even so small a thing as putting up-down scrolling controls side-by-side can be shown to continue to lead to user errors regardless of the thoroughness of training or duration of use.

Some of the transgressions in this area are egregious. Shown below is the layout of the entry keys for a VCR remote control. Two rows of six keys have been allocated to numeric entry for channel numbers. Perhaps the engineers were partially reformed duodecimalists. The result is that the user must always read the numbers carefully. Even so, something approaching a third of the time the wrong key is pressed, and entering a two-digit number almost guarantees an error. Users of this control usually stick to the channel-up/channel-down buttons on the left, which slows down getting the channel you want. In such cases the software developers would have had to get together with the hardware designers. One could always hope that they could be dissuaded from such an infelicitous layout. (The madness continues in the screwball way the same keys are used to enter dates for programming the VCR or resetting the clock. Instead of something marginally sensible like the days being on keys 1-7, they are here on keys 6-0 plus "SP" and "EP." And why did some idiot put the tape speed select up here with the channel select anyway? Ah, yes, two left over keys in the two rows of six!)

The Changeable and the Unchangeable

Three variations on the hardware/software theme in embedded systems interface design must be considered. Some things are beyond control of the software and software developer; some things are within control of the software and software developer; and some things, even if they involve hardware, are potentially within the influence of software developers.

The keyboard on my main desktop system illustrates these three variations. It is a fully programmable marvel with its own processor and memory. In principle, any key can be remapped to any other key, and any key can be assigned to a macro string. It can even download and upload maps and keystroke macros to and from its internal memory. It also has quite a good feel to it, with clear tactile and audible feedback.

In the first category is the typing feel of the keyboard. Nothing the program or programmer
can do can possibly affect this, at least on this keyboard. Some keyboards are mechanically silent and provide audible feedback through a programmed chirp or click. How this is programmed can profoundly affect usability. I know of one keyboard with a chirp that could not be silenced under keyboard control and that was so annoying the owner finally opened the case and cut the traces to the speaker. However, there will always be basic electromechanical characteristics beyond the reach of programming or programmers to influence.

In the second category is an odd interaction between key remapping and keystroke macros. On my keyboard, even the shift keys can be remapped. Nothing prevents the user from deciding that the “L” key will become the “CapsLock” key. The user can also assign different macros to the shifted and unshifted state of any key, so <Ctrl><PgUp> can generate a different character string than <Alt><PgUp>. Pretty keen, huh? Well, except I can’t both remap shift keys and still use them to condition different macros from the same key. If I remap <Ctrl> and <Alt> so that they are on the keys above and below the left-hand <Shift> key, as many of us writers prefer, I can’t have different macros on <Ctrl><PgUp> and <PgUp>. By analyzing what happens when you try to do this I concluded that it was just a matter of programming some tests in one order rather than another. Too bad. The programmers unnecessarily limited the generality of their user interface.

To illustrate the third category, consider the diagram below, showing the keys and LED indicators in the upper right of my keyboard. Here we have the typical abomination that comes about when nobody actually examines carefully what a physical interface looks like. The PROGRAM indicator is, logically enough, above the <ProgrmMacro> key. Except this indicator is on if the keyboard is operating with macros active and off if not active. It is turned on and off by the <SuspndMacro> key to the right below it. Then there is the CAPS LOCK indicator above the <ScrollLock> key, whose indicator is to the right above the <Pause> key. The NUM LOCK indicator is farthest from the <NumLock> key, over on the side toward the main keyboard where the <CapsLock> key is to be found. Alas, these LEDs are not under user control, but they are under control of the embedded program. The principle here is the same as the one governing the placement of light switches on switch panels for rooms with multiple lights or for multiple rooms. The switch closest to the light should control it. With this keyboard arrangement, to know what is happening, the user cannot just rely on the position of the indicator, but must actually read the legend above it. And it requires “double-think” to remember that you turn PROGRAM mode (as indicated by the LED) back on using the <SuspndMacro> key, not the key under the light.

The embedded systems programmer at the keyboard company should have looked at more than just the logical relationship between keys and indicators. The actual physical relationship should also have been considered. If necessary, the programmers should have alerted the hardware design people so that the legend could be printed differently and/or key tops be swapped.

Bells and Whistles

In some cases an improved interface can be achieved with only relatively minor software changes. As a frequent international traveler, I appreciated the gift of a pocket international time zone/monetary exchange rate calculator. In addition to an enhanced calculator keypad, this little device has this a nice user-friendly interface in the form of a world map. Associated with the map are small pads with currency symbols and the names of cities. Touch one and you get the time for that city or the previously entered currency exchange rate, depending upon what mode the unit is in. Alas, not all the time zones are represented by one of the 17 cities presented and there is no provision for entering a time zone other than those preprogrammed. So you do the extra adding or subtracting in your head, no great mental challenge, but a silly annoyance that seems to defeat the purpose of the calculator. And it certainly would be no programming challenge to provide such a feature.

Embedded systems programmers should be on the alert for such potential enhancements to a user interface, especially where expanded capability can be provided through a simple, generalized facility. On the other hand, some things are so easy to do in code that developers are tempted to throw them in without being asked. This can lead to the kind of “creeping featurism” that plagues so many spreadsheets, word processors, and graphics packages. Like the baby carriages that is also a high chair and a stroller and a bassinet, a system that does too many things can actually be harder to use.
can do can possibly affect this, at least on this keyboard. Some keyboards are mechanically silent and provide audible feedback through a programmed chirp or click. How this is programmed can profoundly affect usability. I know of one keyboard with a chirp that could not be silenced under keyboard control and that was so annoying the owner finally opened the case and cut the traces to the speaker. However, there will always be basic electromechanical characteristics beyond the reach of programming or programmers to influence.

In the second category is an odd interaction between key remapping and keystroke macros. On my keyboard, even the shift keys can be remapped. Nothing prevents the user from deciding that the “L” key will become the “CapsLock” key. The user can also assign different macros to the shifted and unshifted state of any key, so <Ctrl><PgUp> can generate a different character string than <Alt><PgUp>. Pretty keen, huh? Well, except I can’t both remap shift keys and still use them to condition different macros from the same key. If I remap <Ctrl> and <Alt> so that they are on the keys above and below the left-hand <Shift> key, as many of us writers prefer, I can’t have different macros on <Ctrl><PgUp> and <PgUp>. By analyzing what happens when you try to do this I concluded that it was just a matter of programming some tests in one order rather than another. Too bad. The programmers unnecessarily limited the generality of their user interface.

To illustrate the third category, consider the diagram below, showing the keys and LED indicators in the upper right of my keyboard. Here we have the typical abomination that comes about when nobody actually examines carefully what the physical interlace looks like. The PROGRAM indicator is, logically enough, above the <ProgrmMacro> key. Except this indicator is on if the keyboard is operating with macros active and off if not active. It is turned on and off by the <SuspndMacro> key to the right below it! Then there is the CAPS LOCK indicator above the <ScrollLock> key, whose indicator is to the right above the <Pause> key. The NUM LOCK indicator is farthest from the <NumLock> key, over on the side toward the main keyboard where the <CapsLock> key is to be found. Alas, these LEDs are not under user control, but they are under control of the embedded program. The principle here is the same as the one governing the placement of light switches on switch panels for rooms with multiple lights or for multiple rooms. The switch closest to the light should control it. With this keyboard arrangement, to know what is happening, the user cannot just rely on the position of the indicator, but must actually read the legend above it. And it requires “double-think” to remember that you turn PROGRAM mode (as indicated by the LED) back on using the <SuspndMacro> key, not the key under the light.

Bells and Whistles

In some cases an improved interface can be achieved with only relatively minor software changes. As a frequent international traveler, I appreciated the gift of a pocket international time zone/monetary exchange rate calculator. In addition to an enhanced calculator keypad, this little device has this a nice user-friendly interface in the form of a world map. Associated with the map are small pads with currency symbols and the names of cities. Touch one and you get the time for that city or the previously entered currency exchange rate, depending upon what mode the unit is in. Alas, not all the time zones are represented by one of the 17 cities presented and there is no provision for entering a time zone other than those preprogrammed. So you do the extra adding or subtracting in your head, no great mental challenge, but a silly annoyance that seems to defeat the purpose of the calculator. And it certainly would be no programming challenge to provide such a feature.

Embedded systems programmers should be on the alert for such potential enhancements to a user interface, especially where expanded capability can be provided through a simple, generalized facility. On the other hand, some things are so easy to do in code that developers are tempted to throw them in without being asked. This can lead to the kind of “creeping featurism” that plagues so many spreadsheets, word processors, and graphics packages. Like the baby carriage that is also a high chair and a stroller and a bassinet, a system that does too many things can actually be harder to use.
User-Centered Design

Because it is the same species of critters using the systems, the same methods of human interface design apply to embedded systems software as to any other software (Constantine, 1990; 1991). To get substantially more usable user interfaces requires putting users at the center of the design process rather than treating them as secondary, at best an annoyance and at worst a potentially real obstacle to “just getting the darned programming done.” Appropriately, an approach that makes the user the heart of the process is called “User-Centered Design” (Norman, 1988). This is more than just being “user-oriented”; it makes the end user the focus that informs the entire design process. User-centered design is not about “user friendliness” either, or a matter of the aesthetics of the interface, but about making systems that are substantially easier for users to make use of—simply, quickly, and reliably—that make it easier to do things well and right than poorly and wrong.

User-centered design rests on the simple but profound idea that, to the user, “the system” is what the user actually sees—the visible system—and how the user thinks about this visible system. If the visible system is not “right” for the user it does not matter how clever or well-structured the code may be or how carefully thought out the algorithms are. To the user, the user interface is the system; it represents the system to the world of users and is not merely some peripheral aspect of system design.

Fortunately for the embedded systems developer who may have only limited control over the shape and appearance of the user interface, good user interface design is more about how the interface works than how it looks. Gimmicks and style may appeal in the short run, but real usability wins in the end. Well established methods for user interface design can be applied as successfully to embedded systems software as to any other kind of software.

RULES AND PRINCIPLES

In general there are several paths to better user interfaces: rules, principles, reviews, and research. Rules and principles are easy to learn and apply, but the greatest payoffs may be from careful study of user interfaces through systematic interface reviews and through ongoing usability studies.

Rules provide canned answers and set solutions to standard problems. Ideally these are based on a large base of experience or accumulated research evidence. For example, an application may require a comparison of two things presented to the user. We know that quicker and more reliable comparisons are made when information is presented side-by-side or over-and-under than when its is offset both vertically and horizontally. For another example, a large number of studies have demonstrated that the most rapid and reliable comprehension results when the user is presented with both pictorial/symbolic and textual/verbal information together. Thus, symbols should always be accompanied by labels or explanatory annotations. If interfaces must cross language barriers, then software could be table driven to supply the local language text to accompany symbols or icons.

On General Principles

Better user interfaces can be designed if you keep in mind some broad principles about human-machine interaction. Here I will hit on a few of the more important ones as they apply to user interfaces for embedded system applications.

Visibility

The knowledge needed to use a system at any given time should be visible to the user. That means that the needed information is on the system itself, not filed away in a thick manual. This is especially important in carrying out complex sequences of activities. The system should guide the user through the process in an obvious way, whether through menus, sequenced lights, or just a series of arrows printed on a control panel.

My microwave manages to walk me through defrosting frozen chicken parts with nothing more than a beeper and a five-character display. After it beeps long and loud for my attention, it tells me to

```
RE-ARRNG
```

until I open the door and “re-arrng” things. Once I close the door again, it reminds me to

```
PRESS
```

until I do.

Feedback

The system should always give feedback to the user about important information and events, including changes in internal state, the progress (or lack thereof) of any activity, and the disposition of any input from the user. If the internal state of the system affects how inputs are processed or what actions are possible or in any other way can be of interest to the user, the system should make this visible. Significant changes in internal state should be reflected externally in ways that make the state visible or apparent to the user. When a programmed telephone is in program rather than dial mode, this should be clear to the user through
User-Centered Design

Because it is the same species of critters using the systems, the same methods of human interface design apply to embedded systems software as to any other software (Constantine, 1990; 1991). To get substantially more usable user interfaces requires putting users at the center of the design process rather than treating them as secondary, at best an annoyance and at worst a potentially real obstacle to “just getting the darned programming done.” Appropriately, an approach that makes the user the heart of the process is called “User-Centered Design” (Norman, 1988). This is more than just being “user-oriented”; it makes the end user the focus that informs the entire design process. User-centered design is not about “user friendliness” either, or a matter of the aesthetics of the interface, but about making systems that are substantially easier for users to make use of—simply, quickly, and reliably—that make it easier to do things well and right than poorly and wrong.

User-centered design rests on the simple but profound idea that, to the user, “the system” is what the user actually sees—the visible system—and how the user thinks about this visible system. If the visible system is not “right” for the user it does not matter how clever or well-structured the code may be or how carefully thought out the algorithms are. To the user, the user interface is the system; it represents the system to the world of users and is not merely some peripheral aspect of system design.

Fortunately for the embedded systems developer who may have only limited control over the shape and appearance of the user interface, good user interface design is more about how the interface works than how it looks. Gimmicks and style may appeal in the short run, but real usability wins in the end. Well established methods for user interface design can be applied as successfully to embedded systems software as to any other kind of software.

RULES AND PRINCIPLES

In general there are several paths to better user interfaces: rules, principles, reviews, and research. Rules and principles are easy to learn and apply, but the greatest payoffs may be from careful study of user interfaces through systematic interface reviews and through ongoing usability studies.

Rules provide canned answers and set solutions to standard problems. Ideally these are based on a large base of experience or accumulated research evidence. For example, an application may require a comparison of two things presented to the user. We know that quicker and more reliable comparisons are made when information is presented side-by-side or over-and-under than when its is offset both vertically and horizontally. For another example, a large number of studies have demonstrated that the most rapid and reliable comprehension results when the user is presented with both pictorial/symbolic and textual/verbal information together. Thus, symbols should always be accompanied by labels or explanatory annotations. If interfaces must cross language barriers, then software could be table driven to supply the local language text to accompany symbols or icons.

On General Principles

Better user interfaces can be designed if you keep in mind some broad principles about human-machine interaction. Here I will hit on a few of the more important ones as they apply to user interfaces for embedded system applications.

Visibility

The knowledge needed to use a system at any given time should be visible to the user. That means that the needed information is on the system itself, not filed away in a thick manual. This is especially important in carrying out complex sequences of activities. The system should guide the user through the process in an obvious way, whether through menus, sequenced lights, or just a series of arrows printed on a control panel.

My microwave manages to walk me through defrosting frozen chicken parts with nothing more than a beeper and a five-character display. After it beeps long and loud for my attention, it tells me to

\[
\text{RE-} \\
\text{ARRNG} \\
\text{RE-} \\
\text{ARRNG}
\]

until I open the door and “re-arrng” things. Once I close the door again, it reminds me to

\[
\text{PRESS} \\
\text{START} \\
\text{PRESS} \\
\text{START}
\]

until I do.

Feedback

The system should always give feedback to the user about important information and events, including changes in internal state, the progress (or lack thereof) of any activity, and the disposition of any input from the user. If the internal state of the system affects how inputs are processed or what actions are possible or in any other way can be of interest to the user, the system should make this visible. Significant changes in internal state should be reflected externally in ways that make the state visible or apparent to the user. When a programmed telephone is in program rather than dial mode, this should be clear to the user through
indicator lights or audible changes. The watchword here is:

*Keep the user informed!*

Keeping the user informed means that signals or messages to the user must be clear and clearly distinguishable. Sometimes the output devices for embedded systems applications are somewhat simplified. There might be only a set of LEDs or a small LCD panel, but even such simple devices can be used remarkably effectively to communicate a range of information.

It is important to distinguish different messages to the user clearly and in a way that naturally varies combinations for different conditions, depending on what is going on or going right or going wrong with the printing. Unfortunately, they blink at the same, somewhat sleepy rate no matter what is happening. It is not always easy to tell the difference between flashing that means all is going well and flashing that means everything is a mess. Even reading the legends next to the lights is not terribly informative, since lights flashing in combination can change the meaning. To add to the clumsiness of this interface, the Postscript cartridge I have plugged into the printer uses the three lights (and the eight membrane switches) in completely different ways largely unrelated to the captions printed on the control panel.

The programmers of the printer code and the cartridge code did not think through very carefully just what they wanted to communicate with users and how they could best go about it. An error condition ought to grab the user’s attention. The lazy, half on, half off, once-a-second blink just doesn’t do it. It is precisely because such a blink is not very distracting that cursors tend to work that way. Instead, a misfeed or out-of-paper condition ought to be signaled by a flash of about 4-6 per second with a shorter on than off cycle.

Users also want reassurance when all is right with the world. How do I know the printer is actually getting information as opposed to sitting there mulling over the high price of toner cartridges? What I’d really like is to be able to see the data actually flowing along the cable into the printer, like water through a hose. For this, a rapid fluttering of the “ready” light, reminiscent of the flickering TXD/RXD light on my first Hayes modem, would naturally suggest that data was pouring through at a decent clip.

Even when systems are equipped with very limited or even primitive output facilities, often more can be done than the typical embedded system programmer thinks about. For example, a hand-held laboratory instrument may be equipped with a gated sound generator that emits a fixed-frequency beep. Nevertheless, a clever programmer can create several very distinctive and useful outputs quite simply. In addition to the standard beep of 1/4 to 1 second, one can get a click (very short pulse), a continuous tone, or even a warbling tweet, like a police whistle. The least abrasive of these (such as the click) can be used to signal normal conditions, such as accepted input, while the slightly stronger standard beep could be used to signal simple input error, reserving the most annoying, such as the warbling whistle, for high priority attention getting.

**REVIEWS AND RESEARCH**

Perhaps the most effective way to get better user interfaces is to take a critical, investigative approach, studying the user interface, studying users, and studying users using the user interface. Usability studies that take a close look of users in regular working contexts trying to solve actual problems with a system can yield the most important kind of information: information about what it is that you don’t know you don’t know.

The acid test is to sit someone down with a prototype, mockup, or simulation, and tell them what the system is supposed to do. Then turn them loose without a manual or further instructions. Of course, we are here interested in the user who has relevant experience with the application, but not necessarily with this or comparable systems. If it’s going to be a blood gas analyzer, you want to have a medical technician; if the system is a computerized router for shippers, you probably want a truck driver. To get the most from these studies, you should video tape them, then review the tape with the user, keeping in mind that any problem the user has, however minor, is not a symptom of their stupidity or a sign that they didn’t pay attention, but a problem in the user interface that ought to be examined more closely. Every mistake or misstep they make indicates a probable design flaw.

Even if you can’t get access to real end users to play with systems or prototypes, there are things you can do that go a long way in the right direction. Consultant Lucy Lockwood likes to use junior staff members to test out user interfaces. Relative newcomers can often find problems that are invisible to the old timers and regular developers who have lived with the system since the beginning and who have their accepted and unquestioned ways of doing things. Often the neophytes or new hires have just the right mix of ignorance and a need to prove themselves so that they ask the right question. Basically, you just hand them the new toy and tell them to figure out how to use it, then figure out how to misuse it, then figure out how to break it. Have them keep notes or repeatedly debrief them along the way. You will be amazed at what you will learn.

Another trick is to learn how to think like a user, especially a naive user. Programmers are smart people and they know a lot, especially about how programs work and how programmers think. To think like a user you need to set this familiarity aside and approach the system afresh. It helps to take a break and be away from it for awhile, so this training and expertise must be set aside temporarily to successfully play the role of the naive user. It also helps to be critical. Get yourself into a foul and hypercritical mood, then pick up the gizmo or mockup and see how many things you can find wrong with it. Even if it is your own work,
Keep the user informed!

Keeping the user informed means that signals or messages to the user must be clear and clearly distinguishable. Sometimes the output devices for embedded systems applications are somewhat simplified. There might be only a set of LEDs or a small LCD panel, but even such simple devices can be used remarkably effectively to communicate a range of information.

It is important to distinguish different messages to the user clearly and in a way that naturally communicates. My Hewlett-Packard laser printer has three lights on it that flash in various combinations for different conditions, depending on what is going on or going right or going wrong with the printing. Unfortunately, they are the same, somewhat sleepy rate no matter what is happening. It is not always easy to tell the difference between flashing that means all is going well and flashing that means everything is a mess. Even reading the legends next to the lights is not terribly informative, since lights flashing in combination can change the meaning. To add to the clumsiness of this interface, the Postscript cartridge I have plugged into the printer uses the three lights (and the eight membrane switches) in completely different ways largely unrelated to the captions printed on the control panel.

The programmers of the printer code and the cartridge code did not think through very carefully just what they wanted to communicate with users and how they could best go about it. An error condition ought to grab the user’s attention. The lazy, half on, half off, once-a-second blink just doesn’t do it. It is precisely because such a blink is not very distracting that cursors tend to work that way. Instead, a misfeed or out-of-paper condition ought to be signaled by a flash of about 4-6 per second with a shorter than off cycle.

Users also want reassurance when all is right with the world. How do I know the printer is actually getting information as opposed to sitting there nulling over the high price of toner cartridges? What I’d really like is to be able to see the data actually flowing along the cable into the printer, like water through a hose. For this, a rapid fluttering of the “ready” light, reminiscent of the flickering TXD/RXD light on my first Hayes modem, would naturally suggest that data was pouring through at a decent clip.

Even when systems are equipped with very limited or even primitive output facilities, often more can be done than the typical embedded system programmer thinks about. For example, a hand-held laboratory instrument may be equipped with a gated sound generator that emits only a fixed-frequency beep. Nevertheless, a clever programmer can create several very distinctive and useful outputs quite simply. In addition to the standard beep of 1/4 to 1 second, one can get a click (very short pulse), a continuous tone, or even a warbling tweet, like a police whistle. The least obtrusive of these (such as the click) can be used to signal normal conditions, such as accepted input, while the slightly stronger standard beep could be used to signal simple input error, reserving the most annoying, such as the warbling whistle, for high priority attention getting.

REVIIEWS AND RESEARCH

Perhaps the most effective way to get better user interfaces is to take a critical, investigative approach, studying the user interface, studying users, and studying users using the user interface. Usability studies that take a close look of users in regular working contexts trying to solve actual problems with a system can yield the most important kind of information: information about what it is that you don’t know you don’t know.

The acid test is to sit someone down with a prototype, mockup, or simulation, and tell them what the system is supposed to do. Then turn them loose without a manual or further instructions. Of course, we are here interested in the user who has relevant experience with the application, but not necessarily with this or comparable systems. If it’s going to be a blood gas analyzer, you want to have a medical technician; if the system is a computerized router for shippers, you probably want a truck driver. To get the most from these studies, you should video tape them, then review the tape with the user, keeping in mind that any problem the user has, however minor, is not a symptom of their stupidity or a sign that they didn’t pay attention, but a problem in the user interface that ought to be examined more closely. Every mistake or misstep they make indicates a probable design flaw.

Even if you can’t get access to real end users to play with systems or prototypes, there are things you can do that go a long way in the right direction. Consultant Lucy Lockwood likes to use junior staff members to test out user interfaces. Relative newcomers can often find problems that are invisible to the old timers and regular developers who have lived with the system since the beginning and who have their accepted and unquestioned ways of doing things. Often the neophytes or new hires have just the right mix of ignorance and a need to prove themselves so that they ask the right question. Basically, you just hand them the new toy and tell them to figure out how to use it, then figure out how to misuse it, then figure out how to break it. Have them keep notes or repeatedly deconstruct along the way. You will be amazed at what you will learn.

Another trick is to learn how to think like a user, especially a naive user. Programmers are smart people and they know a lot, especially about how programs work and how programmers think. To think like a user you need to set this familiarity aside and approach the system afresh. It helps to take a break and be away from it for awhile, so when you return there is a better chance of spotting the mistakes. Writers and programmers often use this trick to make it easier to edit their own work.

When, on occasion, I have to diagnose and repair hardware, whether it’s a VCR or a balky lawn mower, I try to think like the engineer who designed it. Why would there be a screw in this particular place? If this spring holds that wheel, then there must be something holding the other one. To think like a user, you have to do just the opposite. All this training and expertise must be set aside temporarily to successfully play the role of the naive user. It also helps to be critical. Get yourself into a foul and hypercritical mood, then pick up the gizmo or mockup and see how many things you can find wrong with it. Even if it is your own work,
it is essential not to defend or explain the design.

It may be harder to prototype embedded system interfaces than standard applications for the latest GUI. If you don’t have a working prototype or a serviceable mockup, you may be able to create a serviceable simulation in software. Windows or another GUI on a PC or workstation with a touch screen can “become” almost anything you want it to be and, with the help of a little imagination, can do a fairly good job of simulating most interfaces. If nothing else, imagine yourself as the user and walk yourself through a complete scenario in your mind.

If a prototype or mockup is available, developers should try to use it for actual work, to solve real problems not just briefly, but repeatedly over an extended period of time. Don’t just measure and record one signal, use the thing to measure and record the signals at every accessible point on some circuit board. Don’t just make the robot arm move from here to there. Spend a day programming and reprogramming it to reorganize your desk. If you don’t have a prototype or simulation of the system you are developing, use a competitor’s product or an earlier version of the same line to study. Along the way you are likely to learn some of the important things you do not want to do on the new version.

The most important thing to keep in mind is to actually look at the interface. Even if you are “just a programmer,” insist on seeing a mockup or pictures of the actual hardware interface. Don’t be content to be told to “read this signal and send an on-code to that indicator.” Know what these mean and how they are arranged in relation to each other.

If you keep your eyes open and your mind on basic principles of user interface design, you can help design better user interfaces for embedded systems.

REFERENCES


