Individual Project

Final Report
Supervisor: Susan Eisenbach

Written by Paul Harrison

Date: 17th June 1999
Degree: 4th Year MEng Computing
Co-ordinator: Tony Field
Personal Tutor: Jill Pavne
Abstract:

Diagnosing, and repairing faults with electronic circuits are common everyday activities for engineers. Although both these activities are closely related, they are currently supported by separate toolkits.

The primary aim of this project is to explore the concept of integrating the functionality of these separate tools into a single, portable, programmable device. The secondary aim is to see if the recent increase in power and functionality of leading-edge microcontrollers might provide a suitable platform for implementation.

This report follows the clear distinction between the hardware and software aspects, providing a general overview of how they interact before examining both individually in depth. Finally the report summarises the results and examines the commercial potential for such a device.

Acknowledgements:

My thanks go to both my supervisors, Susan Eisenbach and Kevin Twidle for their guidance, belief, and foresight, that such an idea might be possible.

I would also like to thank James Rice for reminding me that functionality comes before flashiness.

Thanks also to Dave Avery from GCHQ for his help with the surface mount circuit board, and for providing the expertise and equipment required to build the casing.

Thanks also go to all the interviewers I discussed the project with, and anyone else I have talked to about it, but have not specifically mentioned by name above.
Contents:

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract:</td>
<td>1</td>
</tr>
<tr>
<td>Acknowledgements:</td>
<td>1</td>
</tr>
<tr>
<td>Chapter 1 – Introduction:</td>
<td>3</td>
</tr>
<tr>
<td>The Challenge:</td>
<td>3</td>
</tr>
<tr>
<td>Aims:</td>
<td>3</td>
</tr>
<tr>
<td>Concept Demonstrator:</td>
<td>4</td>
</tr>
<tr>
<td>Layout of Report:</td>
<td>6</td>
</tr>
<tr>
<td>Chapter 2 – Background:</td>
<td>7</td>
</tr>
<tr>
<td>Existing Equipment:</td>
<td>7</td>
</tr>
<tr>
<td>Design Background:</td>
<td>12</td>
</tr>
<tr>
<td>Summary:</td>
<td>22</td>
</tr>
<tr>
<td>Chapter 3 – Overview:</td>
<td>23</td>
</tr>
<tr>
<td>Design Stages:</td>
<td>23</td>
</tr>
<tr>
<td>Hardware Overview:</td>
<td>24</td>
</tr>
<tr>
<td>Using the Device:</td>
<td>26</td>
</tr>
<tr>
<td>Operating System:</td>
<td>27</td>
</tr>
<tr>
<td>User Code:</td>
<td>30</td>
</tr>
<tr>
<td>Putting it all together:</td>
<td>33</td>
</tr>
<tr>
<td>Summary:</td>
<td>36</td>
</tr>
<tr>
<td>Chapter 4 – Hardware:</td>
<td>37</td>
</tr>
<tr>
<td>Ports:</td>
<td>37</td>
</tr>
<tr>
<td>Microcontrollers:</td>
<td>38</td>
</tr>
<tr>
<td>LCD Screen:</td>
<td>39</td>
</tr>
<tr>
<td>Indicators:</td>
<td>42</td>
</tr>
<tr>
<td>Keypad:</td>
<td>43</td>
</tr>
<tr>
<td>Sound:</td>
<td>44</td>
</tr>
<tr>
<td>Serial Port:</td>
<td>45</td>
</tr>
<tr>
<td>Digital I/O:</td>
<td>46</td>
</tr>
<tr>
<td>A/D Converter:</td>
<td>47</td>
</tr>
<tr>
<td>D/A Converter:</td>
<td>48</td>
</tr>
<tr>
<td>Testing:</td>
<td>50</td>
</tr>
<tr>
<td>Chapter 5 – Software:</td>
<td>52</td>
</tr>
<tr>
<td>Introduction:</td>
<td>52</td>
</tr>
<tr>
<td>Microcontroller Programming:</td>
<td>53</td>
</tr>
<tr>
<td>Operating System:</td>
<td>55</td>
</tr>
<tr>
<td>Standard Test Equipment:</td>
<td>69</td>
</tr>
<tr>
<td>User Code:</td>
<td>72</td>
</tr>
<tr>
<td>Testing:</td>
<td>79</td>
</tr>
<tr>
<td>Chapter 6 – User Guide/Examples:</td>
<td>81</td>
</tr>
<tr>
<td>Basic Examples:</td>
<td>81</td>
</tr>
<tr>
<td>Advanced Examples:</td>
<td>82</td>
</tr>
<tr>
<td>Chapter 7 – Conclusions:</td>
<td>86</td>
</tr>
<tr>
<td>Commercial Potential:</td>
<td>87</td>
</tr>
<tr>
<td>Achievements:</td>
<td>88</td>
</tr>
<tr>
<td>Possible Extensions:</td>
<td>89</td>
</tr>
<tr>
<td>The Future:</td>
<td>89</td>
</tr>
<tr>
<td>Appendix 1 – Bibliography/References:</td>
<td>90</td>
</tr>
<tr>
<td>Glossary of Terms:</td>
<td>92</td>
</tr>
</tbody>
</table>
Chapter 1 – Introduction:

This chapter introduces the project by explaining what the project is trying to achieve, and why it really is solving a real-world problem.

The Challenge:
So much of our lives today depend on technology, that we take for granted that it is reliable. Unfortunately this is not the case, as electronic circuits do fail regularly. How serious a faulty electronic circuit is, varies considerably, from the almost unnoticeable effect of a defective street lamp, to the life threatening failure of an army communications device.

It is the everyday job of engineers to find and diagnose these faulty electronic circuits. Having found the fault, they then have to repair the problem, or find a temporary solution, until a permanent repair can be made.

Surprisingly, although the activities of diagnosing and repairing faults are very closely related, the type of equipment used is totally separate for both jobs.

Another problem facing engineers dealing with such circuits is that they are often located outside of the normal laboratory environment. This often necessitates taking bulky test equipment out into the field. Once in the field, another problem regularly arises – lack of mains power supply. This can be solved using either long extension leads (where possible), or in extreme cases, portable generators.

Putting all these factors together suggests that there might be a better solution. Finding that better solution is the motivation for this project.

Aims:
This project has two aims:

• The main aim is to explore the concept of integrating the functionality of the tools used for locating, detecting and diagnosing faults with those that are used to provide a solution. This will be achieved through the development of a ‘Concept Demonstrator’, which will attempt to prove that such an idea does have real benefits. It will also show whether it is practically possible to combine the separate tools into a single, portable, programmable device.

• The secondary aim is to see if the recent increase in power and functionality of leading-edge microcontrollers might provide a suitable platform for implementation.
Concept Demonstrator:
The concept demonstrator will be actual proof that the integration of diagnostic and repair equipment is implementable and that such an implementation is useable.

The idea is to design the device so that it can be connected into a faulty electronic circuit. Once attached, it will have two main functions:

1. To analyse the fault, by perhaps acting as a voltmeter or another item of test equipment.

2. To provide a temporary solution, by allowing the user to program the device to act as a replacement for the faulty part of the circuit.

The advantage that this device will provide over other methods, is that it will be small and light enough to be carried around on a belt, and as everything needed will be provided on the device, it will not require any additional pieces of equipment for normal use. On top of this the device will be re-programmable as many times as required.

The power of the device will come from a range of functional blocks, which can be combined by the user, to suit their particular needs. These ‘functional blocks’ will be accessible via a programming language, built into the device.

The functionality will be similar to that of an electronic version of the Swiss Army Knife - some basic facilities will be provided as standard, but the user will be able to extend these and create their own set of functions, to provide a solution to their particular problem.

At the heart of the system will lie a microcontroller which can be programmed in the field, using a small integrated display and keypad. This will be connected to analogue and digital inputs, and analogue and digital outputs. The interconnection of which, will be defined by code, provided by the user, running on the microcontroller.

As the device may be providing a temporary solution to a faulty circuit, it may be left connected to the circuit, until it is possible to produce a permanent solution to the problem.

As this is a research and development based project, with a very open specification, it is expected from the outset that the demonstrator will have some areas of deficiency. The objective is to show that the principles work, rather than building a ‘perfect’ tool, thus detailed specific implementation issues may not be covered where appropriate.
**Design:**
It is intended to design the device around the following outline configuration:

The basic idea is that the control unit will control how the analogue or digital input is sampled, and how the analogue or digital output is generated. It will achieve this by executing code which the user has written, using the built-in keypad and miniature screen. Both sampled data and user code will be able to be stored for later retrieval.

**Programmable:**
One of the device’s main advantages will be it’s ability to be programmed by the user, to perform complex one-off operations, such as sampling a data stream or outputting a particular analogue waveform after a triggering event has occurred. Programming will not require an external PC.

**Target Environment:**
The device is intended as a tool for experienced engineers, it is not aimed at general users. There will be some assumed knowledge, for example that the engineers can program and know how to use equipment such as oscilloscopes and volt-meters. The device will be designed to be an engineering tool, so it is more important for the device to be functional than to have a flashy interface.
Desirable Characteristics:
There are a number of desirable characteristics that the device will ideally have. The main aspects which I will try to achieve are:

- **Ease of use** – as the device may often be used outside, it is preferable to avoid the need to carry a reference manual. This aim is quite difficult to achieve with such a complex device, but it can be achieved, to some extent, if the device is able to offer choices of the functions it is capable of, thus helping the user to know what is available.

- **Self-contained** – part of the main aim is to integrate the functionality of a number of items of test equipment into a single device, therefore no additional pieces of equipment (such as a PC) should be required.

- **Must run from battery power** – it is expected that the device will be used outside, hence no mains or fixed supply can be assumed.

- To simplify the concept demonstrator, it need only be able to analyse TTL circuits, with frequencies up to 1KHz.

Name ‘M G B $\dagger$’:
The original idea for this project came from watching the American TV series ‘MacGyver’. In the show, the central character manages to solve most problems using a Swiss Army Knife and a roll of ‘duct’ tape. I thought it would be interesting to see if the same sort of tool could be made for the electronics industry, and so the project was born. The name M G B $\dagger$ stands for The ‘MacGyver Box Support Tool’. Note that in this report, the terms ‘M G B $\dagger$’ and ‘device’ will be used interchangeably.

Layout of Report:
The rest of this report covers the theory behind the device, and details how the concept demonstrator has been implemented, in particular:

*Chapter 2* discusses the background to the project, explaining the early design decisions.
*Chapter 3* provides a high-level overview of the complete device
*Chapter 4* discusses the concept demonstrator hardware in depth
*Chapter 5* discusses the concept demonstrator software in depth
*Chapter 6* shows how to use the device, with examples of it in action
*Chapter 7* concludes, focussing on whether the idea of integration is a success, and looks at the possible commercial potential
Chapter 2 – Background:

This chapter looks at other similar devices and discusses some of the initial choices made, giving reasons why one solution was selected over another.

Existing Equipment:

This section surveys the market for similar devices.

One of the side effects of this background research is finding out whether the concept of integration is original. As far as I know, and my research has borne this out, there does not exist an integrated electronic support device on the market currently. I have also discussed my ideas with a number of experienced engineers in industry, as well as a number of lecturers at Imperial College, who have all supported the claim that the idea is original. Although this is not proof that such a device doesn’t exist, it does make it unlikely.

Although it does not appear that the concept of integration has been developed before, all the functions that will be provided by the device are already available separately. The M GB st is aiming to integrate a range of features into a small portable device, linking them in a way which makes the whole greater than the sum of the parts.

The processing capability of the device is available in a number of portable computers, from the fully fledged PCs notebooks (such as the Toshiba Libretto [30]) down to any device with just a serial port and some processing facilities (such as the HP48-series calculators [31]).

It is interesting to note that none of the more recent Windows CE palmtop computers [32,33], have specific hardware add-ons. This is probably due to the fact that they only provide a limited number of ports, usually only serial and parallel. Although in theory it would be possible to connect hardware devices to these ports, the current software base tends to be for DOS or Windows 95/NT based systems.

Psion [34] has an industrial division [35] which specialises in mobile data collection, however most of their applications are targeted at the human-entry commercial market, where data is entered by hand or bar-code rather than being sampled from a data source. Psion have two main devices which are similar to the M GB st:

---

Page 7  M GB st - Final Report  Paul Harrison
Psion Organiser II: [16]
- 2 or 4 line text-LCD screen
- Up to 512K FLASH memory
- ABCDE keyboard
- Programmed using built-in BASIC-like OPL [36] language
- RS-232 serial interface
- Communication port for additional peripherals, such as bar-code scanners and magnetic card swipe readers [37]

Psion HC: [17]
- 160 x 80 reflective graphic-LCD screen with optional back-lighting
- 2 drives for Solid State Disks
- Rechargeable NiCad battery module
- Two expansion slots
- Loudspeaker and microphone
- Uses 80C86 processor, thus can be programmed using standard PC development tools, including C and OPL [36]

Neither device has the same number or type of I/O capabilities that the M G B $f$ will have.

Integrated Solutions:

I have only been able to locate one machine which has integrated measurement facilities as an add-on. The machine is the Atari Portfolio [38], which is no longer manufactured. It only has an 40 char x 8 line screen and 128k of memory in total, of which usually 32k is allocated as a RAM disk. However, it does have a full expansion port and a number of data logging add-on peripherals were (and some still are) available. Examples include:

- Portalog [39] produced by IBP. Compact DMM-logger. “Turns the Portfolio into a powerful measurement logger.” It’s main features are:
  - 10 measuring channels, 1 with auto-ranging facility
  - Voltages of up to 400V AC/DC and currents of up to 400mA
  - AC/DC voltage & current measurement
  - Sampling times from 50 msec to 24 hours
  - Stores up to 250,000 values
  - Continuous graphic printout of all channels
  - Data transfer to PC through included software
• **Palmdac** [39] produced by vMS. Data acquisition module that plugs into the Portfolio’s expansion bus. It’s main features are:
  
  • 6 simultaneous data acquisition channels  
  • Custom applications are developed quickly using provided libraries  
  • Programmable sample rates up to 1024Hz  
  • Programmable timer with interrupt capability  
  • 12 bit resolution, 10 bit accuracy  
  • 2 independent amplifiers  
  • 2 independent analogue filters  
  • No external power source required  

**Separate Solutions:**

Most data-logging and measurement add-ons are connected to a PC via the serial port (for low speed) or the parallel port (for higher speeds). The company most famous for such add-ons is *Pico* [10]. They produce a number of devices ranging from simple low-cost single channel analogue/digital converters, through to high-speed multi-channel data-logging equipment. As an example, here is a selection of their products:

• **ADC-10** [40]. Features:
  
  • Entry-level Analogue to Digital converter  
  • Single channel  
  • Digital resolution: 8 bits  
  • Voltage range: 0V to 5V  
  • Sampling rate: 15KHz  
  • Cost: £59

• **ADC-22** [41]. Features:
  
  • Mid-range device, with 22 input channels  
  • Digital resolution: 10 bits  
  • Voltage range: 0V to 2.5V  
  • Sampling rate: 10KHz  
  • Outputs: Single digital line for control  
  • Cost: £199
• **ADC-216 [42].** Features:
  - Top of the range, PC based data-logging oscilloscope replacement
  - Dual channel
  - Timebase: 20 μS/div to 50 s/div
  - Spectrum Range: 0 - 166 KHz
  - Sample rate: 625 Ksps (single channel), 150 ksps (dual channel)
  - Voltage range: ± 20 mV to ±20 V in 9 ranges
  - Digital resolution: 16 bits
  - Buffer size: 32Kb
  - Cost: £499

All the above units come with *PicoScope* [43], a Windows based interface to the units. It provides on-screen controls and displays sampled waveforms in a window. The software also allows spectrum analysis of sampled signals - something not normally available at this price.

Most of the ‘add-ons’ are less powerful than a standard 20MHz scope, in that they aren’t capable of taking as many samples per second or providing the same resolution as a scope is able to. However, they do offer the ability to store and view waveforms on a PC, which is often a big advantage.

Recently, reproducing the facilities of a standard oscilloscope more accurately has become popular. By using the power of a PC, it is possible to add additional services that were once confined to only the most expensive test equipment, such as Spectrum Analysis and Digital Storage.

One of the first devices was the **K7103 Digital Storage PC Oscilloscope [44]** from *Maplin:*

It features:

- Available ready-built or in kit form
- Dual channel (2nd channel optional extra)
- Sensitivity: 10mV to 5V per division
- Frequency range: DC to 16MHz
- Maximum input voltage: 100V
- Timebase: 100ns to 100mS per division
- Digital resolution: 8-bit
- Storage capacity per channel: 4Kb
- Windows control software provided as standard
- Order Code: VF75S
Built with similar aims in mind, *TiePie Engineering* from The Netherlands, sell the **Handyscope 2 [45]**:

It features:

- Dual channel
- Digital resolution: 8-12 bits
- Frequency range: 200KHz - 50MHz
- Voltage range: 100mV - 1200V
- Storage capacity: 64Kb shared between the channels
- Windows control software provided as standard

Finally *Pico* [10] have recently introduced the **OsziFox [46]**. This is a cross between a multi-meter and a digital storage oscilloscope. The device is held in the hand, and has a small 16x32 backlit LCD display to show waveforms. It also has the ability to capture a set of results and display them on a PC.

Features:

- Single channel
- Sample rate: 50nS to 1mS
- Voltage range: 1V to 100V
- Resolution: 6 bits
- Buffer: 128 bytes
- Built-in AC/DC volt meter
- Serial connection to PC
- Windows software provided as standard

**Summary:**

All of the above devices are capable analysis and data storage devices. However, once a problem is found, they cannot help to solve it. This is whether the **M G B s†** comes in. Not only will it be able to analyse the fault, and give a basic display of a waveform, but it will also be programmable, allowing the user to inject signals into a circuit to provide a temporary solution. It’s this combined ability that gives the **M G B s†** it’s advantage.
Design Background:

This section discusses the research completed for the initial design.

**Microcontrollers**

The second aim of this project is to see if the latest microcontrollers are powerful enough to support the control unit requirements for the MGB $f$. I will start by discussing why I am using a microcontroller and not a microprocessor.

Microcontrollers and microprocessors are aimed at two different markets. Microprocessors are designed to form the central intelligence of a system. A range of external components support the microprocessor, such as data memory, CMOS battery backed memory, I/O modules and UARTs.

Microcontrollers are designed as ‘all-in-one’ solutions for handling I/O - they not only contain the central intelligence (the CPU) but also small amounts of memory and built-in UARTs. As I/O functionality is core to their purpose, it is usual to find anywhere between 1 and 32 direct I/O lines directly built into the chip.

The most recent microcontrollers have FLASH program memories, which means they can be electrically programmed up to 1000 times, without the need for erasing them first with ultraviolet light. This makes them ideal for prototyping designs. It also allows easy in-system field upgrading of system software.

Until recently the processing ability of microcontrollers was limited. When mobile phones and other consumer goods required I/O facilities with higher processing needs, new more powerful microcontrollers entered the market. It is this new breed of microcontroller that allows this type of project to be undertaken.

Many companies produce microcontrollers, a selection of the more common variants are catalogued below:

<table>
<thead>
<tr>
<th>Company</th>
<th>Microchip [47]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microcontroller</td>
<td>PIC range [48]</td>
</tr>
<tr>
<td>Details</td>
<td>Large range of devices available. Maximum memory: 8Kb</td>
</tr>
</tbody>
</table>
Maximum EEPROM: 128 bytes
Other facilities: I²C, USART, SPI
Most of the PIC range are One Time Programmable (OTP), although a few FLASH programmable chips are now appearing.

<table>
<thead>
<tr>
<th>Company</th>
<th>Microcontroller</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atmel [49]</td>
<td>AVR range, including AT90S8515 [50] and ATmega103 [4]</td>
<td>Atmel have only recently entered the microcontroller market</td>
</tr>
<tr>
<td></td>
<td></td>
<td>All their devices are FLASH programmable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AT90S8515: 8Kb FLASH, 2Kb SRAM, 2Kb EEPROM, 8MHz</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ATmega103: 128Kb FLASH, 4Kb SRAM, 4Kb EEPROM, 6MHz</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Stack is in SRAM, thus can be up to 4Kb</td>
</tr>
<tr>
<td></td>
<td></td>
<td>In-system programmable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>23 Interrupt sources</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8 multiplexed 10-bit analogue comparator</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other facilities: USART, SPI</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AT90S8515 Package: 40 pin DIP, 48-pin TQFP surface mount</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ATmega103 Package: 64-pin TQFP surface mount only</td>
</tr>
<tr>
<td>Hitachi [51]</td>
<td>H8/3777Y [52]</td>
<td>8-bit microcontroller with 60K FLASH program memory, 2K RAM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Maximum frequency: 16MHz</td>
</tr>
<tr>
<td></td>
<td></td>
<td>8 x 10-bit A/D lines, 2 x 8-bit D/A lines, 58 I/O lines</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other features: I²C, ISA Bus Host, Keyboard controller</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Package: 80-pin QFP Surface mount</td>
</tr>
<tr>
<td>Scenix [53]</td>
<td>SX [54]</td>
<td>Fastest microcontroller on the market - 50MHz</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2Kb FLASH program memory</td>
</tr>
<tr>
<td></td>
<td></td>
<td>136 Bytes RAM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>In-system programmable</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Internal RC oscillator up to 4MHz</td>
</tr>
<tr>
<td></td>
<td></td>
<td>On-chip analogue comparator</td>
</tr>
</tbody>
</table>
8 Level hardware stack
Interrupt facilities
Pin compatible with PIC16C5X chips

Company: **Parallax** [55]
Microcontroller: **BASIC Stamp**
Details: A PIC and interfacing circuitry integrated onto a small surface mount board. It comes in two different versions:

**Stamp 1** [56]: 8 I/O lines
- 80 lines of BASIC
- 256 bytes program EEPROM
- RS-232 comms to 2400 baud
- 2mA at 6-12V DC
- Based on PIC16C58
- Cost: £25

**Stamp 2** [57]: 16 I/O lines
- 600 lines of BASIC
- 2048 bytes program EEPROM
- RS-232 comms to 50Kbaud
- SPI interface
- 7mA at 6-12V DC
- Based on PIC16C58
- Cost: £39

The next stage is to select the most appropriate microcontroller to start the project. The clear contenders are the PIC, Hitachi H8/300, and AVR ranges. Although the Hitachi is very well specified (the only microcontroller to have onboard A/D and D/A), it is only available in 80-pin surface mount - making prototyping difficult. Hitachi have also revealed that this chip is designed for OEM, and thus is only available in multiples of 1000, at around £25 each. This unfortunately limits the choice down to the PIC or AVR.

The PIC range has a clear advantage, because it has been available for many years, it has built-up a strong development environment, including assemblers, C-compilers and in-circuit
emulators (ICE). The AVR range, being a more recent entrant, only has an assembler, a software simulator, and an expensive (£2000) C-compiler, and thus does not have a good development environment.

Taking memory into consideration: the PIC range does not go above 8K, whereas the top AVR has 128K, allowing a much more powerful device to be designed. Therefore, although the development environment is weak, with the current choice of microcontrollers (and these options may change during the project lifetime), the initial choice will be the Atmel AVR ATmega103.

---

**Operating System**

The main role of the operating system is to provide an ‘overall higher level’ control of the device. It allows multiple processes to co-exist and ensures they can access the hardware layer safely. The operating system is also responsible for ensuring the correct modules are running at the right time and will make sure that time critical events happen when they should. Finally it will also be responsible for ensuring that users code cannot cause the entire system to crash.

Operating systems are fairly specific to the underlying architecture and so comparing specific operating system is not particularly helpful, although examining the type of facilities provided by similar real-time operating systems, will be beneficial.

A number of operating systems specifically for the AVR, are currently under development, although at the time of writing none of them has been commercially released.

---

*Jon Anders Haugum*, a project engineer from Norway, is developing a kernel for the AVR range called ‘avrk’. His website [18] lists it’s main features:

- Pre-emptive multitasking
- Priority levels, with round-robin scheduling at equal priority levels
- Unlimited number of tasks
- Simple signalling system between tasks

Unfortunately no source code is available currently.
Myke Predko, technical author of ‘Handbook of Microcontrollers’ developed a basic operating system for the 68HC05 controller. However it, in his own words, ‘…had a few deficiencies that were largely due to the architecture and device’. The extra facilities provided by the PIC and AVR microcontrollers have given him reason to update his real-time operating system (RTOS) to take advantage of the extra memory and power of the new microcontrollers; his website [19] describes the new RTOS features:

- More than four tasks
- Tasks are able to call subroutines
- Each task is given a priority level
- Keep part of the bank registers as well as context registers in the task block
- Pointers to messages
- Simplify interrupt operation

Source code for the UMPS [20] microcontroller simulator is available, although no direct assembler source is provided as yet.

Although not strictly an operating system in its own right, Antti Lukats from SiStudio [21] is working on an interpreting kernel, which offers similar facilities. Called ARTI (A Real Tiny Interpreter), it is a minimal kernel designed to be implemented on microcontrollers with only a couple of hundred words of memory. It’s design was conceived before FLASH microcontrollers became the norm, and targeted the OTP (One-Time Programmable) microcontroller market.

It’s basic theory was to run Forth-like user code from EEPROM, thus allowing the user to change the EEPROM code, after the OTP-chip had been programmed.

It also circumvents one of the major problems with OTP-chips - if a bug is discovered after they have been programmed, the chip has to be thrown away. If ARTI is used instead, the code can be changed as many times as required.

As the code resides in EEPROM, it would be possible to switch between several different blocks of user code, simulating a type of multitasking, although it would probably be fairly slow in practice.

ARTI was originally designed for the PIC range, but would be easily ported to AVRs.
Normally writing propriety software is best avoided because using standard software allows greater compatibility, but in the case of the M GB st’s operating system this rule will have to be broken, because the functionality required is so specific, and no operating systems are currently available. Using the above research as a starting point, there are a number of features that the basic operating system should provide:

- **Multi-tasking** – this will allow the user to do more than one task at a time. For example, it will allow them to output a test waveform (running the process in the background), while at the same time another process can be sampling and storing the results from the circuit under test.

- **Priority Scheduling** – different processes will have different urgencies to run. A process which outputs a square wave each minute need not have as much processor time as a 1KHz analogue sampling process. Ideally these priorities should be dynamically modifiable during execution, to utilise the processor in the most efficient manner.

- **Accurate time delays** – as the device will be handing real-time data, it is important that processes can delay for precise periods of time. A process sampling a data stream at 10th of a second intervals needs to be able to wait until the correct time to proceed.

- **Message passing between modules** – once a module generates data it must be able to pass it to other modules to make that data available. The module handling the keypad will need to pass the current state of the keys to whichever process is currently active.

- **User code interruption facility** – users will be able to write code that loops continuously, thus it is important that the operating system is able to stop such processes under user control. This type of execution-stopping facility is known as a ‘break-in’ [22].

---

**Source Language**

One of the key advantages that the M GB st has over other items of test equipment is that it is programmable, allowing the user to customise the device for each situation encountered. It therefore requires a method for the user to instruct the device what to do. As the user will need to be able to write sequences of commands, a full source language is needed. There are
a number of different issues to consider when choosing a language, and these are discussed in
the next sections.

The first choice is whether to compile or interpret the source language. Compiled code gives
the most compact, fastest representation of the code possible. Normally this would be the
chosen option, however this is not possible on a microcontroller due to their Harvard
architecture.

This is a crucial point, as it effects the whole design of the system. A microprocessor can run
code which modifies memory, and then sets the program counter to point to the modified
code and run it. This is the quintessential technique used for compiling code.

The Harvard architecture uses separate buses for program data and user data. Combine this
with a FLASH program memory, and the designer has no choice but to write all code in
advance - no code can be dynamically generated. It is therefore impossible to compile the
source language for the microcontroller, leaving some form of interpreted language the only
option.

Richard Bornat [25] notes 'In an environment where program design is constantly changing,
where programmers are experimenting with the design of programs... an interpreter certainly
saves human time and effort because it permits more effective interaction with the object
program than is possible when the program is compiled.'

A large number of source languages currently exist. Only a small proportion of these fit into
the type of memory constraints the M G B will have. A description of the main contenders
is given below:

<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Description:</td>
<td>Acronym for Tiny Beginners All Symbol Instruction Code. Designed to be relatively readable, and easy to use for a beginner.</td>
</tr>
<tr>
<td>Advantages:</td>
<td>Very widely known and understood. Short learning curve.</td>
</tr>
<tr>
<td>Disadvantages:</td>
<td>Slow, not very elegant. Has a reputation of not being very capable.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Language:</th>
<th>MINOL [12]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description:</td>
<td>1.75Kb implementation of BASIC. Similar to Tiny BASIC</td>
</tr>
<tr>
<td>Advantages:</td>
<td>Very small memory requirements. Code still reasonably readable.</td>
</tr>
<tr>
<td>Disadvantages:</td>
<td>Limited number of BASIC commands implemented.</td>
</tr>
</tbody>
</table>
Forth [13]

Description: Directly executable language based on an abstract stack model. All code is written in reverse polish notation (RPN). Functions are known as ‘words’, and the user can add new ‘words’ to solve new problems.

Advantages: Very easy to implement in a small amount of memory. Fast to execute. User code is very compact.

Disadvantages: Difficult to read, steep learning curve.

C

VSEL [14] (Very Small Embedded Language) is a scripting language based loosely on C. It is interpreted, and runs internally on a stack machine.

Advantages: Allows code looking like C to be written and implemented in little memory. C is widely known and understood, and is preferred to BASIC.

Disadvantages: In the memory available, it is not possible to provide full ANSI C compatibility, which limits the usefulness of VSEL. Interestingly, it’s author, Scott Bigham notes ‘...the core VSEL interpreter adds only 20-25K.’ VSEL is aimed at small Linux systems, not the very small constraints of a microcontroller!

In an ideal world, a subset of C would be implemented, but due to the restricted memory size, it is unlikely that enough of a C interpreter would be implementable, to give enough benefit.

The second choice, from an implementation point of view, would be a Forth variant. It is easy to write Forth interpreters and user code executes quickly. However, the M G B $f is designed to be used in conditions where putting together a quick program to solve a temporary problem is important, and this is where Forth fails badly - the code is very difficult to read without considerable training.

This leaves BASIC. Although BASIC is often not well-respected and considered too simple for ‘serious’ programming, it is ideal in the type of environment where getting short blocks of code up-and-running quickly is essential. It is interesting to note that Psion have also opted for a BASIC-like language, OPL [36], in their organisers [16].

I have therefore decided to implement a language from the BASIC family, with one difference - to improve it’s speed I will implement an intermediate language, that will
hopefully offer some of the advantages of Forth. This intermediate language is discussed further in the next section.

---

**Internal Language:**

Converting direct from source language to machine code (called interpreting) is slow. One method to improve speed is to use an intermediate or internal language. The idea being to convert the source code to the internal language and then convert the internal language to machine code (or run it directly). This is based on the assumption that converting to, and running the internal language is faster than converting from source direct.

These are the properties an ideal internal language should possess [22]:

- It should be relatively easy to convert from the source to the internal language, although speed is not critical as this conversion can be done as code is entered.

- Converting to, and executing the internal language must be faster than executing direct from source, otherwise there is little point in the conversion.

- It is very advantageous if it is possible to reproduce the original source from the internal language, as this means only one representation of the source has to be stored.

- A useful by-product is often that the intermediate language representation is more concise than the original source, requiring less storage space.

- It is also possible to apply optimisations to an intermediate representation, although this aspect won’t be covered by this project.

A number of internal languages already exist. A selection of these are discussed below [23]:

<table>
<thead>
<tr>
<th>Internal language:</th>
<th>p-code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Used with:</td>
<td>Pascal</td>
</tr>
<tr>
<td>Theory:</td>
<td>Cross between machine code and Pascal, a simplified version of Pascal</td>
</tr>
</tbody>
</table>
Internal language: Syntax tree
Used with: Wide range of languages, C for example uses this technique
Theory: A tree structure is created for the program, which is then walked over by the compiler

Internal language: Reverse Polish / Postfix notation
Used with: Forth
Theory: All statements are converted into postfix notation and then executed from the stack

The space advantages of Forth can be regained by implementing the BASIC source language as a stack machine [22]. This can be achieved by converting the source language into a Reverse Polish notation. Not only will this reduce the storage space required for the code, while still allowing fast execution, but it will also allow the source to be reconstructed from the internal language, thus removing the need to store both the original source and the interpreted version.

**Human-Computer Interaction:**
The user-interface provided by the device will be important, because a number of complex functions are available at any one time, and the user should be able to use the device to its maximum potential with relative ease.

It was decided early on, that although a graphic LCD screen would be the most appropriate solution, it would not be realistic to drive it in the time available. A small 12 character by 2 line text-screen was therefore the next best option.

After many hours of research I could find no documents that refer specifically to such small screens, although as Clive Akass points out [24] ‘The proliferation of web-access devices has created the problem of how to format information for displays such as those on a mobile phone, a TV and a PC’. He suggests that new markup languages may be the solution. However, even mobile phone displays tend to use graphics and several lines of text, against which will only have 24 characters in total.

The screen strongly influences the choice of interaction method, and in this case a menu system of some form is the obvious choice. Jenny Preece writes that ‘Unlike command-driven systems, menus have the advantage that users do not have to remember the item they want, they need only recognise it.’[27]. This is a very good point, as it address the ‘Ease of
Use’ desirable characteristic mentioned in chapter 1. Having menu options will reduce the amount of information the user will have to remember.

She also notes that ‘In the majority of cases it is useful to organise the commands in a hierarchical way.’ As such a small space is available to display menu entries, it will be difficult to show where the user is in a hierarchical system, and thus this is not an appropriate solution for this device, so the initial device will use only single depth menus where possible.

To access the menus, a keypad will be required. The device simply doesn’t have enough space for a full-size keyboard, so a compromise has to be made. There are two clear options which would both be suitable for menu access: soft-keys or cursor keys and a select key.

Soft-keys (or function keys) change their function depending on the current context. Donald Norman [26] points out that a designer should try to ‘make it easy to determine what actions are possible at any moment’, soft-keys don’t allow advance calculation of possible actions, because it depends on the current context.

This is where cursor keys and a select key provide an advantage: it doesn’t matter whether the user is selecting from a menu, or entering code, they always know that pressing the ‘UP’ key will scroll upwards through a list, the ‘DOWN’ key will scroll downwards, and ‘SELECT’ will allow them to make their choice known to the system. It is interesting to note that Hewlett Packard, when designing their new range of ‘easy to use’ Infinium oscilloscopes [28], have found that soft-keys aren’t popular with engineers either.

Summary:
This chapter has shown the background research which has led to the following initial design decisions:

- **Control Unit:** Atmel AVR ATmega103
- **Operating System:** Propriety, priority scheduled multi-tasking
- **Source Language:** BASIC
- **Internal Language:** Reverse Polish
- **User Interface:** Menus with cursor-keys and a selection key

The next chapter explains how these will fit together with the other required hardware and software elements, to produce the final device.
Chapter 3 – Overview:

This chapter discusses the general theory behind the M G B $sl$, and how the individual parts interact. Technical details of how each component works can be found in the following two chapters.

Design Stages:
Due to the research and development type nature of this project, an incremental style of design was chosen as being most suitable. The plan was to design three prototypes, each improving and correcting features from the previous. This way of working proved to be very successful, as it allowed minor problems to be ignored until the following stage. These are features of each prototype:

**Prototype 1:**

Timescale: October - December 1998

Purpose: To ensure using the Atmel microcontrollers didn’t pose any serious problems, and to give a feel for using the on-screen menus.

Features: Simple microcontroller – 5V AT90S1200A device running at 1MHz 12x2 LCD screen with software controllable backlighting Switches controlling a simple on-screen menu RS-232 serial connection to PC

---

**Prototype 2:**

Timescale: January – May 1999

Purpose: To develop and implement the main technical aspects of the device, including the operating system, user code facilities, and I/O capabilities.

Features: Powerful microcontroller – 3.3v ATmega103L running at 3.686MHz 16x2 LCD screen without backlighting Switches controlling a simple on-screen menu RS-232 serial connection to PC, Piezo buzzer for sound output Single channel analogue input, Single channel analogue output
Pre-emptive multi-tasking operating system
Multiple system processes, multiple user processes
User code editor, user code execution unit
Volt-meter, Scope and fast-sampling facilities.

Prototype 3:
Timescale: June 1999

Purpose: To show the device at a realistic size, to give a better feel for how it would be to use.

Features: 5V ATmega103 running at 6MHz
No additional technical features
Minor corrections and improvements to prototype 2
Surface mount circuit board
Realistic-sized plastic casing

Prototype 2 uses a low-power 3.3V version of the microcontroller because at the time of development this was the only version available already mounted on a carrier board, allowing easy access to all the surface mount pins. As Prototype 3 uses a custom designed circuit board, a 5V version could be used directly. The 5V version has the advantage that the logic levels are directly TTL compatible, and it is also capable of running at up to 6MHz, against the 4MHz maximum of the 3.3V version.

Note that this report concentrates on Prototype 2, which was used to cover the main technical aspects of the project. Prototype 1 was used purely to ensure the idea was feasible, and Prototype 3 was used to make everything fit into a small case.

Hardware Overview:

The M G B st interacts with the user through a 12 character by 2 line LCD screen and a keypad, which consists of UP/DOWN/SELECT and SYSTEM buttons. These allow the user to scroll through menu options, when they find what they want they can then press SELECT. The SYSTEM button allows access to a higher-level control, enabling different processes to
be selected. The LCD is also backlit, making it easier to read in dark conditions. Two LEDs are provided: a green LED is lit continuously when power is applied, and a red LED is software controllable. A piezo buzzer is software controllable, allowing a basic sound output, designed to allow warning bleeps to be used.

Two BNC connectors are used for Analogue Input and Analogue Output, and a 9-way D-type connector carries both the serial lines (Tx, Rx and Gnd) and 4 Digital Input / Output lines.

Prototype 2 also has an additional 9-way D-type which allows direct access to the microcontroller programming lines for rapid development. This port is not provided on the final version, although the lines are accessible internally (for system updates).

All the above lines are connected either directly or indirectly to the main ATmega103 microcontroller. The serial lines go through a MAX233CPP voltage conversion chip, and the analogue output is generated via an external Digital-To-Analogue Converter (an AD7391). The keys, LCD and LEDs are connected directly to the microcontroller. The LCD backlight is transistor (BC107) controlled via an I/O line on the microcontroller.

The power supply from the 9V battery is regulated and dropped down to +5V via a 7805 voltage regulator. The following block diagram show how all the parts are connected:
Using the device:

**Menu System:**
The menu system allows the user to navigate around the various functions the device offers, by using the cursor keys and the select key. The main menu allows the following choices:

**Main Menu:**

<table>
<thead>
<tr>
<th>Option</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Volt Meter</td>
<td>Displays a volt-meter</td>
</tr>
<tr>
<td>Scope</td>
<td>Display a scope trace</td>
</tr>
<tr>
<td>Sampler</td>
<td>Grabs a block of samples and outputs them to a PC via the serial port.</td>
</tr>
<tr>
<td>Download</td>
<td>Copy all code in memory to a PC via the serial port.</td>
</tr>
<tr>
<td>Upload</td>
<td>Copy a file containing code from a PC to the device, via the serial port.</td>
</tr>
<tr>
<td>Backlight</td>
<td>Toggle the LCD screen backlight on and off</td>
</tr>
<tr>
<td>Restart</td>
<td>Restarts the device, this stops all running code, but doesn’t erase any user code.</td>
</tr>
<tr>
<td>Full Reset</td>
<td>Stops all running programs, clears all program memory, and restarts the device, as if it just powered on for the first time.</td>
</tr>
<tr>
<td>About</td>
<td>Display a message with the current version number.</td>
</tr>
</tbody>
</table>

Each of the four user virtual consoles also have their own menu, allowing the engineering functions to be accessed:

**Console Menu:**

<table>
<thead>
<tr>
<th>Option</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Run Code</td>
<td>Allows the user to run a piece of code they have written.</td>
</tr>
<tr>
<td>Edit Code</td>
<td>Allows the user to edit a piece of code.</td>
</tr>
</tbody>
</table>

**Switching between processes:**
A special ‘SYSTEM’ button is provided which when pressed, immediately runs a high-level control console allowing various system features to be controlled. The concept demonstrator currently uses this purely as a method to allow processes to be switched, but it is envisioned
that many extra facilities could be added here, such as the ability to change the priority levels of processes, or to reset a virtual console which has gone into an infinite loop.

**Volt-Meter, Scope and Sampler functions:**
A number of default equivalents to standard test equipment have also been provided. In their current form they are not as powerful as their desktop equivalents, but given more time, they could be improved, and they do have the advantage of all being available in a single package.

These are the standard functions provided:

| Volt-meter: | Samples the analogue input and displays the voltage, accurate to 2 decimal places. |
| Scope: | Displays a graphical representation of the waveform on the analogue input. It achieves this even though the display is designed to only display text. It is possible to vary the timebase using the cursor keys. |
| Sampler: | This grabs a block of 256 samples as quickly as possible, and then dumps all the samples as a comma separated file to a PC, via the serial port. This allows fast waveforms to be captured and displayed on a PC. |

Details of how these are designed can be found in chapter 5.

---

**Operating System:**

The MG B st is made up from a number of both hardware and software components which interact with each other, controlled by the operating system. As the operating system is the key piece of software, it was developed first. However, before the OS could be written, a basic hardware structure was required to test it. I therefore designed and built the microcontroller programmer, LCD screen connections, four buttons and an LED to allow work on the OS to begin.

The OS has two primary functions:

- Allow multiple processes to run simultaneously (multi-tasking)
- Provide access to hardware features in a controlled manner

Each of these is discussed individually below:
**Multi-tasking:**

Allowing multiple processes to run simultaneously is achieved by giving each process a very short amount of time, perhaps a few 1000ths of a second, before moving onto the next. Swapping between two processes is possible by a method known as a ‘context-switch’ [8]. This involves copying the contents of all registers, including the status and program counter to memory, and then loading in the old values from memory of the next process to run.

Making the OS efficient involves balancing the amount of time processes are given to run, and the amount of time taken to switch between them. This ratio varies depending on the type of processes involved, and so it is advantageous to allow dynamic priorities to allow different processes to run for different amounts of time. My OS partially implements this feature. It allows different processes to have different priorities (and thus run for different lengths of time), but it doesn’t change them as the system is running. As the priorities are in SRAM, it would be easy to add this facility later on.

As well as knowing how to switch between processes, it is also necessary to know when to carry out the switch. There are two options [8]:

- **pre-emptive:** in this method, processes are given a set quantum of time, which when elapsed, control is taken back by the operating system.

- **run to completion:** with this method, processes are allowed to run until they have done as much as they want to, and they then pass control back to the operating system.

‘run to completion’ is easier to implement, but suffers from the major disadvantage that if a process crashes before it has given control back to the OS, the entire system locks up. Pre-emptive has the advantage that the time quantums can be decided in advance to be the most suitable. For example a process displaying the current date and time only needs to run every second, whereas a real-time input process may need to be run every 100\(^{th}\) of a second to ensure it works as required. For these reasons I decided to implement a pre-emptive operating system.

The AVR microcontroller I am using, has three built-in ‘interrupt-generating’ timers, which are ideal for starting the context-switch. The idea is simple, a timer is set for a period of time, perhaps a 10\(^{th}\) of second. Process code is then run, and when the time is up, the timer generates an interrupt, which forces the microcontroller to execute the interrupt handler,
which does the context-switch, then starts the timer counting for another period of time, and finally returns to the new process. Each time the handler is run, it loads the next process in sequence (known as ‘Round-Robin scheduling’ [8]), and so each process gets a short amount of time to execute. This gives an overall effect of all the processes running simultaneously. Figure 1 shows this diagrammatically.

Operating System Services:
The second primary function of the operating system is to bridge the gap between the hardware and the software layers. In practice this means providing a controlled mechanism to allow running processes to access hardware devices, such as the serial port or analogue input.

It is very important that only one process at a time is allowed access to a particular device, otherwise corruption may occur. For example if one process sent the half a nibble to the serial port, and another process was then switched in and tried to send a byte, the receiving equipment would get corrupted data. There are two mechanisms in the operating system to avoid this problem.

The first is the ‘Active Process’ or AP concept. One process is selected by the user to be ‘Active’ and it is given direct access to the screen and keypad. Other processes wanting to read the keypad are blocked, and other processes wanting to write to the display get routed through to a private display data area by the screen service. This screen data is then switched when the user requests a new ‘Active Process’, giving the effect of multiple virtual consoles.

The second method is to force a service to complete before releasing control back to the process. This is needed with the time critical services, such as the serial port, which has to get data out at a constant rate. Forcing completion is achieved by temporarily disabling the timer.
interrupt during the critical sections. Unfortunately this does slow down the system and is only used where absolutely necessary. Currently only the serial port and EEPROM routines use this technique. A full list of all services, and how they are used is provided in chapter 5.

User Code:
Allowing the user to enter, and then run their own code is the key advantage of this device. However, achieving this functionality in limited memory is not easy. Returning to the aim of the concept demonstrator, it is to demonstrate how the device might be used, rather than being concerned with detailed implementation issues. One of the ways of reducing the complexity is to reduce the power of the language used.

For a full working version, it would be best to store user code on an external medium which can be removed and backed up on a PC for safety. One possible solution is the use of a PC card (PCMCIA) which could be interfaced directly to the microcontroller. However, there was not enough time to look at this option, so the choice became between the two internal user memories provided by the microcontroller – SRAM and EEPROM.

SRAM has the great advantage that it is much faster (by an order of magnitude) over EEPROM and it effectively has an infinite life, it main disadvantage is that when power is removed, any stored data is lost. If the user has written a number of routines they use on a regular basis, it would be preferable if the code was not lost when powering down. This indicated that EEPROM was the most appropriate choice for storing user’s code.

EEPROM has a write limit of around 100,000 times. Assuming that an engineer updates his code 10 times a day, 5 days a week, this would allow the device to work for:

$$\frac{100,000 \div (10\text{updates} \times 5\text{days})}{52\text{weeks}} = 38.5\text{years}$$

This is long enough not to be considered a problem. By the time the EEPROM wears out, technology will have moved on, and new devices will be on the market.

It was known in advance that EEPROM was slow, but it wasn’t until code was actually executed that it became clear just how slow it was. The concept demonstrator does not try to correct this problem, but one possible solution would be to copy the program code in SRAM and run it from there.
Each piece of code executes in it’s own private environment, with it’s own local variables, thus two programs can refer to variable ‘A’, and they will each have their own copy which won’t be corrupted by the other code.

The concept demonstrator allows up to 8 programs to be entered and then run. Due to time constraints the facility to edit programs already entered was not completed, however this ability would be required for a full version.

**Entering Code:**
First the user must select the program they wish to enter from an on-screen menu:

```
Edit Proc: 1
```

and then they are shown the first line, with a menu of the possible commands they can enter:

```
10: PRINT
```

The cursor flashes after PRINT to show that there is a choice. Pressing the ‘UP’ and ‘DOWN’ buttons allows the user to scroll through the following available commands:

<table>
<thead>
<tr>
<th>Command</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRINT</td>
<td>Display a value</td>
</tr>
<tr>
<td>INPUT</td>
<td>Ask the user for a value</td>
</tr>
<tr>
<td>LET</td>
<td>Assign a value to a variable</td>
</tr>
<tr>
<td>END</td>
<td>Finish entering the program</td>
</tr>
<tr>
<td>GOTO</td>
<td>Jump to a particular line number</td>
</tr>
<tr>
<td>DELAY</td>
<td>Wait for a period of time</td>
</tr>
<tr>
<td>BEEP</td>
<td>Make an audible beep at the given frequency</td>
</tr>
<tr>
<td>AOUT</td>
<td>Output an analogue value</td>
</tr>
<tr>
<td>AIN</td>
<td>Sample the analogue input</td>
</tr>
<tr>
<td>CLS</td>
<td>Clear the screen</td>
</tr>
</tbody>
</table>

(At the time of writing, it is hoped that several more commands will be available, including ‘IF...THEN...’, ‘DIGIN’ and ‘DIGOUT’ – time will dictate whether these are implemented or not.)
After the user has selected the command they want using ‘SELECT’, the editor then displays the next possible actions. In this case it displays the choice between entering a variable or a number:

```
0: PRINT Num
```

After selecting ‘Num’ for number, the user can then enter the number they want using the next menu:

```
0: PRINT 5
```

Next the editor gives the option of entering a formula by displaying a menu with +, -, *, / and ‘.’ to end. For this example I just want to display the number 5, so I select ‘.’ to finish entering the formula:

```
0: PRINT 5.
```

(Note that the display has scrolled to show the last part of the line)

The editor then displays the next line, and gives the original command menu again:

```
20: PRINT
```

This time I select ‘END’ to show I’ve finished entering the program:

```
20: END
```

Having done this the editor returns to the console menu, where the user can then chose to execute the code they have just written.

It is also possible to download all the code to a PC, and upload code back to the device. Currently it is not possible to edit the downloaded code, because it is in reverse polish format, however it would be a simple job to write a converter to take the reverse polish and convert it
to BASIC. The user could then edit the code and use another converter to put it back to reverse polish, ready to be uploaded to the device.

**Executing Code:**

Once a program has been entered, as above, it can then be run. Each of the four user virtual consoles is capable of both entering and executing code. Executing code is achieved by first selecting ‘Run Code’ from the console menu, and then selecting the process to run:

```
Run Proc: 1
```

Any of the consoles can run any of the user programs. The same user program can be run on multiple consoles at the same time without risk of corruption, as each console has its own private environment.

The result of running a program depends entirely on what the user wrote, but here are a couple of examples of common displays:

Asking the user for input (executing ‘INPUT A’ for example):

```
1
```

The cursor will flash over the number, and the UP and DOWN keys can be used to increase or decrease the value. When the required value is displayed, SELECT is then pressed.

Values that have been ‘printed’ will be displayed directly. Executing the code written earlier will produce:

```
5
```

---

**Putting it all together:**

Once the hardware and software had been developed, it was then decided to try and make a realistic size version, to give a better feel for using the device in action. This smaller version required two new parts – a proper circuit board, and suitable case. Each of these will now be discussed in detail:
Circuit Board:
As the ATmega103 microcontroller is a surface mount chip, this meant the circuit board had to be able to accommodate it. Normally a schematic of the circuit would have been drawn, which after net-listing would then have been laid out onto copper, however due to the limited time constraints it was decided to go straight to the copper layout stage and layout out the connections by hand. This carries the risk of human error, as the net-list allows a computer to ensure all the connections are correct, fortunately as the board was not too complicated, it is hoped the final board will only contain minor errors.

The board was laid by hand out using the ‘ARES IV’ PCB software. This is the final copper layer used to make the actual circuit board:
**Casing:**

The aim of the casing was to try and get the device into as small a package as reasonably possible. It was soon clear that I didn’t have the skills to make the casing myself, and Imperial College did not have suitable facilities either. The only way to solve the problem was to use an outside contractor. Fortunately I had contacts with a manufacturing company - ‘Vosper-ManTech’ [15], who kindly agreed to help at short notice.

After discussing the constraints with them, we decided that the smallest case we could use in the time, would be a plastic case with dimensions: 117x70x25mm. One of the main problems was the BNC connectors – they took up far too much room, and would have forced the use of a much bigger case. On advice from Vosper-ManTech I decided to switch to miniature SMA connectors, which fitted in easily and allowed a very shallow depth case to be used. If the case had been designed from scratch, then full BNC connectors could have been used, but the SMA gave the best effect in the time available.

Once the basic layout had been agreed, I then developed an accurate layout so that the box could be milled to my exact specifications.

This is the drawing was used to create the casing:
**Silk-Screen Printing:**

Once the position of the components was decided the next stage was design the labels to go onto the top of the case, to show what the purpose of each switch and port is. *Vosper-ManTech* have on-site silk-screen facilities, and have agreed to arrange for the case to be silk-screen printed. I provided the following layout for the silk-screen mask:

![Silk-Screen Layout](image)

**Summary:**

Once all the hardware and software had been developed and tested, the casing designed and milled, and the silk-screen printing completed, the final effect should be very impressive. Due to the time taken to outsource the case building, it is not possible to show a picture of the final device here, however it is expected that the device will be completed within a week of this report being completed.

The next two chapters discuss the hardware and software respectively in detail, then chapter 6 draws conclusions on how successful the project has been, and looks at its commercial potential.
Chapter 4 – Hardware:

This chapter covers in depth the hardware systems used within the concept demonstrator.

*Introduction:*

The hardware consists of a number of individual components which are linked together to produce the overall device. The following block diagram shows the high-level layout:

Each hardware component is supported by a software driver (called a ‘service’), details of which are in the following chapter. The rest of this chapter will take each hardware component in turn and describe how it is used.

*Ports:*

The M G B $\ddagger$ has three ports to connect to the outside world. They are as follows:

<table>
<thead>
<tr>
<th>Port</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>9-way D-type</td>
<td>Serial lines, Digital I/O lines</td>
</tr>
<tr>
<td>BNC</td>
<td>Analogue Input</td>
</tr>
<tr>
<td>BNC</td>
<td>Analogue Output</td>
</tr>
</tbody>
</table>
Microcontrollers:

**Background:**
Microcontrollers can be best described as a ‘computer-on-a-chip’, as they consist of a CPU, program and data memory, Input / Output lines and often a serial UART, all on a single piece of silicon. They tend to run at low speeds (up to 10MHz is common) and have very low power consumption. Microcontrollers are designed for applications that require a small amount of processing coupled with direct I/O requirements for control purposes, they were not designed to run more demanding applications.

Within the last year the power and functionality offered by the leading-edge microcontrollers has increased considerably. Until now it has been commonplace to use microcontrollers with 8k of program memory and a few bytes of SRAM and perhaps a few bytes of EEPROM. However many manufacturers have recently introduced far more powerful devices.

In spring 1998, *Atmel*[1], introduced the first of a new range of high-performance microcontrollers, called their ‘ATmega’ range, offering many features beyond those offered by their competitors. After surveying the market, it was clear that the top-of-the-range 8-bit ATmega103 was the obvious choice, and so it was selected to be the core processing facility for the concept demonstrator. It provides the following facilities (taken from the ATmega103L datasheet [4]):

- 2-stage pipelined CPU
- 128K FLASH program memory
- 4K SRAM volatile data memory
- 4K EEPROM non-volatile data memory
- Programmable Serial UART
- 32 I/O, 8 I, 8 O TTL lines
- 64-pin TQFP Surface Mount Package

The aim is to use this microcontroller in a new way – to run a fully multi-tasking operating system, with a programming language and driver code to support the various hardware components.

**Use:**
Initially the surface mount package aspect of the microcontroller provided a obstacle to be overcome – SMDs at this size are very difficult to hand-solder, thus making a development board hard to produce. The solution came from *Kanda-Systems*[3], a microcontroller
specialist who were able to provide carrier boards with a ATmega103L already mounted, and connections to four banks of 2x8 0.1” spaced pins.

The ATmega103L is able to use timing crystals between 0 and 4MHz. Initially it was thought that the faster the better, but the datasheet showed that a crystal of 3.6864MHz would divide down exactly to all the common baud rates for the on-board UART. As it was only slightly slower than the maximum allowable, it was decided to use this frequency.

The microcontroller is powered from a +3.3V supply, which is provided by using an +5V LM7805 voltage converter, and then using a zener diode/resistor arrangement to drop the +5V down to the required +3.3V.

**Programming:**
There are two different ways to program the FLASH memory on the AVR microcontrollers – by using a separate programmer, or by using the in-circuit programming facility. As the uC would be programmed many times, it was decided that in-circuit programming would be the most flexible choice, removing the need to continually plug and unplug the chip, reducing the possibility of damaging it in the process.

The 128K FLASH program memory is programmed by using the Serial Peripheral Interface (SPI) protocol. The protocol uses three lines – Serial Clock (SCK, pin 11), Programming Data Input (PDI, pin 2) and Programming Data Output (PDO, pin 3). To enable programming, the ‘RESET line (pin 20) must be held low, while the protocol pins are toggled appropriately.

Programming consists of placing the data, one bit at a time onto the PDI line, providing a negative edge on SCK, and finally checking the result by monitoring the PDO line.

All four lines are connected to the Parallel port on a PC, via a suitable cable and the programming software ensures the lines are correctly pulsed. Details of the software used is provided in the next chapter.

---

**LCD screen:**
Due to the low power and size requirements of the display for the demonstrator, it was immediately clear that only LCD-type screens would be suitable. Deciding whether to use a graphics-based or text-based display was more difficult.
As it is desirable to display sampled waveforms on the device, a graphics-based display would have been the best solution, especially as many graphic displays can also be programmed to display text as well. However, researching the area showed a number of problems:

- Standard graphic screens tend to be large and bulky compared to their text-based equivalents.
- No one single ‘official’ protocol exists for driving graphics screens. Although there are a number of different standards, none has become the ‘de-facto’ standard yet.
- The software control is much more complex (as would be expected).

These reasons suggested that a compromise should be made, and so I selected a Hitachi HD44780-type [2] 16 character by 2 line text-based display for the project.

**Screen Control:**
The Hitachi HD44780 protocol allows the choice between using 8-bit or 4-bit modes, the latter mode requiring a double nibble to be sent for each byte. The main advantage of the 4-bit mode is that four less I/O lines are required. As I may need the extra lines later, I decided to use the 4-bit mode.

Independent of the physical screen layout, all HD44780 controllers assume a screen memory of 40 characters by 2 lines, with smaller screens having the facility to scroll to see the extra characters. The screen controller does allow bytes to be read as well as written to the screen memory, and so it would be possible to use the extra \((40*2) - (16*2) = 48\) bytes as additional memory.

Although accessing this extra memory is possible, the 4K SRAM available on the microcontroller should be enough, therefore I have decided to only write to the display, not read from it. This means I can tie the display’s Read/Write (R/W) line (pin 5) to GND, saving another I/O line.

Two different types of data can be sent to the display – character data, and control data. The type of data being sent is identified by the Register Select line (pin 4). This line will need to be controlled from software.

Finally, to clock data into the display, the Enable (E) line (pin 6) must be pulsed, while valid data is on the data lines (pins 7-10).
**Backlight:**
To aid readability, the LCD displays can be chosen with or without a backlight. Selecting the backlight option increases the cost (by around 30%), but makes the display readable at night and in dark areas. As the integrated device is designed to be used in inaccessible locations, the extra cost is warranted.

Power for the backlight is separate from the main LCD power supply, via an additional line (pin 15). The grounds for both parts are common. The backlight on the LCD display I am using requires +5V, and also has a contrast option, provided through an external variable resistor. After experimenting with the contrast, I found that once set, it didn’t require alteration during use. I therefore selected a 10K preset, which allows initial adjustment, without taking up too much board space.

Powering the backlight uses a considerable amount of power, especially when the whole device will be running from batteries. To allow the user to save power, in conditions where the display is readable without the backlight turned on, a facility to turn the backlight on and off from a menu option was needed.

To achieve software control of the backlight, one I/O line from the microcontroller is connected to a transistor that switches, via a resistor, the separate +5V backlight power supply. By making the resistor also a preset (4.7K) I am able to control the amount of current the backlight can use, allowing the brightness to be controlled. It would be possible to control the brightness level through software as well (with a small number of extra external components), but through testing, I found that once set, the brightness didn’t need to be adjusted again during operation, so for this version of the demonstrator the software only controls whether the backlight is on or off.

This is the circuit used to control the backlight.

```
+5V
Q1
RV1 4.7K
BC107
PDU
LCD Pin 16
```
The following table summarises the connections required to the LCD:

<table>
<thead>
<tr>
<th>Line:</th>
<th>Pin Number:</th>
<th>Connection:</th>
</tr>
</thead>
<tbody>
<tr>
<td>GND</td>
<td>1</td>
<td>GND</td>
</tr>
<tr>
<td>V_{cc}</td>
<td>2</td>
<td>+5V</td>
</tr>
<tr>
<td>RS</td>
<td>4</td>
<td>PB2</td>
</tr>
<tr>
<td>R/W</td>
<td>5</td>
<td>GND</td>
</tr>
<tr>
<td>Enable</td>
<td>6</td>
<td>PB3</td>
</tr>
<tr>
<td>D0</td>
<td>7</td>
<td>PB4</td>
</tr>
<tr>
<td>D1</td>
<td>8</td>
<td>PB5</td>
</tr>
<tr>
<td>D2</td>
<td>9</td>
<td>PB6</td>
</tr>
<tr>
<td>D3</td>
<td>10</td>
<td>PB7</td>
</tr>
<tr>
<td>Backlight Power</td>
<td>15</td>
<td>+5V via transistor</td>
</tr>
<tr>
<td>Backlight Contrast</td>
<td>3</td>
<td>10K preset</td>
</tr>
</tbody>
</table>

**Indicators:**

Two LED indicators have been included in the design, mainly for testing purposes, but also to show the device is functioning correctly during normal use. They are:

<table>
<thead>
<tr>
<th>Colour:</th>
<th>Use:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Green</td>
<td>Power-On indicator. Lit continuously when correct power is applied.</td>
</tr>
<tr>
<td>Red</td>
<td>Software controllable, in testing one processes flashes this LED on and off every half a second, showing the Operating System is running correctly.</td>
</tr>
</tbody>
</table>

The green LED is powered from across the +5V power supply via a current limiting resistor (220R), and the red LED has it’s power sunk straight from an I/O line (PD6) on the microcontroller.
Keypad:

**Background:**
One of the aims of the M GB s! is to make it as portable as possible. Having a full-size keyboard is therefore ruled out. The decision is how few keys can be provided, while still maintaining easy use. As most options will be via on-screen menus, UP, DOWN and SELECT buttons appeared to be the best choice.

During development it became clear that these three buttons on their own were not enough, as access to higher-level control (provided by the OS) was required. For example when running a menu based application, there needs to be a way to switch to another process. The solution was to add a fourth button (the SYSTEM button) which when pressed gives access to a system menu, allowing the current process to be changed, as well as any other higher-level functions that may be required later on (such as changing the priority level of a process).

**Layout:**
The layout was almost predefined by the buttons themselves, UP must be above DOWN, and SELECT should be to the right. There were a number of options of where to place the SYSTEM button, and after experimentation it was found that having it slightly above and to the right of SELECT was the most convenient position:

![Keypad Layout](image)

**Use:**
The ATmega103L has internal, programmable pull-up resistors on all its I/O lines. Using this option saves a number of external resistors, reducing the board space required and making the device lighter.

Each switch is connected between the microcontroller’s I/O line (which is programmed to be an input via software) and GND. Note that there is no hardware de-bouncing on any of the switches – this is achieved through an Operating System service.
The following table shows how the switches are wired:

<table>
<thead>
<tr>
<th>Switch:</th>
<th>uC pin:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Up</td>
<td>PD2</td>
</tr>
<tr>
<td>Down</td>
<td>PD3</td>
</tr>
<tr>
<td>Select</td>
<td>PD4</td>
</tr>
<tr>
<td>System</td>
<td>PD5</td>
</tr>
</tbody>
</table>

Sound:

**Background:**
Donald Norman writes of user interfaces that ‘there must be adequate and appropriate “system feedback”’ [26] to help the user know what is happening. The M G B sf will be able to provide audio feedback, as well as visual feedback, thus aiding the usability of the device. The frequency and duration of the sound will be controllable from software, and will be provided as one of the OS services, thus any process will be able to produce sound.

The point of having a sound facility in the M G B sf is not to for musical effects, but to allow the user to program an alert bleep if a problem arises.

**Use:**
Due to the limited amount of space and power available, a normal speaker was not an option. The next best alternative was a piezo buzzer. These are very thin, while still producing a reasonable amount of sound. Their main disadvantage is that only certain frequencies can be produced. However, this was not considered a disadvantage in this application.

To make the piezo buzzer vibrate (and thus make a noise) requires the buzzer lines to be toggled. The obvious solution is to connect it straight to an I/O line on the microcontroller and get software to pulse the line at the required speed. Unfortunately the 3.3V that the microcontroller can provide is not enough to make the buzzer vibrate, thus a transistor was therefore required to switch a +5V supply on and off. Testing showed this produced a sound level that could be easily heard, without being too loud.
I/O line PD7 on the microcontroller is used to control the buzzer.

---

**Serial Port:**

**Background:**
Serial ports are almost universal on all modern computers, and are becoming more common on test equipment as well. Thus compatibility with the serial RS-232 standard was considered essential, to allow the device to communicate with as many other devices as possible.

**Technical:**
The ATmega103L has a built-in UART capable of producing TTL level RS-232 signals on two of it’s I/O lines. Having already selected a crystal frequency of 3.6864MHz (which divides down to the common baud rates exactly), using this UART should have been easy.

However, the designers of the microcontroller decided to put the Transmit and Receive lines onto the same pins as used for programming the device. As I am using the in-circuit method for programming, these lines are already used. The only solution would be a 2-way multiplexor on the pins. Due to the added complexity and risk of programming difficulties, it was decided reluctantly to use a software UART and to chose a couple of more suitable I/O lines that are free.

Full details of the software UART are provided in the next chapter. Although both the software and hardware UARTs produce the correct timing, they can only provide 0/3.3V
signals. It is therefore necessary to use an external voltage conversion chip to get the voltages to within the correct RS-232 specification.

The chip selected is the MAX233CPP from Maxim. It offers the following facilities:

- Single +5V power supply
- 2x TTL – RS-232 converters
- 2x RS-232 – TTL converters
- No external capacitors

This chip was chosen mainly because of its requirement for no external capacitors, this does make it over twice as expensive as its rivals, but it uses much less board space and is more reliable as no externals connections are used (except for wire links).

The aim of the concept demonstrator is to show the principles rather than spend time perfecting the physical issues. In keeping with this ideal, it was felt that adding handshaking lines would not add enough benefit to warrant the extra time required to implement them. Because of this, the serial communications on the device are limited to 4800 baud. Although this sounds slow, tests have shown that this is fast enough to show how the device might be used in practice.

**Port:**
A 9-way D-type socket is provided on the device, which is wired as follows:

<table>
<thead>
<tr>
<th>Port Pin Number</th>
<th>Serial Line</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Tx</td>
</tr>
<tr>
<td>3</td>
<td>Rx</td>
</tr>
<tr>
<td>5</td>
<td>GND</td>
</tr>
</tbody>
</table>

This port is also used jointly with the digital I/O hardware.

**Digital I/O:**

**Background:**
The device is designed to be able to both sample, store and detect digital lines, as well as generating new lines.
**Technical:**
The digital I/O lines are perhaps the easiest part of the device – they are simply connected straight from the digital I/O port to a set of I/O lines on the microcontroller. All control (including setting whether a line is an input or output) is achieved through software.

This table summarises the lines used:

<table>
<thead>
<tr>
<th>Port Pin Number:</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 (Ground Pin)</td>
</tr>
<tr>
<td>6 (Digital I/O line)</td>
</tr>
<tr>
<td>7 (Digital I/O line)</td>
</tr>
<tr>
<td>8 (Digital I/O line)</td>
</tr>
<tr>
<td>9 (Digital I/O line)</td>
</tr>
</tbody>
</table>

---

**A/D Converter:**

**Background:**
One of the most important aspects of the device is the ability to sample analogue signals to either store them for later analysis or to display the waveforms on the LCD screen.

**Technical:**
The ATmega103L comes with a built-in Analogue to Digital Converter (ADC), which is prefixed with an automatic sample-and-hold circuit. The microcontroller also has an 8-way multiplexor, allowing eight signals to be sampled, one after another. The concept demonstrator will only sample a single signal to show the theory.

The ADC offers these facilities [4]:

- 10-bit resolution
- 70 – 280 µS conversion time
- Up to 14kSPS (14,000 samples per second)
- 8 multiplexed input channels
- Interrupt on ADC conversion completion
Normally it is very important to ensure that analogue signals are kept separate from digital, and to ensure that separate correctly-decoupled power supplies are used to reduce noise, however due to time constraints these aspects haven’t been covered in this project.

For simplicity the device only uses the 8 most significant bits, as this allows a complete sample to be stored and transmitted as a single byte. It is likely that because no proper grounding / decoupling is used, the lower couple of bits may be in the noise floor anyway.

The ADC can be set to cause an interrupt on completion, though due to the complexity of interrupt handling with the OS, the demonstrator will poll the ADC status register to detect completion. This will slow down the maximum sample rate, but will make the software service easier to design.

The ADC requires a separate power supply to be connected to its own $V_{cc}$ (pin 64) and its own GND (pin 63). Additionally a reference voltage is used by the ADC which must be provided on $A_{ref}$ (pin 62). In practice $A_{ref}$ and $V_{cc}$ are connected via a wire link. Although a separate power supply is recommended, the ADC $V_{cc}$ and A/D GND were commoned with those of the microcontrollers main supply.

The control for sampling data is done entirely through a software service, which is described in the next chapter.

**Port:**
The ADC input (pin 61) is connected to the core of the input BNC connector on the M GB $\star\dagger$, and the BNC tag is connected to GND.

---

**D/A Converter:**

**Background:**
One of the key advantages the M GB $\star\dagger$ will provide, is it’s ability to provide temporary solutions, many of which will require an analogue output facility. The Digital-To-Analogue Converter (DAC) will take digital data that is either stored in memory, or generated on-the-fly and convert it to an analogue signal, to be output via a BNC connector to the target circuit.
Technical:
No internal DAC is provided on the ATmega103L, so an external DAC has to be added. A wide range of DACs are available, but for demonstration purposes it is best to match the number of DAC bits with the number used by the ADC, therefore the DAC must provide at least 8-bits of resolution.

The next choice is whether to use a parallel or serial-driven converter. The serial-converters tend to come in smaller 8-pin DIP packages, against the larger 24+–pin DIP packages of the parallel types. Using a serial converter has the overhead of additional control to load the DAC register, but as a microcontroller will drive the DAC, this isn’t considered to be a problem. Research showed that the most appropriate DAC appeared to be the AD7391 from Analog Devices. It’s datasheet [5] describes it’s functions:

- Single supply - +2.7V to +5.5V operation
- 10-bit resolution
- Serial Interface
- Micropower – 100µA

As the board already has +3.3V and +5V, it is clearly better to select one of these to power the DAC. +3.3V was selected to allow the output from the DAC to be fed back into the ADC for testing and demonstration purposes, if 5V had been used instead, there would have been a risk of damaging the ADC with too high a voltage. Although this limits the range of voltages the DAC can output, this isn’t a problem for demonstrating the concept.

Like the ADC, the AD7391 requires a reference voltage, which in this project has been tied to Vcc (3.3V). As well as the supply lines, the chip also uses four control lines [5] for entering data. A complete listing of all pins is shown in the following table:

<table>
<thead>
<tr>
<th>Line</th>
<th>Pin</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘LD</td>
<td>1</td>
<td>Load Strobe. Transfers shift register data to DAC. This causes the analogue value to change at the output.</td>
</tr>
<tr>
<td>CLK</td>
<td>2</td>
<td>Clock Input. Positive edge clocks data into shift register</td>
</tr>
<tr>
<td>SDI</td>
<td>3</td>
<td>Serial Data Input. Data loads directly into shift register</td>
</tr>
<tr>
<td>‘CLR</td>
<td>4</td>
<td>Resets DAC to zero.</td>
</tr>
<tr>
<td>GND</td>
<td>5</td>
<td>Ground</td>
</tr>
<tr>
<td>VOUT</td>
<td>6</td>
<td>Voltage output</td>
</tr>
<tr>
<td>VDD</td>
<td>7</td>
<td>Power supply</td>
</tr>
<tr>
<td>VREF</td>
<td>8</td>
<td>Reference voltage</td>
</tr>
</tbody>
</table>
When the M G B 51 powers up, there is no need for the DAC to be zero, if this is required, the microcontroller can initialise the DAC to zero, by loading a zero value during the boot-up sequence (which is exactly what is actually done). This means that the ‘CLR line can be tied high, freeing another line.

**Loading Data:**
The DAC consists of two registers – a register which is used to output the voltage, and a shift register, which is used to load the next value in. When the required value has been loaded into the shift register, the load strobe line is pulsed and the new value is moved into the main DAC register, and the analogue output voltage generated.

Data is clocked into the shift register, one bit at a time using the SDI line. Every positive edge of the CLK, shifts the bit into the register. The device has a particularly nice feature in that you can clock in as many bits as you want, but only the last 10-bits are used. This allows a microcontroller to send two bytes with the real data in the 10 least significant bits. This also makes it easy to upgrade to a 12-bit device at a later stage, without having to rewrite the control software.

Once at least 10 bits of data have been clocked in, ‘LD is strobed low, causing the value to appear at V_{out}.

**Port:**
The DAC output (AD7391/pin 6) is connected to the core of the output BNC connector on the M G B 51, and the BNC tag is connected to GND.

---

**Testing:**
Each of the above components were tested by writing a separate test harness to check the hardware behaved as expected, and to prove the software was driving it correctly.

The early test harnesses just used an LED output to allow debug information to be conveyed, but once the LCD routines had been written, they were used in all following test harnesses to allow proper debug information to be displayed. This proved invaluable in locating difficult errors, such as input values which were not quite right, but were very close.

Once a component worked in it’s own private single-tasking environment, it was then added to the multi-tasking environment of the operating system. It was often that code had to be
slightly modified to work correctly under these conditions. For example, sometimes interrupts have to be disabled to allow a high-speed time-critical item of hardware to work.

One of the early problems I came across was that of mis-programming the microcontroller. Sometimes the programmer would fail (for example if the chip was not correct seated) and then it was very difficult to tell whether it was my code that wasn’t working or whether the chip hadn’t been programmed properly.

To solve this problem I borrowed an idea from a hardware development board I had used before. Part of it’s demonstration was to flash an LED, and they called this the ‘heartbeat’.

This turned out to be an ideal solution to my problem - by simply selecting one of the pins on the microcontroller (PD6) to act as the ‘heartbeat’, it work’s as follows: when the AVR is waiting for a command, it pulses the heartbeat pin and this can then be picked up on an oscilloscope. Making it a simple matter to detect a dud programming session.
Chapter 5 – Software

This chapter covers the software used for driving the M G B 51, in particular it discusses the operating system and the programming language provided.

Introduction:
The software component of the M G B 51 is the most important part, as it gives the device it’s flexibility, and moves it away from the standard fixed test equipment arena into the next generation of programmable support devices.

The software is split into three distinct categories:

- Operating System:
  - Process control
  - Services (device drivers)

- System Processes:
  - Program editor
  - Menu system
  - Volt-Meter, Scope, Sampler

- Programming Language:
  - External language
  - Internal representation

As well as these, a number of tools for creating, simulating and downloading code onto the microcontroller are also required. As these are needed to create the above components, they will be discussed first.

Before covering the software in depth, it is worth pointing out again one of, if not the, most important difference between developing code for a microcontroller and a microprocessor. Microcontrollers, in particular the ATmega103L, tend to have Harvard architectures. This means that they use a physically separate bus for user data and program data. This combined with a FLASH program memory is a big disadvantage, as it means that once code is written it cannot be dynamically changed as the microcontroller executes.
This immediately rules out any possibility of writing a compiler to compile user code into AVR byte-code to be executed natively. Thus the only option is to write an interpreter.

---

**Microcontroller Programming:**

When I started this project, because of the very recent launch of the AVR range of microcontrollers, there were no reasonably priced C-compilers available (only one was available, costing £2000). Within the last months of the project, a number of cheaper C-compilers have become obtainable, but they came too late for me to use them. It is interesting to note that all the demo versions of the compilers I have seen, have implemented their own stack frame for function calls. This may have caused problems, as I use multiple context stack frames myself, which are dynamically switched during execution. Reliably switching three separate stacks for each process, would not only have been complex to program, but would also have been extremely difficult to ensure that the switching was work correcting (two stack frames are hard enough).

Due to the above reasons, all code is in AVR assembler. Ideally future versions would be in C, with perhaps the exception of the core of the OS, which may still need to be written in assembler, for both speed and due to it’s design.

**Assembler:**

I used the assembler provided by the manufacturers[1], called wavrasm[6] version 1.30. It provides a basic editor and assembler and can produce a range of different output formats. As the programming software I had available required Motorola S-format files, this was the format I instructed the assembler to produce.

A number of assembler directives are available, including ‘.include “filename.asm”’, which allows file inclusion. This turned out to be a vital feature, as the editor is only able to edit files up to approximately 20K. I do not understand why such a small limit is used, but it does, in effect, force the code to be modular, which promotes good design, so this wasn’t a serious problem.

**Simulator:**

The FLASH program memory on the microcontroller has a limit to the number of times it can be written of around 1000 writes. When developing code, this is a serious, and costly limitation. The solution is to use the software simulator, provided by the manufacturers [1].
AVR Studio Version 1.50 [7] is capable of simulating the complete AVR range, including the ATmega103L used in this project. It allows code to be stepped through, showing the state of all registers and the processor at each stage. This is very useful to locating and correcting loops, which are difficult to debug on an embedded microcontroller.

Another particularly useful feature is the accurate time calculations it allows. These were firstly used to ensure that the operating system was giving each process the correct amount of time, and secondly to check that the timing service provided by the operating system is as accurate as possible.

However, the simulator did cause many hours of extra work, due to it not being fully debugged. The main problem was that it assumes that memory is infinite in both directions – for example it is possible to go to the bottom of SRAM, decrement the pointer by one and the simulator doesn’t wrap the access, as the real chip does. Thus often I got different results on the simulator and the chip. I also found that an early version of the simulator had a bug where the Program Counter was decremented by 3, instead of 2, when a interrupt routine was called. This resulted in the wrong return address being picked up, and my OS appeared to fail because of this, even though it worked perfectly on the real chip.

The simulator is a very useful tool, but it must not be totally relied upon.

**Programming:**

The base of the microcontroller downloading software has been kindly provided by Brent Crosby, an AVR developer in Utah who has written C code to program the ATmega103L via a parallel port. Although his code provided all the routines to toggle the lines, and to read in the Motorola S-format data files, it used a slightly different wiring configuration to the one I had used, and the polled timing delays were too short for my own PC.

It didn’t take long to modify his code, and I was soon able to download sample applications to the microcontroller. Most of my code is under 4K long, and so downloading the full 128K each time started to slow my development time. I therefore modified his code again so that only 4K is downloaded, reducing the time by over 90%.

I could have written this code myself, but it is a complex algorithm, and would have taken me several week’s to get it working correctly, I was therefore very grateful to Brent Crosby for providing the basis, which I could then modify and use.
Operating System:

The Operating System (or OS for short), is the key software component. It allows multiple processes to run simultaneously together, and it also provides the various services required by both the system processes and user processes. In particular it provides a timing mechanism allowing processes to be suspended for a specific length of time.

**Overview:**

The OS provides the following main features:

- **Pre-emptive multitasking** – each process in turn, is given a short amount of time to execute, giving the effect of many processes running simultaneously. Pre-emptive refers to the fact that control is pulled from each process when it’s time has expired, thus stopping any one process from taking up all the CPU’s time.

- **Priority scheduling algorithm** – this allows different processes to have different amounts of time to execute, thus a low priority clock process, could have 10% of the CPU time, while a high priority real-time data gathering process may have 60% of the time.

- **Services** (or device drivers) – these allow safe access to the hardware components, such as the serial port. They stop processes being able to access the hardware directly - in this way, the OS has more control over what goes on, and can ensure the system runs as reliably as possible.

The operating system will be based on the following architecture:

![Operating System Architecture Diagram](image)

The next section will discuss how processes are controlled, then the initialisation completed by the OS on boot-up will be explained, and finally all the services provided will be listed.
**Process Control:**

Control of processes is based around a process table, which has an entry for each process. The operating system executes each process in sequence, for a short amount of time to provide the multi-tasking functionality. The process table is stored in SRAM.

The process table is used to store all the information required for each process. Each process has its own stack which is unusually stored *inside* the process table. Initially this may appear a risky strategy – if the stack overflows then the process table becomes damaged and no part of the system can function. However, if the stack was in a different part of memory, it could still overflow leading to just as high a risk as before. By keeping the stack in the process table, it makes it very easy to context-switch, improving the efficiency and elegance of the context-switcher.

Each entry consists of a 128 byte block, starting at $0100 for process 1, $0180 for process 2, $0200 for process 3, etc. The following table shows an entry for a particular process in the process table:

<table>
<thead>
<tr>
<th>Offset Address:</th>
<th>Value:</th>
<th>Purpose:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$00</td>
<td>PC Low</td>
<td>Program Counter Low Byte</td>
</tr>
<tr>
<td>$01</td>
<td>PC High</td>
<td>Program Counter High Byte</td>
</tr>
<tr>
<td>$02</td>
<td>SREG</td>
<td>Status Register</td>
</tr>
<tr>
<td>$03</td>
<td>SP Low</td>
<td>Stack Pointer Low Byte</td>
</tr>
<tr>
<td>$04</td>
<td>SP High</td>
<td>Stack Pointer High Byte</td>
</tr>
<tr>
<td>$05</td>
<td>Priority</td>
<td>Priority Level, used to determine how long the process is allowed to execute</td>
</tr>
<tr>
<td>$10 - $30</td>
<td>Register Base</td>
<td>Location for storing all registers</td>
</tr>
<tr>
<td>$4F</td>
<td>Stack Top</td>
<td>Stack grows down from this location</td>
</tr>
<tr>
<td>$50-$7F</td>
<td>Display Data</td>
<td>LCD routines write to this area, when the process is not the Active Process (AP).</td>
</tr>
</tbody>
</table>

In this version of the operating system, processes are statically defined in advance, rather than being created dynamically. Dynamic creation would be the preferred option, but it would have considerably increased the complexity of the scheduling algorithm, and so was rejected.

The process table is defined in advance, and a global variable, ‘maxProcesses’ determines how many processes are in the table. When the context-switcher gets to the last process, it loops back and runs the first process again, in a Round-Robin style schedule.
The user is able to select the process that they wish to interact with, and this is known as the ‘Active Process’ or AP, and is allowed direct access to both the LCD and the keypad. A service can check whether it should allow direct access, by comparing the AP with the current process number (held at location $0080$), if equal then the operation is allowed, otherwise it is either blocked or deferred.

An example of deferred access is used by the LCD service. If a process is not active it may still be trying to update the screen, and the user will expect the screen to have been updated, when they switch to that process. To achieve this, if a non-AP process calls the LCD service, the data is instead written into the display data block (offset $50$) in the process table for that process. Then when the user switches to the new process, the current display data held in the process table for the new process is written to the LCD (bypassing the LCD non-AP routine, otherwise the system process screen would be corrupted) just before the new process is allowed to run.

To test whether the processes were being switched at the correct time, I used the cycle time facility on the Simulator [7]. By setting a breakpoint on the timer interrupt handler, I could monitor the time each process was being given to run. After several runs I found that the switch was happening when it was supposed to.

The following table shows the results from running 7 processes (note that process 3, the main menu, is run first):

<table>
<thead>
<tr>
<th>Process</th>
<th>Priority</th>
<th>Cycle 1:</th>
<th>Cycle 2:</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>$20</td>
<td>8.89 (8.89)</td>
<td>62.22 (8.88)</td>
</tr>
<tr>
<td>4</td>
<td>$10</td>
<td>13.34 (4.45)</td>
<td>66.67 (4.45)</td>
</tr>
<tr>
<td>5</td>
<td>$20</td>
<td>22.22 (8.88)</td>
<td>75.56 (8.89)</td>
</tr>
<tr>
<td>6</td>
<td>$20</td>
<td>31.11 (8.89)</td>
<td>84.45 (8.89)</td>
</tr>
<tr>
<td>7</td>
<td>$20</td>
<td>40.03 (8.92)</td>
<td>93.34 (8.89)</td>
</tr>
<tr>
<td>1</td>
<td>$20</td>
<td>48.89 (8.86)</td>
<td>102.22 (8.88)</td>
</tr>
<tr>
<td>2</td>
<td>$10</td>
<td>53.34 (4.45)</td>
<td>106.67 (4.45)</td>
</tr>
</tbody>
</table>

A higher priority value indicates a higher priority process. All times are in mS. The times in the cycle columns show how long since the OS started before a particular context-switch occurred, and the time in brackets show how much time each process was given. It is clear that the two processes with lower priority get half as much time as the higher-priority processes.
The operating system also uses a general block of information stored in SRAM between $0070$ and $00FF$:

<table>
<thead>
<tr>
<th>Absolute Address:</th>
<th>Value:</th>
<th>Function:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0000$ - $0060$</td>
<td>System registers</td>
<td>Used internally by the microcontroller for hardware access. Not allocable by user code.</td>
</tr>
<tr>
<td>$0070$</td>
<td>Timer Base</td>
<td>Each process has it’s own counter in this area, which can be used by calling the delay service.</td>
</tr>
<tr>
<td>$0080$</td>
<td>Current Process Number</td>
<td>Used by the OS to determine the next process to run.</td>
</tr>
<tr>
<td>$0081$ - $0084$</td>
<td>Temp variables</td>
<td>Temporary variables required by the context-switcher. (Timer 0)</td>
</tr>
<tr>
<td>$0090$ - $0094$</td>
<td>Temp variables</td>
<td>Temporary variables required by the timing interrupt. (Timer 2)</td>
</tr>
</tbody>
</table>

One disadvantage with microcontrollers is that it is not possible to control system access to memory, because it is not possible to intercept a memory request before it is executed. This creates a problem as it means one system process could in theory overwrite another system process’s memory. It would also have been useful in that the stacks could have been protected, stopping them from overwriting the process table if they overflow. Although system processes cannot be protected in this manner, user processes can be. All memory accesses go through the User Code Execution Unit (UCEU), where they could be intercepted and range-checked. This is a feature that would be worth implementing in future versions.
**Initialisation:**
The first job of the OS is to initialise the microcontroller into a stable state, and start the first process running. To achieve this, it goes through a boot-up sequence, consisting of the following steps:

1. Disable global interrupts
2. Initialise stack pointer (to allow initialisation routines to call functions)
3. Initialise the I/O lines to be Input or Output
4. Turn on the programmable pull-up resistors, where required
5. Initialise the process table, setting up the standard process entries
6. Initialise the DAC, and set the analogue output to 0V
7. Initialise the ADC, by selecting the correct channel, and enabling conversions
8. Initialise Timer0 interrupt handler, used for process switching
9. Initialise Timer2 interrupt handler, used for timing delays
10. Select initial process to run, by posting it’s id into SRAM
11. Load in stack space of first process
12. Load status register of first process
13. Set the process as Active (giving it access to the LCD and keypad)
14. Enable global interrupts
15. Jump to the first process
**Memory Map:**

The ATmega103 has three areas of memory:

<table>
<thead>
<tr>
<th>Type</th>
<th>Amount</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLASH</td>
<td>128Kbytes</td>
<td>Program memory</td>
</tr>
<tr>
<td>SRAM</td>
<td>4Kbytes</td>
<td>Fast volatile user memory</td>
</tr>
<tr>
<td>EEPROM</td>
<td>4Kbytes</td>
<td>Slow non-volatile user memory</td>
</tr>
</tbody>
</table>

The operating system, services, programming language and user code execution unit are all fairly compact, and only require the first 4K of program memory to be used, allowing much more functionality to be easily added later on. The SRAM and EEPROM are laid out as follows:

**SRAM**

<table>
<thead>
<tr>
<th>User Code 8 Area</th>
<th>$0000 - $00FF</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Code 3 Area</td>
<td>$0A00 - $0AFF</td>
</tr>
<tr>
<td>User Code 2 Area</td>
<td>$0900 - $098F</td>
</tr>
<tr>
<td>User Code 1 Stack (grows downwards)</td>
<td>$0806 - $085F</td>
</tr>
<tr>
<td>User Code 1 Variables</td>
<td>$0800 - $07FF</td>
</tr>
<tr>
<td>Sample Data Area</td>
<td>$0600 - $047F</td>
</tr>
<tr>
<td>Process 7</td>
<td>$0400 - $047F</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>Process 3</td>
<td>$0200 - $027F</td>
</tr>
<tr>
<td>Process 2</td>
<td>$0180 - $01FF</td>
</tr>
<tr>
<td>Process 1</td>
<td>$0100 - $017F</td>
</tr>
<tr>
<td>Timer and Temp Vars</td>
<td>$0094 - $0070</td>
</tr>
<tr>
<td>System Variables</td>
<td>$0060 - $0000</td>
</tr>
</tbody>
</table>

**EEPROM**

<table>
<thead>
<tr>
<th>User Prog 16</th>
<th>$0000 - $0FFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>User Prog 3</td>
<td>$0200 - $03FF</td>
</tr>
<tr>
<td>User Prog 2</td>
<td>$012F - $02FF</td>
</tr>
<tr>
<td>User Prog 1, Line 150</td>
<td>$0110 - $011F</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>User Prog 1, Line 20</td>
<td>$0100 - $010F</td>
</tr>
<tr>
<td>User Prog 1, Line 10</td>
<td>$00FF - $0100</td>
</tr>
<tr>
<td>User Prog 1 Meta-Data</td>
<td>$0000 - $0000</td>
</tr>
</tbody>
</table>

**Key:**

- Unused

*Memory maps are not to scale*
**Services:**
The OS provides a number of services to both system and user processes. A ‘service’ is a set of routines for safely accessing a hardware component, such as writing data onto the LCD screen. A process can call the ‘service’ and the operating system will ensure the action is handled correctly. Thus processes are not allowed to directly access the hardware, they must go through the OS.

The following sections describe each service in detail.

---

**LCD:**
Controls what information is displayed on the LCD screen. Only the active process (AP) gets to have the screen immediately updated to reflect it’s request, all other processes have their requests written into the display area in the processes’ entry in the process table. One register, LCDtx, is set aside for access to the routines, data is loaded into this register, and the required routine is then called.

The LCD service offers the following routines:

<table>
<thead>
<tr>
<th>Routine</th>
<th>Use:</th>
</tr>
</thead>
<tbody>
<tr>
<td>lcd_init</td>
<td>Initialises the LCD, by sending init command sequence</td>
</tr>
<tr>
<td>lcd_com</td>
<td>Sends the data in LCDtx to the LCD display as a command</td>
</tr>
<tr>
<td>lcd_data</td>
<td>Sends the data in LCDtx to the LCD display as pure data</td>
</tr>
<tr>
<td>lcd_int</td>
<td>Displays the number in LCDtx as an integer, i.e. $FF will be displayed as 255. Only the required number of characters are used, i.e. $0A will produce only 2 characters, ‘1’, ‘0’. These are then sent to the LCD via the lcd_data routine.</td>
</tr>
<tr>
<td>lcd_mess</td>
<td>Display a message on the LCD. Messages are stored at the start of the program memory, and are addressed using the Z register. Two versions of this routine are available: lcd_mess and lcd_messd. The first automatically calculates the address offset for the message, while the second allows it to be set directly.</td>
</tr>
<tr>
<td>lcd_home</td>
<td>Returns the cursor to the left of the screen</td>
</tr>
<tr>
<td>lcd_clear</td>
<td>Clears the whole LCD, by calling lcd_com with the LCD clear command.</td>
</tr>
<tr>
<td>lcd_right</td>
<td>Moves the LCD one position to the right</td>
</tr>
<tr>
<td>lcd_left</td>
<td>Moves the LCD one position to the left</td>
</tr>
</tbody>
</table>
Keypad:
The Keypad service allows the state of the keypad to be read, as well as ensuring the keys are debounced. Only the active process (AP) is allowed to access the keypad, all other requests are ignored, and get ‘no key pressed’ responses.

The Keypad service offers the following routines:

<table>
<thead>
<tr>
<th>Routine</th>
<th>Use:</th>
</tr>
</thead>
<tbody>
<tr>
<td>keypad</td>
<td>Returns the state of the keypad on the stack</td>
</tr>
<tr>
<td>wait_SELECT</td>
<td>Wait for the SELECT button to be pressed</td>
</tr>
<tr>
<td>debounce</td>
<td>Ensures the keys are properly debounced</td>
</tr>
</tbody>
</table>

To achieve software debouncing of the switches, the following algorithm is used:

1. Wait for each switch to be released
2. Wait for a short period
3. Wait again for all switches to be released

It was found during testing, that this algorithm was both fast and reliable.

Timing:
The Timing service provides a means for processes to wait for a period of time. Under a single tasking OS they could achieve this with a busy-wait, but under the multi-tasking OS used on the MG B sf, this method fails.

The reason it fails, is because as more processes are added, the delays get longer, so to flash an LED once every half a second is very difficult to achieve. Solving this problem accurately is also difficult to achieve. Andrew Tannenbaum [8] suggests a using a linked list of
processes, which are suspended until their time has elapsed. Implementing this method would have required the ability to allocate processes dynamically, which as mentioned above under ‘Process Control’, would have required a more complex scheduling algorithm.

The method I opted for, makes use of a second interrupt (Timer2), that is set to be generated every 100th of a second. Each process has it’s own counter in the counter block (starting at address $0070 and incrementing for each process), which this interrupt updates. When a process wants to be delayed for say ½ second, it calls the delay routine within the Timing service, which resets that process’s counter, and then monitors it until it reaches the required value, at which time the OS returns control to the process.

This method is not completely accurate – even if the correct time has elapsed, the process won’t get control back until it’s turn comes round again. It would be possible to solve this problem, by using a more complex priority system, but for the demonstration, as long as the delays aren’t shorter than that requested, the few microseconds the delay is out by, is not enough to warrant further work.

The Timing service offers the following routine:

<table>
<thead>
<tr>
<th>Routine:</th>
<th>Use:</th>
</tr>
</thead>
<tbody>
<tr>
<td>delay</td>
<td>Picks up the delay length from the stack, resets the correct process’s counter in the Timer2 interrupt area, then monitors it until it matches the given value, at which time the routine exits, passing control back to the process</td>
</tr>
</tbody>
</table>

EEPROM:
The EEPROM services allow bytes to be written to, and read from, the 4K of EEPROM memory. If a context-switch happened during a read or a write, there could be a risk of corrupting the data, so all interrupts have to be turned off while the memory is being accessed – this only causes a slight delay as just 4 instructions have to be executed in this protected manner.

Currently, any process can write to and read from any location in EEPROM. This leads to a risk of processes overwriting each other’s data, and is an issue that could be looked at in the next version, although this wasn’t felt a serious risk in this prototype.
The EEPROM has its own four control registers:

- EEARL: Low byte of address
- EEARH: High byte of address
- EEDWR: Data to write
- EEDRD: Data read

The EEPROM service offers the following routines:

<table>
<thead>
<tr>
<th>Routine:</th>
<th>Use:</th>
</tr>
</thead>
<tbody>
<tr>
<td>EEread</td>
<td>Read a byte from the EEPROM at the given address, returning it in EEDRD</td>
</tr>
<tr>
<td>EEwrite</td>
<td>Write byte EEDWR to EEPROM at the given address</td>
</tr>
</tbody>
</table>

Note these functions are based closely on App Note 100[9], provided by Atmel.

---

**Sound:**

Sound is generated by toggling the I/O line connected to the piezo buzzer at the required frequency.

A particular frequency is generated by turning the piezo buzzer on, then waiting for a given period, then turning the piezo buzzer off, then waiting again for the same given period. This is repeated as many times as dictated by the length of sound requested.

It is critical that a context-switch doesn’t happen when the sound is being generated or else a totally corrupted noise will be heard. The only way currently to stop a context-switch is to disable interrupts, which is how the sound service is programmed. If processes could request a period of time before they were switched out, this could be avoided.

Frequency and time are passed on the stack, and are recovered by the sound service. The following sound routines are provided:

<table>
<thead>
<tr>
<th>Routine:</th>
<th>Use:</th>
</tr>
</thead>
<tbody>
<tr>
<td>bleep</td>
<td>Generate a bleep at a given frequency for a given period</td>
</tr>
<tr>
<td>beepwelc</td>
<td>Generates welcome bleeps</td>
</tr>
<tr>
<td>beep1</td>
<td>Sound the standard error bleep</td>
</tr>
<tr>
<td>beep2</td>
<td>Sound the standard confirmation bleep</td>
</tr>
</tbody>
</table>
The duration and frequency are linked to cycle counts so at the moment a bass sound will last longer than a treble at supposedly the same duration. This would be easy to correct, but wasn’t felt an important issue for this version of the device.

---

**Serial Port:**
The serial port allows data to be exchanged with any RS-232 compatible system, such as a PC. For the reasons given in the previous hardware chapter, the serial connection is handled by a software UART, rather than using the built-in hardware UART.

This means that the serial service has to generate logic levels with the correct timing for the required baud rate. As no handshaking lines are used, the baud rate is fixed at 4800, although this could easily be made user-selectable, as baud rate is controlled by an internal variable.

*Atmel* provide example code for a software UART in their Application Note 305 [29]. This code is very compact and there was no obvious way of improving it’s efficiency, therefore the serial driver is based on their code, with extra code added to handle the multi-tasking aspects.

As with the sound service, time is critical and the routines cannot be switched out, without causing corruption. Hence interrupts are disabled just before a character is sent, and then re-enabled immediately after it has been sent, this makes sure the multi-tasking is only stopped for the shortest period of time possible. This allows other processes to be run between successive characters. The serial service provides the following routines:

<table>
<thead>
<tr>
<th>Routine:</th>
<th>Use:</th>
</tr>
</thead>
<tbody>
<tr>
<td>sputchar</td>
<td>Outputs a character to the serial port</td>
</tr>
<tr>
<td>sgetchar</td>
<td>Waits until a character is received from the serial port, and returns it. Note that during waiting, the interrupts are not disabled, so this doesn’t slow down the OS</td>
</tr>
<tr>
<td>scr</td>
<td>Sends a carriage return followed by a linefeed character to the serial port</td>
</tr>
<tr>
<td>serial_int</td>
<td>Outputs the decimal equivalent of the character to the serial port, followed by a comma. This allows comma separated values to be very easily output from user code. For example the value $FF would cause the following four characters to be output: ‘255,’</td>
</tr>
</tbody>
</table>
**Analogue Input:**

The job of the analogue input service is to sample the value on the analogue input BNC connector, and return it in an appropriate format to the requesting process.

The Analogue-To-Digital converter is a 10-bit built-in device and thus is extremely easy to use. The first stage is to initialise it, which is done by the OS during boot-up. These are the steps that are needed for initialisation:

<table>
<thead>
<tr>
<th>Step</th>
<th>Action</th>
<th>Use:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Select Channel 0</td>
<td>There are 8 multiplexed channels, and the BNC connector is hooked onto channel 0</td>
</tr>
<tr>
<td>2</td>
<td>Set the clock frequency to be $\frac{1}{4}$ of the main crystal frequency</td>
<td>This was found to allow a reasonable amount of the waveform to be captured, without losing too much detail</td>
</tr>
<tr>
<td>3</td>
<td>Enable the Analogue-To-Digital Converter</td>
<td>Allow samples to be taken later on</td>
</tr>
</tbody>
</table>

Once initialised, it is then a simple matter to take a sample – the ADC status register is simply polled until the ADSC flag (bit 6) goes low indicating completion. This polling method is ideal, as it allows the context-switcher to continue, thus interrupts never have to be disabled during analogue input.

When completed, the 10 bits of data can be read from two registers. To keep the data easy to handle it is much better if it is only 8 bits long. It is also likely that the lower 2 bits may be in the noise floor anyway, so conversion loses very little information.
The data from the ADC is returned in two registers: ADCL and ADCH. ADCL uses all 8 bits, while ADHL only has the 2 least significant bits valid, giving a total of 10 bits. Converting to 8-bits can be achieved very efficiently with 4 rotate right instructions as follows:

<table>
<thead>
<tr>
<th>Step:</th>
<th>Action:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: ROR ADCH</td>
<td>Move LSB (bit 9) of ADCH into Carry</td>
</tr>
<tr>
<td>2: ROR ADCL</td>
<td>Move Carry into MSB of ADCL, losing LSB (bit 0) of ADCL</td>
</tr>
<tr>
<td>3: ROR ADCH</td>
<td>Move LSB (bit 10) of ADCH into Carry</td>
</tr>
<tr>
<td>4: ROR ADCL</td>
<td>Move Carry into MSB of ADCL, losing LSB (bit 1) of ADCL</td>
</tr>
</tbody>
</table>

(ROR is the AVR instruction for ROtate Right through carry)

This results in the MS 8-bits of the sample being left in ADCL. This data is then returned to the requesting process through a temporary variable.

The analogue input service provides the following single routine:

<table>
<thead>
<tr>
<th>Routine:</th>
<th>Use:</th>
</tr>
</thead>
<tbody>
<tr>
<td>anain</td>
<td>Returns the 8-bit result of sampling the Analogue Input BNC connector</td>
</tr>
</tbody>
</table>

It was noticed during testing that the flashing red LED affects the value returned by the analogue input service. This is probably due to poor decoupling of the LED, this should be a simple matter to correct in a future version.

**Analogue Output:**

The ATmega103 does not have a built-in Digital-To-Analogue Converter (DAC), thus an external DAC has to be driven from a set of I/O lines. The AD7391 used is a 10-bit serial data input, voltage output device and requires 10 bits to be clocked into, before the load line is strobed, transforming the binary data into a voltage.

Although the DAC is capable of outputting 10-bits, to keep consistent with the ADC, only the 8 MSB will be used. The lower 2 bits will be set to 0.

Loading data is simple, a byte is passed to the analogue output routines which is then rotated left through carry 8 times. Each time, a branch on carry either sets or clears the line connected to the Serial Data Input (SDI), is then followed by strobing the Clock (CLK) line.
to clock the data in. After all 8 bits have been loaded, a further two low pulses are applied to SDI to take the number of bits to 10.

Having loaded the data into the chip, the Load Strobe (‘LD) line is pulsed low for at least 10nS and this causes the data input to be converted to a proportion of the reference voltage (which in this case is the same as the supply voltage) and output onto the analogue output BNC connector.

A short delay (approximately 130 µS) has also been added to allow the voltage time to settle. During testing it was found that if this delay wasn’t included and the analogue output was changed rapidly, the resulting output was inaccurate.

The analogue output service provides the following single routine:

<table>
<thead>
<tr>
<th>Routine:</th>
<th>Use:</th>
</tr>
</thead>
<tbody>
<tr>
<td>anaout</td>
<td>Outputs the value passed, as a voltage on the Analogue Output BNC connector</td>
</tr>
</tbody>
</table>

Initialisation of the analogue output service is performed by the OS during boot-up and consists of setting the LD, SDI and CLK lines to their initial states, followed by outputting a known low value, so the output port is started with a predictable condition.

**Digital Input:**

Digital input has already been used by the serial routines to detect and receive a character. As adding a digital input service requires no new technical features Prototype 2 did not implement the separate digital port. However, this will hopefully be available for Prototype 3.

The service will simply read 2 bits from the port that the digital lines are connected to, and return them to the calling process. It is expected that the following routine will be implemented:

<table>
<thead>
<tr>
<th>Routine:</th>
<th>Use:</th>
</tr>
</thead>
<tbody>
<tr>
<td>digin</td>
<td>Returns the state of the 2 digital input lines as the 2 LSB of the return value.</td>
</tr>
</tbody>
</table>

The OS initialises the digital input lines as input during it’s boot-up sequence.
**Digital Output:**
Like digital input, digital output has already been demonstrated many times, including toggling the system LED and driving the DAC. Prototype 2 was therefore not concerned with implementing the digital output lines directly, as no new technical features would be required. Hopefully there will be 2 digital output lines available on Prototype 3.

The service will simply use the 2 LSB bits of the value passed to set the lines appropriately. It is expected that the following routine will be implemented:

<table>
<thead>
<tr>
<th>Routine:</th>
<th>Use:</th>
</tr>
</thead>
<tbody>
<tr>
<td>digout</td>
<td>Output the 2 LSB of the given value to the digital output port</td>
</tr>
</tbody>
</table>

The OS initialised the digital output lines as output during its boot-up sequence.

---

**Standard Test Equipment:**
A number of items of standard test equipment are used so commonly, that it is worth providing them as standard on the MGB st. Each function is selected from the main menu and runs until the user holds down SELECT, which returns control back to the main menu. Note that very similar functions (except for the Scope) could be written by a user using the built-in programming language. Each function is discussed individually below:

**Volt-Meter:**
The volt-meter shows the voltage of the input connected to the Analogue Input BNC connector.

The volt-meter uses the analogue input service routine (anain) to sample the input, it then converts the value received into a voltage between 0 and 2.5v. It achieves this by inserting a decimal point into the correct position of the 0-255 value returned. After displaying the voltage, it delays for a short period, clears the display, and then shows the next voltage, in a continuous loop. Holding down SELECT at any point returns to the main menu.

This is what is shown on the screen:

```
VM: 0.44v
SELECT=Menu
```

Note this value gets updated approximately once every half a second.
**Sampler:**
The sampler allows a high-speed set of samples to be taken, which are then transferred via the serial port to a PC in comma separated value (.csv) format. This detailed data can then be viewed and graphed on a PC for later analysis.

The sampler works by taking 255 samples as quickly as possible and storing them in the sample data area in SRAM which is located between $0600$ and $0800$. While this is happening the following message is briefly displayed:

```
Sampling...
```

Once complete, all 255 values are sent to the serial port, one at a time followed by a comma. This takes a little time at 4800 baud. While waiting, the user is shown the following message:

```
Sending...
```

If time allows a 'percentage completed' indicator will be added to this screen, to give the user some feedback as to the length of time until completion.

When the sampler has finished, the system returns to the main menu.

**Scope:**
The most visually interesting feature on the device is the oscilloscope function. It shows a real-time waveform display of the signal on the analogue input BNC connector. Considering the device has a text-only display, it is quite surprising how effective the display looks.

The secret is that, as well as 192 text characters, the Hitachi HD44780 LCD screens also have 8 user programmable 5x8 (7 + 1 cursor line) pixel characters, giving an effective miniature graphics screen of 40x7 – just enough to show a waveform.

The data from the analogue input service is provided as an integer between 0 and 255, thus it is easy to scale this to 0-7 by doing 5 left shifts, to divide by 32. It would then appear a simple case to use 5 values at a time to build up a character, and then display it. Unfortunately the characters have to be built up row by row, not column by column. This effectively means that I have to rotate the data through 90 degrees.
Solving this problem still does not give a scope display, because this provides single dots on the screen – a real scope joins the dots together. Thus the algorithm has to work out which pixels should be filled in as well as rotating the data.

One minor problem is that each character is followed by an empty column of pixels, meaning that there are small gaps throughout the display, however this doesn’t appear to be problem, especially when the display is viewed from a slight distance.

This is the final algorithm used, it can almost certainly be optimised, but it works very well for the concept demonstrator:

1. Grab 40 samples as quickly as possible, storing them in the sample data area at $0600, with a set delay between each sample.
2. Convert all samples to 0-7, by 5 left shifts (divide by 32)
3. For each character (1-8) do
4. For each row (0-7) do
5. For each column (0-4) do
6. IF ((current row >= previous row) AND (current row <= current sample)) OR ((current row <= previous row) AND (current row >= current sample)) THEN place_dot ELSE dont_place_dot
7. Next column
8. Next row
9. Next character
10. Display the 8 characters on the LCD (this routine has to bypass the LCD service, as the LCD service doesn’t support programmable characters yet. As the LCD service is not used, if the scope process is switched, and switched back, corruption of the display occurs.)
11. Check the keypad status (using the keypad service)
12. If UP has been pressed then shorten the delay used in step 1
13. If DOWN has been pressed then lengthen the delay used in step 1
14. If SELECT has been pressed then return to the main menu
15. Loop back to step 1

UP and DOWN change the timebase, and SELECT returns to the main menu. Experimentation showed that changing the timebase logarithmically gave the most effective control.
It is difficult to show exactly what the display looks like, but the following gives an idea:

```
OSC:~
SELECT=Menu
```

The display is updated regularly, the exact frequency depending on the timebase chosen. It is interesting to see the effect caused when the M G B s† is powered from a mains adapter, and the analogue input isn’t connected to anything, and isn’t grounded – just as real scope picks up the mains hum, so does the M G B s†!, demonstrating the scope feature is realistic.

Clearly the text-type screens are not designed to show graphics, and if a proper graphics screen was used, the above complexities would be avoided, but it is still impressive what is possible with a few programmable characters and an analogue input.

---

**User Code:**

Allowing a user to write their own code to control the M G B s† separates it from other items of test equipment, and gives the device a major advantage.

The aim of the concept demonstrator is show the theory can work in practice, so a subset of BASIC, large enough to be useful, but not too large to be unrealistic to implement in the time available, was chosen.

An editor is provided which allows the user to enter their code, line by line. Once entered, they can then go and run the code under one of the execution units. Choosing whether to edit or a run a program can be selected from the console menu.

There are eight program areas which can be programmed with the user’s code. Each area is capable of storing 15 lines of BASIC, in theory it would be possible to join several areas together to make longer programs, but not all commands (in particular GOTO) support this option yet.

**Entering User Code:**

Users can enter their own code through the editor, but there is currently no facility for viewing or modifying code already entered – although it is possible to enter new code as many times as required.
Code entry is achieved through a number of context-sensitive menus. The idea is that the user is shown the first part of the line (including the line number) and can then select from a menu the first part of the line. Once they have made a selection they are then shown a new menu with all the options that are possible from that point. For example after selecting the PRINT command, the user is then given the option of printing a VARiable or a NUMber. Whereas after selecting the LET command, the only option is to enter a variable, so they are shown the variables menu immediately.

Initially there was some concern as to whether such a method of entry would be workable in practice, but after getting use to the system, it allows fast entry of code – in some cases faster than by using a keyboard as whole commands can be entered with a few button clicks.

**Expressions:**

Some careful thought was given to the method of handling expressions, and it was concluded that the sub-system needed to allow any form of expression to be entered would take more time than would be warranted. It is important to bear in mind that it is a concept that is being demonstrated not a full implementation of a language. Of course a full version of such a device would need the ability to enter any type of expression, but for this version being able to show a small number of common expressions will be more appropriate.

The form chosen is as follows:

<table>
<thead>
<tr>
<th>First Operand:</th>
<th>Operator:</th>
<th>Second Operand:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable: A,B,C,D,E</td>
<td>addition: +</td>
<td>Variable: A,B,C,D,E</td>
</tr>
<tr>
<td>Number: 0,1,2,3,4,5,6,7,8,9</td>
<td>subtraction: -</td>
<td>Number: 0,1,2,3,4,5,6,7,8,9</td>
</tr>
<tr>
<td></td>
<td>multiplication: *</td>
<td></td>
</tr>
<tr>
<td></td>
<td>division: /</td>
<td></td>
</tr>
<tr>
<td></td>
<td>stop: .</td>
<td></td>
</tr>
</tbody>
</table>

The ‘stop’ or ‘.’ operator is used to signal to the editor that only the first operand is required.

For example to print the number 5 onto the screen the sequence of options would be:

`PRINT`, `5`, `.

One frustrating feature on the AVR microcontroller range is the lack of a division instruction, thus to achieve division, a multiple subtraction routine is implemented. It is interesting to note that the assembler [6] actually has a ‘div’ instruction, perhaps suggesting that such a
command was intended to be implemented, but never was. It is reasonable to expect that a division instruction will be added in the future.

**Command Representation:**
The original BASIC source is not stored by the device, instead a Reverse-Polish based intermediate language is used to both decrease time taken to interpret the commands, and to store the code in a compact form. The Reverse-Polish representation has been chosen so that the original code can be reconstructed for later editing, although this feature hasn’t been implemented in Prototype 2. It also avoids having to store two copies of the same code.

Prototype 2 supports 9 different commands, which are listed in the table below with their function, their reverse polish representation and their accompanying menu structure. It is possible that more commands will be added to Prototype 3, if time allows.

**Command Table:**

<table>
<thead>
<tr>
<th>Command:</th>
<th>Reverse Polish:</th>
<th>Choice 1:</th>
<th>Choice 2:</th>
<th>Choice 3:</th>
<th>Choice 4:</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>$01</td>
<td>VAR: A,...,E NUM: 0,...,9</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>$02</td>
<td>VAR: A,...,E NUM: 0,...,9</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>*</td>
<td>$03</td>
<td>VAR: A,...,E NUM: 0,...,9</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>/</td>
<td>$04</td>
<td>VAR: A,...,E NUM: 0,...,9</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>PRINT</td>
<td>$05</td>
<td>VAR: A,...,E NUM: 0,...,9</td>
<td>+,-,*,/,.</td>
<td>VAR: A,...,E NUM: 0,...,9</td>
<td>-</td>
</tr>
<tr>
<td>INPUT</td>
<td>$06</td>
<td>A,...,E</td>
<td>VAR: A,...,E NUM: 0,...,9</td>
<td>+,-,*,/,.</td>
<td>VAR: A,...,E NUM: 0,...,9</td>
</tr>
<tr>
<td>LET</td>
<td>$07</td>
<td>VAR: A,...,E</td>
<td></td>
<td></td>
<td>-</td>
</tr>
<tr>
<td>BEEP</td>
<td>$08</td>
<td>VAR: A,...,E NUM: 0,...,9</td>
<td>+,-,*,/,.</td>
<td>VAR: A,...,E NUM: 0,...,9</td>
<td>-</td>
</tr>
<tr>
<td>GOTO</td>
<td>$09</td>
<td>10,...,80</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>DELAY</td>
<td>$0A</td>
<td>0,...,9</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>AOUT</td>
<td>$0B</td>
<td>VAR: A,...,E NUM: 0,...,9</td>
<td>+,-,*,/,.</td>
<td>VAR: A,...,E NUM: 0,...,9</td>
<td>-</td>
</tr>
</tbody>
</table>
Reverse-Polish Notation:

Each item is stored as a byte pair. The first byte represents the type, and the second represents the value. The possible type values are shown in the table below:

<table>
<thead>
<tr>
<th>Type</th>
<th>Reverse-Polish First Byte:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command</td>
<td>$00</td>
</tr>
<tr>
<td>Data</td>
<td>$01</td>
</tr>
<tr>
<td>Variable</td>
<td>$02</td>
</tr>
<tr>
<td>End of Program</td>
<td>$99</td>
</tr>
</tbody>
</table>

Thus to store the command ‘PRINT A+1’, the following is used:

<table>
<thead>
<tr>
<th>Command:</th>
<th>Type:</th>
<th>Value:</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRINT</td>
<td>$00</td>
<td>$05</td>
</tr>
<tr>
<td>A</td>
<td>$02</td>
<td>$01</td>
</tr>
<tr>
<td>+</td>
<td>$00</td>
<td>$01</td>
</tr>
<tr>
<td>1</td>
<td>$01</td>
<td>$01</td>
</tr>
</tbody>
</table>

If this was line 20 of program 2, it would be stored in EEPROM as follows, note that the command is also reversed as follows: ‘A,1,+,PRINT’ thus forcing the expression to be evaluated first:

<table>
<thead>
<tr>
<th>Address:</th>
<th>Contents:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$0220</td>
<td>$02, $01, $01, $01, $00, $01, $00, $05</td>
</tr>
</tbody>
</table>

Once the user has entered their program, they can then select to run it from the console menu. This next section discusses how this reverse polish notation is interpreted and the code then run.
**User Code Execution:**

User code is run direct from EEPROM where it is stored. This method is acceptable for the concept demonstrator, but is too slow for real device. One possible solution is to copy the Reverse-Polish representation from EEPROM into SRAM before execution begins, just as software is copied from disk into memory on a PC.

Execution is based on a Forth-like stack machine, with data and variables being pushed onto the local stack, while commands pop values off the local stack. Each of the 8 run environments has it’s own local private stack, which it uses while executing code. Note that these are different stacks from those used by the processes which are context-switched. In effect each run environment has two stacks – one used for user code, and one used for function calls made by the execution unit.

To start executing code, the user must first select which program area they wish to run from a menu. This is then used as the upper byte index into the EEPROM program area. The lower byte is used to reference a particular line within the program. For example line 10 of program 1 is stored at $0110, and line 70 of program 3 is stored at $0370.

The execution cycle is started by examining the type byte of the first instruction (at offset $10). If it indicates a command byte, then the correct command handler is called. A data value or variable reference is pushed straight onto the local stack.

What a command handler does is dependant on the command it is executing. As an example, the INPUT command requires a variable reference to store it’s value. The editor will have reversed the order, so the Reverse-Polish will be as follows: A, INPUT. Thus A will be pushed onto the stack, then INPUT will be executed. INPUT gets a value from the user by displaying a menu on the screen allowing the user to choose a value from 0-255. Once they have made their choice (I’ll assume 10 was chosen), the value is pushed onto the stack. Thus the stack now contains A,10.

INPUT was chosen as the example because it is a special case – it requires an assignment of a value to a variable, i.e. it has an implicit LET stage at the end. Therefore once the stack contains A,10, the INPUT command handler can call LET which pops a value and a variable reference off the local stack and stores the value in the variable. Once this is done, the execution unit moves to the next line, at offset $20.
Variables:

Richard Bornat [25] notes that 'Programs are mostly made up of names and keyword so the lexical analyser will spend a great of it's time searching the symbol table... therefore the table lookup algorithm must be made as efficient as possible.' The most efficient lookup is a direct index, and this is the method I have chosen to use. This nicely avoids the problem of having to search the symbol table.

Each run environment has it’s own local variables stored at offset $01-$05 from the program area, each variable from A to E is stored in sequence, thus variable B of program 5 is stored at $0C02, and variable A of program 1 is stored at $0801 in SRAM. (Note that each run environment has an $08 high-byte offset)

Example:

It is worth discussing a more complex example which shows how expressions are evaluated and programs ended. To show what happens I will use the following simple program, stored in program area 1:

<table>
<thead>
<tr>
<th>BASIC</th>
<th>Address</th>
<th>Reverse-Polish</th>
</tr>
</thead>
<tbody>
<tr>
<td>10: LET A=2</td>
<td>$0110</td>
<td>$02,$01,$01,$02,$00,$07</td>
</tr>
<tr>
<td>20: PRINT A+1</td>
<td>$0120</td>
<td>$02,$01,$01,$01,$00,$01,$00,$01,$00,$05</td>
</tr>
<tr>
<td>30: END</td>
<td>$0130</td>
<td>$99,$99</td>
</tr>
</tbody>
</table>

Stage 1 (Line 10 is processed):

$02,$01 (‘A’) is read from EEPROM, being a variable it is pushed onto the local stack
$01,$01 (‘1’) is read from EEPROM, being data it is pushed onto the local stack
$00,$07 (‘LET’) is read from EEPROM, being a command, it’s handler is run.

Stage 2 (LET is processed):

The LET handler pops the data to be written, and then the variable from the local stack.
It then works out where the variable reference points to and stores the value there, in this case the variable is ‘A’ so the value $02 is stored at $0801.

Stage 3 (Line 20 is processed):

$02,$01 (‘A’) is read from EEPROM, being a variable it is pushed onto the local stack
$01,$01 (‘1’) is read from EEPROM, being data it is pushed onto the local stack
$00,$01 (‘+’) is read from EEPROM, being a command, it’s handler is run.
Stage 4 (+ is processed):
The + handler takes two values off the local stack.
The first value popped is pure data - $01, so it is put straight into a temporary variable.
As the second value is a variable, it’s location is calculated and the value stored there is read.
In this case it is $02, which is then stored in another temporary variable.
The two temporary variables (values $01 and $02) are added together and the result ($03) is
pushed back onto the stack with a type value of $01, representing pure data.

Stage 5 (Line 20 is continued):
$00, $05 (‘PRINT’) is read from EEPROM, being a command it’s handler is run.

Stage 6 (PRINT is processed):
PRINT pops one value from the stack, as the type is $01 it knows it is pure data and doesn’t
try to reference it.
Instead it simply calls the LCD service and requests the integer ($03) is written to the screen.

Stage 7 (Line 30 is processed):
$99,$99 (‘END’) is read from EEPROM, being the END command, the execution units
finishes, by waiting for SELECT to be pressed, before returning to the console menu.
The above run-through shows how a simple command is executed, all commands follow a
very similar structure, with some requiring less parameters, and some requiring more.

Error checking:
As user code is entered by menu, no syntactical errors are possible, any errors will be logical,
and most aren’t trapped by the UCEU. For example it is possible to write a GOTO that goes
to a line that doesn’t exist – the GOTO command causes an offset to be added to the current
user line number, therefore from the UCEU point of view, it is a legal instruction and will be
executed.

One error that is not legal is a divide by zero, and this is trapped by the division handler. If a
division by zero occurs, the following error message is displayed:

!:Div by 0
Halting user programs:
If a user program loops continuously, it is possible to stop it by using the main menu and selecting ‘Halt Prog’. A menu of program numbers is then offered, from which the user can select the program they want to stop.

The stopping mechanism works by setting a flag (initially cleared when a program is started), which is checked as each line is executed. If the flag is set, execution is stopped, the flag is reset and the console menu reappears. Each program has its own flag stored in the program’s meta-data area at offset $00. For example, the halt flag for user program 3 is stored at $0A00.

Testing:
Each piece of software was tested individually in a single-tasking environment, before being integrated into the Operating System and multi-tasked with other processes.

Operating System Process Control:
Much of the early work on process control and context-switching was done using the Simulator [7], stepping through the assembler one instruction at a time, checking that the correct instruction was being executed at the correct time. Although this took longer than expected, it has resulted in a very stable base upon which all the other parts of the system can rely.

Operating System Services:
Each service was tested using a separate single-tasking program, and the hardware was linked to a standard 20MHz oscilloscope to test that the software was having the desired effect on the hardware.

Carefully constructed test cases showed up any problems quickly, where they could be solved in a controlled environment and re-tested before full integration with the OS.

User code editor:
Testing the editor was perhaps the simplest – a selection of simple programs were entered and the byte sequence created by the editor then dumped to the screen, where they were compared with hand-generated versions. If the two matched, then that part of the editor was considered correct.
Each command was taken one at a time, and every possible combination of parameters was tried and checked. Due to the fairly simple command set, the editor was checked quite quickly, and any errors were obvious and easy to fix.

**User Code Execution Unit (UCEU):**
The UCEU is more difficult to prove correct, as there are, in theory, an infinite number of combinations. However as for the editor, a number of carefully chosen test programs were used and debug information output to the screen to follow what was happening. The simulator was used to double check the correct variable references were being picked up, as if they were one location out, the code would still have executed correctly.

Now that all the hardware had been built, the services implemented and tested, and the programming language added, the device can now be used to solve real-world problems. The next chapter gives a number of examples of how the device could be used in practice.
Chapter 6 – User Guide/Examples

This chapter shows how to use the device to solve a range of problems, those at the start are simple, with later examples covering more complex issues.

Starting:
When the device is powered up, a welcome message is displayed for 2 seconds before the main menu is displayed.

To get to a console menu, press the SYSTEM button, then select process 4, 5, 6 or 7 (the console processes), and press SELECT. The console menu will now be displayed, where the user can chose between editing or running a program.

To get back to the main menu, press the SYSTEM button, then select process 3 (the main menu), and press SELECT. The main menu will now be displayed.

Stopping a Program:
To halt a user program that has got into an infinite loop, press the SYSTEM button, then select process 3 to bring up the main menu. Then select ‘Halt Prog’ from the main menu and select the user program number that should be stopped, then press SELECT. The program selected will be stopped after the next line is executed.

Basic Examples:

The basic examples do not require any code to be written, as they use the built-in standard test equipment features.

Checking the voltage on a line:
To check the voltage of a line, first connect it via a BNC lead to the Analogue Input port, then use the main menu to select ‘Volt Meter’. A continually updated display of the voltage will be shown. To return to the main menu, simply hold down and then release SELECT for one second.
**Viewing a waveform:**
To see an analogue waveform of an input, first connect the source to the Analogue Input BNC connector, then use the main menu to select ‘Scope’. A graphical representation of the signal will be shown. The UP and DOWN keys can be used to change the timebase. Holding down SELECT for one second and then releasing it will cause the main menu to reappear.

**Downloading a waveform to a PC:**
To download a set of samples of an analogue waveform to a PC, first connect the analogue source to the Analogue Input BNC connector, and connect a serial cable from the M GB $f$ to a PC. Have a terminal program running on the PC that can accept 4800 baud data with no parity and one stop bit. Next select ‘Sampler’ from the main menu and a few seconds later the samples will be displayed on the PC in comma separated value format.

The results can then be taken to a graphing program (such as Microsoft Excel) to display the original waveform.

---

**Advanced Examples:**

All the following examples require the user to enter the code shown, by going to the console menu, and selecting ‘Edit Prog’. Then the user can then select the program area (1-8) they wish to use, at which point the editor will be started and the first line can be entered, using the on-screen menu system.

**Ask user for an analogue output voltage:**
This code simply asks the user for a voltage and then outputs that voltage on the analogue output port.

```
10: INPUT A     // Ask user for voltage required
20: AOUT A      // Output the analogue voltage
30: GOTO 10     // Loop back for next voltage to output
40: END         // This line is never reached
```

**Double the current analogue input signal once:**
This code will look at the signal on the analogue input, double it, and output it on the analogue output port.

---

Paul Harrison   M GB $f$ - Final Report   Page 82
10: AIN A  // Sample the analogue input
20: AOUT A*2 // Output double it’s voltage to analogue out
30: END    // End the program

**Biasing the analogue input signal continuously:**
This code biases the analogue input by an amount specified by the user, and outputs it to the analogue output port.

10: INPUT A  // Ask the user for a bias level
20: AIN B    // Sample the analogue input
30: AOUT A+B  // Increase the value by the bias amount
40: GOTO 20  // Loop back and sample again
50: END      // This line is never reached

**Generating a square wave:**
This code will ask the user for two voltage levels, and a time delay, and will then generate a square wave between the two voltage levels at the given frequency.

10: INPUT A  // Ask the user for the low value
20: INPUT B  // Ask the user for the high value
30: INPUT C  // Ask the user for the delay value
30: AOUT A   // Set the analogue output to low value
40: DELAY C  // Delay
50: AOUT B   // Set the analogue output to high value
60: DELAY C  // Delay
70: GOTO 30  // Loop back
80: END      // This line is not reached

**Generating a stepped wave:**
This code will create a simple ‘stepped’ output wave, which can be displayed on the scope.

10: LET A=0  // Start the wave at 0 volts
20: AOUT A   // Output the value to the analogue port
30: LET A=A+1 // Increment the variable
40: DELAY 3  // Wait for a short period
50: GOTO 20  // Loop back and output the new value
60: END      // This line is not reached

If the analogue input and output ports are looped back-to-back, and this code is run on one of the consoles, and the scope is then run, the following stepped wave will be seen:

(OSS::  
SELECT=Menu)

(This is an actual screenshot from running the above code)
Waiting for a digital trigger, before outputting a preset analogue voltage:
This code asks the user for a voltage level and then waits for either digital input to go high, before outputting the specified voltage level on the analogue output port.

```
10: INPUT D  // Ask user for analogue output voltage
10: DIGIN C  // Sample the digital lines
20: IF C>0 THEN 40  // If one is raised then output voltage
30: GOTO 10  // Loop back waiting for trigger
40: AOUT D  // Output the analogue voltage
50: END  // Finish the program
```

(Note that DIGIN has not been implemented at the time of writing this report)

Waiting for an analogue trigger, before raising a digital line:
This code asks the user for a voltage level on which to trigger. When that voltage level is exceeded a warning beep is sounded and the 1st digital line goes high.

```
10: INPUT C  // Ask user for trigger level
20: AIN A  // Sample analogue input
30: IF A>C THEN 50  // Check if over trigger level
40: GOTO 20  // Loop back for next sample
50: BEEP 3  // Sound warning beep
60: DIGOUT 1  // Raise digital line 1
70: END  // Finish the program
```

(Note that DIGOUT has not been implemented at the time of writing this report)

Averaging a signal over time:
This code will take 255 values, dynamically average them, and output the result at the end.

```
10: LET A=1  // Set the samples count to 1
20: LET C=1  // Initialise the average to 1
30: AIN B  // Sample the analogue input
40: LET C=C+B  // Add the new sample
50: LET C=C/A  // Divide by the total number of samples
60: LET A=A+1  // Increment the sample count
70: IF A>254 THEN 90  // Check if last sample reached
80: GOTO 30  // Loop back for next sample
90: PRINT C  // Display the average of last 255 samples
100: END  // Finish the program
```
**Applying a correction factor to an analogue signal:**
This code asks the user for two voltage levels, and upper limit and a lower limit, and if the analogue input goes below the lower level, it is increased by a 0.02 volts, and if it goes above the upper limit, it is decreased by 0.04 volts. Once the user has entered the values, the software runs without further user intervention.

```
10: INPUT A    // Ask the user for the lower limit
20: INPUT B    // Ask the user for the upper limit
30: AIN C      // Sample the analogue input
40: IF C<A THEN 80 // Check if below the lower limit
50: IF C>B THEN 110 // Check if above the higher limit
60: AOUT C     // Output the corrected value
70: GOTO 30    // Loop back for next sample
80: BEEP 2     // Sound a ‘too-low’ warning beep
90: LET C=C+2   // Correct it by increasing by 2
100: GOTO 60   // Go back and output the correction
110: BEEP 5     // Sound a ‘too-high’ warning beep
120: LET C=C-4  // Correct it by decreasing by 4
130: GOTO 60   // Go back and output the correction
140: END        // This line is never reached
```

**Build-your-own volt-meter:**
This code shows how a user can build their own volt-meter, this is very similar to the built-in version. It first samples the analogue input, then clears the screen, displays the reading before pausing and repeating the sequence.

```
10: AIN A        // Sample the analogue input
20: CLS          // Clear the screen
30: PRINT A      // Display the voltage
40: DELAY 3      // Wait for a short period
50: GOTO 10      // Loop back to read next voltage
60: END          // This line is never reached
```
Chapter 7 – Conclusions:

This chapter looks at the results of the project, discusses whether integration is viable, and examines whether microcontrollers do provide an appropriate implementation platform. It concludes by looking at the commercial potential for such a device.

“Is integration a good idea?”

This project has shown that it is possible to integrate the functions of a number of items of test and support equipment into a single unit, whilst adding value to the individual parts by the introduction of a programming language.

It has also clearly demonstrated that such a solution does have real benefits, offering functionality not currently available on a single device.

“Are current microcontrollers powerful enough?”

The microcontrollers currently available are not fast enough to simulate in software, test equipment such as a 20MHz oscilloscope. However, they are capable of easily simulating lower speed devices, such as volt-meters, and low speed (perhaps around 1-5MHz) sampling devices. These type of applications still have a very wide use, and thus any device based on current microcontroller technology could provide a solution with real benefits. It is also reasonable to expect that it will not be long before microcontrollers operating in the low 100MHz regions will be generally available.

The concept demonstrator has been successful in trying to show the types of application that might be used in the future, although in it’s current state it is not ready for everyday use quite yet. The following are the main limitations of the concept demonstrator that need to be addressed:

- **Improve speed** – with a combination of a faster microcontroller and/or optimised code, with more emphasis on running native code, by a more intelligent programming language
- **Improve accuracy** – this could be achieved with minor modifications to the operating system, full de-coupling on all chips and separate analogue ground planes
- **Improve robustness** – provide over-voltage and over-current protection to the inputs
- **Add increased storage** – high speed sampling generates large numbers of results, which need to be stored in a larger capacity memory, this could either be an internal memory chip, or an external storage device.
Commercial Potential:

The commercial success of the M G B sl will depend on the type of target market it is aimed at. The original market was engineers working in inaccessible locations trying to troubleshoot low speed faulty circuits. This is still the area where the M G B sl excels the most, however this project has shown it also has many other applications.

During circuit development, engineers often have to generate low speed complex test signals which this device can produce with ease. The ability to take programmable samples for later analysis, whether in a laboratory environment or outside, makes this an attractive device which offers value above that offered by current test equipment.

One major advantage of having a software based device is that the manufacturing costs are kept low. The concept demonstrator could easily be built for under £50, allowing a considerable profit margin, while still keeping the device competitive with other items of test equipment. It also offers the following benefits:

- Integrated solution
- Portability, battery powered
- Dynamic re-configurability

Home constructors and enthusiasts may also be interested in the power and flexibility offered at a low price. Add to this the fact that they actually get a number of items of test equipment, they might not otherwise be prepared to pay for, and the device could be very popular. Whether it is successful in this market, would depend on the price point. The original specification gave a figure of approximately £100, which I feel is very competitive with other devices currently available.

One of the main drawbacks to its commercial potential is the small target audience. This type of device is never going to become something the general public would buy. But because of its flexibility it could be expanded to cover more diverse markets, perhaps a version could be designed for the automotive industry. If this expansion of target audience could be achieved, this would considerably increase its potential.

Taking all the above points, coupled with the fact that the idea is original, and does provide benefits over and above other products in the same market, and the fact that it is not currently available in another form, the M G B sl does appear to have commercial potential.
Achievements:
This project has consisted of many parts, all of which have had to work together to produce the final result. Due to the very broad range of topics, some areas haven’t been covered in as much depth as would have been liked, but all areas been examined in enough detail to give a good feel for the possibilities this type of device could achieve. The following summarises the main achievements of the project:

- Introduced an original, new concept for integration

- Hardware:
  - Designed microcontroller circuit
  - Controlled LCD screen
  - Implemented a keypad
  - Built Serial Port
  - Analogue Input / Output
  - Digital Input / Output

- Software:
  - Designed a multi-tasking priority-schedule operating system
  - Added services to drive the hardware components safely
  - BASIC Programming Language
  - Reverse-Polish internal representation of the programming language
  - Stack-based Reverse-Polish execution unit
  - Menu systems
  - Standard test equipment: Volt-meter, oscilloscope, high-speed sampling

- General:
  - Designed and built a surface mount circuit-board
  - Designed and worked with an outside contractor to build a suitable casing
  - Designed silk-screen mask for casing

- Integrated all the above components into a single working unit

- Proved integration is viable, and microcontrollers are close to being powerful enough
Possible Extensions:
Due to the research and development type nature of this project, a large number of possible extensions have emerged. Those that I feel would add most benefit and should be tackled first, are listed below:

1. Explore the use of a graphic-based display, rather than the text-based screen, used for the demonstrator
2. Use the built-in hardware UART, rather than a software process
3. Provide proper grounding and chip decoupling
4. Provide over-voltage protection for the I/O lines
5. Provide more accurate time delays in the operating system
6. Improve the priority scheduling algorithm
7. Use a second chip to handle sound and other time critical functions
8. Allow a process to prematurely give control back to OS
9. Provide access to a PC card (PCMCIA) for removable storage
10. Copy EEPROM code into SRAM for faster execution
11. Allow users to edit code on the device
12. Allow code to be edited and displayed on a PC, by providing using a converter
13. Protect user code memory access
14. Extend the language to handle more complex statements and more commands
15. Stop the user from editing the same program area in two consoles at the same time
16. Use plug-in modules for high-speed, high-voltage applications
17. Allow values to be entered as volts/milli-volts and seconds/hundredths of a second
18. Increase the number of analogue and digital I/O lines
19. The final device could be ‘ruggedised’, e.g. waterproofed etc.
20. Create a smaller more limited version – perhaps credit-card sized
21. Provide an On/Off switch (!!!)

The Future:
I personally don’t have any plans to try to market this device, although I feel with only a few more months development, it could be turned into a commercial product. I do, however, plan to build myself a next-generation version, because I have come across so many occasions in the past where a device like the M G B $f$ would have been invaluable, that I want to make sure I have such a device for the future.

Paul Harrison
Appendix 1 – Bibliography/References:


[18] avrk. Written by Jon Anders Haugum. Website: colargol.tihlde.hist.no/~jonah/el/


(Note that as of June 1999, this site no longer exists)
Published by Wiley-Interscience. 1979.
By. A.Aho, R.Sethi, J.Ullman. Published by Addison-Wesley. 1986
Published by MIT Press. 1998.
Published by Addison-Wesley. 1994.
Website: http://www.tmo.hp.com/tmo/country/GB/CtryTop/feature/UKEnglish
/UK_Infinium.html
March 06 01:25:16 UTC 1998
Website: http://www.hp.com/calculators/products/product_info/48gx_info.html
Website: http://www.compaq.co.uk/products/handheld/aero/
[37] Barcode reader. ACT Engineering. Email: act.engineering@innet.be
Dirk van den Oudenhoven, Kleine Bollostraat 48, B-3120 Tremelo, Belgium
TEL = +32 (0)16 530524, FAX = +32 (0)16 538120
[38] Atari Portfolio. February 99
http://www.tardis.ed.ac.uk/~alexios/MACHINE-ROOM/Atari_Protfolio.html
DIP Systems, DIP Ltd, 32 Frederick Sanger Road,
Surrey Research Park, Guildford, Surrey. GU2 5BR.
Website: http://www.maplin.co.uk/velleman/k7103.htm. Order Code: VF75S


(The websites were checked just before the report was submitted to get the most up-to-date URLs. Therefore many of them are dated June 99, whereas they were actually used much earlier in December 1998)

Glossary of Terms:

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVR</td>
<td>A range of RISC-based microcontrollers made by Atmel. Atmel claim the initials do not stand for anything, although I personally favour AdVanced Risc.</td>
</tr>
<tr>
<td>BNC</td>
<td>Type of analogue connector, very commonly used in the field of test equipment.</td>
</tr>
<tr>
<td>OS</td>
<td>Short form of Operating System</td>
</tr>
<tr>
<td>uC</td>
<td>Short form of Microcontroller</td>
</tr>
</tbody>
</table>

Copies of the full source code, and .pdf and .ps versions of this report are available at:

~prh/PROJECT/