

CONTROLS & SENSORS



RUNNING RELIABLY

*The GUI should
be as dependable
as the appliance.*

by **ken klask**

*Ken Klask is CEO,
Amulet Technologies, Santa Clara, Calif.*

Flat panel displays aren't just for computers anymore. They are now becoming commonplace in many other products, such as automobiles, cell phones, digital cameras, portable digital music players, and even major household appliances. Just as the automotive industry discovered in the early 1990's, appliance manufacturers are now discovering that electronics and industrial design are the keys to providing value to consumers, and in turn, differentiating their products. They are also discovering that a common intersecting point where design and electronics meet is at the flat panel graphic user interface. A well-designed GUI provides an intimate, stylish and easy-to-use conduit to the underlying device functionality. This makes the product appear more upscale and sophisticated, yet more convenient and easier to use.

However, household appliances have some unique characteristics that can be challenging for GUI integration. First, they are typically never turned off and are required to run reliably 24 hours a day, 7 days a week, for years on end. Second, they typically employ small to medium-sized microcontrollers that don't have the resources to control a GUI and the device at the same time. Third, the look and feel of the product is developed by industrial designers instead of software programmers.

Reliability concern

Most of the reliability concerns about GUIs stem from bad experiences with personal computers. However, many of those issues arise from a design requirement that is absent in most household appliances. The general-purpose nature of PCs require them to be able to run random combinations of differing application programs that are installed after the system software has been written. For this reason, memory and other computing resources cannot be determined at the time the system software is built. This unique requirement forces PC operating systems to employ dynamic memory management and other run-time techniques for sharing resources.

As a result, memory leaks and fragmentation (which, in practice, are very difficult to avoid in systems that employ dynamic memory allocation) will eventually bring the system to an out-of-memory condition, and the system will crash. With PCs, such crashes can usually be averted for a long enough period of time by simply adding more memory. Unfortunately, there is no amount of memory that could be added to prevent out-of-memory crashes when a system is left to run continuously for years at a time.

CONTROLS & SENSORS

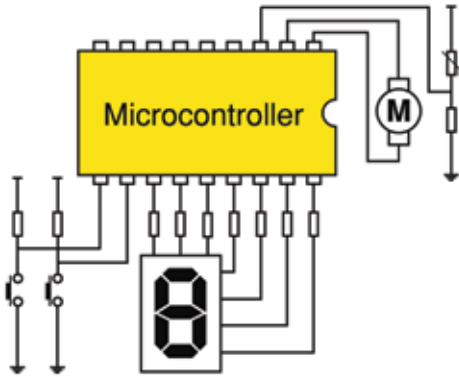


Fig. 1. System with a simple user interface.

Since many major appliances are never really turned off, and are expected to run 24/7 for years and years without any problems, dynamic memory allocation should be avoided. Since most, if not all, GUI libraries based on C++ require dynamic memory allocation, these libraries should be avoided for such applications. The GUI example here is built with tools from Amulet Technologies. Amulet employs a GUI compiler that converts HTML authored screens to statically allocated and initialized GUI resources for reliable “leak free” operation. GUI systems that employ similar static allocation are a good fit for appliances.

Simplicity’s appeal

From a software engineering standpoint, systems without a GUI are smaller and easier to develop and test. As shown in Fig. 1, a single MCU is connected directly to both the user interface hardware and the controlled system. The user interface hardware consists of two buttons and a 7-segment LED. As a result, all that is required of the MCU is a few digital I/O ports, and the software to monitor the inputs and bang out the bits to turn a handful of LED segments on and off. The software required for this type of user interface is typically very small and it places a very small burden on the MCU. From an engineering view, the benefits of this simple interface over a more complex GUI interface are obvious. It has lower component costs, and it is easier to develop, easier to test and probably more reliable.

The catch is that although that type of interface is simple to implement, it has about as much visual appeal as a 1970s digital alarm clock, and is about as easy to use as the front panel of a VCR. Furthermore, the software only stays simple if user interaction is kept to a minimum. Start to add any advanced personalization or functionality to the underlying device electronics, and the

user interface starts to grow in the number of displays and controls. One eventually reaches a point of diminishing returns, and the front panel becomes very cluttered, difficult to code, difficult to use and, worse yet, unattractive and impersonal to the end user. This is the time to consider adding a GUI. But, how does one retain the simplicity of the Fig. 1 implementation with its easy-to-develop and easy-to-test attributes?

Avoiding complexity

The answer lies in the golden rule of software engineering, which is to keep things small, and break big problems into little ones. Taking a lesson from the automotive industry, partitioning a design into multiple, small processors can make the software development and testing tasks more manageable and less prone to errors. This is the design strategy adopted by Amulet Technologies in its GUI coprocessors.

While keeping the reliable and field proven real-time control algorithms in the MCU, the system shown in Fig. 2 now places the user-interface control in a separate GUI coprocessor. This approach effectively splits the load, leaving the reliable legacy 8-bit MCU in place, fully leveraged, and virtually unmodified. In other words, the GUI functionality gets “bolted on” to existing code. Modifications to legacy code are limited to a very-low-bandwidth bi-directional serial interface with the Amulet MCU.

As far as the embedded controller is concerned, the GUI can still be seen, conceptually, as a very large front panel full of user controls and displays. In reality, the displays are not all visible at the same time, nor are all the controls accessible at the same time. But, the software interface to the user-interface components is still very similar in that they can be accessed via a virtual memory-mapped interface.

In Fig. 2, the block labeled Shared Memory is a virtual, dual-port RAM. In the Amulet GUI coprocessor, the shared memory consists of 256 different byte variables, 256 different word (16-bit) variables, and 199 different 18-character null-terminated string variables. The on-screen user interface components can read or write to these Internal RAM variables. So, for example, if a variable is to be displayed on the GUI, all the embedded MCU needs to do is

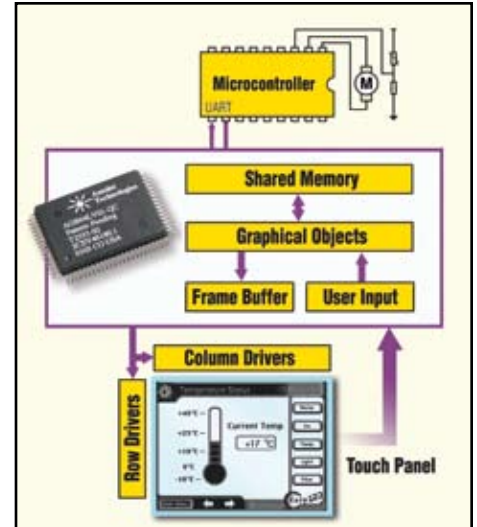


Fig. 2. Amulet system architecture.

write the data value to the appropriately mapped internal RAM variable. The GUI coprocessor figures out how and when to visually present the data, and it takes care of doing it without requiring any interaction from the embedded MCU. In fact, the embedded MCU wouldn’t even have to know whether the current UI screen needs to display the data or not. Likewise, a visual touch-sensitive control could be used to manipulate a unique internal RAM variable. The embedded MCU could read that RAM location at a regular rate, just as it would read a memory-mapped A/D converter value, and adjust the operation of the appliance accordingly.

To read and write these memory-mapped variables, the microcontroller can send a simple serial message to the GUI. This means that the microcontroller is not required to be the GUI coprocessor’s slave. Optionally, the GUI coprocessor could initiate the communications to inform the MCU of any events requiring immediate action or that certain variables have changed.



Fig. 3. Visual screen prototype.

Beyond Your Basic LCD Controller.



**Visually Appealing.
Simply Reliable.**

Amulet Technologies™ Graphical User Interface (GUI) solutions empower your designers to utilize your LCD's full design potential. Chips and software from Amulet are optimized for 24/7 reliability and are compatible with most microcontrollers, LCD's and touch panels.

Call Amulet today at (408) 244-0363 to discuss your GUI needs.



© 2005 Amulet Technologies, LLC.
Amulet Technologies is a trademark of Amulet Technologies, LLC.
US Patents Pending, European Patents Granted

CONTROLS & SENSORS

First step

Shown in *Fig. 3* is an example of a screen prototype for a portable digital music player that will be used to illustrate the steps of building a GUI. The screen prototype represents the first phase in GUI design — determining the visual appearance by prototyping and user-testing the screens. This exercise may be considered more of an artistic and marketing driven endeavor, but it is typically fraught with many technical challenges. The problem stems from the fact that not all LCD displays are created equal, and, as a result, the images prototyped on a computer's LCD screen will rarely look like the images that will be seen on the production LCD — even if viewed under the same lighting conditions and viewing angles experienced with the final product. So, it is very important that the screen prototypes be proofed on the actual LCD that will be used in the final product.

Seeing the screen art on the actual display hardware gives the screen artist a very realistic sense of the capabilities/limitations of the artistic medium. Only with this level of understanding will the artist be able to compensate for the medium and be able to deliver screen art that will be true to the final product implementation. Skipping this step almost always ensures that the software engineer will be given screen art that will have to be tweaked to work with the LCD. Unless the software engineer is also a gifted artist, this usually means that the resultant GUI will not have the look and feel envisioned by the original screen designer.

Furthermore, to ensure that the correct LCD is chosen for the given application, it usually makes sense to have the screen prototyping and LCD selection tasks as parallel efforts. Doing so simplifies the evaluation of LCD contrast, viewing angle, pixel pitch, column bleeding, and color depth contouring issues in actual user environments and with actual user screen shots. When evaluating displays and prototype screen art, the evaluation should be done under the same lighting conditions that the customers will have in the field. Finally, the LCD should be mounted at the same height as in the final product so that the screens can be evaluated at the actual viewing angle.

Slicing phase

Once the look of the GUI has been determined, the next step is to create the feel of the product by transforming the static screen prototypes into interactive user interfaces. This is done by slicing up the static screen shots into individual graphic elements based on their function in the UI. Functionally, these graphic elements will fall into one of the following three categories:

- ▶ Decorative elements.
- ▶ User controls.
- ▶ Status or data displays.

As seen in *Fig. 4*, our example GUI has been sliced up into a background image and eight interactive components labeled A-H. The background image is purely a decorative element and forms a backdrop for the other elements. Since Amulet employs an HTML screen layout methodology, the HTML DIV tag is used to place the background image in a layer behind the other elements. The other elements are also positioned with DIV tags, but they are each positioned above the background layer and located with the Absolute Position attribute of the DIV tag.

The interactive screen elements get their interactive behavior from a library of GUI components that Amulet calls widgets. Each Amulet widget is placed on the screen using the HTML APPLET tag. The visual aspect of each widget comes from the bitmap components that are referenced via the APPLET tag, while the interactive element is inherent to the specific widget. Listed below are the Amulet widgets that make up each of the screen elements.

- ▶ **Animation:** Shows that the music is playing by cycling through multiple frames at prescribed time intervals. The animation is paused when the music is not playing. The animation images and times are specified using an animated GIF file.
- ▶ **Numeric Field:** Shows play time in minutes and seconds. Font chosen has the look of a 7-segment LED.
- ▶ **Radio Buttons:** The Play/Pause/Forward/Reverse buttons are grouped together into a group of radio buttons because only one button in the group can be active at any time.

CONTROLS & SENSORS

- ▶ **String Field:** Shows the name of the current play list and song being played. Font chosen is a proportionally spaced sans serif font with white text on black background.
- ▶ **Group of String Fields:** The list of selectable song choices are displayed in a group of string fields. Like the string field above, all of the strings in this group of fields are displayed using a proportionally spaced sans serif font. However, most of the strings are displayed as black on white. As the user scrolls through the list, the color is inverted to white on black to indicate its selection status.
- ▶ **Image Sequence:** The volume of play is displayed as an image that is indexed from an array of images. This is similar to an animation object, but instead of sequencing with time, the frame is chosen as a function of the value of a parameter.
- ▶ **Image Bar:** The battery level is presented with a bar graph that sweeps from left to right. The bar graph is made from two images — an Empty image and a Full image. A portion of each image is drawn as a function of the value of a parameter.
- ▶ **Slider:** The slider is a control that is made up of two images: the slide control knob and the background of the slider. As the user manipulates the knob, a variable is changed. That variable is used as the song index.

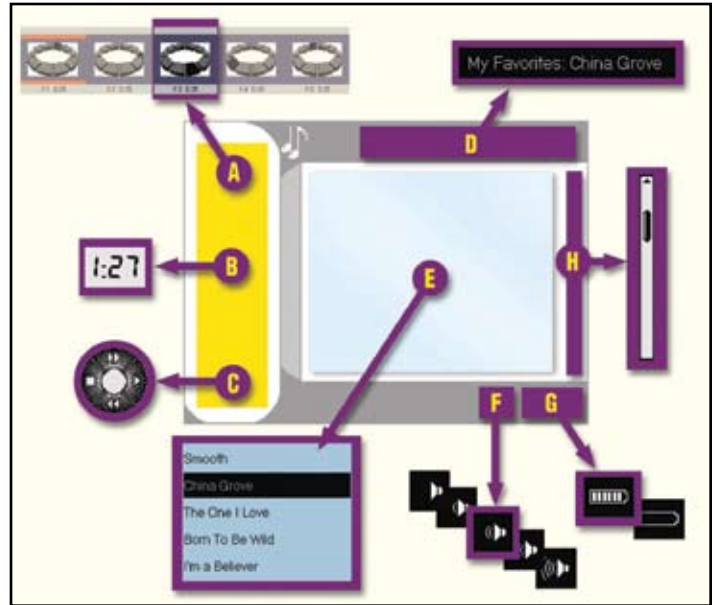


Fig. 4. Map of visually interactive components.

Adding interaction

Since the GUI is the bridge between the user and the machine, the final phase is to add the interaction between the GUI and the underlying electronic control system. This is done by memory mapping the variables referenced by each on-screen widget to entries in the shared memory arrays. Read-and-write access to these variables is then made possible via a simple serial protocol. To illustrate the serial protocol and the mapping of the shared memory interface, Fig. 5 shows how a serial stream is used to update the playtime numeric field in our example GUI.

The column of numbers to the left of the screen represents an array of byte variables in the GUI coprocessor's shared memory, while the column to the right shows an array of shared string variables. The table at the top of the figure shows a string of serial bytes being sent from the UART of the embedded MCU to the UART of the GUI coprocessor.

The structure of the serial message dictates that the first byte of the message is an opcode that defines the type of operation to be performed. In this example, Byte 0 has the value of (0xD5) which is the opcode for a byte write

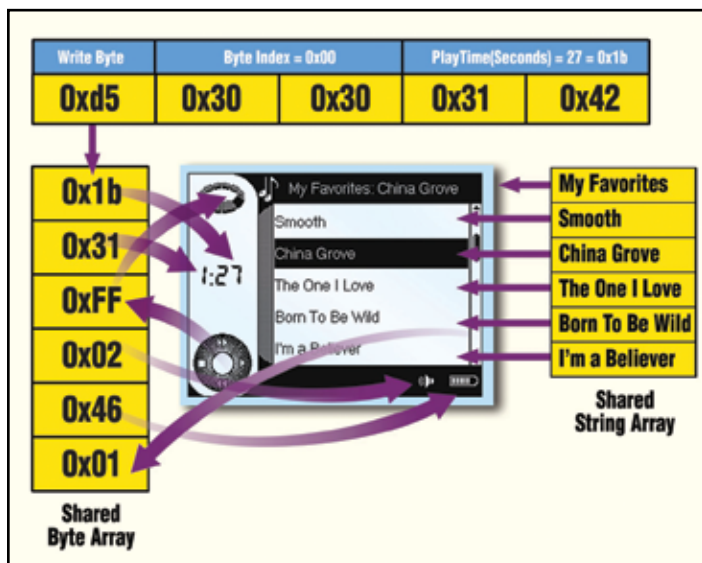


Fig. 5. Serial interface between GUI and microcontroller.

operation. Bytes 1 and 2 identify, in ASCII, which byte to read or write. In this example, (0x30 and 0x30) correspond to ASCII 0 and ASCII 0 for a byte index of 0x00. The remaining bytes form the payload of the message. In this byte write message, the data value of 0x1b is the value that is to be written to Internal RAM byte location 0x00. This data value represents the Seconds portion of the total play time. The value of 0x1b (27 decimal) is the actual seconds.

As can be seen on the sample screen shot of the LCD, there are actually multiple UI objects that either control, or are controlled, by the shared memory arrays. To summarize, the first two entries in the byte array control the play time in seconds and minutes, respectively. The next entry is the play status, and it is manipulated via the on screen play/pause button cluster. The play status variable is also monitored by the play-status animation to pause and play the screen animation. The next byte represents play volume and it is monitored by the volume image sequence. The next byte represents the battery life as is displayed by the battery image bar. The final value in the byte array is the song index. The song index is manipulated by the slider control.

The string array is also to be updated with ASCII strings that represent: the current play list and the songs in that play list. The song index byte is to be monitored and used by the music library manager to determine which strings write to the string array.

Simply sophisticated

Albert Einstein once said, "Things should be made as simple as possible, but not any simpler." The primary design goal of this GUI architecture was to make it as simple as possible for the designers and engineers, but not so simple as to compromise the usability and appeal for the end consumer. Using the Amulet approach, appliance designers can now build sophisticated, yet easy-to-use appliances without fear of compromising the rock-solid reliability that they have worked so hard to achieve. ■