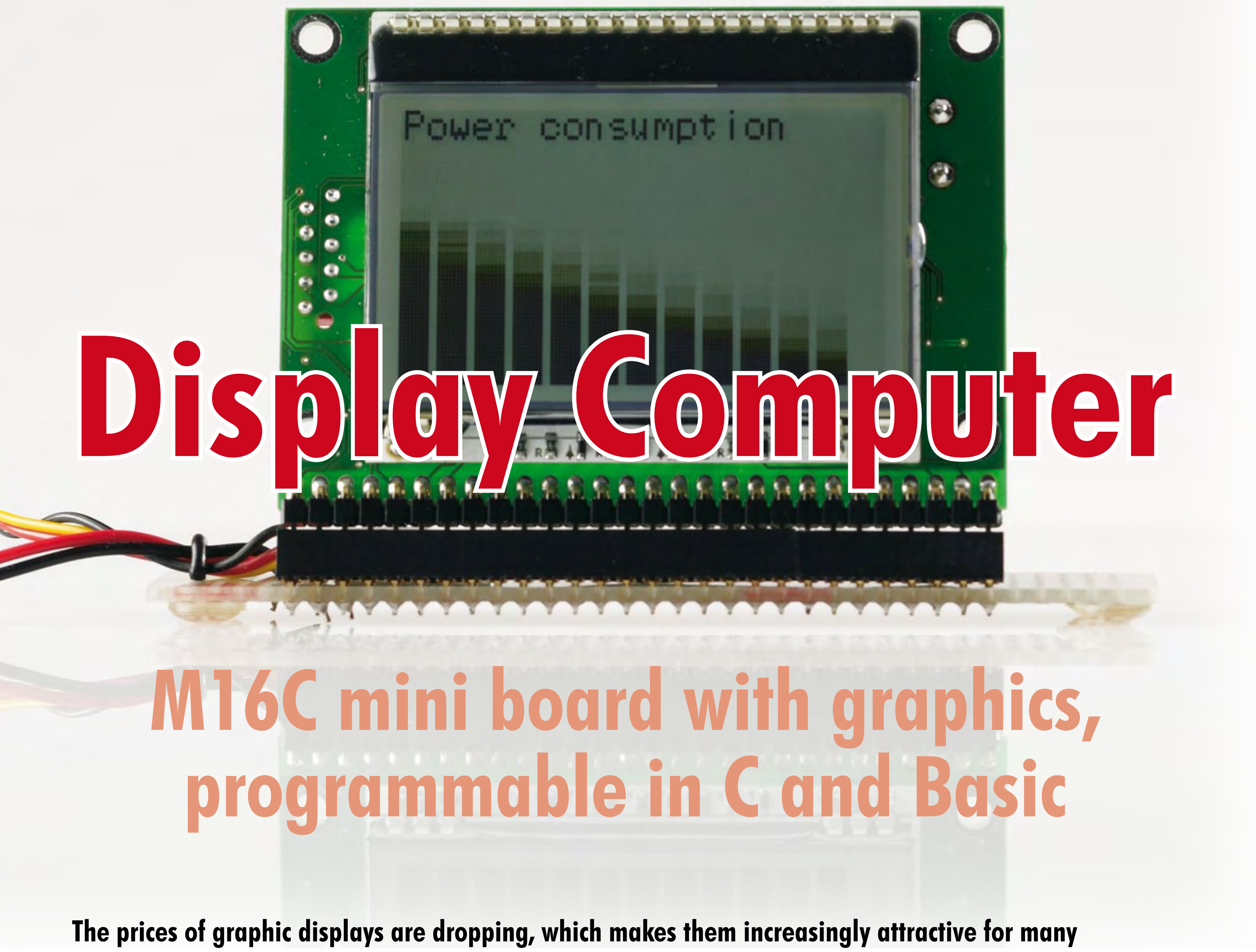


Display Computer

M16C mini board with graphics, programmable in C and Basic

The prices of graphic displays are dropping, which makes them increasingly attractive for many applications. However, programming a graphic display is distinctly more difficult than programming a text display. Our mini microcontroller board features a new display-on-glass module and a high-performance Renesas M16C microcontroller. The board is available fully assembled, and the microcontroller is pre-loaded with a TinyBasic interpreter to simplify the development of graphics applications – even for novices.

Dr Uwe Altenburg

There are countless applications for a stand-alone microcontroller board with a graphic display – everything from model railways or regulating the temperature in your home or conservatory to robotics.

However, driving the display and programming the associated microcontroller are tasks that exceed the skills of quite a few beginners. For this reason, in this article we present a ready-made board equipped with a display, a high-performance 16-bit microcontroller, and even a Basic interpreter [1].

The powerful M16C – the big brother of the R8C, which is well known to many of our readers – can also be programmed in C just like any other microcontroller, so advanced users have plenty of opportunity to develop their own applications. In any case, you can take advantage of features such as 128 kB of flash memory, a 10-bit ADC, PWM signal generation and much more, which make this mini board a truly versatile module.

Display

Electronic Assembly has released a novel 'display on glass' module with the part number EA-DOGM128, which is driven via an SPI interface [2]. This interface can be clocked at up to 20 MHz, so data transmission is not a significant bottleneck. Data is only transmitted in one direction here (from the microcon-

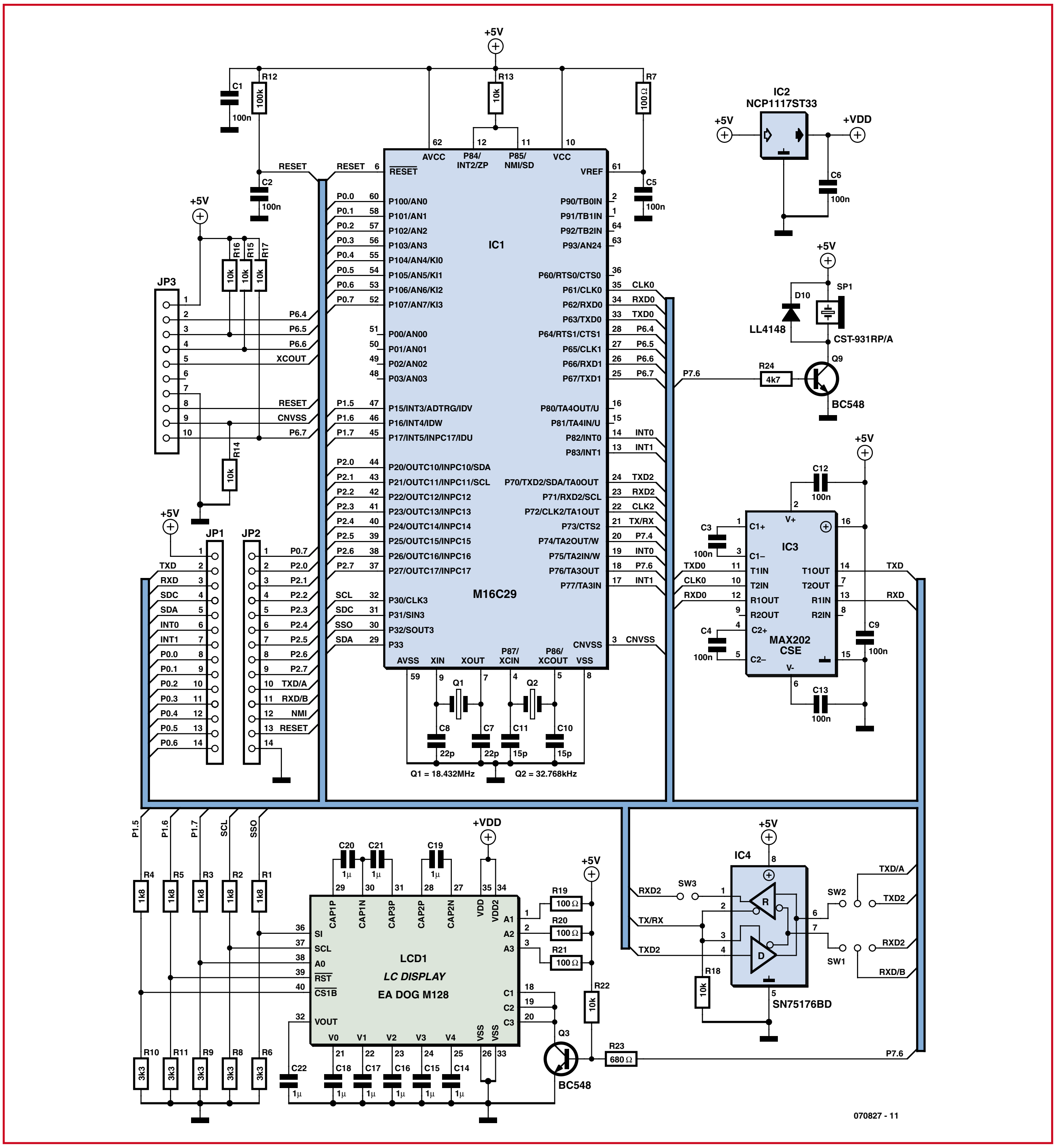


Figure 1. The circuitry around the M16C is relatively uncomplicated.

troller to the display), so only two signal lines are necessary. All in all, only five microcontroller I/O pins are needed – two for the data lines and three for the control lines (RESET, /CS and DATA). The display also features a very low profile of only 5.8 mm. The integrated LED backlight and automatic contrast adjustment ensure good legibility under all conditions, combined with low current consumption. Thanks to its pin-header contacts spaced at 2.54 mm, the module is easy to fit on a PCB. And on top of all this, it is available in several colour combinations (e.g. from Reichelt Germany [3]).

Microcontroller

Our search for a suitable microcontroller led us to choose the Renesas M16C28/29 [4]. This 16-bit machine has an impressive array of features. With 128 kB of flash program memory, 4 kB of flash data memory and 12 kB of RAM, it is generously endowed with storage capacity. Although the display has its own graphic memory, the data for the display must still be assembled in the microcontroller. A monochrome display with a resolution of 128 × 64 pixels requires an image memory of 1 kB in the microcontroller for this purpose (128 × 64 ÷ 8). The M16C28/29 has two DMA channels, so data can be copied directly from the image memory to the display without imposing any significant load on the microcontroller.

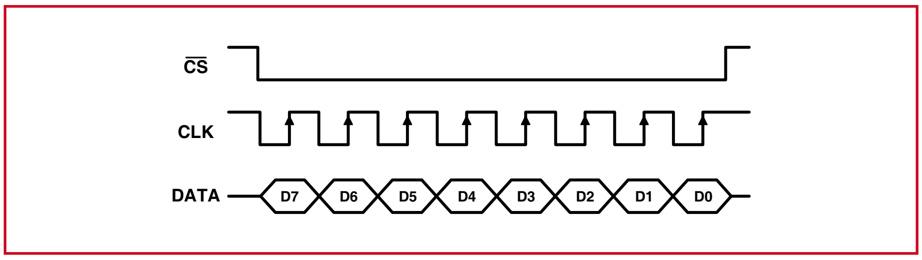


Figure 2. The data transferred via the SPI link is clocked into the display on the rising clock edge.

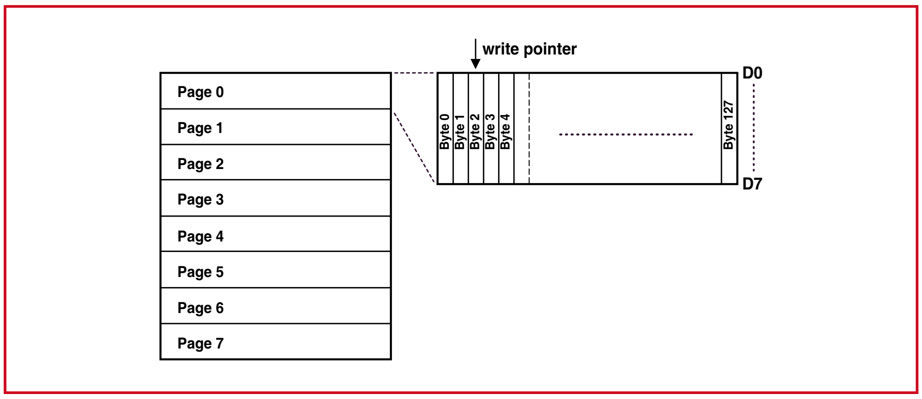
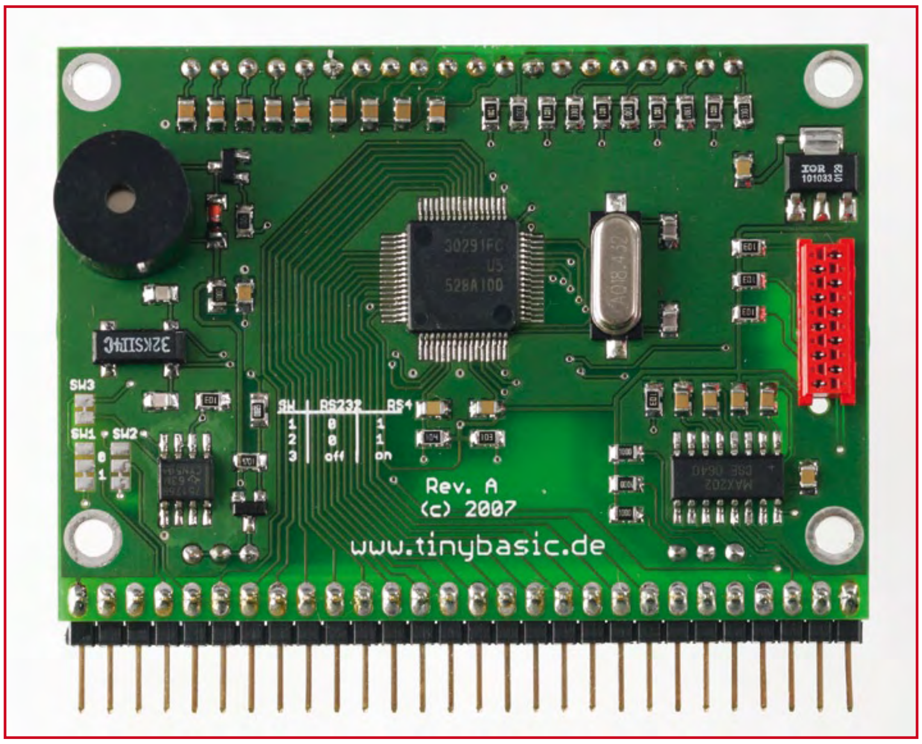


Figure 3. The image memory structure of the display.

Naturally, this little powerhouse in its 64-lead P-LQPFP package also has a lot more to offer. Along with a 10-bit ADC with 16 input channels, it has several timer units, one of which can generate up to eight PWM signals with a resolution of 16 bits. There is an SPI interface for the display (of course), as well as two UARTs available for user-defined functions. A third UART is used for the ISP/debug interface.

Without wishing to revive the old debate on the relative merits of RISC and CISC architectures, we can simply mention here that the instruction set of this CISC processor is extremely efficient. The instruction execution time is only 50 ns with a 20 MHz clock. Several registers can be saved on the stack at the same time with a single assembly-language instruction when an interrupt routine is called. This results in extremely short interrupt response times. For information on other features, such as clock generation using a PLL, we suggest that you consult the datasheet.



Circuit

The schematic diagram of the circuit (**Figure 1**) is relatively uncomplicated. It is built around an M16C29 microcontroller (IC1) with a minimum of peripheral components. The Reset input manages nicely with a simple RC network (R12/C2). The primary clock signal is generated using an 18.432-MHz crystal (Q1). The microcontroller has a specified maximum clock frequency of 20 MHz. Given the capabilities of the internal dividers and the requirement to have the serial interfaces support all standard baud rates from 300 to 115,200 baud, the maximum clock frequency that can be used is 18.432 MHz. However, the SMD crystal used here is a standard component.

In the interest of maintaining a low profile, the display is soldered flush against the surface of the circuit board. All other components are mounted on the solder side. This means that SMD components must be used (with a few exceptions). This applies to the second crystal as well (Q2), which operates at 32.768 MHz to provide a secondary clock signal. There are essentially two situations in which the secondary clock can play a role. The first is when a timer is programmed to act as a real-time clock, for instance in order to trigger a low-priority interrupt once per second. Alternatively, the secondary clock can be used in place of the primary clock in order to operate the microcontroller with the lowest possible power consumption.

A 10-way MicroMatch connector is used for programming and debugging the microcontroller. The signals available on this connector correspond to the signals needed by the Renesas E8 emulator for this type of microcontroller. The Renesas emulator is available from Reichelt [3], among other sources, but it can also be obtained from Rutronik or Glyn along with an evaluation board. The emulator is accompanied by a very good C compiler, which

can handle a code size of up to 64 kB in the free version.
But that's not all: for the C fans among our readers, the next issue of *Elektor* will have an article describing a small circuit that lets you program the microcontroller without using purchased hardware. All you actually need to flash into the M16C is a serial interface and a few freely downloadable tools (for example, from Renesas), and many readers are probably already familiar with them from the R8C 'Tom Thumb' project.

Display power

The display (LCD1) requires only a single supply voltage of 3.3 V. The microcontroller can be operated at 3.3 V or 5 V. As the display module is intended to be used in other circuits, we decided on a supply voltage of 5 V.

A supplementary low-drop 3.3-V voltage regulator (IC2) generates the supply voltage for the display. Simple voltage dividers (R1–R10) are used to adjust the levels of the signals from the microcontroller to the levels required by the display. Here the fact that the display is driven by only five lines, and only in one direction, is a distinct advantage. It keeps the amount of hardware necessary for level adjustment within reasonable limits.
The display microcontroller (an ST7565) needs higher internal voltages to drive the LCD segments. For this purpose, a set of external capacitors (C14–C21) must be connected to its integrated charge pump circuit.

The LEDs for the display backlight are driven via series resistors R19, R20 and R21 and a simple transistor stage (Q3). The display illumination is always on unless it is programmed otherwise. In the simplest case, the illumination can be switched on and off under software control. However, the I/O pin used for this purpose (P7.4) can also be programmed as a PWM output to provide a conveniently adjustable brightness level under software control.
One of the two free serial interfaces is connected to one of the 14-way headers via an RS232 level converter (IC3, a MAX202). This interface can be used for connection to a PC or a modem. The TinyBasic interpreter, which is described later on in this article, also uses this interface for downloading program code. (Note: if you're not afraid of a bit of soldering work, you can tap off another V24 signal from the T2OUT pin of the MAX202 – see the schematic diagram.)
The second serial interface can be connected to the header either directly or indirectly via an RS485 level converter by suitable configuration of jumpers SW1, SW2 and SW3. This means that the module can also be connected to a network and share a bus with other microcontrollers.

Drive logic

Now let's turn our attention to the graphic display. The SPI interface is

Listing 1. Display initialisation
(Data types used: BYTE = 8 bits unsigned, WORD = 16 bits unsigned, INT8 = 8 bits with sign, INT = 16 bits with sign, LONG = 32 bits with sign)

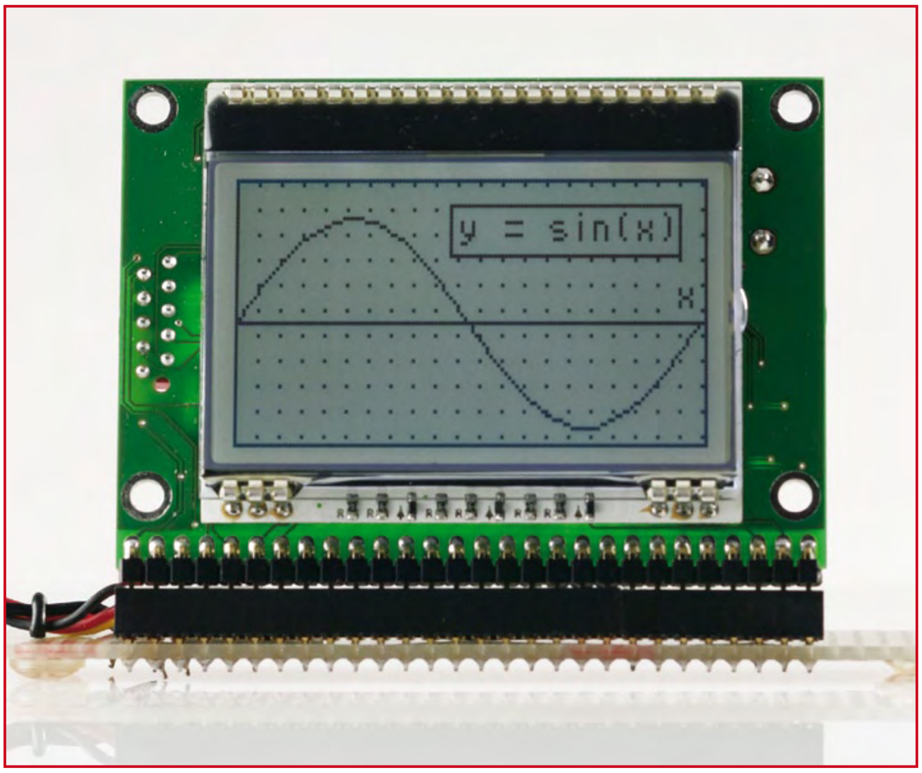
```
// --- Init sequence ---
const BYTE InitList[] =
{
        0x40,                                   // start line
        0xA1,                                   // normal layout
        0xC0,                                   // normal COM0..63
        0xA6,                                   // normal display
        0xA2,                                   // set bias 1/9
        0x2F,                                   // booster regulator on
        0xF8,0x00,                              // booster to 4x
        0x27,0x81,0x16,                         // set contrast
        0xAC,0x00,                              // no indicator
        0xAF                                    // display on
};

// --- Init display ---
void InitDisplay()
{
        BYTE nCmd;

        LCD_CS = 1;                             // no chip select
        LCD_RES = 0;
        Sleep(50);                              // 50ms reset delay
        LCD_RES = 1;
        Sleep(50);                              // 50ms power-up delay

        LCD_MODE = 0;                           // command mode
        for (nCmd = 0; nCmd < sizeof(InitList); nCmd++)
        {
                SPISend(InitList[nCmd]);        // send cmd
                Sleep(1);                       // wait 1ms
        }
}
```

Listing 2. Page copying

```
// --- Copy a single page ---
void CopyPage(BYTE nPage)
{
        BYTE nPos;

        LCD_MODE = 0;                           // command mode
        SPISend(0x40);                          // memory base
        SPISend(0xB0 + nPage);                  // select page
        SPISend(0x00);                          // col low
        SPISend(0x10);                          // col high

        LCD_MODE = 1;                           // data mode
        for (nPos = 0; nPos < 128; nPos++)      // copy page
                SPISend(Pixels[nPage][nPos]);   // send byte

        LCD_MODE = 0;                           // command mode
        SPISend(0xE3);                          // send nop
}
```

operated in Mode 0, which means that the data is clocked into the display on the rising edge of the clock. The timing diagram for this is shown in **Figure 2**. The individual data bytes are transferred to the display sequentially using this clocking scheme. In the code examples, the SPISend() routine is called for this purpose (see **Listing 1**).
The display must be initialised before any image data is sent to it. The routine that does this is called InitDisplay(). This routine sends several commands after the Reset pulse and a short startup time delay. The A0 line of the display must be held low while the commands are being transferred. Consult the data sheet for the display microcontroller (ST7565) for the specific functions of the individual commands. The sequence of commands shown in Listing 1 is designed to initialise the display to an operational state.
The next question is how to send image data to the display. To answer this question, you first have to understand how the image memory of the display is organised. The EA-DOGM128 is divided into eight sections called 'pages'. Each page consists of a 128 × 8 array of pixels. This means that each page needs 128 bytes. The top left pixel corresponds to bit 0 of the first byte of the top page.

The display has a write pointer in addition to the image memory (see **Figure 3**). The write pointer can be set to a specific position in a page by sending commands to the display. All data bytes sent after this are written to the image memory starting from this position. The write pointer is incremented automatically during this process.
Naturally, each data byte modifies eight pixels at the same time. Modifying a single pixel is thus not possible, and it would anyhow not be especially efficient. Instead, a copy of the image memory is maintained in the microcontroller, and it is also organised in pages. This can be achieved by declaring a suitable variable: BYTE Pixels[8][128]. All drawing operations are performed directly on this internal image memory. This simplifies the graphics routines, and it is significantly faster.
Of course, the internal image memory must be copied periodically to the display. In the simplest case, this can be done by an interrupt routine that copies only one page at a time to the display. The CopyPage() routine first sends several commands to the display to set the write pointer to the start of the page to be copied. After this, the page data is sent to the display, and the process is terminated by a NOP command (see **Listing 2**).
At a data rate of 1 Mbit/s, the copy process takes approximately 1.1 ms.



Listing 3. Pixel setting

```
// --- Set a single pixel ---
void SetPixel(BYTE x,BYTE y)
{
        if (x < 128 && y < 64)              // clip
        {
                BYTE nPage = y / 8;         // calc page
                BYTE nMask = 1 << y % 8;    // calc mask

                Pixels[nPage][x] |= nMask;  // set pixel
        }
}
```

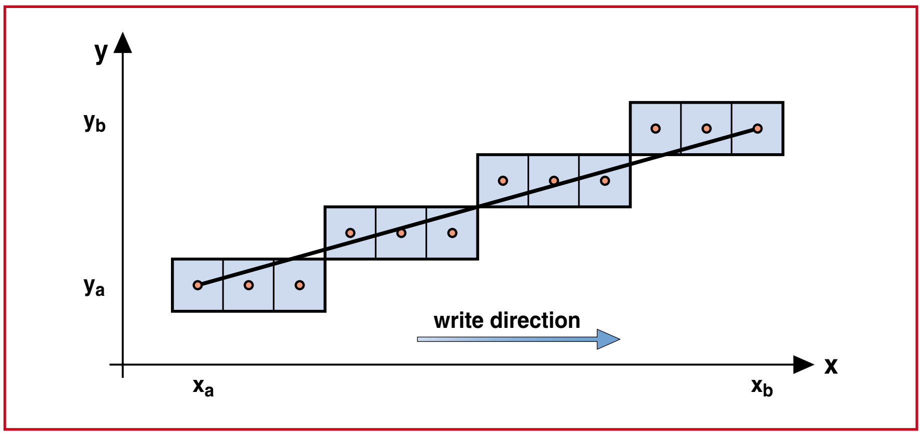


Figure 4. The Bresenham algorithm constructs lines as sequences of adjacent pixels. Here three steps in a straight line are followed by a diagonal step (see Listing 4).

If the interrupt routine is called every 10 ms, it takes around 80 ms to transfer a full image. Naturally, it is only necessary to copy the pages where changes have actually occurred. For this purpose, a 'Dirty' flag is maintained for each page. It is set by the graphics routines. The alternative 'high-end' solution is to use a DMA channel for the page data.

Listing 4. Line from point A(ax,ay) to point B(bx,by)

```
// --- Draw a line ---
void DrawLine(BYTE ax,BYTE ay,BYTE bx,BYTE by)
{
        INT  x  = (INT)ax;                    // start
        INT  y  = (INT)ay;
        INT  dx = (INT)bx - ax;               // distance
        INT  dy = (INT)by - ay;
        INT8 sx = 1;                          // step width
        INT8 sy = 1;

        if (dx < 0)                           // x orientation
        {
                sx = -1;
                dx = -dx;
        }

        if (dy < 0)                           // y orientation
        {
                sy = -1;
                dy = -dy;
        }

        if (dy <= dx)                         // select direction
        {
                INT c = 2 * dx;
                INT m = 2 * dy;
                INT d = 0;

                while (x != bx)               // draw in x direction
                {
                        SetPixel(x,y);        // set pixel

                        x += sx;              // step x
                        d += m;

                        if (d > dx)
                        {
                                y += sy;
                                d -= c;
                        }
                }
        }
        else

        {
                INT c = 2 * dy;
                INT m = 2 * dx;
                INT d = 0;

                while (y != by)               // draw in y direction
                {
                        SetPixel(x,y);        // set pixel

                        y += sy;              // step y
                        d += m;

                        if (d > dy)
                        {
                                x += sx;
                                d -= c;
                        }
                }
        }
}
```

Graphics programming

Now that you know how to connect and program the EA-DOGM128, you might think that you're all set – but in fact you're just getting started. Here we want to talk about drawing line and circles, which is not as simple as it seems.

Of course, the first thing you need is a routine for setting the values of individual pixels, which goes by the name SetPixel(x,y) – see **Listing 3**. As the image memory is virtually located in the RAM of the microcontroller, the routine only has to set the right bit. One of the most important tasks in this connection is range checking, as otherwise it would easily be possible to set bits that have nothing at all to do with the image.

When you hear the term 'straight line', you may well recall the standard algebraic formula for a straight line: $y = ax + b$. It can be used to calculate the y value of a straight line for all possible values of x. This is a good initial approach, but it requires very precise calculation of the factors, in particular the slope a ($\Delta y/\Delta x$), and even with a precise calculation the formula for a straight line results in a broken line if the slope is steep, due to the quantisation of the x values.

Avoiding the need for floating-point arithmetic another is another factor that makes revisiting the 'stone age' of computer technology worthwhile. At that time, clever approaches were often devised to compensate for low processing power – something that seems to neglected more and more these days. In the 1960s, Jack Bresenham was working at IBM on graphic output for plotters, and around 1962 he developed an algorithm that bears his name: the Bresenham algorithm [5a][5b] – see **Figure 4** and **Listing 4**. Even now, 40 years later, this algorithm is just as important as ever.

Lines and circles

The Bresenham algorithm first considers only straight lines with slope $0 < a < 1$ (first octant). The line is drawn by processing the x coordinates incrementally starting from the initial point and deciding for each point

whether the *y* coordinate should be increased. The *y* coordinate is increased if the error relative to the straight line is greater than half a pixel.
The error is calculated for each step in a manner that only requires an integer comparison. Finally, the algorithm can be applied to the other seven octants by mirroring or by reversing the direction of drawing. This yields a very fast and exact line-drawing algorithm. The Bresenham algorithm can also be used to draw circles and ellipses, which avoids the need to calculate sine and cosine values. A suitable listing can be downloaded from the Elektor website [6].

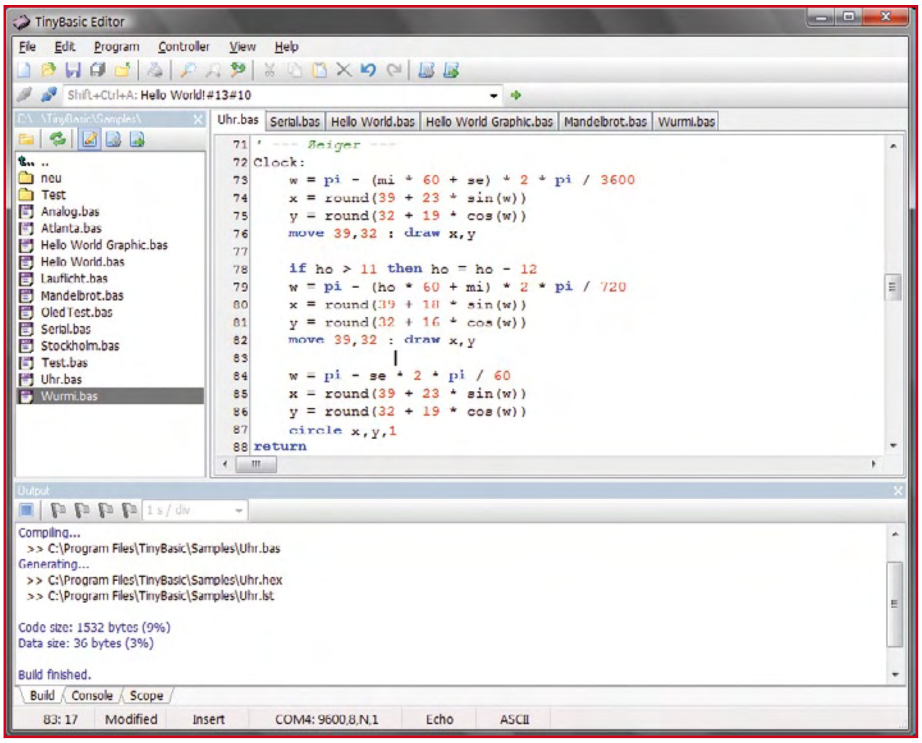
TinyBasic interpreter

As mentioned above, the microcontroller used here has enormous potential, including a relatively large memory capacity, analogue inputs, and a variety of interfaces. It would be a sin to use this potential for nothing more than driving the display. To make it even easier for novice programmers to create their own graphic applications, the author has developed a Basic interpreter.

Programs can be generated using a convenient PC-based text editor that highlights keywords in colour (**Figure 5**). The program code can then be downloaded directly to the flash memory of the microcontroller, where it will be launched immediately when the module is switched on.
An overview of the language and the available commands is available online [1]. Naturally, it supports conditional branching and loop commands, as well as mathematical functions such as SIN(), COS(), EXP() and LOG(). There are also commands for graphic output (PLOT, MOVE, DRAW, COLOR, FRAME, CIRCLE, PICTURE, BARGRAPH) and for accessing the hardware (POKE, PAUSE, SOUND, SETCOM, SETPWM, SETPORT, SETCLOCK, SETDRIVE, SETKEYPAD, SETDISPLAY, SETNETWORK, SETCOUNTER, SEND, RECV, I2CIN, I2COUT, and SPISHIFT).
As an example, up to eight R/C servos could be connected to the module, which is enough for a small walking robot with facial expressions and gestures.

A sample application…

As a sample application to get you started, here we describe how to use



```
' --- Zeiger ---
Clock:
    w = pi - (mi * 60 + se) * 2 * pi / 3600
    x = round(39 + 23 * sin(w))
    y = round(32 + 19 * cos(w))
    move 39,32 : draw x,y

    if ho > 11 then ho = ho - 12
    w = pi - (ho * 60 + mi) * 2 * pi / 720
    x = round(39 + 18 * sin(w))
    y = round(32 + 16 * cos(w))
    move 39,32 : draw x,y

    w = pi - se * 2 * pi / 60
    x = round(39 + 23 * sin(w))
    y = round(32 + 19 * cos(w))
    circle x,y,1
return
```

```
Compiling...
  >> C:\Program Files\TinyBasic\Samples\Uhr.bas
Generating...
  >> C:\Program Files\TinyBasic\Samples\Uhr.hex
  >> C:\Program Files\TinyBasic\Samples\Uhr.lst

Code size: 1532 bytes (9%)
Data size: 36 bytes (3%)

Build finished.
```

Figure 5. The PC-based text editor provides a convenient development environment for Basic programs.

Listing 5. Analogue clock in TinyBasic (excerpt)

```
' --- Definitions ---
#define BTN1_PRESSED   (port0.5 = 0)          ' Button 1
#define BTN2_PRESSED   (port0.6 = 0)          ' Button 2
#define BTN3_PRESSED   (port0.7 = 0)          ' Button 3
#define BACKLIGHT       port7.4               ' LCD backlight
#define T20SEC          20000                 ' Backlight time

' --- Hardware ---
setdisplay LCD_DOGM128x64                     ' Display type
setclock REAL_CLOCK                           ' Real-time clock
setport 7,$10                                 ' Backlight output
setport 0,0,$E0                               ' PB switch pull-ups

' --- Variables ---
float w,t0,t1
byte  ho,mi,se,da,mo,ye,x,y
byte  Icon[18]

' --- Init ---
BACKLIGHT = 1                                 ' Backlight on
Timer0 = T20SEC                               ' Start timer

gosub Scale                                   ' Draw clock face

' --- Main loop ---
do
        if BTN1_PRESSED or BTN2_PRESSED or BTN3_PRESSED then
                Timer0 = T20SEC               ' Start timer
                BACKLIGHT = 1                 ' Backlight on
        elsif Timer0 = 0 then
                BACKLIGHT = 0                 ' Backlight off
        endif

        if Time.Second <> se then             ' New second
                gosub UpdateTime              ' Update time
                gosub UpdateTemp              ' Display temperature
        endif
```

the module to make an analogue clock with an integrated display of the indoor and outdoor temperature. This requires a small amount of additional circuitry for measuring the temperature in a suitable manner.
A temperature-dependent current source, such as the AD592, can be used as the temperature sensor. The advantage of this sensor is that it has a linear characteristic. You only have to connect a 10-kΩ resistor in series with the sensor in order to covert the current into a temperature-dependent voltage with a scale factor of 10 mV/K (see **Figure 6**). Unfortunately, this sensor is relatively expensive. As an alternative, you could construct a similar circuit using the more economical LM344 sensor.

The AD592 supplies a current of 273 μA at a temperature of 0 °C. This results in a voltage of 2.73 V across the resistor. The analogue inputs (in this case P0.0 and P0.1) have a resolution of 10 bits, which means they supply values in the range of 0 to 1023. The currently measured temperature can be calculated using the simple formula 'temp = (analogue_value – 559) ÷ 2.04'.

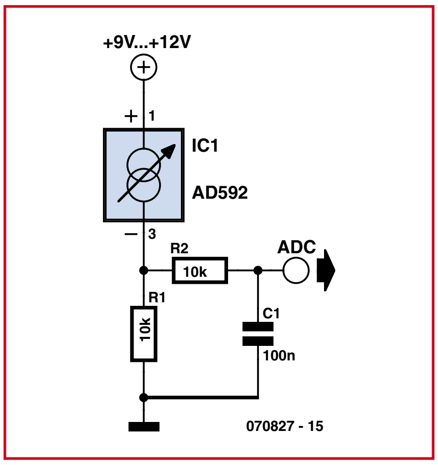


Figure 6. This additional circuitry can be used for temperature measurement.

Naturally, the previously described routines for lines and circles can be used for the analogue clock display. The sin() and cos() functions must be used to calculate the positions of the hands. TinyBasic provides these trigonometric functions (and others) – see **Listing 5**.
As this is supposed to be an analogue clock, the hands should of course move smoothly. For this reason, the positions of the hour and minute hands are interpolated. This gives the impression of continuous motion. The second hand is represented by a small circle, which is more a question of design than necessity.

```
loop

' --- Clock face ---
Scale:
        frame 0,0,127,63                        ' Draw frame
        circle 39,32,29,25                      ' Draw clock

        for w = 0 to 2 * pi step 2 * pi / 12
                x = round(39 + 25 * sin(w))
                y = round(32 - 22 * cos(w))
                plot x,y
        next

        for w = 0 to 2 * pi step 2 * pi / 4
                x = round(39 + 26 * sin(w))
                y = round(32 - 23 * cos(w))
                plot x,y
        next
return

' --- Hands ---
Clock:
        ' Hour hand...
        w = pi - (mi * 60 + se) * 2 * pi / 3600
        x = round(39 + 23 * sin(w))
        y = round(32 + 19 * cos(w))
        move 39,32 : draw x,y

        ' Minute hand...
        if ho > 11 then ho = ho - 12
        w = pi - (ho * 60 + mi) * 2 * pi / 720
        x = round(39 + 18 * sin(w))
        y = round(32 + 16 * cos(w))
        move 39,32 : draw x,y

        ' Second hand...
        w = pi - se * 2 * pi / 60
        x = round(39 + 23 * sin(w))
        y = round(32 + 19 * cos(w))
        circle x,y,1
return
```

...and your own applications

If you buy the display board from the Elektor Shop, you receive a fully assembled and tested module. The TinyBasic interpreter is already installed in the microcontroller. The Basic development environment can be downloaded from the Elektor website [6] free of charge, along with additional examples and listings.

For beginners, we have also put together a step-by-step guide that describes how to install the necessary software on the PC and how to download your own application.

(070827-1)

Web Links
[1] www.tinybasic.de/
[2] www.electronic-assembly.de/deu/dog/dog.htm
[3] www.reichelt.de/ (in German)
[4] www.m16c.de/
[5a] http://en.wikipedia.org/wiki/Bresenham_algorithm
[5b] http://www-lehre.inf.uos.de/~cg/2002/skript/node30.html (in German)
[6] www.elektor.com

Component availability notice

Ready assembled and tested board including display and pre-programmed microcontroller (bootloader and Basic Interpreter) available from the Elektor Shop, order code 070827-91.

Bare PCB, ref. 070827-1 available from www.thepcbshop.com.