Atmel AVR 2004 Design Contest Entry #A3754

AVR COG
Atmel AVR Contest Entry #A3754

Overview:

A home automation and security system needs a way to keep the occupants informed about what's happening around the home. Like many households, mine has several TV's, at least one of which is usually on whenever someone is awake in the house. It makes sense to use them as a display device for the control system.

There are devices available off the shelf that will overlay text onto a video signal, but they are all more expensive than my budget would allow to equip all my TV's and video monitors with one. So I built the AVR Character Overlay Generator, or COG, using the AT90S2313 and a handful of other components as one of the important parts of my system. It is small, simple, versatile, and best of all, inexpensive enough that I can have as many as I need without breaking my budget.

What it's for:

The main focus of the COG is to provide a status display on a TV or security monitor. It's designed to go between a video source (tuner, DVD player, camera, etc.) and a TV or monitor and put status updates onscreen under control of a remote host. It has other uses, especially for robotics or remote monitoring, which I will discuss later. But first, I will describe what the COG does and how it does it.

What it does:

The COG hooks in line between an NTSC video source and display device. It locks itself to the NTSC video and displays text received over a 9600 bps RS-232 link. The text is monochrome and consists of six rows of twenty columns each displayed in the lower half of the screen. The COG works much like the display side of a dumb terminal, recognizing most normal ASCII control codes such as carriage returns and line feeds, as well as a few other codes to control the display. It has 128 displayable characters and can accept data as fast as the serial port can send it. The display can be cleared of all text by a control code, or blanked by a front panel button. If the display is blanked, it will become visible again whenever another character is received from the serial port. There is also a 4 bit output port with a strobe line. This port is intended to select alternate video sources, such as a security camera, under control of the system. However, the port is not dedicated to that and can be used for any other purpose. The input of the serial port is designed to allow up to 16 COGs to be driven from the same port, displaying the same data on every attached device.

How it works:

Hardware:

The COG's hardware is the simple part. Most of the complexity is in the software. The features of the AT90S2313, especially the analog comparator, fast execution speed, and clean RISC architecture, allow the AVR to work standalone. The only other active devices used are a 5 volt regulator and an NPN transistor for RS-232 level translation and inversion.

The block diagram shows the major functional blocks of the COG. The power supply is a simple bridge rectifier and 5 Volt regulator. The RS-232 input consists of an npn transistor to invert and translate the voltage from the serial port for the 2313. The digital output port consists

of 5 bits from the 2313 to interface to external circuitry. The video output section is one diode and one resistor to isolate the AVR's output from the video signal and limit the brightness of the displayed video. The heart of the circuit is the bias and clamp block and the AT90S2313. The video I/O block consists solely of the two video jacks with a through connection and a connection to the rest of the circuit.

The schematic shows how simple the circuitry is. Almost all the work is done by the AVR. Most of the other components are only to signals back and forth between the NTSC levels and the 5 volt logic levels.

The RS-232 input, consisting of Q1, D4, R7, and R8, translates the +- 3 to 12 V signal to a 0-5V level and inverts it for the AVR's UART. It has a high impedence input to allow multiple COG's to be driven from one port. It is connected to a DB9F connector to interface with a PC serial port. The inverted signal then goes to the AVR UART input at pin 2.

J3 and J4 are the video input and output. The video signal passes through from one to the other so they are interchangeable. A connection from the junction of those two connectors is made to the input circuit. The signal is AC coupled through C3 into a clamp circuit consisting of R1 and D1. Since a standard NTSC video signal is only 1 volt p-p, the clamp circuit is biased by D2, C4, and R2 to approximately 0.6 V so that the bottom of the sync signal is very near ground level. This brings the entire video signal above ground, allowing the onboard analog comparator to detect the low-going sync pulses. The black level of the signal is typically 0.3V and the bottom of the sync pulse should be slightly above ground (about 100mV). Voltage divider R3 and R4 divide the reference voltage from D2 in half to provide a 0.3 V reference to the analog comparator's negative input. Whenever the clamped video signal drops below 0.3V, the comparator will generate an interrupt to the CPU, indicating a sync pulse. The software determines if it is a horizontal or vertical sync pulse and synchronizes the display to the signal. The video signal is completely generated inside the AVR and port B7 sends the digital video signal through D3 to isolate it and R5 to limit the brightness.

The COG is powered by an 8.5VAC wall wart that is rectified and filtered by BR1 (D4 to D8) and C1, then regulated to 5VDC by IC2, a 78L05. The COG circuitry itself only draws about 10 ma, but the supply is designed to provide 50 ma to allow circuitry to be added to the 4 bit output port, such as a latch or mulitplexer. The supply is filtered more than necessary to minimize ripple getting into the generated video signal.

The COG can accept serial input at the maximum rate, so no flow control is needed. The blanking input, SW1, is read every time a vertical sync pulse occurs. The switch is not debounced since it any bounce will only cause the blanking to turn on repeatedly which causes no harm. If the switch is low when read, the video output is turned off to blank the text. Reception of any character on the serial port turns it back on. I initially used the internal pullup on the pin, but found that noise on the wire to the switch caused the display to blank at random. The 10K pullup added externally solved the problem.

The last part of the circuit doesn't involve any extra hardware. It is simply five pins of the AVR that output the low four bits of character codes 10 through 1F hexadecimal. Port D2 is the LSB and Port D5 is the MSB. The fifth line, Port D6, is a high going strobe for 300 nanoseconds. As mentioned above, this port was intended to control a video multiplexer, but it can be used for any purpose, such as to turn on or off relays, etc. Since the pins of the AVR can sink up to 20 ma, each one could drive a solid state relay directly.

Software:

When the inspiration struck for this project, I expected the software to be the most difficult part. I wasn't disappointed. About 90% of the development time was spent on software. However, it

wasn't nearly as bad as I expected. The AVR is a joy to program. The well thought-out instruction set and large number of registers make a lot of problems much easier than on most processors I've used. The code isn't pretty, but it works. All the code was written in assembler, using AVR Studio. It was debugged mostly by crash and burn, since the sleep instruction, a critical part of the program, isn't simulated in AVR Studio.

The AT90S2313 has 1024 words of program memory (2048 bytes), 128 bytes of RAM, and 32 byte-wide registers. The EEPROM is not used in the COG. Half the program flash holds a 1024 byte character table (128 characters X 8 bytes). About 250 more words hold the program, which includes what amounts to a terminal emulator program. Of the 128 bytes of RAM, 120 are used for a display buffer and four are used as a ring buffer for the incoming serial characters. That leaves four bytes for the stack and anything else that might be needed. Since the video sync causes an interrupt which uses two bytes of stack space, I decided not to use subroutines at all. I tried to structure the code well and there are lots of comments to explain what's happening, but it is rather hard to follow.

The flowchart shows the overall structure of the program. As you can see, most of the work is done either in the comparator interrupt routine or as a result of it. There is another interrupt, the 8 bit timer 0. It acts something like a watchdog, allowing serial data to be received if there is no video source present.

Since the software is the heart of the project, I will describe it in some detail. You can refer to the code listing to follow along. As mentioned, there are no subroutines other than the two interrupt routines, so most of the code is straight line with some jumps thrown in. For speed and to limit stack usage, many registers are used to hold constants and variables. There are also 7 registers used as a working set for many purposes. They are labeled TEMP1 through TEMP7.

At reset, the code initializes the hardware and many of the constants. Most of that is straightforward, except the row increment pointer, RowIncL. This is a constant used to speed up the addition in the video generation section to read the proper row of bytes from the character table for each line of video. The video ram is cleared and then a signon message is loaded into it to let you know the COG is ready to accept data. Once everything is initializes, interrupts are enabled and the main loop is entered.

The main loop itself is quite simple. The processor goes to sleep and waits for an interrupt. The sleep is primarily to eliminate interrupt latency so that the video starts at the same time after each horizontal sync pulse. After the interrupt is processed and returns, the main routine checks to see if a character has been received on the serial port. If one has arrived, it is put into the ring buffer for later processing.

The next step is to process a character from the ring buffer. Since the timing is critical during video generation and some characters may require more than an entire video line time to process, a check is made before processing the character. If the current line is not at least two lines away from the active video lines, the character won't be processed.

If there is time before generating the video a character is retrieved from the ring buffer, if present, and checked to see what action should be taken. It is first checked to see if it is a control character (ASCII code below 32). If it is not then the high bit is masked off, resulting in a displayable code from 0 to 127. This allows the low 32 characters to be displayed by setting the high bit. Otherwise, they would be interpreted as control codes and not displayed. The displayable character is then put into the video ram and the screen updated as needed.

If the character is a control code, it is processed by what is effectively a large "switch" statement. First it is tested to fall between 16 and 31. If it does, then the low 4 bits are output to the output port and the strobe line is pulsed. If not, further testing is done. The rest of the tests for recognized codes are in numerical order. There is no reason other than programmer

convenience why this is so. It just seemed logical. Each code is tested in turn, and the appropriate action taken. The codes recognizes are listed below:

code	name	action
8	backspace	Moves backward one space on current line only. Does not erase.
9	tab	Moves to the next tab stop. Tabs are 4 spaces. Moves to next line at end.
10	line feed	Moves cursor to same position on next line.
11	vertical tab	Moves cursor up one line at same position. Will not scroll up.
12	form feed	Clears screen and moves cursor to home position.
13	carriage ret	Moves cursor to beginning of current line without erasing anything.

Any other codes are ignored. After each character is processed, the cursor is updated as appropriate. If the cursor moves past the last line, the display is scrolled one line to make a blank line at the bottom and the cursor is updated to that line.

As mentioned above, the serial port is checked after every interrupt. Since the cpu is put to sleep waiting on those interrupts and the UART doesn't use interrupts, then data from the serial port will be lost if there are no interrupts. This can occur if there is no active video source. Since we may still need to receive data while there is no active video source, timer 0 is loaded with a timeout value of about 80 uSeconds. Since a video line is about 63.5 uSec and the timer is reset at each interrupt, the timer interrupt won't ever happen as long as there is active video. However, in case the sync pulses are missing, the timer takes over and interrupts the cpu allowing the serial processing to take place.

The interrupt routines are where the interesting work gets done. The timer interrupt is simple. It's main function is to allow the AVR to come out of sleep mode and process the serial port. However, it also resets the timer and sets the line pointer to 0, indicating the top of the screen, so that the system doesn't think it is on an active video line if that is where it last got a sync pulse. Once a video signal is present again, the program will lock back on to the sync signal.

The analog comparator interrupt is the real workhorse. It's main function is to generate the video signal. It has some housekeeping to do also, but most of the code is used to turn ASCII codes into a serial bitstream and put those bits out to the NTSC signal.

Upon entering the IRQ, the timer is reset to prevent accidental interrupts. The processor then goes into a wait loop for about 5 microseconds, the length of a normal horizontal sync pulse. After that, the comparator output is tested and if it is still active the vertical sync portion of the routine is entered. If it is not still active, then it must be a horizontal pulse. The line counter is incremented and tested. If no video needs to be generated for this line, the routine returns with no further action.

If video does need to be generated, another wait of 10 uSec allows the characters to be centered on the display. Then the registers are preloaded with the constants and addresses needed to generate the video as fast as possible. The bytes to display are fetched in sequence from the program memory, using the ASCII code as an index. The load program memory instruction reads the byte for the current row of the character into R0, then outputs and shifts all the bits to port B7. The characters are in a 5 x 8 matrix shifted to the most significant bits of the byte. The low 3 bits of each byte must be 0 to prevent interference with the comparator as the bits are shifted out and to turn off the video when the last bit is shifted out. Once all 20 characters for the current row are output, the routine does a little more housekeeping to prepare for the next row and returns.

A word is in order about the video generation. To make the code as fast as possible, the character codes in program memory must be arranged so that the first (top) row of each character

are all contiguous in memory, followed by the second row of all characters, all the way to the last (eighth) row. I wrote a crude QBASIC program to accept a text file and convert it to the proper format and output an include file for the assembler.

During the video generation, each pixel needs to be shifted out as fast as possible. Then each row of dots for the next character needs to be loaded. The time it takes to load the next byte for shifting sets the minimum time between characters. The spacing is slightly wider than I would like, but is quite readable.

Now that the horizontal sync has been described, what happens on vertical sync? Not much. Mostly, the registers and counters are reset to prepare for the next frame of video. The COG doesn't distinguish between even and odd fields, so the video is generated on both, providing characters without lines between the rows as was common on many older computers that used non-interlaced NTSC video output.

One thing to note about the vertical routine: The equalizing pulses that arrive during vertical sync, which are designed to keep a television synced horizontally during the long vertical sync period, cause the COG to actually see several vertical sync pulses in a row. This won't cause any harm, but is worth mentioning.

The last action the vertical sync routine performs is to read the blanking switch and turn off the video output if it is pressed. The video output pin driver is simply turned off. Everything else continues to operate as normal and receipt of any character on the UART turns the video back on.

How to use it:

Using the COG is very simple. Connect a video source to either of the RCA jacks and a monitor or TV to the other. Connect an RS-232 cable to the DB9 port and plug in the power. A signon message will be displayed and the COG is ready to receive data from the serial port.

The serial data should be 9600 bps, 8 data bits, and no parity. There are 128 displayable characters. However, the low 32 ascii codes are reserved for control and not displayable. Any code from 128 to 255 will have the MSB masked off and display the proper character. So to print the lowest 32 characters, you must send them with the high bit set (128 – 159).

There are several control codes that are recognized. Most work as expected. A carriage return returns the cursor to the start of the current line. A line feed moves to the next line. Tab stops are at every fourth position (0,4..16,0). I wanted to be able to move the cursor up and to clear the screen. A somewhat standard form feed (ascii 12) is used to clear the screen and home the cursor, and a vertical tab (ascii 11) will move up one line without erasing anything. A backspace (ASCII 8) backs up one character without erasing the character. The backspace and vertical tab will not back up past the start of the current line or the first line of the screen.

To use the four bit output port, character codes 16 – 31 are used. When one of these characters is received the low four bits are output on the port pins and the strobe goes high for 300 ns.

Without a video source connected there will be no sync pulses. Since the serial port is read as a result of the sync interrupts, not having a video signal would prevent reception of characters. To allow reception of data even without an active video source, a timer will fire an interrupt if no sync is received and allow data to be received even when no video is present.

After powerup, a formfeed, character 12, should be sent to the COG to remove the signon message.

Going Further:

Although the COG works well as it is, there are several improvements that can be made. There are also extensions that could be added and other ways to use it than addressed so far.

First are the improvements. As it stands, the video jitters somewhat. That is probably because of the 10MHz processor clock not matching the video timing of the NTSC signal. Also, the characters as well as the spaces between them are wider than I would like. That is a function of the speed also. Scrolling and clearing the screen are time consuming. Speeding up those functions would make adding other capabilities, such as scrolling up or clearing a line, easier to add. All of these could be improved by running at a faster clock rate. I have seen several projects on the Internet where people were running the AT90S2313 at much higher speeds than specified. I breadboarded a version of the COG early on running at 14.31818 MHz. It worked fine. The characters looked much better and the jitter seemed to improve. It was built on a solderless breadboard and used the capacitance of that breadboard instead of actual load capacitors on the crystal so the frequency may have been way off. Using proper capacitors may have a more dramatic effect. I chose to stick with 10 MHz since it works well enough and is within the specifications of the processor. I left some of the timing constants in place in the code and plan to come back to it in the future. There are other changes that need to be made as well. For that, though, I will probably wait until I get some of the newer ATtiny 2313's, which are rated up to 24 MHz.

As for extensions, I have several ideas in mind. Another small AVR could be added to scan a keyboard and create a complete video terminal for interaction with the control system. Also, a wide range of devices could be connected to the 4 bit output port. A VCR could record the scene from a video camera where motion is detected, security lights could be turned on, or a robot with video camera could be remotely operated.

That brings up other uses of the COG. It is small and inexpensive enough to place with each camera of a security system. A local processor with sensors attached could display its status on that camera's video signal. And as mentioned above, a robot with a video camera could display status messages on the video signal instead of a separate link to the operator. Whatever method used to transmit the video signal would carry the status messages also.

There are about 250 words of program memory, a few registers, and the entire eeprom unused in the COG, as well as a 16 bit timer. It would be interesting to see what else could be put into those resources.

Conclusion:

This project was started because I needed a cheap and simple character overlay generator. The COG has exceeded my initial requirements and expectations and was easier to develop than I anticipated. This was my first real design with the AVR (it shows when reading the code) and I was surprised how easy it was to program. I now have a video overlay module that I can connect to the several televisions and video monitors in my home and receive status updates from my computer. Best of all, I can build all of them I need for about the cost of one commercial module.

Atmel AVR 2004 Design Contest Entry #A3754

AVR COG Flowchart