

Xtreme Debugging

Using the Monochrome Display to Debug Your Apps

TRon

This article is about debugging or shall we say minimizing the number of features. In the real world product has to be done yesterday and it has to be relatively bug free. This means that it must be debugged! Debugging is an artform in itself. Luckily, compiler designers have been working long and hard to help in this effort by creating tools that allow code to be debugged in real-time as it executes. However, there are a number of problems with this approach. First, since we are interested in making games and most of us write for the PC platform, this equation should be very familiar:

Universal PC Game Programming Law 1.0.0.0

[(Direct X) + (C++) + (debugger) + (video game)]^{Windows 95} = CRASH!

Simply put, trying to debug complex applications in Windows 95 such as video games, is a rather complex problem. The act of debugging can change a lot of assumptions about the code and the very structure of the code itself. Of course, the latest tools that come with Borland, and Microsoft compilers work much better than they used to, they still don't like debugging graphical applications that use up a lot of system resources. Not to mention, that optimizing compilers change the code around when debugger information is included in the .EXE's. This means that the debugger version of your code may actually work better or worse than the release version! I have seen this many times.

In all fairness, debuggers are incredibly useful and most of them support dual monitor debugging, i.e. you can use a second monitor to watch the debugger display while your game runs on the main video display. However, sometimes all we want or need to do is sprinkle a few **printf**'s and watch some variables without generating debugger versions of the code and without running a second large App (the debugger) to suck up system resources. Unfortunately, printing doesn't work that well on real-time graphical video displays. Yes, you can overlay the text output on top of your graphics output, but sometimes this is hard to do and, well, it's just plain ugly! What would be nice is if we could control the access to the monochrome display and output variables, status, and other debugging information there without the use of a debugger.

Well, the first question that comes to mind is, "what if I don't have a monochrome display and adapter?" My answer to this is that a monochrome monitor is about \$50.00 new and \$10.00 from a surplus store. A Monochrome Graphics Adapter or MDA is about \$20 from any reputable computer store, so for about \$50 bucks you can build a Duel Monitor Debugging system or DMD. This is well worth it since it will make your life so much easier.

In the remainder of this article we are going to learn how to access the monochrome display and control the MDA. Also, we are going to create a C++ class that will work under DOS 16, Protected Mode DOS, and Windows 95.

Duel Monitor Debugging - DMD

Figure 1.0 - A Duel Monitor Debugging Setup. (insert)

Just to make everything totally clear on what we are trying to accomplish, take a look at Figure 1.0. It is a graphical representation of what we want to do with DMD. As our game runs, we want to output status and variables on this second display, so that we can run our game in whatever graphics mode we like without having to consider text output to display debug information on the game display itself. In essence, what we want to do is create some functions or a C++ class that allows us to communicate with the MDA and print on the display just as we would with **printf(...)**. This way we can simply sprinkle debugging prints within our code and then see the output on the second monitor in real-time as our game runs.

To accomplish the above goal we need to know how to talk to the MDA and tell it to display text. The only problem is that there are no C/C++ libraries or classes to do this that come with the standard compilers, so we will have to write one. This means that we need to do a little research and figure out how the MDA works...hold on a minute....O.K. done. I'll explain how to communicate with the MDA in a moment, but next let's cover some old debugging techniques that are used, so we have some good ideas how to use our new C++ class once we have it.

Back to Basics

Back in the days of TRS-80's, Atari's and Apple's, I dreamt of having debuggers since once in a while it would be nice to let the computer figure out what was wrong with the 50,000 lines of assembly language I was staring at, but that was just a dream. Alas, game programmers had to come up with techniques to "see" what their programs were doing as non-obtrusively as possible. Let's cover a few of these techniques.

Technique 1 - Printing values out - This is the oldest trick in the book. In your code, you want to watch a variable or know when an *if* statement is entered etc., you would simply put some print statements in the area of interest and then as your program ran, the output would tell you what was going on.

Technique 2 - Function Identification - This simple technique just prints out when a function is entered and exited to help figure out where a program is (or where it has been) as a problem occurs. This is similar to stack tracing.

Technique 3 - Reference Counts - This is a more advanced technique used to help figure out how many times a function was called or if it was called at all. Using global or static local variables, a counter is updated whenever a function is called. This counter along with others is then displayed at the end of the program run or in real-time. This helps track how many times a certain function is getting called. This can assist with optimization and logic problems.

Technique 4 - Output Files - This technique is used when it is too difficult or time consuming to display information in real-time and we want a record of what happened for analysis. A disk file is created when the program starts and then debug information such as variables, reference counts, tables, function calls etc. are written to the file. After the program terminates then the file can be analyzed slowly and the problem area can be found.

Using these techniques or others that are derived from these, you can debug most programs in a reasonable amount of time. Now let's move on to the MDA and see how it works.

The Monochrome Display Adapter MDA

Figure 2.0 - MDA Memory Layout. (insert)

The MDA of today is based on the original *Hercules Graphics Adapter* invented about 1,000,000 years ago. Most MDA's are capable of both graphics and text output. Figure 2.0 shows the layout of the MDA screen in both modes. As you can see, the graphics resolution of the MDA is 720x348 and the text resolution is 80x25 (actually other modes are possible with re-programming). The video memory is located at absolute address 0xB00000 or in 16 bit machines B000:0000. The MDA allows both graphics and text, but we are only interested in text today since graphics are much more complex and we don't really need them.

Figure 3.0 - Microscopic Breakdown of Character Cell. (insert)

The MDA memory in text mode is arranged as a matrix of 80x25 cells. Each cell is 2 bytes or 16 bits. The low order byte is the ASCII code of the character to be displayed and the high order byte is the attribute of the character to be displayed which is further broken down into the foreground attribute and background attribute packed in the upper and lower 4 bits. Figure 3.0 shows this breakdown. The ASCII code of the character is self explanatory, but the attribute needs a little explanation. The attribute controls the intensity and style of the character and background under the character. Since, a monochrome display only has shades of green or amber (depending on your monitor) there is no concept of color. The style part of the attribute controls things like underlining, and blinking. Table 1.0 illustrates the various values for the attributes.

Table 1.0 - Settings for Various Intensity and Style Attributes

(Foreground)				
(Background)				
Black	00	NA	07	0F
Dark	NA	88	87	8F
Normal	70	78	NA	NA
Bright	F0	F8	NA	NA

<u>Underlined Version of Character</u> (Foreground)				
(Background)				
Black	NA	NA	01	09
Dark	NA	NA	81	89
Normal	NA	NA	NA	NA
Bright	NA	NA	NA	NA

Reviewing Table 1.0, you can see there are a lot of values that can be used for the attribute, but to keep things simple, I usually stick to NORMAL, and BRIGHT text, and forgo the use of underlining. But they are there if you need them. Getting back to controlling the MDA; we know the memory region that the character/attribute buffer exists at, and we know the format of the data, so let's write a sample piece of code to write a character anywhere on the display. Take a look at Listing 1.0.

Listing 1.0 - A Function to Draw a Single Character on the MDA.

```
void Draw_Char(int x, int y, short c, short attr)
{
    static short *mda_display = 0xB00000; // this will work for Watcom/Win 95, but for DOS 16, you will need to set up segment and offset

    SHORT attr_char=0; // this will hold the combined code

    // combine character and attribute
    attr_char = ((attr << 8) | c);

    // now draw the character
    mda_display[y*80 + x] = attr_char;
} // end Draw_Char
```

Cool Huh? We won't use this function in our C++ class, but it was good for illustrative reasons. Let's quickly review how it works. First, we pass the x, y, character, and attribute we want displayed. The function is entered and we see the definition of **mda_display**, this is totally legal and we won't get any **GPF's** since it is in the first megabyte of memory. However, it is important to note that **mda_display** is a pointer to a short. This is very important because pointer arithmetic will be done on 16 bit boundaries and not 8 bit boundaries. Anyway, moving on, we combine the character with the attribute and then write the data into the MDA with the line:

```
mda_display[y*80 + x] = attr_char;
```

Now, you should look twice at this and make sure this works for you. We said that the MDA memory holds 80x25 characters, but each character is composed of an ASCII code *and* an attribute this means that there are 2 bytes per character or 160 bytes per line. So how can we get away with multiplying y by 80? The answer is pointer arithmetic! Since **mda_display** is a pointer to a short, all math is computed based on 16 bit or 2 bytes boundaries, so it all works out. So (**mda_display**+0) = 0xB00000 and (**mda_display** + 1) = 0xB00002. Get it?

Now that we know how to print a single character on the display, it's pretty much all over. We are now gods of the MDA and with the atomic action we can draw strings, clear the screen, scroll etc. But before we continue with the design of the C++ class, let's stop for a moment and talk about the differences between each OS and how it relates to the MDA memory.

Multiplatform Considerations

The monochrome display code that we are writing can be used on either Real-Mode DOS, Protected Mode 32 bit DOS or on Windows 95. However, we need to see why this all works. First, in standard DOS, pointers are either **near** or **far**. If they are **near**, then they are 16 bits and relative to the current data segment. If they are **far**, then they are 32 bit, well almost. The "almost" stems from the fact the Real-Mode DOS uses the Segment:Offset method of memory access. This means that a pointer must have a *segment* and an *offset*. So the MDA memory is at segment 0xB000, offset 0x0000 or in Intel notation B000:0000. In 32 bit operating systems such as Protected Mode DOS and Windows 95. The Segment Registers are used differently and the memory model becomes **FLAT**. This allows us to use absolute addressing and forget about Segment Registers. Actually, the memory manager in the 486 and Pentium take care of this to make it seem like memory is flat, and the Segment Registers are used as selectors into the descriptor tables, but we don't care. All we care about is where is the MDA memory? It is at 0xB00000.

If you're a Protected Mode DOS programmer and you are using a DOS Extender other than Rational DOS 4G or Pmode then the absolute mapping of 0xB00000 may not work, but under Rational the first 1 meg of memory is directly mapped and you can access it with absolute 32 bit pointers. Also, in Windows 95 absolute 32 bits pointers work in the first 1 meg of memory to access the MDA. So what I am saying is that if you want this code to work in DOS 32 or Windows 95, it will work unchanged, in DOS 16 you will have to make a couple changes such as the declaration of the MDA memory pointer to set the segment and offset. And if you are using another DOS extender that doesn't map the 1st megabyte linearly then you will have to refer to the docs on how to gain access to the MDA memory.

It's been almost 15 years and we are still haunted by segmented memory. Is there no end to the torture?

Designing the C++ Debugger Class

All right, now we are getting somewhere, we have all we need to create a C++ class that encapsulates the functionality of printing and accessing the MDA. Now, we won't be writing a full-featured debugger. In fact, we aren't writing anything except some methods to allow us to print on the MDA. Use these methods in whatever way you see fit to help debug your code. However, at the last minute I decided that the class needed something cool in it, so I added the ability to "watch" variables of almost any type and added this ability to the class. We will get this new functionality later, but for now let's take a look at the C++ class.

The C++ class is called **mono_print**. It contains almost everything we need for the class except for the "watch" variable structure. The watch record is a separate structure and **mono_print** contains an array of these. These structures along with all the defines and types are contained in the file MONO.H which is shown in Listing 2.0.

Listing 2.0 - The header file MONO.H.

```
// MONO.H - HEADER FOR MONOCHROME PRINTING ENGINE ////////////////

#ifndef MONOCHROME_H                                // watch out for multiple inclusions
#define MONOCHROME_H

// colors and styles for printing

#define MONO_BLACK                0
#define MONO_DARK                 7
#define MONO_DARK_UNDERLINE       1
#define MONO_BRIGHT              15
#define MONO_BRIGHT_UNDERLINE     9
#define MONO_BYTES_PER_LINE      160

// dimensions of monochrome display

#define MONO_ROWS                 25
#define MONO_COLUMNS              80

// watch variable defines
```

```

#define MONO_MAX_WATCHES 64 // maximum number of watches

#define MONO_WATCH_CHAR 0 // types of watches, needed to compute RTTI
#define MONO_WATCH_UCHAR 1
#define MONO_WATCH_SHORT 2
#define MONO_WATCH_USHORT 3
#define MONO_WATCH_INT 4
#define MONO_WATCH_UINT 5
#define MONO_WATCH_STRING 5
#define MONO_WATCH_STRING_HEX 7
#define MONO_WATCH_FLOAT 8
#define MONO_WATCH_PTR 9

// TYPES ///////////////////////////////////////////////////////////////////

// some convenient types

typedef unsigned short USHORT;
typedef unsigned char UCHAR;

// this structure is used to hold the parameters for a "watch" variable

typedef struct mono_watch_tag
{
    char name[32]; // the name of the variable
    void *data; // pointer to the data to watch
    int type; // type of data being watched
    int x,y; // position to display watch
    int field_width; // the width you want cleared in watch display
                  // this is so you don't get garbage
} mono_watch, *_monowatch_ptr;

// CLASSES //////////////////////////////////////////////////////////////////////

class mono_print // monochrome display class
{
public:
    mono_print(void); // constructor
    ~mono_print() {} // destructor does nothing for now

    void print(char *string); // prints a string at the current cursor location
                          // and style, similar to printf supports scrolling etc.

    void draw(char *string, // "draws" a string anywhere with sent style
              int x, // no scrolling or logic
              int y,
              int style);

    void set_cursor(int x, int y); // positions the cursor in the 80x25 matrix
    void get_cursor(int &x, int &y); // retrieves the position of the cursor
    void set_style(int new_style); // sets the style of output
    void enable(); // enables output
    void disable(); // disables output
    void scroll(int num_lines); // scrolls the display from the bottom
    void clear(); // this function clears the display

// methods for debugging watch display

    void delete_watches(void); // deletes all watches

    int add_watch(char *name, // ascii name of variable, up to 15 chars
                 void *addr, // address of variable to watch
                 int type, // type of variable
                 int x, // position of variable
                 int y, // position of variable
                 int field_width); // size of output field

    void update_watches(void); // simply displays all the watches

private:
    int cx,cy; // position of printing cursor on 80x25 matrix
    int style; // style of output, dark, bright, underlined, etc.
    int output_enable; // used to "gate" output to monitor
    USHORT *mono_video; // pointer to the monochrome video buffer

    int num_watches; // number of active watches
    mono_watch watch[MONO_MAX_WATCHES]; // pre-allocate the watch structures to keep
                                      // code simple, IRL use a linked list
};

#endif

```

The class **mono_print** contains a set of basic methods to print, scroll, clear the screen, set the printing style, position the cursor, retrieve the cursor position, and control the output trace. Let's cover what each method does and the underlying data, so we have an understanding of what everything is for.

First let's cover the **private** class variables:

cx,cy	- The position of the virtual cursor on the MDA. Range is (0..79, 0..24).
style	- The style to print with, holds the actual attribute code.
output_enable	- Holds a Boolean 0 or 1 which is used to enable/disable output to the MDA.
*mono_video	- This is the pointer to the MDA video RAM.
num_watches	- This holds the number of active variable watches.
watch[]	- The array of watch record, statically allocated.

Now let's look at the class methods:

mono_print(void) - This is the NULL constructor, it is responsible for resetting all the internal variables and for clearing out all the watch records.

~mono_print() - This is the default destructor, currently it does nothing.

void print(char *string) - This function is similar to the C/C++ library function **printf(...)** except that it only takes a character string. However, the clipping, scrolling, and cursor tracking behavior is similar to the standard **printf(...)** function. This function is the workhorse of the debugging output, use it as you would **printf(...)**, but you will always have to build up your output string with **sprintf(...)** and then pass the resulting string to **print(...)**. The function prints at the last cursor position and current style, also it only understands one extended control code which is '\n'.

void draw(char *string, int x, int y, int style) - This function is similar to **print(...)**, but it allows for more control. This function allows you to send the exact position, and style of the string you want printed. The resulting output has no effect on the current cursor position or printing style. This function also has primitive clipping, so if the string is out of bounds or too long, it will be clipped and displayed.

void set_cursor(int x, int y) - This function is used to set the position of the invisible cursor. The cursor position is used in conjunction with the **print(...)** method, that is, **print(...)** starts printing at the current cursor position.

void get_cursor(int &x, int &y) - This function simply retrieves the current cursor position, note the use of references.

void set_style(int new_style) - This function sets the printing style for all calls to **print(...)**. The printing styles are basically equated to the various attributes in Table 1.0. the available constants are:

```
#define MONO_BLACK          0
#define MONO_DARK          7
#define MONO_DARK_UNDERLINE 1
#define MONO_BRIGHT       15
```

void enable() - This function is used to enable the output to the MDA. This is useful as a "gate" in your programs, so you can turn debugger info on and off by software control.

void disable() - This function is the opposite of **enable()**, it disables the output to the MDA.

void scroll(int num_lines) - This function is used to scroll the display vertically. It was needed to simulate the functionality of **printf(...)**, however, it can be used in a dynamic type display to keep information scrolling instead of printing statically at the same location.

void clear() - This function simply clears the MDA memory and resets everything.

void delete_watches(void) - This deletes all the watch variable entries, more on this shortly.

int add_watch(char *name, void *addr, int type, int x, int y, int field_width) - This is used to add a watch to the watch system, more on this shortly.

void update_watches(void) - This is called each game loop to update and display the watches on the MDA.

Functional Definitions

Figure 4.0 - Including the Source Code. (insert)

The actual code for all the methods is contained in the file MONO.CPP, therefore, to use the system in your code you must include the header file MONO.H and the C++ file MONO.CPP in your project. Figure 4.0 shows this relationship. The code is listed in Listing 3.0 for your review.

Listing 3.0 - The main code module that implements all the methods for the class **mono_print**.

```
// MONO.CPP - C++ METHOD IMPLEMENTATION FOR MONOCHROME PRINTING ENGINE //
```

```

// INCLUDES //////////////////////////////////////

#include <conio.h>
#include <stdlib.h>
#include <malloc.h>
#include <memory.h>
#include <string.h>
#include <stdarg.h>
#include <stdio.h>
#include <math.h>
#include <io.h>
#include <fcntl.h>

#include "mono.h" // include the monochrome header

// CLASS METHODS IMPLEMENTATION //////////////////////////////////////

mono_print::mono_print(void)
{
    // the constructor simply initializes the system and sets the cursor to the upper
    // left hand corner

    cx = cy                = 0;                // position cursor at (0,0)
    style                  = MONO_BRIGHT;      // set style to bright text
    output_enable          = 1;                // enable the output to mono monitor
    mono_video = (USHORT *)0xB0000; // pointer to the monochrome video buffer
    num_watches            = 0;                // set number of watches to 0
} // end mono_print

////////////////////////////////////

void mono_print::print(char *string)
{
    // this function is similar to printf in that it will scroll, wrap, and can interpret
    // newlines, note: we probably could have used the draw function, but the logic needed to control
    // it from this function would be as long as copying the draw function as changing it

    USHORT char_attr, // the total character attribute
            char_part, // the ascii part of the character low byte
            attr_part; // the color part of the character high byte

    int index; // looping index

    // only print if gate is enabled

    if (!output_enable) return;

    // enter main loop and print each character

    for (index=0; index<(int)strlen(string); index++)
    {
        // extract the character and attribute

        char_part = (USHORT)string[index];
        attr_part = ((USHORT)style) << 8;

        // merge character and attribute

        char_attr = (char_part | attr_part);

        // test if this is a control character?
        // for now only test '\n = 0x0A

        if (char_part==0x0A)
        {
            // reset cursor to left edge
            cx=0;

            // advance cursor down a line and test for scroll
            if (++cy>=25)
            {
                scroll(1);
                cy=24;
            } // end if
        } // end if

        else
        {
            // display character

            mono_video[cy*(MONO_BYTES_PER_LINE/2) + cx] = char_attr;

            // update cursor position

            if (++cx>=MONO_COLUMNS)
            {
                cx=0;
                // test for vertical scroll
                if (++cy>=25)
                {
                    scroll(1);
                }
            }
        }
    }
}

```

```

                                cy=24;
                                } // end if
                        } // end if
                } // end else

        } // end for index
} // end print

////////////////////////////////////

void mono_print::draw(char *string,int x,int y,int style)
{
// this function is lower level than print, it simply prints the sent string at
// the sent position and color and doesn't update anything
// note that the function has simple clipping

USHORT char_attr,      // the total character attribute
char_part,             // the ascii part of the character low byte
attr_part;            // the color part of the character high byte

int index,             // looping index
length,               // length of sent string
offset=0;             // used in clipping algorithm

char temp_string[256]; // holds working copy of string

// only print if gate is enabled
if (!output_enable) return;

// do trivial rejections first
length = strlen(string);

if (y<0 || y>(MONO_ROWS-1) || x>(MONO_COLUMNS-1) || (x<=-length)) return;

// make working copy of string
strcpy(temp_string,string);

// now test if string is partially clipped on X axis
if (x<0) // test left extent
{
// set offset into string
offset = -x;

// reset x
x=0;

} // end if

// note that we test both cases, since the string may be longer than the width of display
if (x+length>MONO_COLUMNS) // test right extent
length = MONO_COLUMNS-x;

// enter main loop and print each character
for (index=0; index<length; index++)
{
// extract the character and attribute

char_part = (USHORT)temp_string[index+offset];
attr_part = ((USHORT)style) << 8;

// merge character and attribute
char_attr = (char_part | attr_part);

// display character
mono_video[y*(MONO_BYTES_PER_LINE/2) + x+index] = char_attr;

} // end for index
} // end draw

////////////////////////////////////

void mono_print::set_cursor(int x, int y)
{
// this function sets the position of the printing cursor

// check if x position is valid
if (x<0) cx=0;
else
if (x>=MONO_COLUMNS) cx=MONO_COLUMNS-1;
else
cx = x;

```

```

// check if y position is valid

if (y<0) cy=0;
else
if (y>=MONO_ROWS) cy=MONO_ROWS-1;
else
    cy = y;

} // end set_cursor

////////////////////////////////////

void mono_print::get_cursor(int &x, int &y)
{
// this function retrieves the position of the cursor
x = cx;
y = cy;
} // end get_cursor

////////////////////////////////////

void mono_print::set_style(int new_style)
{
// this function sets the printing style

// make sure the style is somewhat reasonable

if (style<0 || style>255)
    style = MONO_BRIGHT;
else
    style = new_style;
} // end set_style

////////////////////////////////////

void mono_print::enable()
{
// this function sets the output enable gate so that output is sent to the display
output_enable = 1;
} // end enable

////////////////////////////////////

void mono_print::disable()
{
// this function is used to disable the output gate to the monitor
output_enable = 0;
} // end disable

////////////////////////////////////

void mono_print::scroll(int num_lines)
{
// this function scrolls the display upward the requested number of lines
// only print if gate is enabled
// note that mono_video is a USHORT pointer!

if (!output_enable) return;

// the display is 25 lines long, all we need to do is move the last 24 up one
// line and blank out the last line

while (num_lines-->0)
{
    // scroll the last 24 lines up, use memmove since dest & source overlap
    memmove((void *)mono_video, (void *) (mono_video+MONO_BYTES_PER_LINE/2), 24*MONO_BYTES_PER_LINE);

    // now blank out the last line
    memset((void *) (mono_video+24*MONO_BYTES_PER_LINE/2), 0, MONO_BYTES_PER_LINE);
} // end while
} // end scroll

////////////////////////////////////

void mono_print::clear()
{
// this function clears the monochrome display

// only print if gate is enabled
if (!output_enable) return;

// clear the display
memset((void *)mono_video, 0, 25*MONO_BYTES_PER_LINE);

//reset the cursor
cx = cy = 0;

```



```

} // end clear

////////////////////////////////////

void mono_print::delete_watches(void)
{
    // this deletes all the watches

    num_watches = 0;
    memset(watch, 0, sizeof(mono_watch)*MONO_MAX_WATCHES);
} // end delete_watches

////////////////////////////////////

int mono_print::add_watch(char *name,          // name or symbol for watch, up to 15 chars
                        void *addr, // address of variable to watch
                        int type,    // type of variable
                        int x,       // position of watch on display
                        int y,
                        int field_width) // size of output field, for blank padding
{
    // this function creates a watch variable and enters it into the watch list

    if (num_watches>=MONO_MAX_WATCHES)
        return(-1);

    // insert this watch at current location

    strcpy(watch[num_watches].name, name);
    watch[num_watches].data      = addr;
    watch[num_watches].type      = type;
    watch[num_watches].x        = x;
    watch[num_watches].y        = y;
    watch[num_watches].field_width = field_width;

    // increment number of watches

    num_watches++;

    // return the position of this watch
    return(num_watches-1);
} // end add_watch

////////////////////////////////////

void mono_print::update_watches(void)
{
    // this function updates the visual display with the current values of all
    // the watches

    int index;                // looping var
    char temp_string[256];    // temporary working string

    // only print if gate is enabled
    if (!output_enable) return;

    // loop thru all the watch variables and display them
    for (index=0; index<num_watches; index++)
    {
        // first generate display string based on type of variable being watched

        switch (watch[index].type)
        {
            case MONO_WATCH_CHAR:
            {
                sprintf(temp_string, "%s=%c", watch[index].name, ((char *) (watch[index].data)));
                break;
            }

            case MONO_WATCH_UCHAR:
            {
                sprintf(temp_string, "%s=%u", watch[index].name, ((UCHAR *) (watch[index].data)));
                break;
            }

            case MONO_WATCH_SHORT:
            {
                sprintf(temp_string, "%s=%d", watch[index].name, ((short *) (watch[index].data)));
                break;
            }

            case MONO_WATCH_USHORT:
            {
                sprintf(temp_string, "%s=%u", watch[index].name, ((USHORT *) (watch[index].data)));
                break;
            }

            case MONO_WATCH_INT:
            {
                sprintf(temp_string, "%s=%d", watch[index].name, ((int *) (watch[index].data)));
                break;
            }

            case MONO_WATCH_UINT:
            {

```

```

        sprintf(temp_string,"%s=%u",watch[index].name,*((unsigned int *)(watch[index].data)));
    } break;

case MONO_WATCH_STRING:
{
    sprintf(temp_string,"%s=%s",watch[index].name,(char *)(watch[index].data));
} break;

case MONO_WATCH_STRING_HEX:
{
    sprintf(temp_string,"%s=",watch[index].name);

    // merge in hex values for characters
    for (int bc=0; bc<(int)strlen((char *)watch[index].data); bc++)
    {
        char hexvalue[8];
        sprintf(hexvalue,"%X",*(bc+(UCHAR *)watch[index].data));
        strcat(temp_string,hexvalue);
    } // end for bc

    } break;

case MONO_WATCH_FLOAT:
{
    sprintf(temp_string,"%s=%f",watch[index].name,*((float *)(watch[index].data)));
} break;

case MONO_WATCH_PTR:
{
    sprintf(temp_string,"%s=%X",watch[index].name,*((unsigned int *)watch[index].data));
} break;

default:break;

} // end switch

// fill in tail of string with blanks
strncat(temp_string,"",
watch[index].field_width - (strlen(temp_string) - strlen(watch[index].name) - 1));

// print out the watch
draw(temp_string, watch[index].x, watch[index].y, MONO_BRIGHT);

} // end for index

} // end update_watches

////////////////////////////////////

```

Review all of the methods carefully and make sure that you understand what they all do and how they do it. The only methods that are at all complex are the ones that implement the "watch" system. Let's take a look at the watch system and its architecture.

The "Watch" System

As you know, if you want to watch the value of a variable then you simply print it out somewhere on the display and watch it. Well, this is such a common operation that it would be nice if we could somehow automate this process with code. If we could pass a variable to a function and that function would add the variable to a list of variables to watch and display them for us would be perfect. This is exactly what the "watch" system should do. But how can we accomplish this? The trick is using pointers. You see, we can get the address of any object in C/C++, so if we were to create a function that used the address of the variable along with its type then it could print it out correctly. This is exactly how the "watch" system was designed.

Figure 5.0 - The Main Components of the Watch System. (insert)

The "watch" system consists of 4 components as shown in Figure 5.0. There is a global data structure that is contained within the class, a way to add watches, a way to delete watches, and finally a function to display them all. The neat thing about the system is that once a watch variable has been entered into the watch list then you can forget about it. Just use it without caring that it is being watched. The watch system knows its address and type, therefore, it can always figure out its value at any time. We have already seen the code for the watch methods and what they do, so review them now if you need to. The methods are:

- **int add_watch(char *name,void *addr,int type,int x,int y,int field_width)**
- **void delete_watches(void)**
 - **void update_watches(void)**

The key to understanding how the watch system works is understanding the record that contains each watch variable. Here it excerpted from MONO.H for analysis:

```

typedef struct mono_watch_tag
{
    char name[32];           // the name of the variable
    void *data;             // pointer to the data to watch
    int type;               // type of data being watched
    int x,y;                // position to display watch
    int field_width;        // the width you want cleared in watch display
                          // this is so you don't get garbage
}

```

```
} mono_watch, *_monowatch_ptr;
```

The structure is fairly simple, it contains an ASCII name for the watch variable, a **void** pointer to the data to be watched, the type of the data (**int**, **char**, **short**, **float** etc.), the position and field width of the watch display. The key to all this working is the use of the **void** pointer and a separate field to remember the actual type of the data to be watched. The system works by adding watches to the global list **watch[]** contained in **mono_print** via **add_watch(...)**. For example, if we wanted to watch a **int** variable named *count* then we would write something like this:

```
mono_print debug;
int count=0;

debug.add_watch("count",(void *)&count, MONO_WATCH_INT,0,10,10);
```

That's it. The first parameter is the name we want on the display, in this case we use the name of the variable. The second parameter is the address of the variable, notice we cast it to (**void ***). The third parameter is the type of the variable. This very is important, since the code needs this to dereference and print the value correctly. The valid type supported are:

```
#define MONO_WATCH_CHAR      0
#define MONO_WATCH_UCHAR     1
#define MONO_WATCH_SHORT     2
#define MONO_WATCH_USHORT    3
#define MONO_WATCH_INT       4
#define MONO_WATCH_UINT      5
#define MONO_WATCH_STRING     5
#define MONO_WATCH_STRING_HEX 7
#define MONO_WATCH_FLOAT      8
#define MONO_WATCH_PTR        9
```

The fourth and fifth parameters are the x,y position of the watch, i.e. where you want the output to be displayed and the final parameter is the width of the white space padding field to print in. This should be set to the maximum number of characters you expect the data to take up. Once you have added all the watches that you want to display then all you need to do is call **update_watches()** each cycle and the software will take care of the rest! When you are finished with the watches or want to reset them, simply call **delete_watches()**, it will delete all the watches.

Before we finish up with some examples, I want to say that there are more efficient ways of designing the watch system such as dynamic memory allocation along with virtual functions to print out each of the types, but I'm running out of space and I used static arrays to simplify the code and explanation along with encoded types instead of virtual functions.

Debugging Demos

No article would be complete without some demos of the code we have developed, therefore, I present you with 3 demo programs to illustrate all of the functionality of our new class. The first program simply uses all of the method functions, so you can see them in context. The second program, creates a number of watches and displays them in real-time. The final program is the world's simplest game, it creates a starfield, a ship, and allows you move with <A> and <S> while tracking your position with a watch. Here are the listings:

Listing 4.0 - A demo of the basic debugger class functions.

```
// INCLUDES //////////////////////////////////////

#define WIN32_LEAN_AND_MEAN // make sure certain headers are included correctly
#include <windows.h>         // include the standard windows stuff
#include <windowsx.h>        // include the 32 bit stuff
#include <conio.h>
#include <stdlib.h>
#include <malloc.h>
#include <string.h>
#include <stdio.h>
#include <math.h>
#include <io.h>
#include "mono.h"

// MAIN //////////////////////////////////////

void main(void)
{
    mono_print debug;        // create an instance of a debugger port

    int x,y;                 // used to get the cursor position
    char buffer[80];         // used to build up strings

    // clear the debugging display
    debug.clear();

    // demo of simple methods

    // demo of draw
    debug.draw("This is the draw method",0,0,MONO_BRIGHT);
    Sleep(2000);

    // demo of set_cursor
    debug.set_cursor(10,10);
    debug.print("Using the print method after setting the cursor position to 10,10");
```

```

Sleep(2000);

// demo of set_style
debug.set_style(MONO_DARK);
debug.print("\nUsing the print method again with a differnt style");
Sleep(2000);

// demo of get_cursor
debug.get_cursor(x,y);
sprintf(buffer,"Cursor at (%d,%d)",x,y);
debug.print(buffer);
Sleep(2000);

// demo of disable method
debug.disable();
debug.print("\nYou can't see this.");
Sleep(2000);

// demo of enable method
debug.enable();
debug.print("\n");
debug.print("\n");
debug.print("\nNow you can see me, but not for long...\n");
Sleep(2000);

// demo of set_style
debug.set_style(MONO_BRIGHT);
debug.print("\nJust changed style to bright!");
Sleep(2000);

// demo of clear
debug.clear();
Sleep(2000);

// demo of scrolling
while(!kbhit())
{
    debug.print("hit any key to exit      ");
    Sleep(50);
} // end while

// demo of clear
debug.clear();

} // end main

```

Listing 5.0 - A demo of watches.

```

// INCLUDES ////////////////////////////////////////

#define WIN32_LEAN_AND_MEAN // make sure certain headers are included correctly
#include <windows.h> // include the standard windows stuff
#include <windowsx.h> // include the 32 bit stuff
#include <conio.h>
#include <stdlib.h>
#include <malloc.h>
#include <string.h>
#include <stdio.h>
#include <math.h>
#include <io.h>
#include "mono.h"

// MAIN ////////////////////////////////////////

void main(void)
{
    // variables used to watch, one of each type

    char c =0;
    UCHAR uc =0;
    short s =0;
    USHORT us =0;
    int i =0;
    unsigned intui =0;
    char string[]="Game Developer";
    float f=(float)0.0;
    int *ptr= (int *)&i;

    mono_print debug; // create an instance of a debugger port

    // clear the debugging display
    debug.clear();

    // first create watches, note that we simply send the address of each variable cast to void *
    debug.add_watch("char c", (void *)&c, MONO_WATCH_CHAR, 0,0,16);
    debug.add_watch("unsigned char uc", (void *)&uc, MONO_WATCH_UCHAR, 0,1,16);
    debug.add_watch("short s", (void *)&s, MONO_WATCH_SHORT, 0,2,16);
    debug.add_watch("unsigned short us", (void *)&us, MONO_WATCH_USHORT, 0,3,16);
    debug.add_watch("int i", (void *)&i, MONO_WATCH_INT, 0,4,16);
    debug.add_watch("unsigned int ui", (void *)&ui, MONO_WATCH_UINT, 0,5,16);
    debug.add_watch("string ASCII", (void *)&string, MONO_WATCH_STRING, 0,6,16);

```

```

debug.add_watch("string HEX", (void *)string, MONO_WATCH_STRING_HEX, 0, 7, 16);
debug.add_watch("float f", (void *)&f, MONO_WATCH_FLOAT, 0, 8, 16);
debug.add_watch("ptr", (void *)&ptr, MONO_WATCH_PTR, 0, 9, 16);

// enter main loop

while(!kbhit())
{
    // update the watch display with current data
    debug.update_watches();

    // update watch variables each cycle

    c++;
    uc++;
    s++;
    us++;
    i++;
    ui++;
    f=f+(float)0.1;
    ptr++;

    // wait a sec
    Sleep(200);

} // while

} // end main

```

Listing 6.0 - Putting it all together in a simple game demo.

```

// INCLUDES //////////////////////////////////////

#define WIN32_LEAN_AND_MEAN // make sure certain headers are included correctly
#include <windows.h> // include the standard windows stuff
#include <windowsx.h> // include the 32 bit stuff
#include <conio.h>
#include <stdlib.h>
#include <malloc.h>
#include <string.h>
#include <stdio.h>
#include <math.h>
#include <io.h>
#include "mono.h"

// MAIN //////////////////////////////////////

void main(void)
{
    int done = 0; // exit flag
    int ship_x = MONO_COLUMNS/2; // initial position of debugger ship

    mono_print debug; // create an instance of a debugger port

    // clear the debugging display
    debug.clear();

    // show instructions
    debug.print("Controls:\n\n<a> to move left\n<s> to move right.\n\nTo exit press <ESC>.");
    Sleep(5000);

    // add variable to debugger display
    debug.add_watch("ship_x", &ship_x, MONO_WATCH_INT, 0, 0, 8);

    // loop until user hits keys other than 'a' and 's'
    while(!done)
    {
        // draw display
        debug.draw(".", rand()%MONO_COLUMNS, 24, ((rand()%2) ? MONO_BRIGHT : MONO_DARK));
        debug.scroll(1);

        // draw ship
        debug.draw(" |(-)-| ", ship_x, 0, MONO_BRIGHT);

        // move ship
        if (kbhit())
        {
            switch(getch())
            {
                case 'a': if ((ship_x-2)<0) ship_x=0; break;
                case 's': if ((ship_x+2)>MONO_COLUMNS-7) ship_x=MONO_COLUMNS-7; break;

                default: done=1; break;
            } // end switch
        } // end if

        // update debug display
        debug.update_watches();
    }
}

```

```
// wait a sec
Sleep(100);

} // end while

// clear the debugging display
debug.clear();

} // end main
```

The only thing that I slipped in was the use of the Win 32 **Sleep(...)** function. It is a delay that waits for a number of milliseconds. That is the only reason for the inclusion of the windows stuff. Otherwise, you can delete the windows *includes* and use your own delay function.

TRoff

Well that's it. With the **mono_print** class you will be able to display debug information very quickly and cleanly. Of course, you shouldn't throw away your debugger, but you will find that in many cases you can use this class to find simple problems. You can also use the class to display status of various variables in your engines as you develop your games.