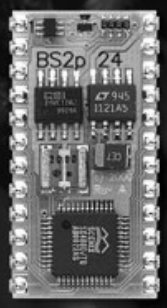
BASIC Stamp 2p

Commands, Features and Projects



by Claus Kühnel and Klaus Zahnert
for Parallax, Inc. Press

Warranty

Parallax warrants its products and printed documentation against defects in materials and workmanship for a period of 90 days. If you discover a defect, Parallax will, at its option, repair, replace, or refund the purchase price. Simply call for a Return Merchandise Authorization (RMA) number, write the number on the outside of the box and send it back to Parallax. Please include your name, telephone number, shipping address, and a description of the problem. We will return your product, or its replacement, using the same shipping method used to ship the product to Parallax.

14-Day Money Back Guarantee

If, within 14 days of having received this book, you find that it does not suit your needs, you may return it for a full refund. Parallax will refund the purchase price of the product, excluding shipping / handling costs. This does not apply if the book has been altered or damaged.

Copyrights and Trademarks

Disclaimer of Liability

Parallax, Inc. is not responsible for special, incidental, or consequential damages resulting from any breach of warranty, or under any legal theory, including lost profits, downtime, goodwill, damage to or replacement of equipment or property, and any costs or recovering, reprogramming, or reproducing any data stored in or used with Parallax products. Parallax is also not responsible for any personal damage, including that to life and health, resulting from use of any of our products. You take full responsibility for your microcontroller application, no matter how life threatening it may be.

Internet Access

We maintain Internet systems for your use. These may be used to obtain software, communicate with members of Parallax, and communicate with other customers. Access information is shown below:

Web: http://www.parallax.com

Internet BASIC Stamp Discussion List

Parallax maintains an e-mail discussion list for people interested in BASIC Stamps. The “basicstamps” list server includes engineers, hobbyists, and enthusiasts. The list works like this: lots of people subscribe to the list, and then all questions and answers to the list are distributed to all subscribers. It’s a fun, fast, and free way to discuss BASIC Stamp programming issues and get answers to technical questions. This list generates about 40 messages per day and has 2,300 subscribers. Subscribe at www.yahoogroups.com under the group name “basicstamps”.

Preface

The Parallax BASIC Stamp 2p extends the family of the well-known BASIC Stamp microcontrollers with a wider feature set for an increased number of applications. The BASIC Stamp 2p has an extended instruction set suitable for more complex projects, making BASIC Stamp 2 projects easier as well. The BS2p supports the Philips I^2C protocol and the Dallas Semiconductor 1-Wire bus along with the direct control of Hitachi-compatible HD44780 alpha-numeric LCDs.

For a long time Parallax customers requested an interrupt capability. The implementation of a powerful polling feature is an answer to this request. Enhancements of additional memory and I/O were also added to let the BS2p to do "double duty" as a datalogger.

In this book we will explain the entire BASIC Stamp 2 family and provide numerous code snippets for using these Parallax microcontrollers.

This book is an English translation of our German book "BASIC Stamp 2 – Neue Eigenschaften – neue Projekte" (ISBN 3-907857-02-X) published in 2002 with numerous updates for the American market.

As manufacturer of the BASIC Stamp, Parallax offers significant information and application support on its web site or in printed form. Do not miss a visit on Parallax's web site [www.parallax.com]. This level of support has differentiated Parallax for years and is certainly a key to the product line's success.

All code examples used in this book are available for download from http://www.parallax.com/bs2pbook.

The authors thank Parallax and especially Ken Gracey for the support of this book. Though Parallax edited the book in detail, the possibility that German language or mannerisms appear in the book remains. If you find such errors please report them to Parallax using e-mail (info@parallax.com).

Klaus Zahnert and Claus Kühnel

Table of Contents

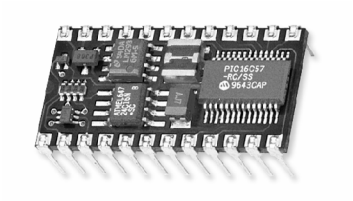
1 BASIC Stamp – an Overview

For years Parallax's BASIC Stamp microcontrollers have been well-known for their ease of use, comfortable programming language and easy debugging using a PC.

BASIC Stamps are not just for engineers (one could also say they are not just for hobbyists, too). Everybody interested in measurements, control and human interaction with electronic circuits will find ease of entry and continuous success with these devices. It's amazing what can be done with a small feature set.

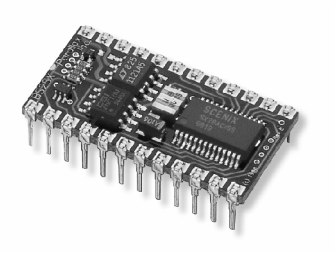
Parallax's educational program "Stamps in Class" introduces new interested parties to this subject with well-designed free tutorials and ready-to-implement class curriculum.

The BASIC Stamp 2 family consists of several variations. Though an overview of technical specifications of all models is helpful to see the differences, Parallax often leads newcomers to the BASIC Stamp 2 and BASIC Stamp 2p due to extensive application support.



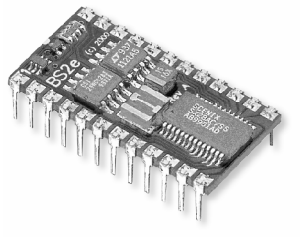
BS2

Microcontroller	PIC16C57 SMD
Clock	20 MHz
EEPROM	2K Bytes
Length of program	500 Lines PBASIC Code
RAM (Variable)	6 I/O, 26 Variable
Input/Outputs	16
Output current(Source/Sink)	20 mA / 25 mA
Current consumption	7 mA Run, 50 µA Sleep
PC Interface	Serial Port
Package	24-Pin DIP Module (green PCB)
Dimensions	30 mm L x 16 mm W x 9 mm H



BS2sx

Microcontroller	Ubicom SX28AC/SS
Clock	50 MHz
EEPROM	8 x 2K Bytes
Length of program	4000 Lines PBASIC Code
RAM (Variable)	6 I/O, 26 Variable plus 63 Byte Scratch Pad RAM
Input/Outputs	16
Output current(Source/Sink)	30 mA/30 mA
Current consumption	60 mA Run, 200 µA Sleep
PC Interface	Serial Port
Package	24-Pin DIP Module (blue PCB)
Dimensions	30 mm L x 16 mm W x 9 mm H



BS2e

Microcontroller	Ubicom SX28AC/SS
Clock	20 MHz
EEPROM	8 x 2K Bytes
Length of program	4000 Lines PBASIC Code
RAM (Variable)	6 I/O, 26 Variable plus 63 Byte Scratch Pad RAM
Input/Outputs	16
Output current(Source/Sink)	30 mA/30 mA
Current consumption	20 mA Run, 200 µA Sleep
PC Interface	Serial Port
Package	24-Pin DIP Module (red PCB)
Dimensions	30 mm L x 16 mm W x 9 mm H

BS2p

Microcontroller	Ubicom SX48AC
Clock	20 MHz Turbo
EEPROM	8 x 2K Bytes
Length of program	4000 Lines PBASIC Code
RAM (Variable)	12 I/O, 26 Variable
Input/Outputs	32 Bytes (6 for I/O and 26 for Variables) plus 32 Byte Scratch Pad RAM
Output current(Source/Sink)	30 mA / 30 mA
Current consumption	40 mA Run, 60 µA Sleep
PC Interface	Serial Port
Package	24-Pin or 40-Pin DIP Module (gold PCB)
Dimensions	24-Pin: 30 L x 16 W x 9 H (mm) 40-Pin: 53 L x 16 W x 9 H (mm)

1.1 What is a BASIC Stamp?

Anyone experienced with a BASIC Stamp can already able to answer this question. Skip to the next subchapter if you do not need this information.

Users with no BASIC Stamp experience will find the next few pages to be useful background information.

The BASIC Stamp is a single board computer containing the microcontroller, an EEPROM, a voltage regulator and reset circuitry. As Figure 1 shows, 24 I/O pins are available for peripherals.

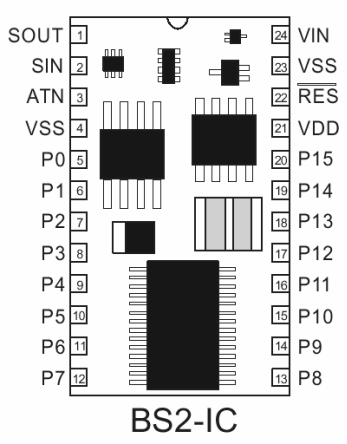


Figure 1 BS2-IC

The pinout listed in the following table shows all I/O resources in detail. In the left column the notation BS2x-24 stands for all 24-pin BS2 modules according to Figure 1, while the second column applies to the 40-pin BS2p only.

Pin BS2x-24[1]	*Pin BS2p-40*	*Name*	*Function*
1	1	SOUT	Serial output (to RxD of the PC COM Port)
2	2	SIN	Serial input (from TxD of the PC COM Port)
3	3	ATN	Attention (to DTR of the PC COM Port)
4	4	VSS	Ground (to GND of the PC COM Port)
5-20	5-20	P0-P15	Digital I/O pins
	21-36	X0-X15	Digital I/O pins
21	37	VDD	+ 5 V DC I/O (stabilized)
22	38	RES	Reset I/O
23	39	VSS	Ground
24	40	VIN	+ 5,5 - 12 V DC input (not stabilized)

Table 1. The "x" refers to a model number of BASIC Stamp in the BS2 series.

Connecting 5 to +12 VDC to V_{IN} and the internal voltage regulator provides a stabilized voltage of +5 VDC at V_{DD}.

If you want to power the BASIC Stamp with a reliable voltage of +5 VDC then connect it to V_{DD} directly. The V_{IN} pin could remain disconnected in this case.

With the reset pin it is quite similar. An internally generated Reset (power-down reset) pulls the RES pin low during the reset phase. An external pulling of RES to low can also force a reset. Both pins (V_{DD} and RES) have I/O characteristics.

The BASIC Stamp Module must be supplied with one voltage only (possibly a battery) and will run a program after program immediately after downloading.

This sounds very easy and it is! Figure 2 shows the complete programming infrastructure required.

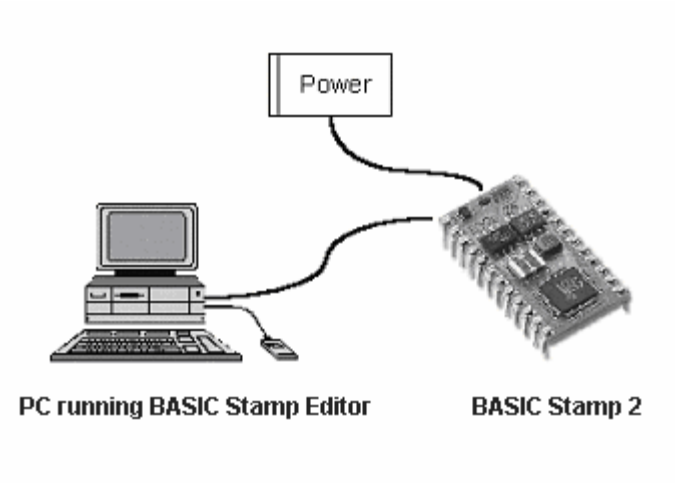


Figure 2 BASIC Stamp Development Environment

If you want to save some money, you can make the required download cable connections from a standard serial cable with the BASIC Stamp module plugged into a breadboard. Figure 3 shows the programming connection. The BASIC Stamp Editor is available as a Windows or DOS program from Parallax's web site for free. At the time of this publication Parallax is about to release a link for Linux, Macintosh and Palm operating systems that would enable developers to design their own IDE and download environments for different operating systems.

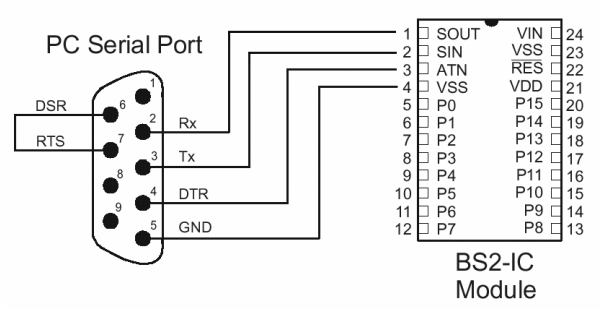


Figure 3 BS2 Download Connection

Let's have a look at how an application could be coded to run on the BASIC Stamp.

The BASIC Stamp Editor must first be installed on the development PC. Parallax offers a detailed "Quick Start Guide" in the BASIC Stamp Manual.

Parallax offers a DOS Editor for each type of BS2 and a single Windows Editor for all types of BS2s.

Figure 4 shows the BASIC Stamp Editor StampW. This programming software is very easy to use.

Figure 4 BASIC Stamp Editor StampW

If the PBASIC source is free of any errors, then it can be compiled and downloaded to the BASIC Stamp.

The downloaded tokens will be saved in the BASIC Stamp's external EEPROM. The BASIC Stamp's microcontroller contains the PBASIC firmware called a "token interpreter". This token interpreter is responsible for running the downloaded tokens and represents Parallax's core intellectual property. The download procedure is the same for all types of BS2s and should not considered in detail here but is discussed in some detail in other resources found on the web (see Brian Forbes' book "Inside the BASIC Stamp 2").

A list of all available PBASIC commands follows in one of the next chapters.

1.2 StampW Editor

Using StampW you can develop programs for all types of BS2s in a Windows environment. StampW can be used from an intuitive standpoint like other Windows application programs. Therefore, this book describes only the more specific features.

To tell the compiler which type of BS2 is described in the PBASIC some innovations were introduced:

STAMP Directive

Different file extensions

Default Stamp Mode (set over the menu **Edit>Preferences**)

The STAMP directive must be used at the beginning of the program. For example, this directive looks for a BS2p:

```
'{$STAMP BS2p}
' Directive shows that this is a BS2p program
```

If you don't insert the directive the BASIC Stamp Windows Editor will add it for you by presenting a pull-down menu and asking you to select a BASIC Stamp.

For the other members of the BS2 family the directives are as follows:

```
'{$STAMP BS2}           ' valid for BS2
'{$STAMP BS2sx}         ' valid for BS2sx
'{$STAMP BS2e}          ' valid for BS2e
```

For the file extensions the following is valid:

filename.bs2 characterizes a source file for BS2
filename.bsx characterizes a source file for BS2sx
filename.bse characterizes a source file for BS2e
filename.bsp characterizes a source file for BS2p

Using the **Edit>Preferences** menu you can set the Default Stamp Mode and different directories for saving the source programs. Figure 5 shows the setup possibilities in the menu **Edit>Preferences>Editor Operation**.

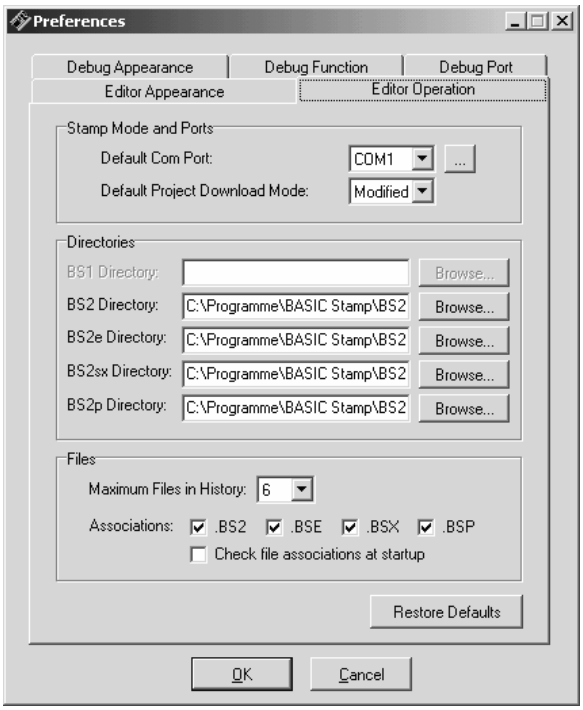


Figure 5 Directory Setup

To test the communication with a connected BASIC Stamp, you can use the Identify Function in the **Run>Identify** menu. Figure 6 shows the response from a BS2p after you've sent the Identify command.

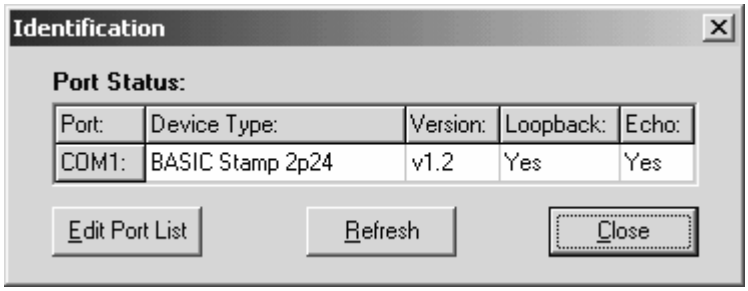


Figure 6 Answer to Identify

Before download you can check the syntax with **Run>Check Syntax**. Syntax errors found will be marked – but undefined commands as “nonsense“, for example, stay unrecognized.

The use of the BASIC Stamp resources can be inspected via **Run>Memory Map**. Figure 7 shows a memory map from this pull-down menu.

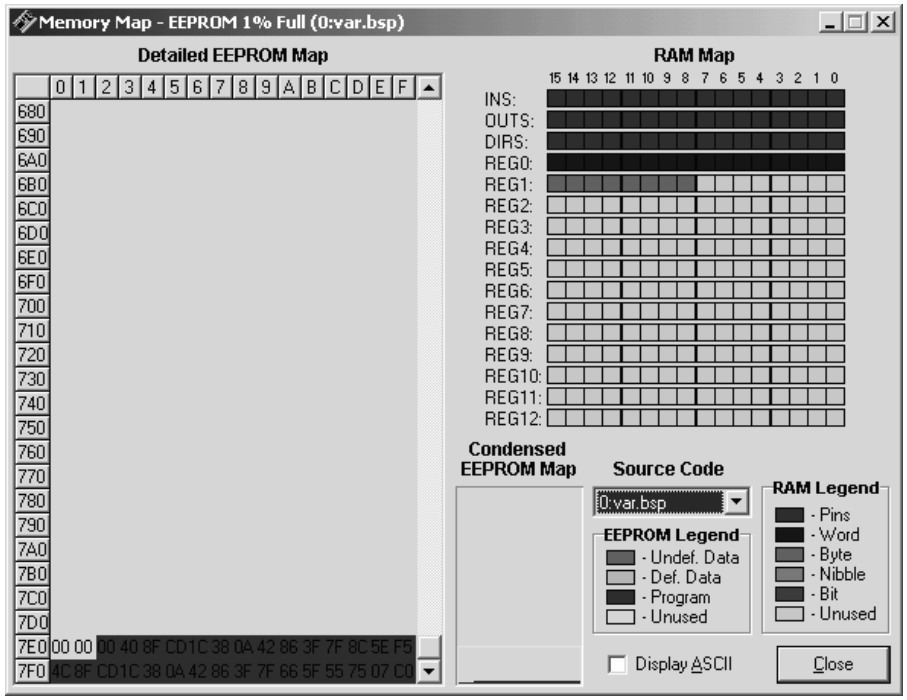


Figure 7 Memory Map

In the left half of the memory map you'll see the 2 KByte EEPROM. For BS2p and BS2sx this is only one slot of program memory. On the right side all 32 bytes of RAM are listed. The Scratch Pad RAM is not shown.

To the usage of EEPROM and RAM by the BASIC Stamp is detailed in the following examples.

StampW offers some new features with the Debug Window. Using the **Run>Debug** menu you can open the Debug Window. Output from the BASIC Stamp's Debug command will be redirected automatically to the Debug Window. But this isn't all it can do.

The programming port can be used for bi-directional communication between a Debug Window and a BS2. The following example demonstrates bi-directional communication on a BASIC Stamp Activity Board:

```
'{$STAMP BS2p}

serstring var byte(3)

loop:
     DEBUG CLS,"D: Waiting for 3 chars from Debug Window..."
     ' SERIN 16, 84, [STR serstring\3]                          ' for BS2
     SERIN 16, 240, [STR serstring\3]                           ' for BS2p
    DEBUG CR,"D: String = ",STR  serstring, CR, CR
    ' SEROUT 16, 84, ["Reflected characters: ",STR serstring\3]' BS2
    SEROUT 16, 240, ["Reflected characters: ",STR serstring\3] ' BS2p

L1:
     IF IN11 = 0 THEN L1
     ' press red key on activity board to proceed
goto loop
```

After the start of the program the DEBUG command sends the string “D: Waiting for 3 chars from Debug Window…” and the SERIN statement waits for exactly three characters. Operating with I/O “Pin 16”, SERIN is redirected to the PC’s serial port and the BASIC Stamp’s DEBUG window. The programmer interface communicates with the Debug Window with the DEBUG command.

After receiving these three characters the second DEBUG command sends the string "`D: String = ___`". The characters received replace the underlined part of the string. After the DEBUG command is executed the BS2p sends the received characters with SEROUT command to the DEBUG Window.

At the end of the loop I/O Pin 11 is checked (if you’re using the BASIC Stamp Activity Board this is the red button) and the program will decide whether or not the whole procedure should be repeated.

Sometimes long instruction lines in our program examples will be broken with what appears to be a carriage return. Please pay attention that in the first column where a label begins a routine. Exceptions are the definitions at the beginning of the program. If you see a character in the first column in the body of the program then this instruction line was too long to print. During compilation such a broken line would be found and signalizes an error. Note that all program examples are available from download from www.parallaxinc.com/bs2pbook.

Figure 8 shows the communication between the BS2p and the Debug Window for the test program.

The Debug Window has two important aspects. The large area in the lower half of the window shows all output of DEBUG and SEROUT 16,..., while the white box above serves as an input line for those characters expected from SERIN.

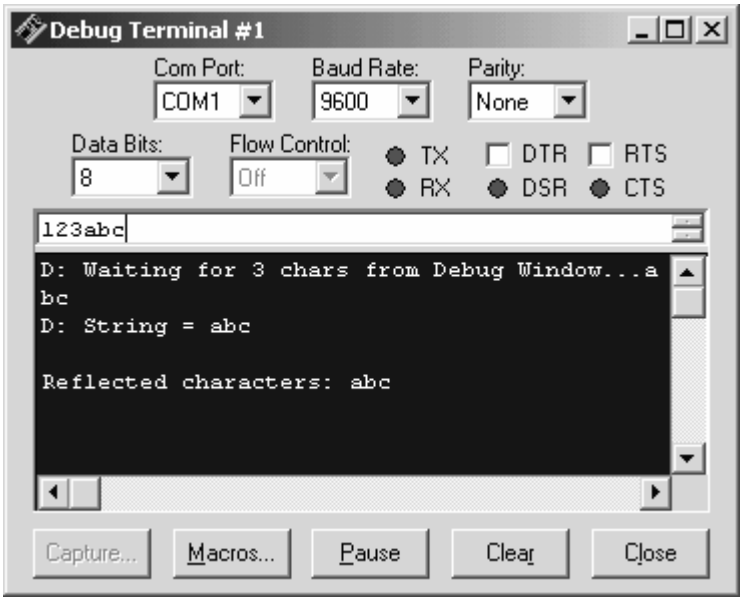


Figure 8 Debug Window

Defining macros can help to simplify the repeated key-pressing during debugging.

Figure 9 shows two macros we defined called *String1* and *String2*. If one presses Ctrl+Shft+A during debugging then the characters "xyz" will be sent to the BASIC Stamp.

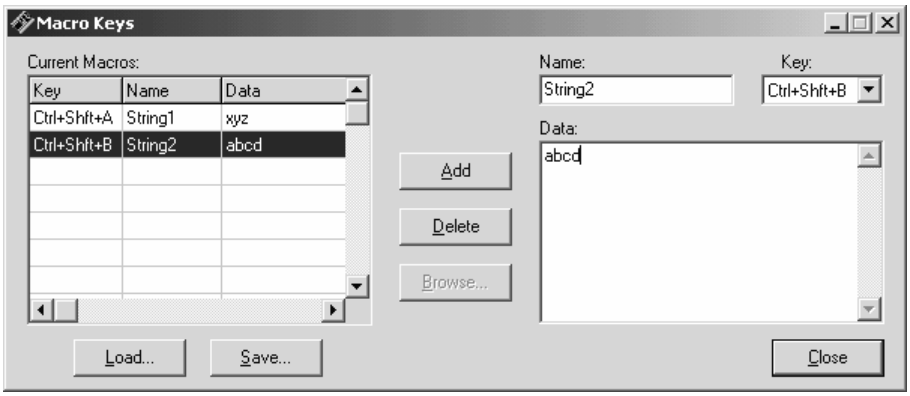


Figure 9 Defining Macros

You can save these definitions in macro files with the extension MCR for further usage.

1.3 BS2 Memory

In the ROM of a BASIC Stamp the token interpreter is saved. The program is saved as tokens in an external EEPROM located on the BS2-IC. Rarely changed information such as configuration data, for example, can be saved in this EEPROM too.

Data is stored in the BASIC Stamp's RAM. After restarting the BASIC Stamp the entire contents of RAM is cleared.

1.3.1 Program Memory

Aside from the original BS2 all other BS2 types (e, SX, p) support a program memory of eight times 2K bytes. With the newer BS2 styles you can define projects consisting of up to eight different programs. Each sub-program can be saved in one program slot of 2 K bytes.

You can save additional program modules, mathematical routines, drivers, text prompts for the user dialogue, calibration data or setup information in these additional program slots.

In this case you may utilize a Stamp directive option Listing all the files to be considered. This Stamp directive must be placed in slot #0 at first.

The examples explain a project with program modules in three different program slots.

The following figures show the creation of three separate source files named project.bsp, aaa.bsp and bbb.bsp. In the files there is no useful source code, just comments. Even though there's no code you can compile these files without any error.

Figure 10 shows all three source files in the Editor window. The leading "0:" marks that each of these source files is prepared for slot #0 by default. This means, we have three separate and independent programs.

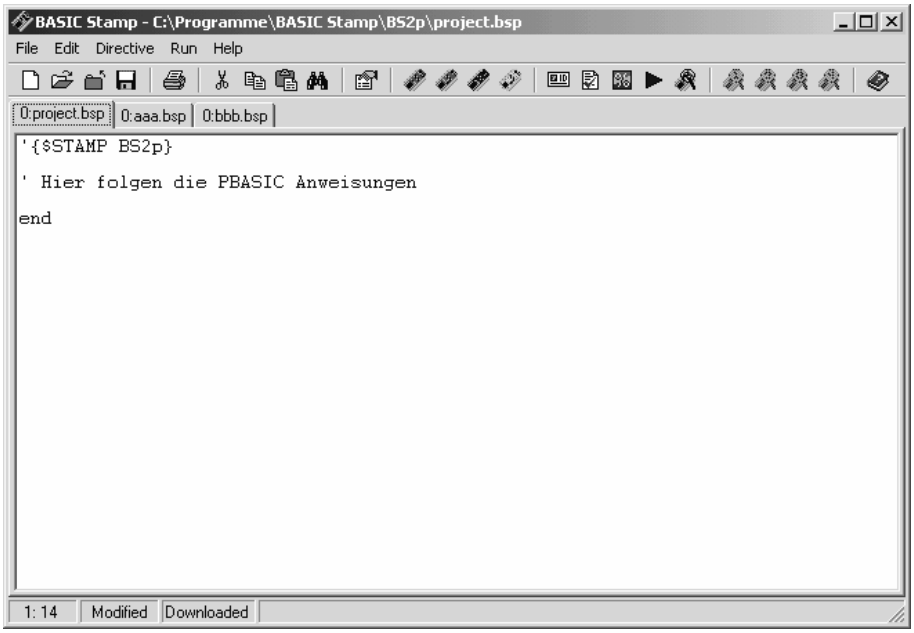


Figure 10 Creating independent source files

To link these programs to one project we have to add these programs to the Stamp directive in the main program in slot #0 `'{STAMP BS2p, aaa, bbb}`. After a new compilation the files are linked.

After reset the BASIC Stamp starts the program in slot #0 by default. Therefore, we place the main program with the extended Stamp directive in slot #0.

Figure 11 shows the revised Stamp directive and the files aaa.bsp and bbb.bsp linked to the project project.bsp.

The tabs of the sub-programs for slot #1 and slot #2 show with their naming [project]1:aaa.respectivelly [project]2:bbb.bsp to which project (project.bsp) they were linked and in which program slot they are located.

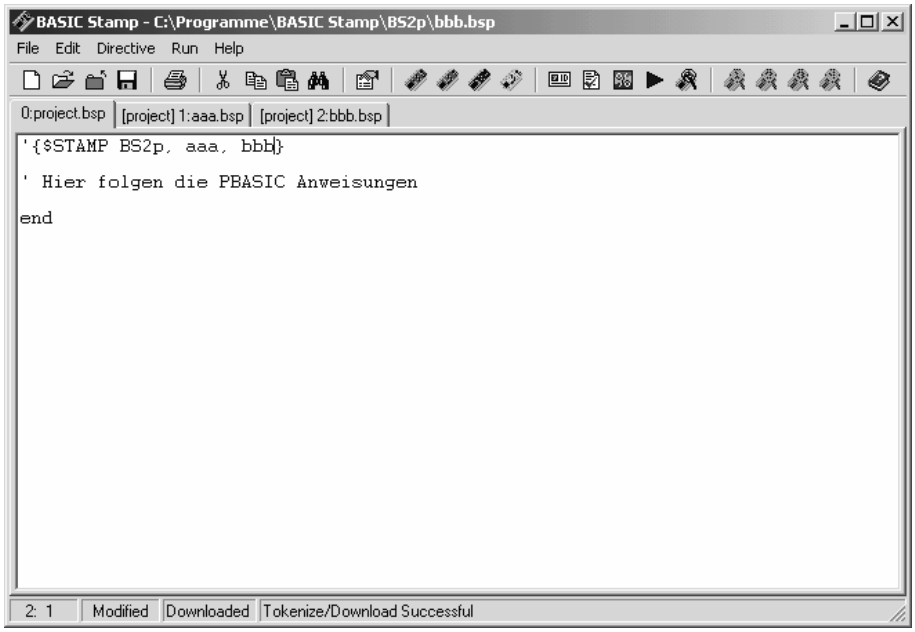


Figure 11 Linking of source files to a project

The RUN command starts the programs saved in different 2 K memory slots. As previously mentioned, after reset the program in slot #0 starts by default and therefore the main program must be located in program slot #0.

You can launch a program from slot #0 use the command RUN 1, for example. After RUN 1 the first instruction in program slot #1 will be operated.

If you want to redirect to program slot #0 again then use RUN 0 from within program #1. Here too, the first instruction in program slot #0 will be executed. You can jump back and forth between programs, always starting with the first instruction.

Pay attention to this difference from normal subroutine techniques. If you switch to another program slot the first instruction of the new slot will be the next instruction executed.

In the highest location of the Scratch Pad RAM (127 in BS2p) is the number of the actual program slot saved. Saving this information before branching to another program slot holds the pointer for return.

Some examples will show that working with different program slots is less difficult than you may have expected.

1.3.2 Data Memory

All BS2s have data memory of 32 bytes. Each bit of these 32 bytes is individually addressable using modifiers. Table 1 shows the RAM organization valid for all BS2 types.

Word Name	*Byte Name*	*Nibble Name*	*Bit Name*	*Purpose*
INS	INL INH	INA, INB, INC, IND	IN0-IN7 IN8-IN15	Inputs
OUTS	OUTL OUTH	OUTA, OUTB, OUTC, OUTD	OUT0-OUT7 OUT8-OUT15	Outputs
DIRS	DIRL DIRH	DIRA, DIRB, DIRC, DIRD	DIR0-DIR7 DIR8-DIR15	I/O Control
W0	B0 B1			
W1	B2 B3			
W2	B4 B5			
W3	B6 B7			
W4	B8 B9			
W5	B10 B11			
W6	B12 B13			
W7	B14 B15			
W8	B16 B17			
W9	B18 B19			
W10	B20 B21			
W11	B22 B23			
W12	B24 B25			

Table 1 BS2 RAM Organization

The access to the RAM is organized by variables. You have to declare all variables in the source code.

```
x var byte      ' Declares a Variable
y var word

z con $5555     ' Defines a Constant

y    = z        ' Initialize Variable
DEBUG HEX y, CR

B0   = $AA      ' Write Register B0 directly
DEBUG HEX y, CR
```

In our program example two variables (x and y) were declared. Variable x has a byte format and variable y word format. For completeness we find the constant z. A constant consumes no memory in RAM.

Have a look at the Memory Map in Figure 12. You can see the usage of the registers and you will recognize a word variable in REG0 (W0) and a byte variable in REG1 (W1).

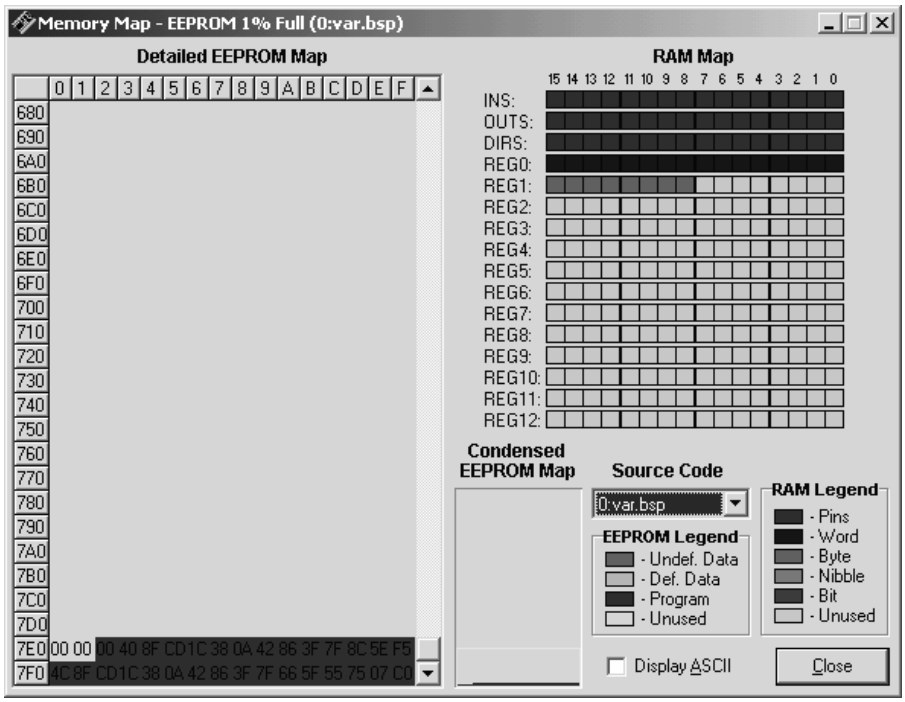


Figure 12 Memory Map

After initialization of the variable y their value can be displayed with DEBUG. The first DEBUG (in single step) will display the hex value $5555 in the Debug Window as expected.

In the next step the value $AA is written to register B0 directly. The second DEBUG shows a different low byte of the word variable y.

In more complex declarations of variables the direct access to registers is fully customizable yet requires some care to avoid program errors.

PBASIC has a lot of modifiers for variables and you can work without direct access to registers. Table 2 lists these modifiers.

Symbol	*Definition*
lowbyte	Low byte of a word
highbyte	High byte of a word
byte0	Low byte of a word
byte1	High byte of a word
lownib	Low nibble of a word or a byte
highnib	High nibble of a word or a byte
nib0	Nibble 0 of a word or a byte
nib1	Nibble 1 of a word or a byte
nib2	Nibble 2 of a word
nib3	Nibble 3 of a word
lowbit	Lowest bit of a word or a byte or a nibble
highbit	Highest bit of a word or a byte or a nibble
bit0	Bit0 of a word or a byte or a nibble
bit1	Bit1 of a word or a byte or a nibble
bit2	Bit2 of a word or a byte or a nibble
bit3	Bit3 of a word or a byte or a nibble
bit4…bit7	Bit4 to Bit7 of a word or a byte
bit8…bit15	Bit8 to Bit15 of a word

Table 2 Modifiers for Variables

These variable modifiers organize access to the content of these variables. Try some experiments with a small test program to get more familiar with the usage of these modifiers.

```
'{$STAMP BS2}

x var word
y var byte
z var NIB

x = $ABCD

DEBUG "Word:              ", HEX x , CR
DEBUG "Low Byte of Word : ", HEX x.lowbyte , CR
DEBUG "High Byte of Word: ", HEX x.highbyte , CR
DEBUG "Nibble 2 of Word:  ", HEX x.NIB2 , CR, CR
```

```
y = x.lowbyte

DEBUG "Low Byte Word:                   ", HEX y, CR
DEBUG "High Nibble of Low Byte of Word: ", HEX y.highNIB, CR
```

All BS2x types with exception of the original BS2 have a so-called Scratch Pad RAM. The Scratch Pad RAM is 64 bytes for BS2e and BS2sx and 128 Bytes for BS2p.

For access to Scratch Pad RAM we use the commands GET and PUT.

The highest location of Scratch Pad RAM (127 for BS2p, otherwise 63) contains the number of the actual program slot.

For the BS2p the high nibble of this cell contains a pointer to that program slot where the commands READ and WRITE operate.

Here is a small program example with DEBUG outputs (Figure 13).

```
'{$STAMP BS2p}

pointer var byte

STORE 5                     ' READ/WRITE in Slot #5

GET 127, pointer            ' Read Slot Pointer in Scratch Pad

DEBUG CR, "Running Program in Slot #", DEC pointer & $0F
DEBUG CR, "READ/WRITE      in Slot #", DEC pointer >> 4

end
```

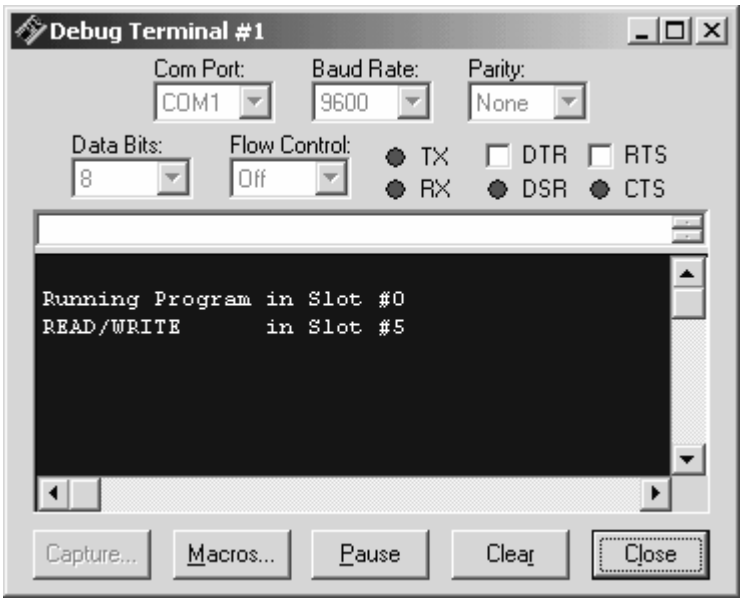


Figure 13 Query for slot numbers

1.4 Which BASIC Stamp is good for my application?

Now you will ask more than ever "Which BASIC Stamp should I use for my application?".

Parallax now offers (including the BS1) seven different devices or modules and you have to calculate performance versus costs to choose the best module for your project. But it's really not that difficult – we think you have two choices.

If the costs are not so important then use the BS2p. If this is not realistic and must be rejected decide on the module that has sufficient performance and a favorable price. Start with the BS2-IC and consider the BS2e-IC or BS2SX-IC only if you have an identifiable need for the extra speed and memory these modules feature. Otherwise use the BS2p or BS2-IC.

Development can always be justified with the module having the highest performance or cost, but in production you have to lower the costs. This can be accomplished in all cases by using the PBASIC Interpreter chips, the core of a BASIC Stamp.

To get an overview to the prices see Table 3. These prices come from Parallax's Product Catalog (2003) and will always differ from Parallax distributors.

BASIC Stamp	*Price*
BS2	US$ 49
BS2e	US$ 54
BS2sx	US$ 59
BS2p-24	US$ 79
BS2p-40	US$ 99

Table 3 BASIC Stamp 2 Prices (USA)

2 PBASIC

The BASIC Stamp understands a special BASIC dialect – called P(arallax) BASIC. The instruction set differs by BASIC Stamps slightly, mostly through the addition of more instructions for newer BASIC Stamps and adjustments in execution speed.

These differences are the basis for different performance and the resulting price.

2.1 BS2 Instruction Set

In the following pages the PBASIC instructions of all BS2 types will be presented.

This compilation cannot replace the original documentation. Download the BASIC Stamp manual V.2.0c containing 351 pages for current information. If needed you can download this 2 MB manual from Parallax's website for free (it also comes on the CD-ROM with your on-line orders).

The BASIC Stamp Editor StampW (now in V.1.31) offers a comfortable on-line help system as well. Figure 14 shows the PBASIC command reference.

Figure 15 shows the explanation to the BRANCH command as an example, while Figure 16 shows a program example to this command.

The PBASIC help system details all PBASIC commands for all BASIC Stamps.

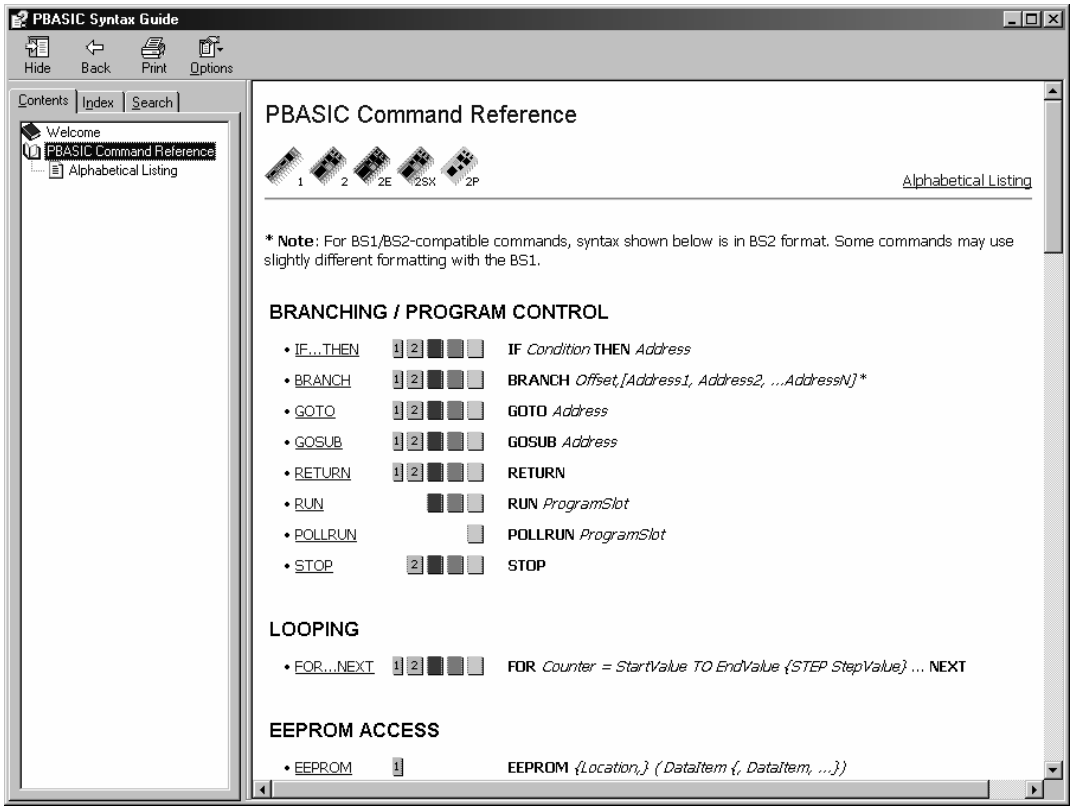


Figure 14 PBASIC Help System – Command Reference

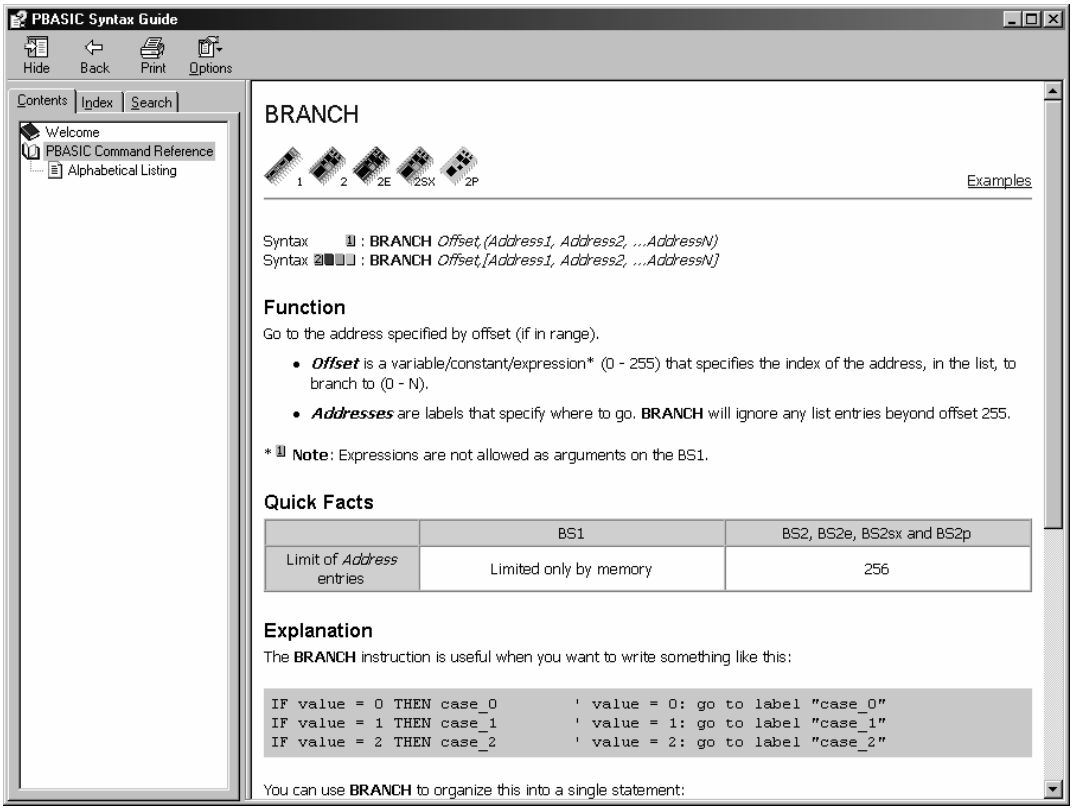


Figure 15 PBASIC Help System – Explanation BRANCH command

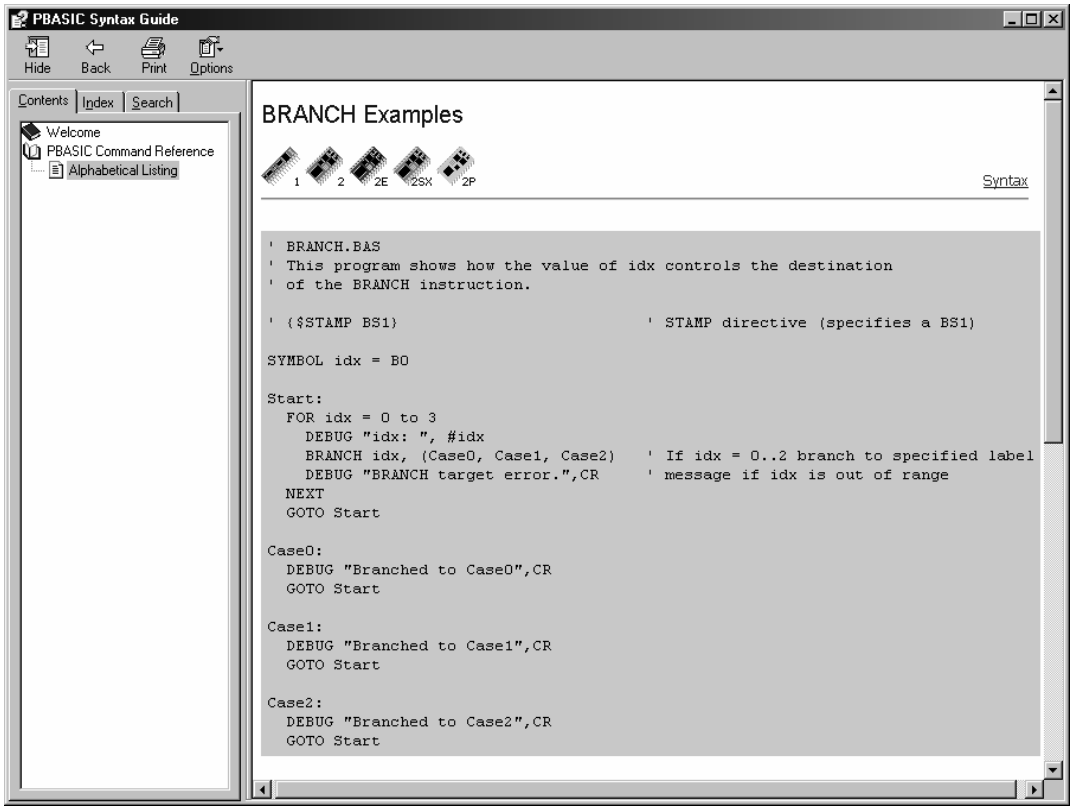


Figure 16
PBASIC Help System – Program example BRANCH

In the program examples we will often use commands that have not been explained. It is normal in such a reference as this book to page forward and backward sometimes and use the on-line help for more detail.

To get a clear structure we will start with each PBASIC command on a new page. You can use the empty room for some comments.

The command for serial I/O SERIN and SEROUT will be directed to I/O pin 16. In these cases the serial I/O works over the programming connection and the serial I/O is redirected to the

Debug Window in the Stamp Editor. The DEBUG command automatically connects to these I/O pins.

If I/O Pin 16 (*Rpin = 16* or *Tpin=16*) used for serial communication then the BS2 works with inverted polarity and active output of the transmitter independent of the Baud mode parameter. In this mode Bit 14 (Polarity) and Bit 15 (output driver) are not relevant.

AUXIO BS2p-40

AUXIO

The commands AUXIO, MAININO, and IOTERM control the access to the I/O pins of the BS2p-40.

AUXIO redirects all I/O operations from MAININO (P0-P15) to AUXIO (X0-X15).

After reset MAININO is default.

For BS2p-24 the commands AUXIO, MAININO, and IOTERM have no effect.

Example:

```
'{$STAMP BS2p}

        high 0    ' Output0 (Pin5) Hi
        AUXIO     ' Switch to Auxiliary I/O Pins
        low 0     ' Output0 (Pin21) Lo
        MAININO   ' Switch to Main I/O Pins
        low 0     ' Output0 (Pin5) Lo
        AUXIO     ' Switch to Auxiliary I/O Pins
        high 0    ' Output0 (Pin21) Hi
        end
```

Remark:

I/O Pin 0 (P0) is set and subsequently the BS2p-40 will be switched to AUXIO. Low 0 resets I/O pin X0 to low afterwards.

After redirection to MAININO low 0 resets P0 to low.

After a new redirection to AUXIO high 0 sets X0 to high.

BRANCH BS2 BS2e BS2sx BS2p

`BRANCH offset,[addr0, addr1,..., addrN]`

Branch of program flow

It means:

offset Pointer in address list

addr element in address list (address, label)

Example:

```
char      var byte
value     var byte

loop:     value = 255 ' initialize value to FFH

          ' Read serial interface
          SERIN 16,84+$4000, [char]     ' 9600 Baud for BS2

          ' Filter character "A", "B" and "C"
          LOOKDOWN char,[65,66,67],value

          ' Branch
          BRANCH value, [aaa, bbb, ccc]
          DEBUG CLS
          goto loop

aaa:      DEBUG CLS, "Character was A"  ' It was an A
          goto loop
bbb:      DEBUG CLS, "Character was B"  ' It was a B
          goto loop
ccc:      DEBUG CLS, "Character was C"  ' It was a C
          goto loop
```

Remark:

Characters will be received via serial input (here from Debug Terminal) and checked for “A,” “B,” and/or “C”. All other characters will be ignored. The resulting value (A = 0, B = 1, C = 2) controls the branch by a jump to the labels aaa, bbb or ccc.

BUTTON BS2 BS2e BS2sx BS2p

`BUTTON pin#, downstate, delay, rate, bytevar, targetstate, label`

Debounce button input, perform auto-repeat, and branch to address if button is in target state. Button circuits may be active-low or active-high.

It means:

pin# Pin for key (0-15)

downstate State of the pressed key (0 or 1)

delay Delay time before Auto-Repeat Function starts. If Delay is 0, Button performs no debounce or auto-repeat. If Delay is 255, Button performs debounce, but no auto-repeat.

rate Auto-Repeat Rate

bytevar Variable for Button command – must be cleared before first use.

targetstate State for branch (0 = not pressed; 1 = pressed)

label Specifies where to branch

Example:

```
key  var byte                  'Defining workspace

         key = 0               'Initialization

loop:    BUTTON 1,0,9,1,key,0,wait
         low 0
         end

wait:    TOGGLE 0
         pause 100
         goto loop
```

Remark:

Query a key connected to I/O Pin 1. The key is low-active, that means Lo at input P1 for pressed key..

If the key is not pressed (targetstate = 0) then the program branches to label wait and toggled I/O Pin 0 each 100 ms afterwards.

If the key is pressed then the program does not branch, P0 is reset to low and ends the program. Connecting a key to an I/O Pin is explained later.

COUNT **BS2 BS2e BS2sx BS2p**

`COUNT pin#, period, wordvar`

Count pulses on an I/O Pin during a defined time

It means:

pin#	Pin as count input (0-15)
period	Time for pulse counting (see Table)
wordvar	Word variable for result

Timing	*BS2*	*BS2e*	*BS2sx*	*BS2p*
Units in *period*	1 ms	1 ms	400 µs	287 µs
Period range	65536 ms	65536 ms	26214 ms	18809 ms
Min. pulsewidth	4.16 µs	4.16 µs	1.66 µs	1.20 µs
Max. frequency	120 kHz	120 kHz	300 kHz	416,7 kHz

Example:

```
'{$STAMP BS2}

countinput    con  1    ' Count input is Pin1
counttime     con  10   ' For BS2 count time is 10ms
countvalue    var  word

loop:     COUNT countinput,counttime,countvalue
          DEBUG DEC? countvalue
          goto loop
```

Remark:

Counts pulses at I/O Pin 1 For a time of 10 ms. The result will be saved in the variable `countvalue` temporarily. The DEBUG command displays the result in the Debug Window of StampW.

DATA **BS2 BS2e BS2sx BS2p**

`{symbol} DATA {at,} dat1 {, dat2, dat3, …}`

Writes data to EEPROM during program download.

It means:

symbol	Symbolic name for the address of first data element (optional)
at	Address of first data element (optional}
dat#	Data element (0-65535)

Example:

```
'{$STAMP BS2p}

addr var word
char var byte

Text1    DATA "0123456789"
         DATA @$10, "A", "B"
Text2    DATA @32,0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
Spc      DATA @$30, " " (16)
Res      DATA @$40, (16)
         DATA 27
Text3    DATA @$100, "Das is ein Beispiel für einen mit ASCII 0 abgeschlossenen
String.", 0

         addr = Text3
         DEBUG CLS, HEX4 addr
         SEROUT 16, 240, [CR]

loop:    READ addr, char
         IF char = 0 then exit
         SEROUT 16, 240, [char]
         addr = addr + 1
         goto loop
exit:    DEBUG CR, HEX4 addr

end
```

Remark:

This example shows different ways to save data in the EEPROM during program download. DATA commands without the address operator @ save the data element to the next free memory location. You can denote the address operator in different number formats (DEC,

HEX). Repetitions can be input by a number in brackets (number). If there is no data element then memory is reserved (here 16 bytes beginning at address $40), but not initialized.

Figure 17 shows the memory map for the example above. After the start of the program the repeated READ fetches the bytes saved between addresses $100 and $142 from EEPROM and SEROUT and then sends them serially to the Debug Window.

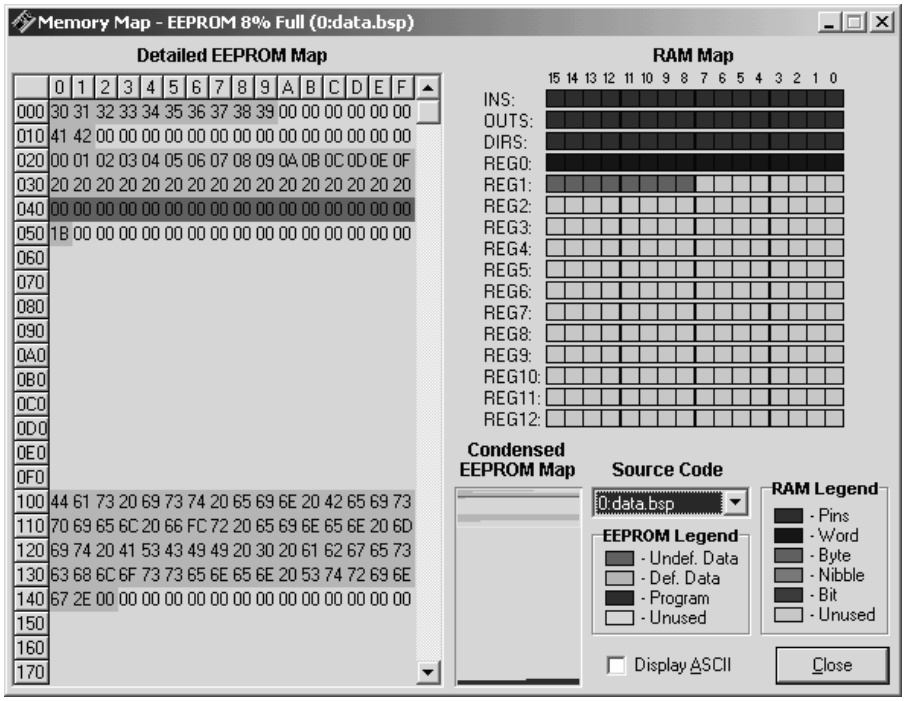


Figure 17 Memory Map for DATA Example

DEBUG BS2 BS2e BS2sx BS2p

```
DEBUG  dat1 {, dat2, ...}
```

Sends variables and messages to the Debug Window for display. Formatter control the display format.

Example:

```
string var byte(4)

string(0) = "A"
string(1) = "B"
string(2) = "C"

value var word
value = -123

DEBUG CLS,"DEBUG BS2",CR     'DEBUG BS2
DEBUG REP "="\20,CR             '====================
DEBUG STR string,CR             'ABC
DEBUG STR string\2,CR        'AB
DEBUG DEC? value             'value = 65413
DEBUG DEC value,CR           '65413
DEBUG DEC3 value,CR             '413
DEBUG SDEC value,CR             '-123
DEBUG SDEC5 value,CR         '-00123
DEBUG HEX4 value,CR             'FF85
DEBUG SHEX4 value,CR         '-007B
DEBUG IHEX4 value,CR         '$FF85
DEBUG ISHEX4 value,CR        '-$007B
DEBUG BIN value,CR           '1111111110000101
DEBUG SBIN value,CR             '-1111011
DEBUG IBIN value,CR             '%1111111110000101
DEBUG ISBIN value,CR         '-%1111011
```

Remark:

The comments in each line show the result of the regarding formatter.

There are additional DEBUG options. CLS clears the Debug Window. HOME places the cursor to the top left position. BELL sounds the PC speaker. BKSP places the cursor one position back while TAB places the cursor to the next tabulator position.

DTMFOUT BS2 BS2e BS2sx BS2p

`DTMFOUT pin#,{ontime,offtime,}[key,key,...]`

Generates DTMF tones (dial tones of the phone)

It means:

pin#	I/O Pin for tone output
ontime	Duration of the tone
offtime	Duration of the pause after a tone
key	Phone key (0 to 9 are numbers, 10 is * and 11 is # and 12 to 15 means A-D (not available in most telephones)

	BS2, BS2e	*BS2sx*	*BS2p*
Default OnTime	200 ms	80 ms	55 ms
Default OffTime	50 ms	50 ms	50 ms
Resolution	1 ms	0.4 ms	0.265 ms

Example:

```
number   DATA "0815"

key  var NIB(4)
i    var NIB

         for i=0 to 3
         READ number+i,key(i)
         'DEBUG DEC key(i),CR
         next
loop:    DTMFOUT 0,250,100, [key(0),key(1),key(2),key(3)]
         pause 2000
         goto loop
```

Remark:

The first loop (FOR…NEXT) reads four keys from EEPROM while the second loop dials them.

For demonstration purposes the dialing is repeated permanently.

The tone output must be filtered. Circuit examples are in the appendix, chapter 7.1.1.

END **BS2 BS2e BS2sx BS2p**

END

End the program and placing the BASIC Stamp into low-power mode.

Example:

```
low 0
pause 2000
high 0
end
```

Remark:

End the program and place the BASIC Stamp into low-power mode - the I/O pins stay active. Every 2.3 seconds the I/O pins go for about 18 ms into tri-state (High-Z).

Only a hardware reset can wake-up the BASIC Stamp from that state after an END is encountered.

FOR | NEXT BS2 BS2e BS2sx BS2p

```
FOR var = start TO end {STEP incr} | NEXT {var}
```

Creating a for ... next loop.

It means:

var	Variable as loop counter
start	Start value of the loop counter
end	End value of the loop counter
incr	Increment of the loop counter. Default is 1.

Example:

```
idx       var byte

          DEBUG "Step Up", CR
          for idx = 0 to 8 STEP 2
               DEBUG ? idx
          next

          DEBUG "Step Down", CR
          for idx = 7 to 2
               DEBUG ? idx
          next
```

Remark:

The program runs the FOR...NEXT loop beginning with the start value till the end value. The loop index will be incremented as long the index is between start and end value. The loop index will be incremented if the start value is smaller then the end value. Otherwise, the loop index would be decremented.

The default increment is 1. If you need another value you have to use STEP incr. The increment value is always positive.

It is possible to nest FOR...NEXT loops in up to 16 levels.

FREQOUT BS2 BS2e BS2sx BS2p

`FREQOUT pin#, period, freq1, {freq2}`

Generate one or two sine-wave tones for a specified period.

It means:

pin#	I/O Pin for tone output
period	Duration of tone
freq1	First frequency
freq2	Second frequency (optional)

	BS2, BS2e	*BS2sx*	*BS2p*
Unit in *period*	1 ms	0.4 ms	0.265 ms
Unit in *freq1,2*	1 Hz	2.5 Hz	3.77 Hz
Frequency range	0 – 32768 Hz	0 – 81917 Hz	0 - 123531 Hz

Example:

```
'{$STAMP BS2p}

melody DATA word 500, word 700, word 900, word 500
' frequency     1885 Hz,  2639 Hz,  3393 Hz,  1885 Hz

tone      var  word
tonelo    var  tone.lowbyte
tonehi    var  tone.highbyte
i         var  nib

          for i=0 to 3
          READ 2*i + melody,tonelo
          READ 2*i + melody + 1, tonehi
          tone = tonehi * 256 + tonelo
          DEBUG DEC? tone
          FREQOUT 11, 500, tone
          next
```

Remark:

The tone output can be filtered. Circuit examples are in the appendix, chapter 7.1.1.

GET **BS2sx, BS2p, BS2sx, BS2p**

`GET location, variable`

Read value from Scratch Pad RAM location and store in variable.

It means:

location	Memory location in Scratch Pad RAM (BS2sx: 64; BS2p: 128)
variable	Variable for value read

Example:

```
a         var word              'general purpose variables
b         var byte

addr      con 44                'sratchpad address

          GET addr, b           'fetch the value
          a = b+1               'perform some calculations
          PUT scr-addr, a.byte0 'save the byte
```

Remark:

The highest memory location (BS2sx: 63; BS2p: 127) is Read-Only and contains the number of the actual program slot.

GOSUB | RETURN **BS2 BS2e BS2sx BS2p**

`GOSUB label | RETURN`

Call of a subroutine and return after processing.

It means:

label Entry point in the subroutine

Example:

```
idx       var nib

          DEBUG CLS, "Main...", CR
          for idx = 0 to 3
          gosub show          'call subroutine show
               DEBUG "Main...", CR
          next
          DEBUG "End."
          end

          'subroutine for display
show:     DEBUG "Subroutine: ", DEC ? idx
          return
```

Remark:

At a subroutine call the address of the next instruction is saved as a return address. The program branches to the label after GOSUB.

RETURN jumps back to return address, more specifically the next instruction encountered after the GOSUB.

GOSUB can be nested in up to four levels. You can use 255 GOSUBs in one program.

GOTO **BS2 BS2e BS2sx BS2p**

`GOTO label`

Jump to a label in the program

It means:

label Target (where to go)

Example:

```
start:    TOGGLE 0
          pause 1000
          goto start

          end    ' program does not come to this point
```

Remark:

After the instruction `pause 1000` the program jumps to the label start. In this case the instruction end is never encountered and can be removed.

HIGH **BS2 BS2e BS2sx BS2p**

`HIGH pin#`

Switches an I/O Pin to output and set it to Hi

It means:

pin# I/O Pin for digital output

Example:

```
start:    low 0
          pause 200
          high 0
          PAUSE 200
          GOTO start
```

Remark:

I/O Pin 0 is set to high and low and then repeated.

I2CIN **BS2p**

`I2CIN pin#, SID, Addr {\LowAddr}, [InputData]`

Receive of data via I^2C bus

It means:

pin#	I/O Pin for SDA line of the I^2C bus
SID	Address of the I^2C device
Addr	Address of a register in the I^2C device
LowAddr	Part of an address of a register in the I^2C device (optional)
InputData	List of variables and modifiers for received data

	BS2p	
I/O Pin	0	8
	I/O Pin 0: SDA	I/O Pin 8: SDA
	I/O Pin 1: SCL	I/O Pin 9: SCL
Transmission rate	about 81 kits/sec	
Remark	SDA and SCL need a Pull-Up resistor of 4,7 kΩ	

Example:

```
'{$STAMP BS2p}

Idx  var word      ' Index variable for address
Result   var byte(16)  ' 16-byte array for returned value

         DEBUG CR, "Reading I2C EEPROM...", CR
         pause 1000
         for Idx = 0 to 2047 STEP 16   'Read 2K EEPROM
              'Read 16 bytes at once
              I2CIN 0, $A1+((Idx>>8)*2), Idx, [STR Result\16]
              DEBUG "Addr: ", HEX4 Idx, "-", HEX4 Idx+15, " Value: ", STR Result, CR
         next
```

Remark:

Reading an I^2C EEPROM with I^2C address $A1 in blocks of 16 bytes. DEBUG generates a memory map in the Debug Window.

I2COUT **BS2p**

`I2COUT pin#, SID, Addr {\LowAddr}, [OutputData]`

Transmit data via I^2C bus

It means:

pin#	I/O Pin for SDA line of the I^2C bus
SID	Address of the I^2C device
Addr	Address of a register in the I^2C device
LowAddr	Part of an address of a register in the I^2C device (optional)
OutputData	List of variables and modifiers for sent data

	BS2p	
I/O Pin	0	8
	I/O Pin 0: SDA I/O Pin 1: SCL	I/O Pin 8: SDA I/O Pin 9: SCL
Transmission rate	about 81 kbits/sec	
Remark	SDA and SCL need a Pull-Up resistor of 4.7 kΩ	

Example:

```
'{$STAMP BS2p}

Idx  var word   'Index variable for address
Result   var byte(16)  '16-byte array for returned value

         DEBUG CR, "Writing I2C EEPROM...", CR
         pause 1000
         for Idx = 0 to 2047 STEP 16                  'Read 2K EEPROM
          'Write 16 bytes at once
          I2COUT 0, $A0+((Idx>>8)*2), Idx, [REP Idx>>4\16]
          pause 5
          DEBUG "Addr: ", HEX4 Idx, "-", HEX4 Idx+15, " Value: ", DEC3 Idx>>4 & $FF, CR
         next
```

Remark:

Writing an I^2C EEPROM with I^2C address $A0 in blocks of 16 bytes. DEBUG generates a protocol in the Debug Window.

IF | THEN BS2 BS2e BS2sx BS2p

```
IF variable ?? value {AND/OR variable ?? value ...}
```

THEN label Conditional branch

It means:

variable	Variable for compare
??	Compare operation (=, <>, >, <, >=, <=, AND, OR NOT, AND, OR, XOR)
value	Value for compare
label	Target for branch

Example:

```
start:    input 8               'I/O Pin 8 is input
again:    if in8 = 0 then work  'wait for I/O Pin 8 = Lo
          goto again
work:     toggle 11             'Reaction
          pause 1000
          goto start
```

Remark:

If the result of the compare operation is valid then the program branches to the label. If the result is not valid the next instruction will be executed.

With the different compare operators building logic is very easy with the BASIC Stamp.

The compare operation uses integer numbers without sign (16 Bit).

INPUT **BS2 BS2e BS2sx BS2p**

`INPUT pin#`

Switches an I/O Pin to input.

It means:

pin# I/O Pin as a digital input

Example:

```
low 8               'Output Lo at Pin8
input 8             'Change I/O Pin8 to input
debug ? in8         'Read and display I/O Pin8
reverse 8           'Change I/O Pin8 to Output
debug ? in8         'Read and display Pin8
```

Remark:

I/O Pin 0 is switched from output to input.

IOTERM **BS2p-40**

`IOTERM block#`

Switching the I/O blocks of BS2p-40.

It means:

block# I/O block number (0=MAINIO; 1=AUXIO)

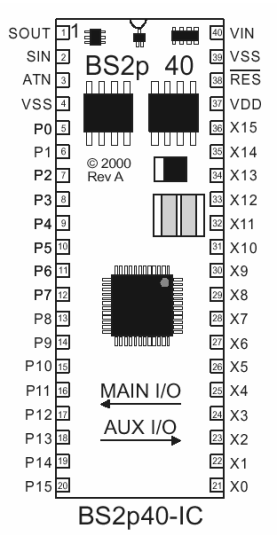
Example:

```
high 0          ' Output0 (Pin5) Hi
ioterm 1        ' Switch to Auxiliary I/O Pins
low 0           ' Output0 (Pin21) Lo
ioterm 0        ' Switch to Main I/O Pins
low 0           ' Output0 (Pin5) Lo
ioterm 1        ' Switch to Auxiliary I/O Pins
high 0          ' Output0 (Pin21) Hi
```

Remark:

Use the IOTERM command for switching between the BS2p's I/O blocks with a single parameter (0=MAINIO; 1=AUXIO).

This command only applies to the BS2p-40.



MAINIO = IOTERM 0

P0 – P15 active

AUXIO = IOTERM 1

X0 – X15 active

LCDCMD **BS2p**

`LCDCMD pin#, command`

Sends a command to an LCD.

It means:

pin# I/O Pin for LCD Enable, specifies the other LCD I/O Pins

command LCD Command

Pin#	*0 or 1*	*8 or 9*
LCD Enable (E)	0 or 1	8 or 9
LCD Read/Write (R/W)	2	10
LCD Register Select (RS)	3	11
LCD Data Bus (DB4-DB7)	4 – 7	12 - 15

Command	*Dec.*	*Explanation*
No Operation	0	
Clear Display	1	Clears the display
Home Display	2	Moves the cursor to home position
Auto-Decrement Cursor	4	No move of the display content
Auto-Decrement Cursor	5	Moves the display content one position to right
Auto-Increment Cursor	6	No move of the display content
Auto-Increment Cursor	7	Moves the display content one position to left
Display Off	8	Switches the display off
Display On	12	Switches the display on
Blinking Cursor	13	Blinking cursor
Underline Cursor	14	Switches the cursor to underscore
Cursor Left	16	Moves the cursor to left
Cursor Right	20	Moves the cursor to right
Scroll Left	24	Moves the display content to left
Scroll Right	28	Moves the display content to right
4-Bit Mode	32	1 Line, Font 5x8
4-Bit Mode	36	1 Line, Font 5x10
4-Bit Mode	40	2 Lines, Font 5x8
4-Bit Mode	44	2 Lines, Font 5x10
Wake Up	48	Wake up of the display
Character RAM Address	64 + addr	Address a location in Character RAM
Display RAM Address	128 + addr	Address a location in Display RAM

Example:

```
InitLCD:
         pause 1000          ' Wait for power-up of LCD
         lcdcmd 1, 48        ' Send wakeup command
         pause 10            ' Pause needed due to the LCD specs
         lcdcmd 1, 48
         pause 1             ' Pause needed due to the LCD specs
         lcdcmd 1, 48
         pause 1             ' Pause needed due to the LCD specs
         lcdcmd 1, 32        ' Set data bus to 4-bit mode
         lcdcmd 1, 40        ' Set to 2-line mode with 5x8 font
         lcdcmd 1, 8         ' Turn display off
         lcdcmd 1, 12        ' Turn display on without cursor
         lcdcmd 1, 6         ' Set to auto-increment cursor
```

```
lcdcmd 1, 1         ' Clear the display
```

Remark:

The LCDCMD command sends control commands to a directly-connected LCD with Hitachi's LCD-Controller HD44780A.

These control commands support special modes of the LCD, as initializing the LCD, cursor movement, font setup etc.

Normally, you should initialize the LCD after power-on respectively after reset.

Chapter 3.3 explains working with direct-connected LCD in detail.

LCDIN **BS2p**

`LCDIN pin#, command, [inputdata]`

Reading data from an LCD.

It means:

pin#	I/O Pin for LCD Enable, specifies the other LCD I/O Pins
command	LCD Command (see LCDCMD)
inputdata	List of variables and formatters

Example:

```
'{$STAMP BS2p}

char      var byte(16)          ' Array for 16 characters read from LCD

          gosub lcdinit         ' Init the LCD (not listed here)
          gosub ReadLCDScreen
          end

ReadLCDScreen:
          debug "LCD Now Says: "
          lcdin 1,128,[str char\16] 'Read 16 char starting at 0
          debug str char\16,CR,CR
          return
```

Remark:

The command `lcdin 1,128, [str char\16]` reads 16 characters beginning at location 0 of display RAM (home position) and saves them in the array `char`. For an LCD n x 16 this would be the first lines completely.

LCDOUT **BS2p**

`LCDOUT pin#, command, [outputdata]`

Writing data to an LCD .

It means:

pin#	I/O Pin for LCD Enable, specifies the other LCD I/O Pins
command	LCD Command (see LCDCMD)
inputdata	List of variables and formatters

Example:

```
'{$STAMP BS2p}

        gosub lcdinit         ' Init the LCD (not listed here)
        lcdout 1, 1, ["Hello World"]
        lcdout 1, 192, [rep "="\11]
        end
```

Remark:

The command `lcdout 1,1, [„Hello World"]` sends the string „Hello World" to the LCD and displays beginning at location 1. The command `lcdout,1,192,[rep „="\11]` underlines this string afterwards.

LOOKDOWN BS2 BS2e BS2sx BS2p

`LOOKDOWN value,??[value0,value1, . . . ],index`

Compares a value with some compare values. The following compare operations are possible: =, <>, >, <, <=, >=.

It means:

value	Value for compare
??	Compare operator (default is =)
value0	first compare value
value1	seconds compare value
index	Index of the compare

Example:

```
value     var byte
idx  var byte
i         var nib

          value = 64
          for i=0 to 4
               idx = 99      'setup for idx if no match
               lookdown value, [65, 66, 67], idx
               debug cr, dec2 value, rep 32\2, dec2 idx
               if idx = 99 then nomatch
m1:            value = value + 1
          next
          end

nomatch:
          debug "  No match found."
          goto m1
```

Remark:

The variable `value` will be compared with the values 65, 66 and 67. The variable `idx` points to that position in the compare list for which the result of the compare operation is valid.

If the compare value is not in the list then the variable `idx` does not change. You can detect such a situation by setup variable `idx` before the compare.

LOOKUP BS2 BS2e BS2sx BS2p

`LOOKUP index,[value0,value1,...],var`

Access to a list of values by an index.

It means:

index	Index to a list of values
value0	First value (Index =0)
value1	Second value (Index = 1)
var	Result

Example:

```
idx  var byte
value    var byte
i        var byte

         for i=0 to 8
              value = 255
         lookup i,[1,2,4,8,16,32,64,128], value
         debug cr, dec i, rep 32\2, dec3 value
              if value = 255 then novalue
m1:      next
         end

novalue:
         debug "  Index out of list."
         goto m1
```

Remark:

The index points into the list of values and saves the value in variable `value`.

If the index exceeds the highest location number in the list, the variable is unaffected.

Setting up the variable var before compare helps detect this situation..

LOW **BS2 BS2e BS2sx BS2p**

`LOW pin#`

Initialization of an I/O Pin as output and set to low.

It means:

pin# I/O Pin for digital output

Example:

```
start:    high 0
          pause 200
          low 0
          pause 200
          goto start
```

Remark:

I/O Pin 0 is switched to high and low repetitively.

MAINIO BS2p-40

MAINIO

The commands AUXIO, MAINIO and IOTERM control the access to the BS2p-40's I/O Pins.

The command MAINIO switches the BS2p-40 I/O operations from AUXIO (X0-X15) to MAINIO (P0-P15).

After a reset MAINIO is set by default.

For BS2p-24 the commands AUXIO, MAINIO and IOTERM have no effect.

Example:

```
'{$STAMP BS2p}

         high 0    ' Output0 (Pin5) Hi
         auxio     ' Switch to Auxiliary I/O Pins
         low 0     ' Output0 (Pin21) Lo
         mainio    ' Switch to Main I/O Pins
         low 0     ' Output0 (Pin5) Lo
         auxio     ' Switch to Auxiliary I/O Pins
         high 0    ' Output0 (Pin21) Hi
         end
```

Remark:

P0 (Pin 5) is set to high and the I/O is switched to AUXIO afterwards. Low 0 resets X0 (Pin 21).

After switching back to MAINIO low 0 resets P0 (Pin 5).

After switching to AUXIO again high 0 sets X0 (Pin 21) to high.

NAP **BS2 BS2e BS2sx BS2p**

```
NAP period
```

Switches for a short period into the Sleep-Mode (Low-Power Mode) and reduces for this time the current consumption.

It means:

Period	*Duration of the Nap Period*
0	18 ms
1	36 ms
2	72 ms
3	144 ms
4	288 ms
5	576 ms
6	1.152 s
7	2.304 s

Current consumption during	*BS2*	*BS2e*	*BS2sx*	*BS2p*
RUN	8 mA	25 mA	6 mA	40 mA
SLEEP	40 µA	60 µA	60 µA	60 µA

Example:

```
long      con 7

loop:     nap long          'Nap about 2.3 sec
          goto loop
```

Remark:

Switches for a short time to the Sleep-Mode. At the end of the Nap period all I/O Pins go for about 18 ms to Tri-State (High-Z) and program executes the next command afterwards.

OUTPUT **BS2 BS2e BS2sx BS2p**

`OUTPUT pin#`

Switches an I/O Pin to output.

It means:

pin# I/O Pin as a digital output

Example:

```
low 8               'Output Lo at Pin8
input 8             'Change I/O Pin8 to input
debug ? in8         'Read and display I/O Pin8
output 8            'Change I/O Pin8 to output
debug ? in8         'Read and display Pin8
```

Remark:

I/O Pin 8 is set to Lo before switching to input. Due to the Pull-Up resistor on the Activity Board the Debug command signalizes high after reading this input.

After switching back to output (`output 8`) and a repeated query of this I/O Pins Debug signalizes low (`low 8`).

OWIN **BS2p**

`OWIN pin#, mode, [inputdata]`

Receives data from a 1-Wire Device.

It means:

pin#	I/O Pin as a digital input
mode	Data transfer mode (0-15)
inputdata	List of variables and formatters for received data.

The parameter *Mode* controls the position of the reset impulse, byte- or bit-mode and data rate. The next table has some combinations. The choice of the correct mode depends of the type of 1-Wire device connected to the BS2p.

Mode	*Meaning*
0	No Reset, Byte Mode, slow data transfer
1	Reset before data, Byte Mode, slow data transfer
2	Reset behind data, Byte Mode, slow data transfer
3	Reset before and behind data, Byte Mode, slow data transfer
4	No Reset, Bit Mode, slow data transfer
5	Reset before data, Bit Mode, slow data transfer
8	No reset, Byte Mode, high data transfer
9	Reset before data, Byte Mode, high data transfer

Example:

```
'{$STAMP BS2p}              'specifies a BS2p

OWpin         con 15        '1-wire device pin

OWFERst       con %0001     'Front-End Reset, Byte Mode
OWBERst       con %0010     'Back-End reset, Byte Mode
OWBitMode     con %0100     'No Reset, Bit Mode

ReadROM       con $33       'Read ROM Command
SearchROM     con $F0       'Search ROM Command
```

```
ROMData       var byte(8)
devcheck      var nib

owout OWpin, OWFERst, [SearchROM]
owin  OWpin, OWBitMode, [devcheck.bit1, devcheck.bit0]

owout OWpin, OWFERst, [ReadROM]
owin  OWpin, OWBERst, [str ROMData\8]
```

Remark:

The 1-Wire Bus is connected to I/O P15.

In Bit Mode a query for a connected 1-Wire device occurs.

In Byte Mode eight data bytes followed by a reset will be read from the 1-Wire bus and stored in ROMDATA.

Explanations of data transfer via 1-Wire bus follow in detail in chapter 3.2.

OWOUT BS2p

`OWOUT pin#, mode, [outputdata]`

Send data to a 1-Wire device

It means:

pin#	I/O Pin as digital output
mode	Data transfer mode (0-15)
outputdata	List of variables and formatters for data to send

The parameter *Mode* controls the position of the reset impulse, byte- or bit-mode and data rate. The next table has some combinations. The choice of the correct mode depends of the type of 1-Wire device connected.

Mode	*Meaning*
0	No Reset, Byte Mode, slow data transfer
1	Reset before data, Byte Mode, slow data transfer
2	Reset behind data, Byte Mode, slow data transfer
3	Reset before and behind data, Byte Mode, slow data transfer
4	No Reset, Bit Mode, slow data transfer
5	Reset before data, Bit Mode, slow data transfer
8	No reset, Byte Mode, high data transfer
9	Reset before data, Byte Mode, high data transfer

Example:

```
'{$STAMP BS2p}                  'specifies a BS2p

OWpin           con 15          '1-wire device pin

OWFERst         con %0001       'Front-End Reset, Byte Mode
OWBitMode       con %0100       'No Reset, Bit Mode

SearchROM       con $F0         'Search ROM Command

devcheck        var nib
```

```
owout OWpin, OWFERst, [SearchROM]
owin  OWpin, OWBitMode, [devcheck.bit1, devcheck.bit0]
```

Remark:

1-Wire Bus is connected to I/O P15.

The command SearchROM is sent in Byte Mode to the connected 1-Wire device.

Explanations to data transfer via 1-Wire bus follow in detail in chapter 3.2.

PAUSE **BS2 BS2e BS2sx BS2p**

PAUSE period

Stops the program for a certain time in milliseconds.

It means:

period Duration of break (period = 0...65535)

Example:

```
start:    low 8
          pause 20          'Pause for 20 ms
          high 8
          pause 1000        'Pause for 1 s
          goto start
```

Remark:

The accuracy of time depends on the accuracy of the oscillator frequency only.

POLLIN POLLMODE POLLOUT POLLRUN POLLWAIT BS2p

```
POLLIN pin#, state
POLLMODE mode
POLLOUT pin#, state
POLLRUN slot#
POLLWAIT period
```

The polling commands allow the BS2p to react to certain states on it's digital inputs.

It means:

pin#	I/O Pin as digital input or output
slot#	Specifies the program slot for the polling event program (0 – 7)
mode	Data transfer mode (0-15)
state	State of the I/O Pin for polling event
period	Specifies the duration of the Low-Power state (0-8). The time calculates to $2^{\mathbf{period}} * \mathbf{18\,ms}$ (for period = 0...7). For period = 8 the BS2p does not switch to low-power state and has therefore a faster response.

Remark:

At the end of each PBASIC instruction and before the next the BS2p interpreter checks the given inputs (*pin#*) for defined levels (*state*). These periodic queries (polling) seem to happen in the background. This polling is not a hardware interrupt. The polling event is when a pre-defined state occurs.

The command POLLWAIT is a little bit different to the other polling commands. POLLWAIT switches the BS2p to stop until the polling event occurs. The parameter *period* works quite similar to the NAP command.

If the time defined by *period* is finished then a new polling of the defined inputs occurs. If there is no polling event then the BS2p switches to Stop and waits the time defined by *period*. If when the polling event occurs the program will execute the next PBASIC instruction.

The polling commands are a very complex enhancement of BASIC Stamp functionality. Therefore, the polling is explained in it's own chapter with some examples. Chapter 2.2.4 describes the required details.

PULSIN **BS2 BS2e BS2sx BS2p**

`PULSIN pin#, state, var`

Measure of a pulse length

It means:

pin#	I/O Pin as digital input
state	Specifies the phase to be measured (0 = low; 1 = high)
var	Result of measurement (Variable as byte or word)

	BS2	*BS2e*	*BS2sx*	*BS2p*
Resolution	2 µs	2 µs	0.8 µs	0.75 µs
Max. Pulse length	131.07 ms	131.07 ms	52.428 ms	49.125 ms

Example:

```
pulsinp con 8      'Pulse input I/O Pin8
state   con 0      'Trigger with 1-0
value   var word

loop:     pulsin pulsinp, state, value
          debug dec ? value
          goto loop
```

Remark:

The stated I/O Pin waits for the specified edge of the pulse (the high-low transition in this example) and starts time measurement until the end of the pulse (the low-high transition).

Have a look at the table for the maximum pulse length. If the pulse is longer than the maximum pulse length then a time-out with the result 0 occurs.

The maximum pulse length determines the time window where the pulses are expected. If no trigger arrives during this time the result will be 0 again.

PULSOUT BS2 BS2e BS2sx BS2p

`PULSOUT pin#, period`

Generates a pulse by inverting an I/O Pin for a defined time.

It means:

pin# I/O Pin as digital output

period Duration of pulse (period = 0...65535)

	BS2	BS2e	BS2sx	BS2p
Resolution	2 µs	2 µs	0,8 µs	1,18 µs
Max. Pulse length	131.07 ms	131.07 ms	52.428 ms	55.479 ms

Example:

```
'{$STAMP BS2p}

pulsoutp  con 8                         'Pulse output I/O Pin8
time      con 42373                     'Duration 42373*1,18 µs = 50 ms

          high pulsoutp                 'Pulse=Hi
loop:     pulsout pulsoutp, time        'Pulse=Lo for 50 ms
          pause 1000
          goto loop
```

Remark:

The state of the concerning I/O Pin is inverted for a defined time. The initial state defines the polarity of the pulse.

In the example the I/O Pin is set high. The endless loop generates 50 ms low pulses every second.

PUT **BS2e BS2sx BS2p**

`PUT location, value`

Writes a byte into Scratch Pad RAM

It means:

location	Location in Scratch Pad RAM
value	Byte to save in Scratch Pad RAM

Example:

```
'{$STAMP BS2p}

scraddr   con 123       'sratchpad pad address
pattern   con $abcd     'bit pattern
a         var word      'general purpose variables
b         var byte

          a = pattern
          debug hex2 scraddr, ":", hex4 a, cr
          put scraddr, a.byte0                  'save the byte
          get scraddr, b                        'fetch the value
          debug hex2 scraddr, ":", hex4 b, cr
```

Remark:

Saves a byte in Scratch Pad RAM. The BS2p has 128 Byte Scratch Pad RAM, while BS2e respectively BS2sx have only 64 Byte.

In the eBS2e and BS2sx location 63 contains the number of the actual program slot. In BS2p location 127 contains the number of the actual program slot (low nibble) and the number of the actual Read/Write-Slot (high nibble). These locations of the Scratch Pad RAMs are read-only.

PWM **BS2 BS2e BS2sx BS2p**

`PWM pin#, duty, cycles`

Output of a pulsewidth-modulated (PWM) signal for a defined time

It means:

pin#	I/O Pin as digital output
duty	Specifies the duty resp. the resulting analog voltage (0 - 255 = 0 - 5V)
cycles	Number of output periods (0-255); Period see table

	BS2	*BS2e*	*BS2sx*	*BS2p*
Period	1 ms	1 ms	0.4 ms	0.625 ms

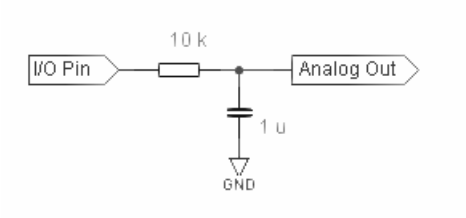
Example:

```
'{$STAMP BS2p}

time     con 5000           'Waiting time

loop:    pwm 8, 51, 255     'U = 1V
         pause time         'Wait 5 sec.
         pwm 8, 102, 255    'U = 2V
         pause time         'Wait 5 sec.
         pwm 8, 153, 255    'U = 3V
         pause time         'Wait 5 sec.
         pwm 8, 204, 255    'U = 4V
         pause time         'Wait 5 sec.
         goto loop          'Repeat endless
```

Remark:

The I/O Pin is switched to an output and generates a defined number of pulses with a defined duty. In the example we generate a pulses of 255 each time. At the end of the pulse package (255 * 0.625 ms = 159 ms for BS2p) the I/O Pin switches to an input again.

When using PWM to generate an analog voltage you need to use a simple R/C filter for smoothing.



The 10 kOhm resistor and 1 uF capacitor are appropriate in this circuit though you can measure the voltages across the capacitor.

You can calculate the charging time of the capacitor by the formula T = 4*R*C. After four periods the capacitor is completely loaded.

The PWM signal of the BASIC Stamp is utilized in a non-common way. Normally a fixed period is separated with a low and high phase relating to their duty cycle. A duty of 0.5 means the same duration is used for the low and the high phase in that period. For example, in a period of 1 ms the low and the high are 0.5 ms both for a duty of 0.5.

BASIC Stamps generate short high/low pulses that number in a period which represents a certain duty cycle.

RANDOM BS2 BS2e BS2sx BS2p

`RANDOM wordvar`

Generates a pseudo-random number

It means:

wordvar Word variable for start value and result

Example:

```
value     var word           'Word variable for random
i         var nib

          value = 999        'Initialize value to 999
          for i=0 to 15
          random value
          debug dec2 i, ":", dec5 value, cr
          next
```

Remark:

Generates pseudo-random numbers depending from an initial value (here 999).

The RANDOM command uses the variable *value* as a start value and saves the result in this variable after operation as a start value for a next call of RANDOM.

RCTIME **BS2 BS2e BS2sx BS2p**

`RCTIME pin#, state, wordvar`

Measures the charge/discharge time for a RC combination

It means:

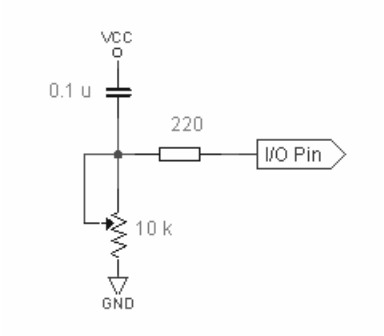
pin#	I/O Pin as digital input
state	Input state (0 or 1)
var	Result of measurement (variable as byte or word)

	BS2	*BS2e*	*BS2sx*	*BS2p*
Resolution	2 µs	2 µs	0,8 µs	0,9 µs
Max. Pulse length	131.07 ms	131.07 ms	52.428 ms	58.982 ms

Example:

```
result   var word

loop:    high 7
         pause 1
         rctime 7, 1, result
         debug ? result
         goto loop
```

Remark:



To measure the unknown value of a resistor and convert it to a digital value use the circuitry connected to an I/O Pin in the left schematic.

In the program example I/O pin 7 is set to Hi for a time of 1 ms disloading the capacity.

The command `rctime` switches I/O pin 7 to input and the resistor (with unknown value) loads the capacity. If the voltage at I/O pin 7 reaches a value of about 1.5 V, then the high phase and internal counting is finished. The counting result is saved in the variable. This count

multiplied with the resolution is the load-time for the RC combination.

READ **BS2 BS2e BS2sx BS2p**

`READ location,variable`

Reads a byte from EEPROM

It means:

location	Location in EEPROM (loc = 0 - 2047)
variable	Variable to save the read byte

Example:

```
'{$STAMP BS2p}

maxmem   con 2048

dta      var byte
addr     var word

         addr = maxmem

         debug cls, "Check EEPROM...", cr

count-tokens:
         addr = addr - 1
         read addr, dta 'read data byte from EEPROM
         debug hex3 addr, ":", hex2 dta, cr
         if dta <> 0 then count-tokens
         debug dec maxmem - addr -1
         debug " token bytes are stored in EEPROM.", cr
```

Remark:

In the program example all tokens are counted and the result is displayed with DEBUG. Check the result with the help of the memory map.

RETURN **BS2 BS2e BS2sx BS2p**

RETURN

Return from a subroutine called by GOSUB

Example:

```
idx       var nib

          debug cls, "Main...", cr
          for idx = 0 to 3
          gosub show  'call subroutine show
               debug "Main...", cr
          next
          debug "End."
          end

          'subroutine for display
show:     debug "Subroutine: ", dec ? idx
          return
```

Remark:

A RETURN without a GOSUB jumps to the first instruction in program.

REVERSE BS2 BS2e BS2sx BS2p

REVERSE pin#

Switches from output to input and vice versa

It means:

pin# I/O pin as input becomes output

Example:

```
low 8              'Output Lo at pin8
input 8            'Change pin8 to input
debug dec ? in8    'Read and display pin8
reverse 8          'Change pin8 to Output
debug dec ? in8    'Read and display pin8
```

Remark:

I/O pin 8 is switched from output to input and with REVERSE back to output.

RUN **BS2e BS2sx BS2p**

`RUN program#`

Switch to another program slot

It means:

program# Number pf program slot (0 – 7)

Example:

```
run 1    ;redirect program execution to page 1
```

Remark:

The program execution is redirected to program slot 1. The program always starts always with the first instruction in a program slot.

Chapter 2.2.2 contains a detailed description of the command using different program slots.

SERIN **BS2 BS2e BS2sx BS2p**

```
SERIN rpin# {\fpin#}, baudmode, {plabel,} {timeout,tlabel,} [inputdata]
```

Receives asynchronous serial data (in accordance with RS-232)

It means:

rpin#	I/O pin for serial input (0-15 = I/O pin, 16 = SIN)
fpin#	I/O pin for handshake (Flow Control)
baudmode	Parameter for baud rate control
plabel	Label to jump to in case of a Parity Errors
timeout	Waiting time for receiving a character (Timeout)
tlabel	Label in case of a timeout
inputdata	List of variables and formatters for serial received data

	BS2, BS2e	*BS2sx*	*BS2p*
Unit in *timeout*	1 ms	0.4 ms	0.4 ms
Baudrate	243 – 50k Baud	680 – 115k Baud	680 – 115k Baud
Max. Baudrate with Flow Control	19.2 kBaud	19.2 kBaud	19.2 kBaud
I/O pins	0 – 15	0 – 15	0 – 15 (MANIO, AUXIO)

Example:

```
'{$STAMP BS2p}

RxD       con 0         'RxD via I/O pin0
Baud      con 16624     'N9600 for BS2p
recchar   var byte
```

```
loop:
     serin RxD, baud, 500, timeout,[recchar]  'Timeout=500ms
     debug "Rec. char.: ", dec recchar, cr
     goto loop

timeout:
     debug "Timeout",CR
     goto loop
```

Remark:

SERIN waits to receive serial data, filters it and converts it for debug display. SERIN is very complex. The example above is quite basic.

Compared to Debug, formatters control the characteristics of serial output on P16. For the SERIN command additional formatter are available:

Special Formatter	*Action*
STR ByteArray \L {\E}	Input a string with length L and saving in an array. If an end character E defined, then the receiving stops with this character and the array is filled with 0.
WAIT (Value)	Waiting for a byte sequence (max. 6 characters) specified by *value*
WAITSTR ByteArray {\L}	Waiting for a byte sequence equal to a string in an array. L is the length of the string. Without L the string must be 0-terminated.
SKIP #	Ignores the number # of characters.

All types of BS2 have a receiver behind their SIN pin (*Rpin=16*). The SIN pin is connected via DB9 connector with the transmit line TxD of the PC's COM port.

Program, download and serial communication during run time use this connection. The interface at the BS2 Carrier Board is assembled only for program downloading.

All types of BS2 can receive serial data on each I/O pin (*Rpin= 0 – 15*). For level conversion a resistor of 22 kΩ connected between the transmit line TxD and the concerning I/O pin is sufficient.

Asynchronous serial communication needs precise timing. Transmitter and receiver must have the same timing. The transmission rate (baud rate) is measured in bits per second (bps) or Baud.

The parameter *baudmode* specifies the time for a received bit, the number of data and parity bits and the polarity.

The parameter *baudmode* is defined as follows:

1.	Bit time (Bit 12-0)	BS2, BS2e:	$\mathbf{INT}\left(\frac{\mathbf{1{,}000{,}000}}{\mathbf{baud\ rate}}\right) - \mathbf{20}$
		BS2sx, BS2p:	$\mathbf{INT}\left(\frac{\mathbf{2{,}500{,}000}}{\mathbf{baud\ rate}}\right) - \mathbf{20}$
2.	Data and parity bit (Bit13)	8-bit/no parity:	0
		7-bit/even parity:	8192
3.	Polarity (Bit14)	True (noninverted):	0
		Inverted:	16384

For the most commonly used baud rates see the *baudmode* parameter listed in the Appendix after Chapter 7.2.

An additional I/O pin handles flow control. If you coded

```
'{$STAMP BS2p}

serdata  var byte
serin 1\0, 240, [dec serdata]
```

then I/O pin 1 serves as input for serial data (Rpin) and I/O pin 0 serves as control output (Fpin). The parameters for serial communication of a BS2p are in accordance with Chapter 7.2 (9600 Baud for eight data bits, 1 stop bit, no parity and normal polarity - no inversion).

This control output (Fpin) signalizes the transmitter with a low level that it is ready for receiving serial data. After receiving all expected characters Fpin switches back to high. This signalizes to the transmitter that the receiver is not ready for receiving additional characters.

For inverted data transmission the polarity of the flow control line also changes.

Chapter 2.2.1 contains some additional program examples with the SERIN command.

SEROUT BS2 BS2e BS2sx BS2p

```
SEROUT tpin#\fpin#, baudmode, {pace,}  {timeout,tlabel,} [outputdata]
```

Transmits asynchronous serial data (in accordance with RS-232)

It means:

tpin#	I/O pin for serial output (0-15 = I/O pin, 16 = SOUT)
fpin#	I/O pin for hand shake (Flow Control, 0-15 = I/O pin)
baudmode	Parameter for baud rate control
pace	Distance of transmitted characters (ms) if no timeout is defined
timeout	Waiting time for ready to receive by Fpin (timeout)
tlabel	Label in case of a timeout
outputdata	List of variables and formatters for serial transmitted data

Example:

```
'{$STAMP BS2p}

TxD       con 16              'TxD at SOUT
baud      con 16624           'N2400
pace      con 200             'Pace = 200 ms

loop:     serout TxD,baud,pace,["Hello world", cr]
          goto loop
```

Remark:

The string *Hello world* is sent via SOUT. Between the characters is a transmission delay of 200 ms.

For the *baudmode* calculation the conditions described in SERIN apply. For the most used baud rates you find the *baudmode* parameter listed in the appendix Chapter 7.2.

The output driver can also be switched to open drain/source. In this case you must add 32768 to the *baudmode* value (Bit15 = 1).

As for DEBUG, previously listed formatter controls change the output format. For the SEROUT command additional formatter are available:

Special Formatter	*Action*
?	Output of a number in format *symbol* = *x* plus CR. Can be combined with BIN or HEX.
ASC ?	Output of an ASCII character in format *symbol* = *x* plus CR.
STR ByteArray {\L}	Sends a string saved in an array. L is the length of that string. If L is not defined then the string must be 0-terminated.
REP Byte \L	Sends a string of character *byte* with the length L.

Chapter 2.2.1 contains some resuming program examples to the SEROUT command.

SHIFTIN BS2 BS2e BS2sx BS2p

`SHIFTIN dpin#,cpin#,mode,[var{\bits},...]`

Receives synchronous serial data

It means:

dpin#	I/O pin for serial input (0–15 = I/O pin)
cpin#	I/O pin as clock output (0–15 = I/O pin)
mode	Mode of operation (0-3)
var	Variable to save received data
bits	Number of bits. Default is eight.

Example:

```
ADres     var byte           'A-to-D result = one byte
CS        con 0              'Chip select is P0
AData     con 1              'ADC data output is P1
CLK       con 2              'Clock is P2

          high CS            'Deselect ADC to start.
again:    low CS             'Activate the ADC0831.
          shiftin AData,CLK,MSBPOST,[ADres\9]
          high CS            'Deactivate ADC0831.
          debug ? ADres      'Show us the result.
          pause 1000         'Wait a second.
          goto again         'Do it again.
```

Remark:

The program example shows the serial data input of nine data bits in MSBPOST Mode. The mode must be selected in accordance with the connected device's shift register.

The following table describes the different modes:

Symbol	*Word*	*Meaning*
MSBPRE	0	First bit is MSB, sample bits before clock pulse
LSBPRE	1	First bit is LSB, sample bits before clock pulse
MSBPOST	2	First bit is MSB, sample bits after clock pulse
LSBPOST	3	First bit is LSB, sample bits after clock pulse

The synchronous serial data exchange reaches high transmission rates. The next table shows the timing:

	BS2 BS2e	*BS2sx BS2p*
Timing T_H/T_L	14 µs/46 µs	5.6 µs/18 µs
Transmission rate	~ 16 kbits/s	~ 42 kbits/s

SHIFTOUT BS2 BS2e BS2sx BS2p

`SHIFTOUT dpin#,cpin#,mode,[var{\bits},...]`

Sends synchronous serial data

It means:

dpin#	I/O pin for serial output (0-15 = I/O pin)
cpin#	I/O pin as clock output (0–15 = I/O pin)
mode	Mode of operation (0-3)
var	Variable with data to sent
bits	Number of bits. Default is eight.

Example:

```
DataP  con 0                    'Data pin to 74HC595
Clock  con 1                    'Shift clock to '595
Latch  con 2                    'Moves data from shift
                                'regiser to output latch.
counter var byte                'Counter for demo program.

again:    shiftout DataP,Clock,MSBFIRST,[counter]
          pulsout Latch,1       'Transfer to outputs
          pause 50                      'Wait 50ms
          counter = counter+1   'Increment counter.
          goto Again            'Do it again.
```

Remark:

The SHIFTOUT command sends the content of the variable `counter` to a shift register. The circuit used in this example (a 74HC595) needs the MSBFIRST mode.

SLEEP **BS2 BS2e BS2sx BS2p**

SLEEP seconds

Switches the BS2 for the given time to Sleep Mode

It means:

seconds Duration of the Sleep Mode in seconds (0...65535)

Example:

```
time     con 5  'sleep time about 5 s

         high 8
         sleep time
         low 8
         end
```

Remark:

Switches into Low-Power Mode (Sleep Mode). The I/O pins stay active. Every 2.3 s the outputs go for about 18 ms to tri-state.

	BS2	*BS2e*	*BS2sx*	*BS2p*
Current consumption during run time	8 mA	25 mA	60 mA	40 mA
Current consumption during sleep	40 µA	60 µA	60 µA	60 µA

STOP **BS2 BS2e BS2sx BS2p**

STOP

Stops the program

Example:

```
tone      var byte

start:    tone = 2000
          freqout 11, 1000, tone
          stop                          'stops the program
          goto start                    'never executed
```

Remark:

The program stops. In opposition to the command end the BS2 does not switch to the low-power mode. All outputs stay active.

STORE BS2p

`STORE Programslot#`

Specifies a program slot for READ and WRITE

It means:

Program slot# Number of the program slot (0...7)

Example:

```
'{$STAMP BS2p, store1}

lang var nib
char var nib
value    var byte
addr var word

         value = 7
         lang = 1

         lookup lang, [0,1], value
         store value
         debug ? value, cr

         addr = text2
start:   read addr, value
         if value = 0 then exit
         debug value
         addr = addr + 1
         goto start
exit:    end

Text1 DATA "Textzeile 1 in Deutsch",0
Text2 DATA "Textzeile 2 in Deutsch",0
Text3 DATA "Textzeile 3 in Deutsch",0
```

Remark:

The variable `value` controls the access to the text strings saved in different program slots of the EEPROM. This way you can implement a multi-language dialog very easy.

For example, here the German text strings are in program slot# 0 and the English text strings are in program slot#1.

The source of the program STORE1.BSP reads as:

```
Text1 DATA "Text line 1 in English",0
Text2 DATA "Text line 2 in English",0
```

```
Text3 DATA "Text line 3 in English",0
```

TOGGLE **BS2 BS2e BS2sx BS2p**

`TOGGLE pin#`

Switches an I/O pin to output and inverts is state

It means:

pin# I/O pin as output

Example:

```
         low 8
loop:    pause 200
         toggle 8
         goto loop
```

Remark:

The state of I/O pin 8 changes every 200 ms.

WRITE **BS2 BS2e BS2sx BS2p**

`WRITE location,variable`

Writes a byte into EEPROM

It means:

location	Memory location in EEPROM (0-2047)
variable	Variable with byte to save

Example:

```
addr      con 0
value     var byte

          value = $AA
          write addr, value
          value = 0
          read  addr,value
          debug hex2 value, cr

          value = $55
          write addr, value
          value = 0
          read  addr,value
          debug hex2 value
```

Remark:

The number of write cycles for EEPROM is limited. For the BS2 a minimum 10 million write cycles are allowed. The other types of BS2 allow 100,000 write cycles.

For BS2p the STORE command selects a program slot for READ and WRITE.

XOUT BS2 BS2e BS2sx BS2p

`XOUT mpin#,zpin#,[house\keyorcommand{\cycles …}]`

Sends a X-10 Powerline Control command to a PL513 or TW523 Powerline Interface Module (60 Hz)

It means:

mpin#	I/O pin for modulation control
zpin#	I/O pin for zero-cross detection
house	House Code (0-15 = “A” - “P”)
key or command	Key Number (0-15 = “1” - “16”) or command cycles

Example:

```
zpin     con  0              'Zero-crossing-detect
mpin     con  1              'Modulation-control pin
houseA   con 0               'House code: 0=A, 1=B . . .
Unit1    con 0               'Unit code: 0=1, 1=2 . . .
Unit2    con 1               'Unit code: 1=2

xout mpin,zPin,[houseA\Unit1]  'Talk to Unit1.
xout mPin,zPin,[houseA\uniton] 'Turn ON.
pause 1000                   'Wait a second.
xout mPin,zPin,[houseA\unitoff]'Turn OFF.
xout mPin,zPin,[houseA\Unit2]  'Talk to Unit2.
xout mPin,zPin,[houseA\unitoff]'Turn OFF.
xout mPin,zPin,[houseA\dim\10] 'Dim unit.
```

Remark:

```
X-10 Commands Value  || Powerline
==================== || Interface    BS2
UNITON         %10010 ||    Pin       Pin
UNITOFF        %11010 || ==================
UNITSOFF       %11100 ||     1        zPin
LIGHTSON       %10100 ||     2        GND
DIM            %11110 ||     3        GND
BRIGHT         %10110 ||     4        mPin
```

2.2 Comments about the Instruction Set

The BS2 commands explained in the last chapter have varying complexity. While the explanation of some commands need less then one page, the explanation of other commands need several pages.

Due to the compact presentation all explanations in the last chapter were short. In this chapter we will give some tips about using some of the more complex commands.

Separate chapters contain extensive explanations to specific commands or command groups like 1-Wire and LCD commands. Therefore they are not listed here.

2.2.1 SERIN and SEROUT

A big strength of all BASIC Stamps are the features of serial data exchange. Practically, each of the 16 I/O pins can be used as a serial input or output.

Questions about serial transmission rates need careful consideration. If you ignore the given possibilities you might be in for a difficult debugging session.

Tracy Allen published a very detailed analysis of the SERIN and SEROUT timing [www.emesystems.com/BS2rs232.htm]. Read this informative web site for more tips.

Note that the serial communication of all BASIC Stamps does not operate in interrupt mode. That means that only a data exchange occurs during execution of the SERIN and SEROUT commands.

This usually doesn't represent a problem for sending data.

Receiving data is more complex. The SERIN command must be executed to get data from a given I/O pin. You have to know when a transmitter will send data to the BASIC Stamp. If nothing is sent the SERIN command waits and blocks the program from further execution. Let us look behind the scene for a proper serial data exchange.

2.2.1.1 SEROUT

The SEROUT command is provided with some parameters briefly explained in chapter 2.1. Here the command is shown with its possible parameters:

```
SEROUT tpin#{\fpin#}, baudmode, {pace,} {timeout,tlabel,} [outputdata]
```

Not all parameter combinations are allowed in command SEROUT.

The simplest serial output operation you will find in many application programs makes use of the used I/O pin, the baud rate and the data to be sent.

```
'{$STAMP BS2}

TxD       con 16             'TxD at SOUT
baud      con 16468          'N9600 for BS2

loop:     serout TxD, baud, ["Hello world", cr]
          pause 1000
          goto loop
```

The short program example above send the string “Hello world“ and a CR in an endless loop. The pause of one second is for a better visualization only and has no importance for the communication itself.

If you comment the PAUSE 1000 command then it is ignored. Inspecting the I/O pin with an oscilloscope you can see short delays between the repeated SEROUT commands yet.

The data stream is not continuous because the SEROUT command must be interpreted at first before the characters can be sent. Beside this there are gaps between every single character.

The next program example shows how the parameter *pace* lengthens the gap.

```
'{$STAMP BS2}

TxD       con 16             'TxD at SOUT
baud      con 16468          'N9600 for BS2
pace      con 500            'Pace = 500 ms

loop:     serout TxD, baud, pace, ["Hello world", cr]
          pause 1000
          goto loop
```

Have a look to the Debug Window of StampW and you can see a delayed serial output of the string “Hello world”.

For the synchronization of the data exchange between receiver and transmitter the BS2 offers the possibility of flow control with an additional I/O pin for handshaking.

The next program example uses I/O pin 15 for flow control.

```
'{$STAMP BS2}

TxD       con 16             'TxD at SOUT
Fpin      con 15             'Fpin is I/O pin 15
baud      con 16468          'N9600 for BS2

loop:     serout TxD, baud, [cls]
          serout TxD\Fpin, baud, ["Hello world", cr]
          goto loop
```

Opposite of the first SEROUT command the second SEROUT command sends data only when I/O pin 15 goes high to signalize that it's "ready to receive".

The parameter *baud* of 16468 switches the BS2 to inverted data transmission mode at 9600 baud (8N1). A receiver must signalize that it is "ready to receive" with high on it's I/O pin responsible for flow control.

If the receiver does not signalize "ready for receive" \then the program example above stops with the second SEROUT command. Use timeouts to avoid such situations.

In our next program example we wait a time of 100 ms for the "Ready to receive". If no "ready to receive" is detected during this time the program branches to a timeout handler beginning with *tlabel*. In our program example the function of this handler is the output "Timeout occurred." in the Debug Window.

```
'{$STAMP BS2}

TxD       con 16         'TxD at SOUT
Fpin      con 15         'Fpin is I/O pin 15
baud      con 16468      'N9600 for BS2
tout      con 100        'Timeout = 100 ms

loop:     serout TxD\Fpin, baud, tout, tlabel, ["Hello world",cr]
          pause 1000
          goto loop

tlabel:   debug "Timeout occurred.", cr
          goto loop
```

Note that a timeout and a delayed transmission of serial data using *pace* can not be combined.

2.2.1.2 SERIN

The SERIN command too is provided with some parameters briefly explained in chapter 2.1. Here the command is shown with its possible parameters:

```
SERIN rpin#{\fpin#}, baudmode, {plabel,} {timeout,tlabel,} [inputdata]
```

We start our short program examples with the simplest input – receiving one single character.

```
'{$STAMP BS2}

RxD       con 16              'RxD at SIN
baud      con 16468           'N9600 for BS2

char var byte

loop:     serin RxD, baud, [char]
          debug cr, "D: ", char
          goto loop
```

The byte variable `char` saves the character received. To simplify matters we use serial input I/O pin 16. We have to note that in this case all received characters will be sent back to the transmitter as an echo.

To distinguish between output from DEBUG and SEROUT we placed ahead the characters “D:” in the DEBUG command.

Sending characters from the PC keyboard to the BASIC Stamp is no problem due to the delays in typing. If the characters come with the specified baud rate (without a gap between the characters) from a measuring device, GPS or something similar then characters can be lost and the communication can hang.

To synchronize the receiver with a transmitter we use flow control. In the next program example we use I/O pin 15 for flow control. The rest of the program is unchanged.

```
'{$STAMP BS2}

RxD       con 16             'RxD at SIN
Fpin      con 15             'Fpin is I/O pin 15
baud      con 16468          'N9600 for BS2

char var byte

loop:     serin RxD\Fpin, baud, [char]
          debug cr, "D: ", char
          goto loop
```

The parameter *baud* of 16468 switches the BS2 into inverted transmission mode with 9600 baud (8N1) again. In this case the BS2 signalizes it's "ready to receive" by showing a high signal on the flow control pin. If the SERIN command is finished then the flow control pin goes low to avoid sending more characters. If the expected data isn't received the program would stop at the SERIN command. Here a timeout would help avoid this situation.

The timeout in the next program example can be used independent of flow control. If there is no data in the `tout` time then the program detects a timeout and branches to the timeout handler `NoData`.

```
'{$STAMP BS2}

RxD       con 16         'RxD at SIN
Fpin      con 15         'Fpin is I/O pin 15
baud      con 16468      'N9600 for BS2
tout      con 1000       'Timeout = 1000 ms

char var byte

loop:     serin RxD\Fpin, baud, tout, NoData, [char]
          debug cr, "D: ", char
          goto loop

NoData:   debug cr, "No Data received - Timeout."
          pause 1000
          goto loop
```

Avoid the values tout = 0 and tout = 1. In these cases the BS2 has a timeout.

For more security of data exchange you can use the parity check (7E1). As shown in the next program example, a parity check works independent of a timeout.

Because the DEBUG command only works with 9600 Baud 8N1 we have to use other I/O pins for serial communication at 9600 Baud 7E1.

```
'{$STAMP BS2}

RxD       con 1              'RxD is I/O pin 1
Fpin      con 0              'Fpin is I/O pin 0
baud      con 24660          'N9600-7E1 for BS2
tout      con 1000           'Timeout = 1000 ms

char var byte

loop:     serin RxD\Fpin, baud, BadData, tout, NoData, [char]
          debug cr, "D: ", char
          goto loop

NoData:   debug cr, "No Data received - Timeout."
          pause 1000
          goto loop

BadData:
          debug cr, "Bad Data - Parity Error."
          pause 1000
          goto loop
```

Receiving single characters is quite rare. Normally we will receive strings and evaluate them in the a PBASIC program. With the many formatters the BS2 gives very good support for these tasks.

The BS2 can use a string as an argument in the SERIN command. We must first declare this string as a byte. After receiving this byte an array is filled with the received characters.

```
'{$STAMP BS2}

RxD       con 16             'RxD at SIN
baud      con 16468          'N9600 for BS2
tout      con 1000           'Timeout = 1000 ms

SerString var byte(11)       'Make a 11-byte array
SerString(11) = 0            'Character 0 is the String Terminator

loop:     serin RxD, baud, [str SerString\10]
          debug str SerString ' Display the string.
          goto loop
```

In our program example we declared an array of 11 bytes. The last bytes were set to 0 during initialization. The SERIN command receives 10 characters and saves them to the byte array.

On the 11th character we encounter a 0-terminated string in the byte array. The DEBUG command expects a 0-terminated string.

A 0 string terminator is often used in strings with a variable length. If you wait for strings with a variable length as input data then SERIN can stop receiving after a defined number of characters or when a special character was detected.

In the next program example a 0-terminated string is expected. Receiving stops after 10 characters or a 0. Use the DEBUG command to visualize this example.

```
'{$STAMP BS2}

RxD       con 16             'RxD at SIN
baud      con 16468          'N9600 for BS2
tout      con 1000           'Timeout = 1000 ms

SerString var byte(11)       'Make a 11-byte array
SerString(11) = 0            'Character 0 is the String Terminator

loop:     serin RxD, baud, [str SerString\10\"0"]
          debug str SerString ' Display the string.
          goto loop
```

The serial data format of a multimeter could look like this:

VDC = 12345 mV<CR> FRQ = 12345 Hz<CR>

If you need the result of the DC voltage measuring only then you can filter these value very simple. The following program example shows the usage of the formatters *wait* and *skip*.

```
'{$STAMP BS2}

RxD       con 16             'RxD at SIN
baud      con 16468          'N9600 for BS2

value     var word

loop:     serin RxD, baud, [wait ("VDC ="), skip 2, dec value]
          debug "Voltage = ", dec value
          goto loop
```

The parameter *wait ("VDC =")* checks the input data stream for the characters “VDC =”. *Skip 2* passes the next two characters – spaces in this example. The characters before CR are changed to a decimal number.

Use the formatter *waitstr* to compare a saved string with the received data. In the next program example a string consisting of four characters is sampled for comparison.

This pattern is compared with the next set of characters received. If the compare finds the strings the same fits then SERIN is finished. The loop repeats after the message “Matching Pattern found.”

```
'{$STAMP BS2}

RxD       con 16             'RxD at SIN
baud      con 16468          'N9600 for BS2

SerString var byte(5)        ' Make a 5-byte array.
SerString(5) = 0             ' Put 0 in last byte.

loop:     serin RxD, baud, [str SerString\4]    'Get the string
          debug "Waiting for: ", str SerString, cr
          serin RxD, baud, [waitstr SerString]  'Wait for a match
          debug "Matching Pattern found.", CR
          goto loop
```

The quickest way to receive data is given in the format `serin RxD, baud, [str SerString\L]`. With this simple syntax the BS2 can receive data with 9600 Baud without gaps. If there are further arguments as *wait* or *skip* then the interpretation needs more resources and the BS2 could loose characters.

For the BS2p there is an enhanced possibility for a temporary saving of the received data using scratch pad RAM.

```
'{$STAMP BS2p}

RxD       con 16             'RxD via SIN
baud      con 16624          'N9600 for BS2p
N         con 16             'Buffer length

char      var byte(N)        'Byte array
addr      var byte           'Address pointer

loop:     serin RxD, baud, [wait("ABC"), spstr N]
          for addr = 0 to 15
               get addr,char(addr)
          next
          debug str char\16, cr
          goto loop
```

The SERIN command saves the characters after recognizing the “ABC”. Then, the next 16 characters are saved to the BS2p’s scratch pad RAM.

To display the received characters with DEBUG the string is first saved in a byte-array and then displayed in the Debug Window.

This way you can save complex strings up to 126 bytes into the scratch pad RAM for further processing or parsing of the individual characters.

You will find such an example in chapter 6.3 for processing GPS data.

2.2.2 RUN

2.2.2.1 Several programs in the controller

Sometimes electronic devices support different functions activated by a query of a switch after power-on. Most of us know these configuration switches as DIP devices mounted on a printed circuit board accessible to the user.

The enhanced EEPROM of the BS2x can contain different applications in different program slots. A DIP switch is connected to the BS2x and after the program starts this switch is queried. Based on this query the program branches to a different program slot.

In the next program example the keys of the BASIC Stamp Activity Board will replace the configuration switch idea by serving as a program selector. The basics of this branching are explained in the next program example.

```
switch    var nib
switch = ins.nib.2 & $03
'debug bin2 switch, cr

BRANCH (switch), [PRG0, PRG1, PRG2, PRG3]
PRG1:     RUN 1  'redirect to page 1
PRG2:     RUN 2  'redirect to page 2
PRG3:     RUN 3  'redirect to page 3
PRG0:            'stay on page 0
...
end
```

The command `switch = ins.NIB2 & $03` queries the I/O pins 11-8 (keys on the BASIC Stamp Activity Board) and masks two bits so only I/O pin 8 (blue key) and I/O pin 9 (black key) can be active.

You can test the branching with the follow programs on the BASIC Stamp Activity Board. To show the links to a project the appropriate Windows Editor screenshots are included.

We have explained the details in chapter 1.3.1. Figure 18 shows the main program in program slot #0. Figure 19 shows one of the executable applications in program slot#1. Here the whole application consists of one DEBUG command only. But this does not matter.

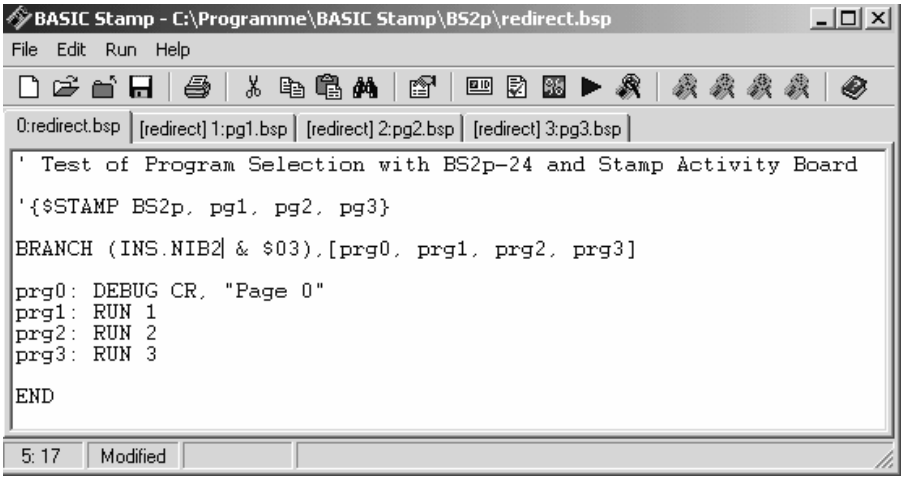


Figure 18 Program REDIRECT.BSP Program Slot #0

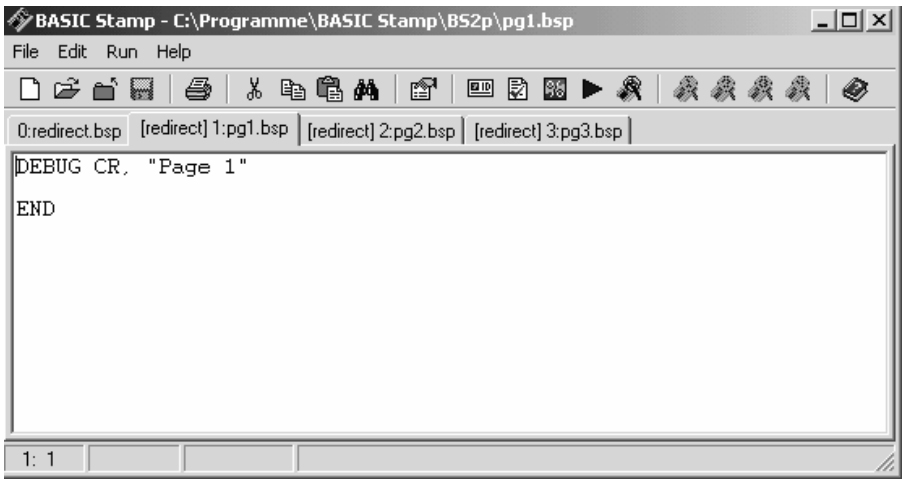


Figure 19 Program REDIRECT.BSP Program Slot #1 (PG1.BSP)

2.2.2.2 Subroutines in different program slots

Some programs are enhanced by using several program slots for program fragments or subroutines.

The goal is to ensure that after processing of such a subroutine that the program jumps to the place it was after the subroutine call back.

Figure 20 shows with dotted lines how to switch between the program slots for the common subroutine technique. The dotted lines show what the RUN command is able to carry out.

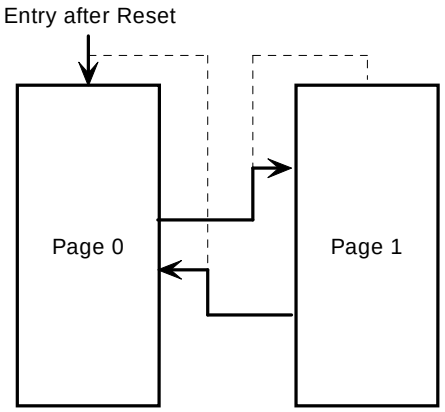


Figure 20 Switch between program slots

A small trick helps us here. At the beginning of each program slot we'll place a branch table. An example shows the principle.

Listing 1 shows the main program in program slot #0. This program slot is active after Reset. Due to the default initialization of the variable `reentry` in slot #0 the main program does not branch and begins at the label `start` to proceed.

As Listing 2 shows, the called subroutines `add` and `mult` are placed in program slot #1.

```
'{$STAMP BS2p, slot1}

'The main program calls subroutines (tasks) in another
'program slot. The point for reentry must be defined
'before the subroutine call.

reentry var nib 'reentry points are _1 and _2
task var nib

x         var byte
y         var byte
z         var word

          branch reentry, [start, _1, _2]

start:    x = 170
          y = 85

          'access task 0 on program slot 1
          reentry = 1 : task = 0 : run 1
_1:       debug dec5 z, cr

          'access task 1 on program slot 1
          reentry = 2 : task = 1 : run 1
_2:       debug dec5 z, cr
          end
```

Listing 1 Main Program in Program Slot #0

```
reentry   var nib       'used by page0
task      var nib       'used by page0

x         var byte
y         var byte
z         var word

          branch task, [add, mult]

add: z = x + y          'operate task 0
          run 0         'reentry in page0

mult:      z = x * y    'operate task 1
          run 0         'reentry in page0
```

Listing 2 Subroutines in Program Slot #1

Use some precaution when calling these subroutines after the initialization of the variables x and y.

The subroutine `add` in program slot #1 is declared as task 0 in the `branch` command, while the subroutine `mult` is declared as task 1. Accordingly the variable `task` must be set in program slot #0 before the call of the respective subroutine.

In the same way we have to organize the return with the help of the variable `reentry`. The line following the subroutine call must be marked with a Reentry Label. After returning to program slot #0 the `branch` command branches to the referred label.

To avoid access conflicts on the variables, both program slots begin with the variable declarations. This ensures the declaration of the variables is global.

2.2.3 Switching the I/O Blocks with the BS2p

In the next program (Listing 3) you can test the functionality of the I/O Block Switching using the BASIC Stamp Activity Board and a BS2p-24.

```
' I/O Test for BASIC Stamp Activity Board with BS2p-24

'{$STAMP BS2p}

         low 11        ' Red key LED on
         gosub brk     ' To leave breakpoint press the blue key
         high 11       ' Red key LED off
         ' auxio
         low 10        ' Black key LED on
         gosub brk     ' To leave breakpoint press the blue key
         high 10       ' Black key LED off
         mainio
         end

brk:     debug cr, "Press the blue key to continue..."
brk1:    if in8=1 then brk1
         pause 100
brk2:    if in8=0 then brk2
         return
```

Listing 3 Switching the I/O Blocks

In this source code example the command `auxio` is commented by the leading character ' and is without any effect.

After the program starts the LED connected to I/O pin 11 (located above the red key on the BASIC Stamp Activity Board) is switched on. Next the first breakpoint is reached.

Leave this breakpoint by a making I/O pin 8 low (or by pressing the blue key on the Stamp Activity Boards).

In the next step the LED connected to I/O pin 11 switches off and the LED connected to I/O pin 10 switches on. Handle the next breakpoint as before and the LED connected to I/O pin 10 switches off, too.

Now activate the I/O Block Switch by removing the comment character ' before `auxio` and download again.

You will see that the LED connected to I/O pin 10 does not work. The I/O operation points to the AUXIO block not available in a BS2p-24. Therefore this I/O operation has no effect in a BS2p-24.

2.2.4 Interrupting by Polling the BS2p

The BASIC Stamp 2 has no hardware interrupt mechanism. For reactions to external events this is sometimes a disadvantage.

Sure, there was the possibility to check any I/O pin periodically for its logical condition, but a short response to a certain logic state makes this option difficult.

The BS2p has a polling mechanism for time-critical applications. This polling mechanism eliminates the disadvantage mentioned above associated with sporadically check I/O states. But, do not confuse this polling with real interrupts. Table 4 lists an overview for the Polling Commands.

Polling Command	*Effect*
POLLIN Pin, State	Configuration of checked inputs
POLLOUT Pin, State	Defines the output activity after a polled-input event occurred
POLLMODE *Mode*	Defines the Polling Mode
POLLRUN *Slot*	Specifies a program running after a polled-input event occurred
POLLWAIT *Period*	Switches the BS2p for a certain time into sleep state and polls the polling inputs afterwards

Table 4 Overview of BS2p Polling Commands

We'll describe this new polling mechanism next with small and clear program examples in significant detail.

Use the `POLLIN` command to declare the I/O pin to poll and define the state of the polled-input event. The output activity depends on the commands `POLLOUT`, `POLLMODE` and `POLLRUN` and is triggered when the polled-input event occurred.

Several I/O pins can be declared for polling. The polled-input event occurs if one of the polling conditions comes true.

The BS2p firmware checks the I/O pins defined by the command `POLLIN` at the end of each instruction and before reading the next PBASIC instruction – practically as a background task in that it is done between lines of code. This mechanism allows a significantly faster response to a polled-input event than a programmed query of I/O pins by PBASIC instructions.

The `POLLOUT` command declares an I/O pin and it's state after a polled-input event occurs.

The `POLLMODE` command defines the polling mode according to Table 5.

Mode	*Effect*
0	Polling disabled, Clears the POLLIN and POLLOUT configuration
1	Polling disabled, Saves the POLLIN and POLLOUT configuration
2	Polling enabled with POLLOUT and POLLWAIT function
3	Polling enabled only with POLLRUN function
4	Polling enabled with all Polling functions
5	Clears the POLLIN configuration
6	Clears the POLLOUT configuration
7	Clears the POLLIN and the POLLOUT configuration
8-15	Identically to 0-7, but POLLOUT states are saved

Table 5 POLLMODE Parameter

With this information we can have a look to the first program example.

```
'{$STAMP BS2p}

InitPolling:
     POLLIN 8,0          ' If this polled input is Lo
     POLLIN 9,0          ' or this input is Lo
     POLLOUT 11,0        ' then the polled output is Lo
     POLLMODE  2         ' Pollout only

Loop:                    ' This in the "main" program
     DEBUG "."
     goto LOOP
```

I/O pins 8 and 9 will be polled. The polled-input event occurs when one or both polled inputs are low.

Using the BASIC Stamp Activity Board this means pressing the blue or the black key.

When the polled-input event occurs I/O pin 11 switches low and the LED switches on.

The defined polling mode allows only the `POLLOUT` and the `POLLWAIT` function. `POLLWAIT` is not used here.

After initializing the polling the program runs an endless loop and displays it's activity with the `DEBUG` command.

In our program example the output defined by `POLLOUT 11, 0` reacts to the polled-input event directly.

As long as one polling input detects low the polling output is also low, the polling output switches to high immediately. The Latch Option of the poll mode can freeze this state. This means after the polled-input event the polling output switches to low as before, but stays low independent what happens on the polling inputs.

In the next program example the poll mode was changed from 2 to 10.

```
'{$STAMP BS2p}

InitPolling:
     POLLIN 8,0     ' If this polled input is Lo
     POLLIN 9,0     ' or this input is Lo
     POLLOUT 11,0   ' then the polled output is Lo
     POLLMODE 10

Loop:               ' This in the "main" program
     DEBUG "."
     goto LOOP
```

With this latch function we have the ability to monitor an I/O pin and to react to the polled-input event later. The polled-input event can be processed after a continuous routine.

Again, we modify our program example.

```
'{$STAMP BS2p}

i var NIB

InitPolling:
     POLLIN 8,0     ' If this polled input is Lo
     POLLIN 9,0     ' or this input is Lo
     POLLOUT 11,0   ' then the polled output is Lo
     POLLMODE 10    ' Enable latched POLLOUT only

Loop:               ' This in the "main" program
     ' Print always 10 dots without any interruption
     for i= 0 to 9
          DEBUG "." : pause 100
```

```
    next
    DEBUG CR
    ' on a latched polling event goto the handler
    IF IN11=0 THEN Event
    goto LOOP

Event:                ' handler for latched polling event
    DEBUG CR, "Latched Polling detected.", CR
    POLLMODE 10       ' restore of polling initialization
    goto LOOP
```

In the main loop DEBUG outputs a package of 10 dots before querying I/O pin 11. If this I/O pin were set to low from a polled-input event occurring during DEBUG outputs then the program branches to the label Event and outputs "Latched Polling detected." The following POLLMODE 10 command re-initializes the polling and I/O pin 11 switches to high again.

The command POLLRUN specifies a program that runs after an polled-input event occurred. This program can be saved in any program slot. The slot number is parameter of POLLRUN.

We add the command POLLRUN 1 to the last program example and change the poll mode to 4 to enable all polling activities.

```
'{$STAMP BS2p, pollrun}

InitPolling:
    POLLIN 8,0    ' If this polled input is Lo
    POLLIN 9,0    ' or this input is Lo
    POLLOUT 11,0  ' then the polled output is Lo
    POLLRUN 1     ' and program in Slot #1 runs.
    POLLMODE 4    ' All polling functions enabled

Loop:             ' This in the "main" program
    DEBUG "."
    goto LOOP
```

As defined, the program running after the polled-input event occurred must be placed in program slot #1.

```
' POLLRUN Activity

loop:
     debug "+"
     goto loop
```

After the program downloads it starts by initializing the polling and running in an endless loop. In this endless loop the DEBUG command outputs one dot after another.

When the polled-input event occurred I/O pin 11 switched to low and the program in program slot #1 begins. You can watch the Pollrun activity by a repeated output of the character "+" by the DEBUG command.

Finally, let's have a look to the POLLWAIT command. POLLWAIT queries (a minimum of) one polling input periodically while the BS2p switches between these queries to its power-down mode with about 60 µA current consumption. The periodic query corresponds to the NAP command and its parameters are listed in Table 6.

Period	*Polling Cycle*
0	10 ms
1	36 ms
2	72 ms
3	144 ms
4	288 ms
5	576 ms
6	1,152 s
7	2,304 s
8	No Power-Down (< 160 µs)

Table 6 POLLWAIT Parameter

Let's modify our program example for some additional experimentation.

```
'{$STAMP BS2p}

InitPolling:
     POLLIN 8,0          ' If this polled input is Lo
     POLLIN 9,0          ' or this input is Lo
     POLLOUT 11,0        ' then the polled output is Lo
     POLLMODE 2          ' All polling functions enabled

Loop:                    ' This in the "main" program
     POLLWAIT 7          ' Polling period 2.3 s
     DEBUG CR, "Polling event detected"
     goto Loop
```

The endless loop contains the command `POLLWAIT 7` which means every 2.3 s both polling inputs will be queried. When the polled-input event occurs the command `DEBUG CR, "..."` is executed. Afterwards the BS2p goes to sleep again.

With `POLLWAIT 8` if an polled-input event is detected in less then 160 µs the BS2p does not switch to the Power-Down Mode.

The Polling Mode of the BS2p is an excellent way for a fast reaction to external events. The possibilities discussed here were explained with some simple program examples. Before using the polling in your own application experiment with similar simple program snippets to reproduce the desired behavior.

To generate a more readable source avoid the use of pin numbers and constant definitions instead. For example:

```
DisPollClrCfg      con 0               ' PollMode 0
DisPollSaveCfg     con 1               ' PollMode 1
EnPolledOutput     con 2               ' PollMode 2
EnPolledRun        con 3               ' PollMode 3
EnPolledAll        con 4               ' PollMode 4
ClrPollInpCfg      con 5               ' PollMode 5
ClrPollOutCfg      con 6               ' PollMode 6
ClrPolIOCfg        con 7               ' PollMode 7

stateHi            con 1
stateLo            con 0

pIN1               con 8               ' PollInput Pin# 8
pIN2               con 9               ' PollInput Pin# 9
pOUT               con 10              ' PollOutput Pin# 10

InitPolling:
     POLLIN pIN1, stateLo              ' If this polled input is Lo
```

```
     POLLIN pIN2, stateLo               ' or this pin is Lo then
     POLLOUT pOUT, stateLo              ' the polled output is Lo
     POLLMODE EnPolledOutput

Loop:                                   ' This in the "main" program
     DEBUG "."
     goto LOOP
```

3 Enhanced I/O

Beyond direct serial I/O control used in the BS2, the BS2p has additional capabilities useful for networking control of more complex devices.

3.1 I²C-Bus

The I²C-Bus was developed for data exchange between different devices such as EEPROMs, RAMs, A/D and D/A converters, RTCs and microcontrollers in a networked environment.

Figure 21 shows all required connections in a typical I²C-Bus network. The SDA and SCL lines connect all members of the network. Pull-up resistors connect these lines to the supply voltage V_{CC}.

In an I²C-Bus network several masters can be connected with several slaves (in a Multi-Master System). The I²C-Bus protocol addresses the members of the network.

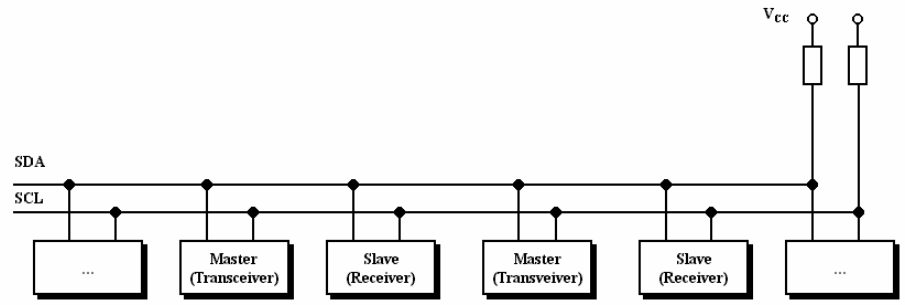


Figure 21 I²C-Bus Network

Figure 22 shows the protocol for writing and reading of one byte. There are further functions, as sequential write and read. We'll concentrate on the basic functions only.

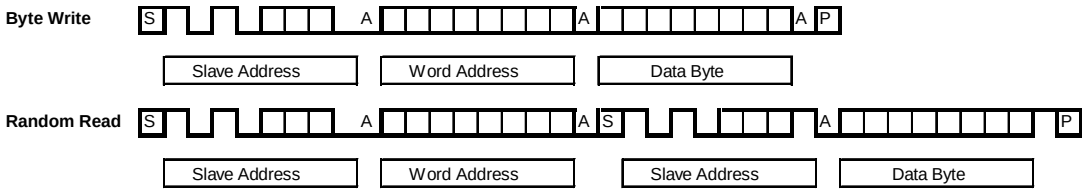


Figure 22 Writing and Reading of one Byte

The peripheral functions depend on the specific device under control. Beside EEPROMs and RAMs from numerous manufacturers there are a lot of further I^2C-Bus devices:

- I/O expander devices
- LCD and LED driver devices
- Video controller
- PAL/NTSC TV Processors
- TV and VTR Stereo/Dual Sound Processors with integrated filters
- Hi-Fi Stereo audio processor interface for color decoder
- YUV/RGB switches
- Programmable modulators for negative video modulation and FM sound
- Satellite sound receiver
- Programmable RF modulators
- BTSC Stereo/SAP decoder and audio processor
- 1.3 and 2.5 GHz bi-directional synthesizer
- 1.4 GHz multimedia synthesizer

The usage of the I^2C-bus is effective when several I^2C-Bus devices are connected using their the SDA and SCL lines. The number of pins used on the BS2p for a data exchange via I^2C-Bus is limited to these two lines. The serial transmission allows a transmission of data to several different addressed I^2C-Bus devices.

3.1.1 Printer Control with I²C Output

With the commands `I2CIN` and `I2COUT` the BS2P communicates using the I²C-Bus.

This chapter shows the I²C output with the `I2COUT` command. This statement, written in one line, handles a lot of data dialogue over the SDA and SCL lines.

The format of the commands `I2CIN` and `I2COUT` were shown in Chapter 2.1. It is useful to remember to the earlier types of BASIC Stamps to see the many PBASIC statements required for creating the necessary pulse scheme used in the I²C-Bus connection. In an additional paragraph we'll show such an example with the BS2 but first we'll use the BS2p's easy–to-use command set.

We will use the PCF8574A I/O expander circuit from Philips as a receiver for the I²C-Bus telegram.

With this integrated circuit a serial transported data byte becomes an eight-bit parallel output.

Using the PCF 8574A I/O expander the BS2 uses only 2 I/O lines (SDA and SCL) to output eight data bits in parallel.

What can we do with the single byte output of PCF8574A?

In this sample application a printer's Centronics port is connected to the PCF8574A. Printers are frequently discarded or replaced upon installation of new computer equipment, making an ample surplus supply readily available. As a result these reliable-working needle-based printers are stored in many corners of your office.

With a BASIC Stamp we can use these printers for an occasional data output device.

In short we'll show the Centronics transmission protocol. Detailed descriptions with web links are shown in Chapter 9.

Figure 23 shows the pulse scheme of a Centronics byte transmission to the printer port.

The byte transmitted to the printer is a printable character or a control byte for the printer itself.

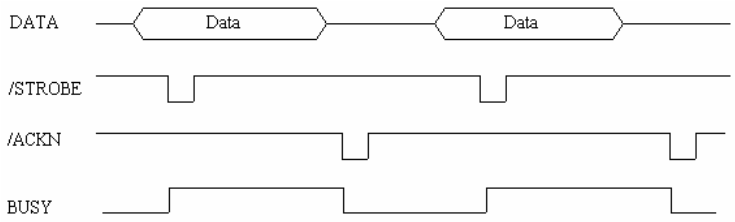


Figure 23 Centronics Port Data Exchange

For our simple example we need a minimum of two lines for DATA and /STROBE.

A data word from lines D1-D8 is adopted and interpreted from the printer with the low-active strobe signal. It is stored in the data buffer for printing after recognizing the CR/LF signal ($0D,$0A). A control byte is transmitted to control the printer state.

It's also practical to include the BUSY signal, which comes from printer output.

A high level on this BUSY line means one of the following situations:

printer receives or interprets a data byte

printer is OFFLINE

printer is in error state

In addition, Figure 23 shows the active low signal /ACKN. With this signal the printer signalizes its readiness for receiving new data. It is prepared for an interrupt signal to a connected data transmitting device to start the transmission of a next data byte. This line is not used in the given example. Also not used are the additional printer control lines like missing paper (PE) or the common error-message /ERROR.

Figure 24 shows the circuit diagram to connect a printer to a PCF8574A using the I²C-Bus.

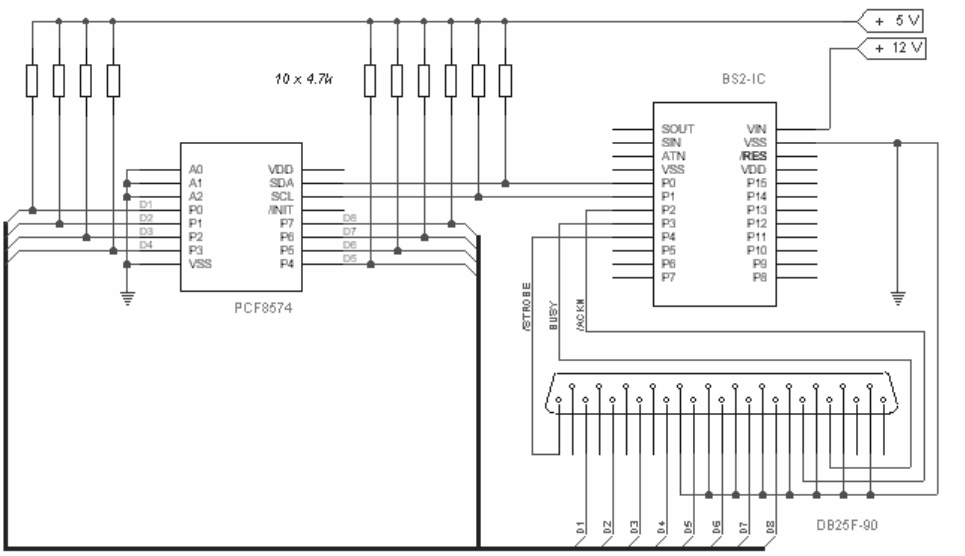


Figure 24 Printer Control with PCF8574A

In Figure 24 the data lines from the PCF8574A's output to the printer connector are drawn on the bus connection. Because the PCF8574A's outputs are not able to drive the connected inputs in a high state, we need pull-up resistors on these eight data lines.

Missing pull-up resistors are often the reason for errors in data transmission with I²C-Bus connections.

There are many ways to print data from the BASIC Stamp with this circuit. One of them is to receive data serially by the command SERIN. Another is the data generation by a connected ADC or RTC. In our example we load a test string from the internal EEPROM. This test string is terminated by ASCII 0 (0-terminated string) to recognize the end of the string.

In a program loop we will read all data bytes in sequence from EEPROM and transmitted them via the PCF8574A to the printer buffer. After receiving the CR/LF the printer starts printing two lines.

Listing 4 shows the program PRN8574.BSP.

```
' -----[ Title ]-----------------------------------------
'
' File......  prn8574.BSP
' Purpose...  Sample for I2COUT - statement.
' Author....  Klaus Zahnert
' Started...  10.6.01
' Updated...
'
' -----[ Program Description ]-----------------------------
'
' Serial data transmission from BS2P to PCF8574 to get
' parallel data for CENTRONICS printer interface.
' Separate lines between BS2P and printer control
' BUSY and /STROBE. ACKN and other controls are not used.
'
' -----[ Revision History ]--------------------------------
'
'
' -----[ BS Type ]-----------------------------------------
'
'{$STAMP BS2p}
'
' -----[ Constants ]---------------------------------------
'
adr9574Aw     con %01110000            'D0 = 0 for write
strb          con   4
datapin       con   0                  'low byte adressing I2C

' -----[ Variables ]---------------------------------------
'
pos           var  byte
rdbyt         var  byte

' -----[ Initialization ]----------------------------------
'
dirl = %00010000                       'P4 = /STROBE for output
high strb

data "the big brown fox jumps over the lazy dog",13,10
data "1234567890!§$%&/()=?",13,10
data $00                               'termination of string

pos = 0

' -----[ Main Code ]--------------------------------------
'
waitbusy:
     if in3 = 1 then waitbusy          'wait for printer
    read pos,rdbyt                     'read act. Byte from EEPPROM
    if rdbyt = $00 then finis          'goto end if recognized $00

     debug dec pos,tab,hex2 rdbyt,tab,tab,rdbyt,cr
                                       'display to screen
```

```
     pos = pos + 1                      'prepare next position

     i2cout datapin, adr9574Aw, 0, [rdbyt] 'output to PCF8574A
     low  strb                          '/STROBE-pulse to printer
     high strb
     goto waitbusy

finis:
     end
```

Listing 4 Printer Control via I²C-Bus (PRN8574.BSP)

The loop starts with label `waitbusy`. Polling the BUSY line is done before a character is transmitted to the printer. This is a test of the readiness of the printer to receive a new character.

After this the actual character is read from the addressed position of EEPROM. The output to the printer is done using the `I2COUT` command with a following /STROBE signal at I/O-pin 9 of BS2P.

The printer starts printing the complete string after receiving CR/LF. On the print-outs we have an image of all the printable characters saved in the EEPROM (see the DATA lines).

An eye-catching fact in Listing 4 is shown following the command `high btrb` after `low strb`. The Centronics specifications indicates that the active-low strobe pulse needs a minimal length of 1 µs. The pulse length resulting from `low strb` followed by `high btrb` is longer due to the time required for the command to execute from the BASIC Stamp's interpreter.

The end of reading from EEPROM is recognized by the byte with value 00 used as string terminator as described before.

3.1.2 Reading and Writing EEPROMs

You're probably already familiar with EEPROMs from working with the BASIC Stamp.

The BASIC Stamp's EEPROM is used as a re-writeable static memory for program tokens and any program data you might want to store. The following program example demonstrates the connection of a 2 KByte 24LC16 EEPROM as an external memory connected to the BS2p's I/O pins. With this circuit you can save data permanently. It is useful as a memory for datalogger applications or to transmit the EEPROM data to a PC via serial output. By changing the direction of this transmission the PC can write data to this EEPROM.

Using a BS2p's program slot as a temporary memory storage it is possible to duplicate (copy) the contents of an entire EEPROM. With two simultaneous connected external EEPROMs the device is able to copy the contents of one EEPROM to another directly.

Both `I2CIN` and `I2COUT` commands transmit the first byte as the device-address. EEPROMs with higher capacity need a second byte as a sub-address.

In our example we use an address byte, whose bits are a device-specific device address (DA) letting you select a particular device from several other devices on the bus. The address selection A2-A0 must correspond to the wired address lines A2-A0 of the concerning device.

Data direction READ or /WRITE

So the address byte is formatted as shown below:

B7	B6	B5	B4	B3	B2	B1	B0
DA3	DA2	DA1	DA0	A2	A1	A0	R/W
1	0	1	0	0	0	0	1/0

The device address for the used EEPROM 24LC16 is $0A and A2-A0 which must correspond to the wired address lines A2-A0.

Figure 25 shows the connection of a 24LC16 EEPROM to the BS2p using the I²C-Bus lines SDA and SCL.

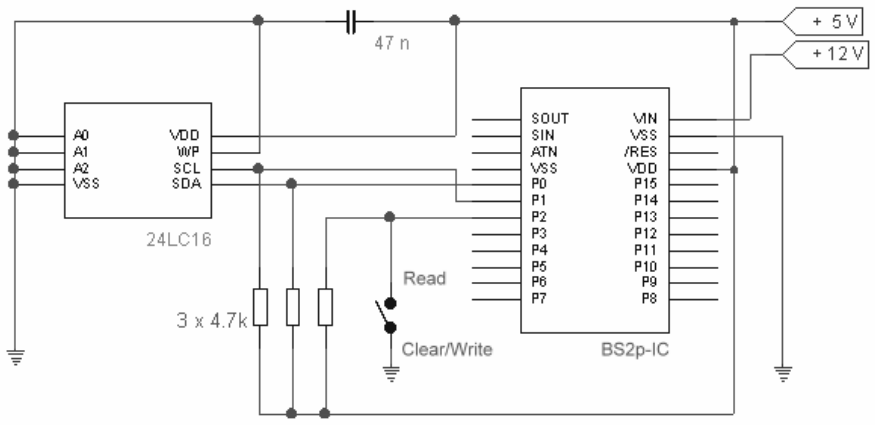


Figure 25 Connection of an EEPROM 24LC16 to a BS2P

The state of the pushbutton is sampled from BS2p's P2. A pressed pushbutton drives I/O pin 2 low and the EEPROM is available for deleting and writing. An unpressed pushbutton (open state, as shown on Figure 25) means the EEPROM is ready for reading its content.

All operations of reading and writing data are secured by re-reading for test purposes. For this we use program slot #5 in our program example for temporary memory.

The progress of reading or writing each data byte involves some additional delay to get a visible (slower) operation you can see in the debug window. If a byte read differs from a byte written an error message in Debug Window occurs. The program also stops.

To delete all data from EEPROM, we send $FF to each cell.

```
' -----[ Title ]-----------------------------------------------
'
' File EEPR_rw1.BSP
' Purpose...Sample for I2IN /I2OUT:
'           clone external EEPROM via bank 5
'
' Author....Klaus Zahnert
' Started...1.7.01
' Updated...

' -----[ Program Description ]--------------------------------
'
' External switched pin 2 runs two separate parts of program.
' With pin 2 = high a plugged programmed EEPROM gives his
' contents to bank 5 as an mirror.
' With pin 2 = low  a plugged EEPROM is deleted and then the
' contents of bank 5 is transmitted (re-mirrored) to this
' EEPROM.
' All transmitted bytes are verified from source to
' destination and displayed on screen.

' -----[ Revision History ]-----------------------------------
'
'
' -----[ Constants ]------------------------------------------
'
datapin   con  0        'high byte addressing I2C
wradr     con  $A0      'address of EEPROM write
rdadr     con  $A1      'address of EEPROM read

' -----[ Variables ]------------------------------------------
'
n         var  word
rdbyt     var  byte
```

```
backbyt  var byte

' -----[ Initialization ]----------------------------------
'
dir2 = 0                              'pin is input for switch

' -----[ Main Code ]-----------------------------------------
'
'{$stamp bs2p}

    debug cls

    if IN2 = 0 then wrEEPR            'IN2=1:read external EEPROM
                                      'IN2=0:del./write ext.EEPR.

rdEEPR:
    debug cr                          'from MASTER-EEPROM to Bank5
    debug"read EEPROM",cr,cr
    pause 1000
    For n=0 to 2047
        i2cin datapin,rdadr,n,[rdbyt] 'read ext. EEPROM-cell
       store 5: write n,rdbyt         'write to bank-cell
       store 5: read  n,backbyt       'read back for verify
       if rdbyt <> backbyt then errstr
         debug dec4 n,tab,hex rdbyt,tab,"read ok",cr
    next
    goto finis

wrEEPR:
    debug cr                          'del. EEPROM with filled $FF
    debug"delete EEPROM",cr,cr
    pause 1000

    For n=0 to 2047
    i2cout datapin,wradr,n,[$FF]      'kill this line for
                                      'test of EEPROM empty
       pause 10
       i2cin datapin,rdadr,n,[backbyt]
       if backbyt<>$FF then errstr
       debug dec4 n,tab,"delete",cr
    next
    'end '==================>this statement for delete only

    debug cr                          'write EEPROM from bank5 cells
    debug"write EEPROM",cr,cr
    pause 500

    for n=0 to 2047                   'from Bank5 to new EEPROM
    store 5: read n,rdbyt
       i2cout datapin,wradr,n,[rdbyt]
        pause 10
       i2cin datapin,rdadr,n,[backbyt]
       if rdbyt<>backbyt then errstr
```

```
         debug dec4 n,tab,hex rdbyt,tab,"write ok",cr
    next
    goto finis


errstr:
     debug "         E R R O R on cell-address  =  ",dec n,cr
     goto finis

finis:
     debug cr
     debug "end of program ",cr
     end
```

Listing 5
Copy the content of an EEPROMs with temporary saving (EEPR_RW1.BSP)

The program contains a commented line with the command `end`. If you activate this line the program ends after deleting the EEPROM contents by writing $FF to each cell.

Changing the data is done with the push button. The I/O pin signalizes its state to “delete only” or to “delete and write” in the same manner as shown with the commands “read “ and “write”.

For direct copy from one EEPROM to another of the same type, a second EEPROM must be added to the circuit. There are two possibilities for doing this.

Either the second EEPROM (destination for written data) is connected with it’s SDA and SCL lines or both EEPROMs are on the same SDA and SCL line - the classical usage of the I²C-Bus.

When using the same SDA and SCL lines the address bits A2-A0 must be used for selecting the device. In the program example the different address bytes `rd_adr` and `wr_adr` must carry the different address bytes A2-A0 to be aligned to the wiring of these lines.

For example, address bit A0 of the EEPROM can be set to high. The EEPROM to be read would have all address lines grounded (A2.. A0 = low). To identify the writeable EEPROM, its address byte is settled to `wr_address = $A3`.

3.1.3 LCD-Controller PCF2116 on I²C-Bus

This chapter describes the Philips PCF2116 LCD controller and its usage with a chip-on-glass LCD module. A BS2p controls the LCD using I²C-Bus.

Because the easy-to-use I²C-Bus commands are only present on the BS2p, we’ll demonstrate the application with a slightly different approach using a standard BS2 module.

The PCF2116 LCD controller has a 4-bit and 8-bit parallel bus on an I²C-bus interface so it's well-suited for this type of application.

The PCF2116 has the following features:

- Single-chip LCD controller and driver
- Control of LCDs with one or two lines with 24 characters/line or two to four lines with 12 characters/line
- 5x8 pixel/character
- Voltage generation for the LCD and oscillator are on chip (external power supply is also possible)
- Display RAM for 80 characters (not all cells of the display RAM are visible)
- Character ROM for 240 characters and character RAM for 16 characters
- I²C-Bus Interface (with device address $74 or $76) and 4-Bit or 8-bit parallel bus
- CMOS/TTL compatible
- HD44780 compatible instruction set (11 instructions)
- Supply voltage VDD-VSS between 2,5 and 6 V DC – low current consumption

Addressing the PCF2116 LCD controller is the same as other I²C-Bus devices. Several devices of the same type can operate on the same bus. Address line SA0 is used for programming the device subaddress, so two PCF2116s can operate in the same I²C-Bus.

The PCF2116's instruction set is compatible to the standard HD44780 LCD controller but is not provided in detail here. The required explanations follow on the program example.

The I²C-bus protocol for LCDs must be understood quite carefully. The lines used in parallel control mode are to be configured as follows:

Control Byte								Command Byte							
Co	RS	R/-W	*	*	*	*	*	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0

The LCD controller's instruction set contains the control bit "Co". Co = 1 in the control byte signalizes that further control bytes follow, while Co = 0 the last control byte marks the end.

Figure 26 shows a sequence for writing data to the LCD module.

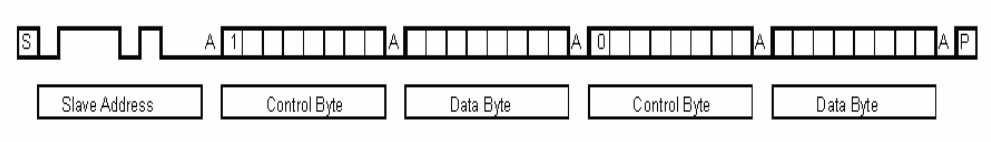


Figure 26 Writing Data to the LCD Module

After the I^2C start condition (S) you'll be sending the slave or device address (here $74). After sending the command one control byte and one data byte always follows. The last command is marked with Co = 0 in its control byte. The whole sequence is finished with the I^2C stop condition (P).

Before explaining the program example it's helpful to learn more about the LCD module.

3.1.3.1 Chip-On-Glass LCD Module MDLS12305COG

The PCF2116 LCD controller is offered in miniaturized chip packages. Several chip-on-glass modules built with this LCD controller are available.

For example, Varitronix offers chip-on-glass LCD modules not thicker than the LCD itself. The thinnest modules available are 1.8 mm thick with 0.7 mm glass substrate. You will find these thin and light-weight LCD modules in mobile phones and pagers.

The MDLS12305COG LCD module from Varitronix shown in Figure 27 functions well with a BASIC Stamp as a complete display unit.

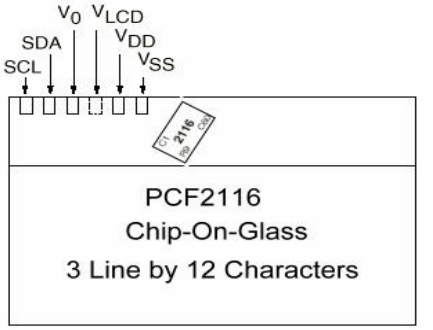


Figure 27 MDLS12305COG

The PCF2116 LCD controller can drive four lines with 12 characters but the MDLS12305COG module offers only three lines with 12 characters.

Figure 28 shows the pixel matrix of this LCD module with the test string "**LCD Control with PCF2116 and StampII.**" displayed. To display one character a 5x8 pixel matrix is used. The bottom pixel line is used to display a cursor.

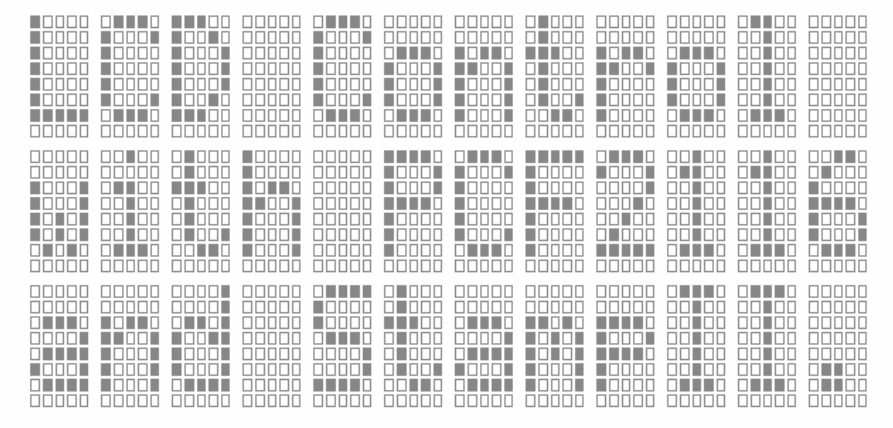


Figure 28 MDLS12305COG Pixel Matrix with Test String

The PCF2116's display RAM contains 80 Bytes. Figure 28 shows that only a part of the display RAM is actually used to display characters.

Figure 29 shows a memory map of the display RAM with display and memory area. The dark cells can be used as common RAM while the contents of the light cells are displayed. Pay attention to these different memory areas in the case of direct addressing of the display RAM.

Spalte	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*
Zeile1	**00**	**01**	**02**	**03**	**04**	**05**	**06**	**07**	**08**	**09**	**0A**	**0B**	0C	0D	0E	0F	10	11	12	13
Zeile2	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**	**28**	**29**	**2A**	**2B**	2C	2D	2E	2F	30	31	32	33
Zeile3	**40**	**41**	**42**	**43**	**44**	**45**	**46**	**47**	**48**	**49**	**4A**	**4B**	4C	4D	4E	4F	50	51	52	53
Zeile4	60	61	62	63	64	65	66	67	68	69	6A	6B	6C	6D	6E	6F	70	71	72	73

Figure 29 Memory Map of the Display RAM

An unfortunate characteristic of the MDLS12305COG LCD module is it's internal grounded address line SA0 for the LCD-Controller. Therefore you can use the device only at address $74 and the I²C-Bus can be used to control only one MDLS12305COG LCD module.

3.1.3.2 Display Unit with MDLS12305COG LCD Module and BS2

The hardware of the display unit is quite simple because we will use the internal features of the MDLS12305COG and the BS2 has to control the both I²C-Bus lines only. Figure 30 shows the required connections between BS2 and LCD module.

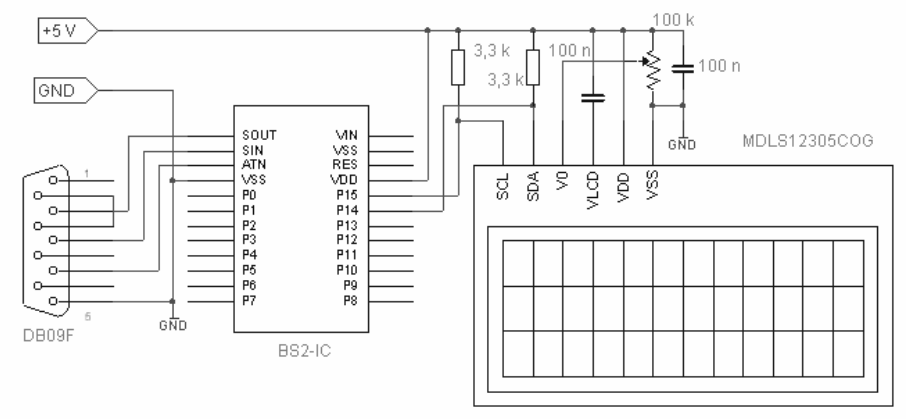


Figure 30 Display Unit

The LCD module generates the LCD voltage internally. The potentiometer delivers the contrast controlling voltage externally. The capacitor provides voltage filtering.

Both I^2C-Bus lines have pull-up resistors which generate a high level on passive bus lines.

A RS-232 interface connects the BASIC Stamp to a PC for debugging and program download.

In our program example (Listing 6) there are three strings for display saved in EEPROM. Each string is terminated by ASCII 0.

```
' -----[ Title ]-----------------------------------------------
'
' File......  I2C_LCD.BS2
' Purpose...  I2C-Interface for LCD driver PCF2116
' Author....  Claus Kuehnel
' Started...  29.12.97
' Updated...  15.09.01

' -----[ Program Description ]-------------------------------
'
' The program demonstrates some control features of the LCD
' driver PCF2116 from Philips.
'
' Some subroutines used in this program
' came from H.Paul Roach, MSU, May 3, '97

' -----[ Revision History ]----------------------------------
'
' 29.12.97: Version 1.0
' 15.09.01: Version 1.1

' -----[ Constants ]-----------------------------------------
'
SDAPIN    con 14
SCLPIN    con 15
TSTPIN    con 0
OUT       con 1
IN        con 0

PCF2116   con $74

CTRL      con $80
WRTDTA    con $40

CLRSCR    con 1
DISPON    con $0C
LFTSHFT   con $18
RGTSHFT   con $1C
MODE      con $3E
```

```
LINE1     con $80
LINE2     con $A0
LINE3     con $C0

delay     con 1

' -----[ Variables ]----------------------------------------
'
SDAOUT    var out14
SDAIN     var in14
SDADIR    var dir14
SCLOUT    var out15

device    var nib         ' device 0-1

obyte     var byte        ' byte to send to device
ibyte     var byte        ' byte received from device
dbyte     var byte

n         var byte        ' index
i         var byte        ' index
addr      var byte        ' text location in EEPROM
b         var bit         ' bit


' -----[ Initialization ]-------------------------------------
'
'   SCLDIR = IN
    SDADIR = IN

    device = 0          ' setting of SA0

' EEPROM Initialization
     text1 DATA "LCD Control ",0
     text2 DATA "with PCF2116",0
     text3 DATA "and StampII.",0

     low TSTPIN

' -----[ Main Code ]-----------------------------------------
'
main:
     gosub blink
     debug ">>>>> Init LCD",CR
     gosub initlcd
     pause 500
     debug ">>>>> Write Text to LCD",CR
     gosub outtext1
     gosub outtext2
     gosub outtext3
     debug ">>>>> Shift Display Right",CR
     for i=1 to 20
```

```
          gosub shiftrght
          pause 200
     next
     pause 1000
     debug ">>>>> Shift Display Left",CR
     for i=1 to 20
          gosub shiftleft
          pause 200
     next
     pause 5000
     goto main
     end

' -----[ Subroutines ]----------------------------------------
initlcd
     gosub start
    obyte = PCF2116 | (device <<1) : gosub send
     obyte = CTRL  : gosub send
     obyte = DISPON: gosub send
     obyte = CTRL  : gosub send
     obyte = CLRSCR: gosub send
     obyte = CTRL  : gosub send
     obyte = MODE  : gosub send
     gosub sstop
     return

shiftleft
     gosub start
    obyte = PCF2116 | (device <<1) : gosub send
     obyte = CTRL  : gosub send
     obyte = LFTSHFT: gosub send
     gosub sstop
     return

shiftrght
     gosub start
    obyte = PCF2116 | (device <<1) : gosub send
     obyte = CTRL  : gosub send
     obyte = RGTSHFT: gosub send
     gosub sstop
     return

outtext1
    gosub start
    obyte = PCF2116 | (device <<1) : gosub send
     obyte = CTRL  : gosub send
     obyte = LINE1 : gosub send
    obyte = WRTDTA : gosub send

     addr = text1
rep1:
     read addr, dbyte
     if dbyte = 0 then end1
```

```
     obyte = dbyte : gosub send
     addr = addr+1
     goto rep1
end1:
     gosub sstop
    return

outtext2
    gosub start
    obyte = PCF2116 | (device <<1) : gosub send
     obyte = CTRL  : gosub send
     obyte = LINE2 : gosub send
    obyte = WRTDTA : gosub send

     addr = text2
rep2:
     read addr, dbyte
     if dbyte = 0 then end2
     obyte = dbyte : gosub send
     addr = addr+1
     goto rep2
end2:
     gosub sstop
    return

outtext3
    gosub start
    obyte = PCF2116 | (device <<1) : gosub send
     obyte = CTRL  : gosub send
     obyte = LINE3 : gosub send
    obyte = WRTDTA : gosub send

     addr = text3
rep3:
     read addr, dbyte
     if dbyte = 0 then end3
     obyte = dbyte : gosub send
     addr = addr+1
     goto rep3
end3:
     gosub sstop
    return

inbyte          ' fetches 8 bits, MSB first
    SDADIR=IN           'input

    ibyte=0
     for n=0 to 7
     pause delay
        high SCLPIN     ' clock high
        pause delay
        ibyte=(ibyte << 1) | SDAIN     'read bit and or
```

```
                                        'with prev
       debug dec SDAIN
       low SCLPIN
     next

     SDADIR=OUT                         'output
     return

outbyte       ' output obyte, MSB first
     low SDAPIN
    for n=0 to 7
     b= (obyte >> 7) & 1
       if (b=1) then outone
       SDADIR=OUT
       debug "0"
_clk:  high SCLPIN
       pause delay
       low SCLPIN
       pause delay
       obyte=obyte << 1
    next
     debug " "
    return

outone
    SDADIR=IN
    debug "1"
    goto _clk

nack
    SDADIR=OUT     ' bring SDA high and clock
    high SCLPIN
    low SCLPIN
    return
send
     gosub outbyte
     gosub nack
     return

start
    low SCLPIN
    SDADIR=IN      ' SDA at logic one
    high SCLPIN
    low SDAPIN
    SDADIR =OUT    ' bring SDA low while clock is high
    low SCLPIN
    debug "START",CR
    return

sstop
     low SCLPIN
    SDADIR=OUT
    high SCLPIN
```

```
    SDADIR=IN      ' bring SDA high while clock is high
    debug CR,"STOP",CR
    return

blink
     toggle TSTPIN
     pause 200
     toggle TSTPIN
     return
```

Listing 6 Test Program (I2C_LCD.BS2)

The commands used for controlling the LCD module are defined as constants.

In the main code area you can find the various program activities as subroutine calls.

At the beginning of the program a test pin is toggled. Connect a LED with a resistor in series to this pin and this LED will signalize the initialization of the LCD module. Most LCD controllers need some time for that initialization so the 500 ms pause is included to provide this delay.

After the LCD is initialized three strings are sent. The subroutines `outtext1` etc. are identically functional. After setting the right address in the display RAM routine the single characters are read from EEPROM and sent to the LCD module. After reading the string terminator (ASCII 0) the read cycle is finished.

After displaying all three text lines in the LCD, the shifting commands (<< or >>) scroll the whole content to the right and to the left afterwards.

Several debug messages explain the state of the running program using the `Debug` Window. The characters sent to the LCD are displayed in binary format in the `Debug` Window. This way you can compare these characters with the instruction table of the LCD controller.

The program I2C_LCD.BS2 serves as a simple demonstration. To use parts of this program in an application, it is helpful to know which resources of the BASIC Stamp are used.

To get an overview of the resources used, Figure 31 shows a memory map. More than a half of the available memory remains free. There is plenty of room to add code for a real application.

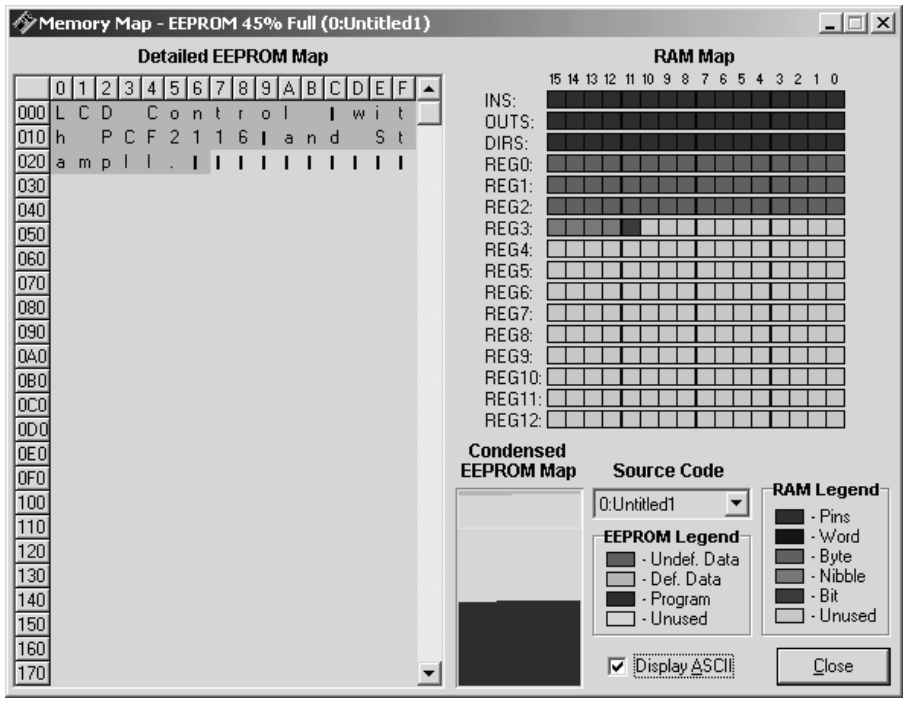


Figure 31 Memory Map

3.2 1-Wire Interface

3.2.1 Some Basics

The 1-Wire Interface connects several so-called 1-Wire Devices to a simple network (MicroLAN).

The interface is built using a two-wire connection (DATA and GND). The pull-up resistor is required as it guarantees the high state. A bus master is responsible for controlling the serial bit stream.

Figure 32 shows the drivers of bus master and slave in a 1-Wire network.

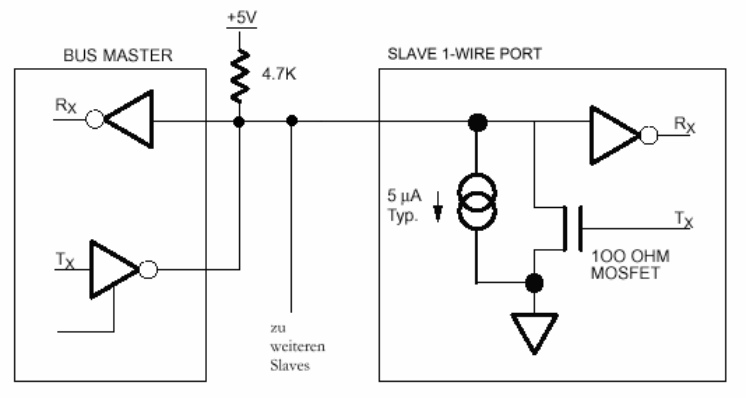


Figure 32 Bus Master and Slave in a 1-Wire network

In an interface using such simple hardware connections the complexity and reliability must rely on software protocols.

Due to the reduced current consumption of the CMOS devices it is possible to power the 1-Wire device in short communication breaks of high signals. The 1-Wire device saves enough energy for running the data exchange until then next charge.

The serial data exchange is half-duplex at discrete time slots.

Acting as the bus master, the BS2p starts the communication by sending a command to the connected 1-Wire device(s). Commands and data will be sent bit by bit starting with the LSB.

A steep rise on the data line by the master synchronizes the master and slave(s). A certain time after this rise (30 µs is standard) the data line is polled depending of the transmission direction from master or slave to read a bit of information (sample time).

This mode is known as data transmission in time slots. Each time slot is synchronized by a sharp high/low edge of the master. Therefore, pauses in the bit stream do not generate errors or other problems.

A data exchange can start after connecting to a 1-Wire device. The 1-Wire devices are not always easy to connect to due to package types. We will come back to this point very soon.

Several microseconds after the connection the 1-Wire device pulls the data line to GND to show the bus master the connection and the waiting for a command. This presence pulse can be requested by the master by sending a reset pulse.

Figure 33 explains the timing briefly. The BS2p hides the details for the programmer, so it is very easy to use the 1-Wire interface in an application.

But you can see too, that only the BS2p's commands `OWOUT` and `OWIN` are able to manage this kind of data exchange. A programmed solution with the other types of the BASIC Stamps is not even possible!

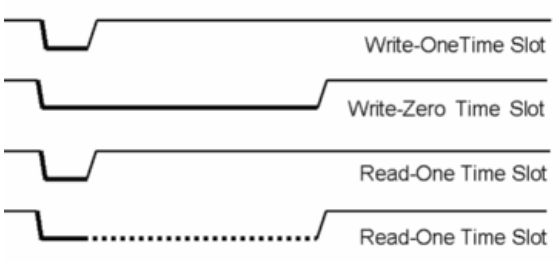


Figure 33 Timing Read/Write Operation

Figure 33 shows the activities of the master with a straight, thick line. Each write or read operation begins with a falling edge of the master followed by a low pulse of about 15 µs.

For writing a "0" the master holds the data line to ground. For writing a "1" the master will be passive and the pull-up resistor pulls the data line high.

Reading data is quite similar. If the slave sends a "0" to the master then the slave holds the data line to GND. This phase is marked by a dotted, thick line. If the master reads a "1" then the slave is passive and the pull-up resistor generates the high level.

3.2.2 1-Wire Devices

After understanding the basics of the data exchange between the bus master and the slave it's helpful to know something about the 1-Wire devices.

Dallas Semiconductor created the iButtons™ and 1-Wire® Chips, all equipped with the 1-Wire interface.

The iButtons are devices in stainless steel package referred to as "MicroCan". These iButtons look more like a big steel horse pill rather than an integrated circuit. Figure 34 shows an iButton.

Figure 34 iButton

The MicroCan protects the iButtons as packaging against external influences and is used as electrical contact at the same time. Numerous connection opportunities exist. One is the iButton Mounting Clip for printed circuit boards (Figure 35). Parallax's BS2p24 Demo Board has such a clip installed.

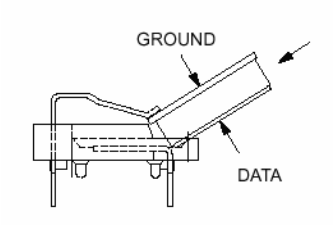


Figure 35 iButton Mounting Clip

Most chips used in iButtons are available as conventional integrated circuits in a plastic package. As we speak about iButtons in the following examples these statements are valid for all 1-Wire devices.

Each iButton and each 1-Wire device have a unique 6-Byte identification number (serial number) saved in a laser-programmed ROM area. The family code indicates the device type. The family code and the serial number describe each 1-Wire device unequivocally.

The ROM area has the same format in all 1-Wire devices. Byte 0 contains the family code describing the device type. Bytes 1 to 6 contain the unique serial number. Byte 7 contains a CRC-8 check sum usable for testing the correctness of the data transmission.

B7	B6	B5	B4	B3	B2	B1	B0
CRC-8	Serial Number						Family Code

The next table shows an overview to the types of iButtons. The number of 1-Wire devices in plastic packages is more extensive yet.

IButton	*Family Code*	*Memory*	*Special Features*
DS1990A	01H	-	
DS1991	02H	512 Bit NVRAM	3 x 384 Bit protected NVRAM
DS1992	08H	1 KBit NVRAM	
DS1993	06H	4 KBit NVRAM	
DS1994	04H	4 KBit NVRAM	RTC, Interval ltimer, Cycle counter
DS1995	0AH	16 KBit NVRAM	
DS1996	0CH	64 KBit NVRAM	
DS1982	09H	1 KBit EPROM	
DS1985	0FH	16 KBit EPROM	
DS1986	0BH	64 KBit EPROM	
DS1920	10H	16 Byte EEPROM	Temperature sensor

For a better understanding of the following application examples we'll explain the most important features of the iButtons we've decided to use. Pay close attention yet understand that this explanation can not replace the data sheet.

You can find all required data at www.ibutton.com respectively dbserv.maxim-ic.com/1-Wire.cfm.

3.2.2.1 DS1990A

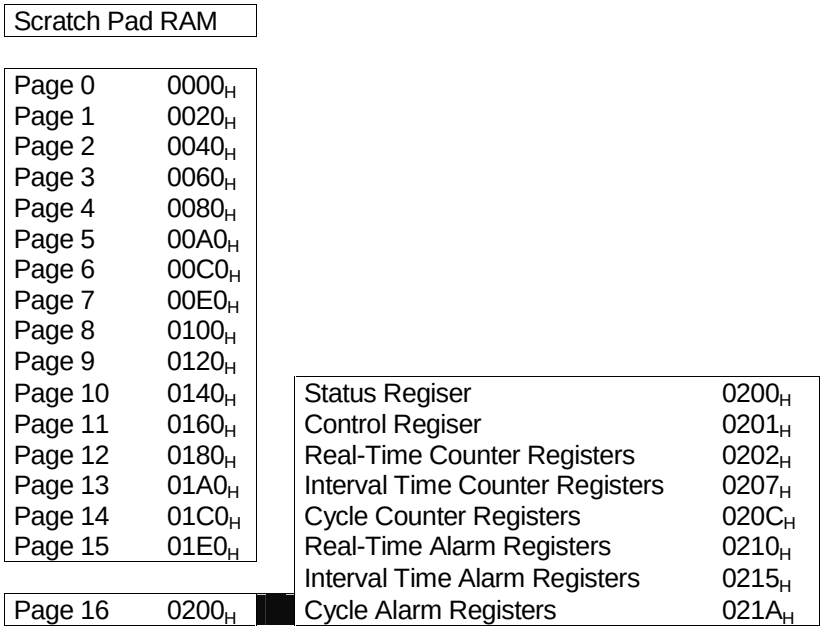
The DS1990A is a serial number iButton containing the 64-Bit ROM and was designed exclusively for identification purposes.

3.2.2.2 DS1994

The DS1994 is a more complete device containing 4-Kbit non-volatile RAM (NVRAM) and three counter/timers (RTC, interval timer, cycle counter) besides the 64-Bit ROM.

The 4-Kbit NVRAM are organized in 16 pages of 256 bits each. The data exchange happens via a Scratch Pad RAM of 256 bits. The counter/timer registers are mapped in the 16th page.

The next diagram shows the memory organization in the DS1994 device.



The bits of the status and control register are named as follows:

	B7	B6	B5	B4	B3	B2	B1	B0
Status Register	-	-	/CCE	/ITE	/RTE	CCF	ITF	RTF
Control Register	DSEL	STOP /START	AUTO /MAN	OSC	RO	WPC	WPI	WR

In the status register are the interrupt enable bits and the alarm flags for the three counter/timers while the bits in the control register control the operation of the device.

The RTC is a 5-Byte binary counter incremented 256 times per second. The low byte counts the parts of a second. The other four bytes count the seconds. This way the counter contains

the number of seconds passed from a reference point defined by the user. An overflow happens after 136 years.

The interval timer operates in a similar fashion but has two different modes controlled by Bit 5 of the control register.

In the AUTO Mode the interval timer starts after the data line was pulled high for a time defined by Bit 7 (DSEL) of the control register. The interval timer stops after pulling the data line low for the same time.

If DSEL = 1 the time is 123 +/- 2 ms. If DSEL = 0 the time is reduced to 3.5 +/- 0.5 ms.

In MAN mode the time measurement is started and stopped by Bit 6 (STOP /START) of the control register.

The cycle counter contains only four bytes. It increments with a falling edge on the data line if the timing defined by DSEL is kept.

The other bits of the control register determine the write protection for the three counter/timer (WPR, WPI, WPC) and move the DS1994 into the Read-Only Mode (RO) and activate/deactivate the oscillator (OSC).

3.2.2.3 DS1920

The DS1920 is an iButton for temperature measurement in a range of –55 °C to 100 °C with a resolution of 0.5 °C.

The measured temperature value has an internal resolution of 9 Bits according to the next table.

Temperature	*Binary Value*	*Hex Value*
+ 100°C	0000000011001000_B	$00C8_H$
+ 25°C	0000000000110010_B	0032_H
+ 1/2°C	0000000000000001_B	0001_H
+ 0°C	0000000000000000_B	0000_H
– 1/2°C	1111111111111111_B	$FFFF_H$
– 25°C	1111111111001110_B	$FFCE_H$
– 55°C	1111111110010010_B	$FF92_H$

The Scratch Pad RAM supports the data exchange again and the temperature levels are saved non-volatile in an EEPROM. Figure 36 shows a block diagram of the DS1920 device.

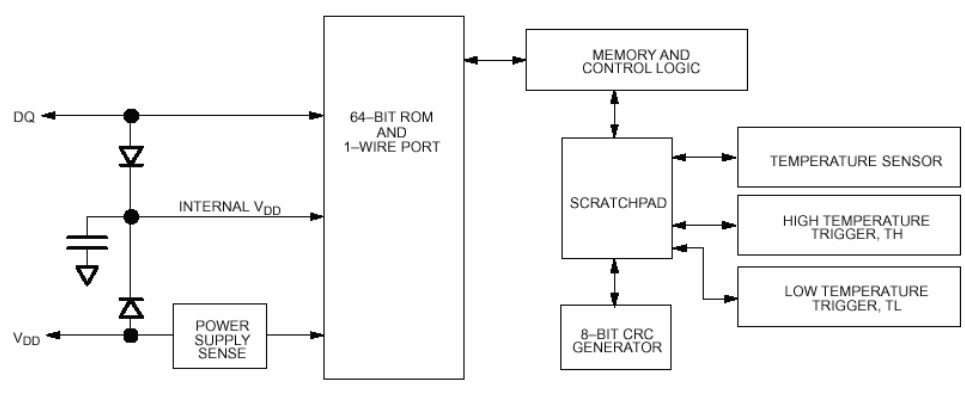


Figure 36 Block diagram of DS1920

The next table explains the contents of RAM and EEPROM:

RAM	*Byte*	*EEPROM*
Temperature LSB	0	
Temperature MSB	1	
TH/User Byte 1	2	TH/User Byte 1
TH/User Byte 2	3	TH/User Byte 2
Reserved	4	
Reserved	5	
Count Remain	6	
Count per °C	7	
CRC	8	

If the measured temperature value exceeds the level TH or falls short of the level TL an alarm flag is set.

To exploit the accuracy of the DS1920 you can process the following steps:

Reading the temperature value and clearing the LSB (TEMP_READ)

Reading the internal counter (COUNT_REMAIN)

Reading the Counts/°C (COUNTS_PER_C)

Calculation of the temperature value according to the formula:

$$\mathbf{TEMPERATURE} = \mathbf{TEMP\_READ} - \mathbf{0.25} + \frac{\mathbf{COUNT\_PER\_C} - \mathbf{COUNT\_REMAIN}}{\mathbf{COUNT\_PER\_C}}$$

3.2.3 Access to iButtons

The access to all iButtons is organized similarly to the ISO/OSI-Model. But, not all layers of this model are implemented. The following table shows the existing layers for the iButtons:

ISO/OSI-Model	*iButton*
Application Layer	No
Presentation Layer	Yes
Session Layer	No
Transportation Layer	Yes
Network Layer	Yes
Link Layer	Yes
Physical Layer	Yes

The Physical Layer defines the electrical conditions, logic levels, and the timing for all 1-Wire devices.

The basic functions of 1-Wire communication including Reset, Presence Detection and Bit transfer are defined in the Link Layer.

In the Network Layer the serial identification of the 1-Wire devices is carried out. The commands of this layer point to ROM exclusively and are therefore named as ROM Commands.

ROM COMMANDS	
Read ROM	Reads the complete ROM content (only possible with a connected iButton)
Match ROM	Addresses an iButton according to the 64-Bit ROM content
Skip ROM	Skip addressing (only possible with a connected iButton)
Search ROM	Search for an iButton in a network
Alarm Search	Search for iButtons (DS1920) in a network, which notify an alarm

The Transport Layer is responsible for the data exchange out of the ROM area. The next table shows a selection of the available Memory Commands:

MEMORY COMMANDS	
Convert Temperature	Starts the temperature measurement
Read Scratch Pad	Reads all bytes from Scratch Pad Memory
Write Scratch Pad	Saves the temperature levels into the Scratch Pad Memory
Copy Scratch Pad	Copies the temperature levels into the EEPROM
Recall EE	Copies the temperature levels back to Scratch Pad Memory
Read Power Supply	Queries the power supply

The next application examples will demonstrate how to organize access to the iButtons.

3.2.4 Identification of iButtons

Each iButton can be identified according to it's ROM data. Before reading the ROM content the iButtons must be connected to the BS2p.

The first program example periodically queries the 1-Wire interface for an iButton connected to I/O pin 15. Listing 7 shows the source of the program 1WIRE_ID.BSP.

If no iButton is found the message “No 1-Wire device present” is displayed in the Debug window and after a pause the whole process repeats. Figure 37 shows this message in the Debug Window.

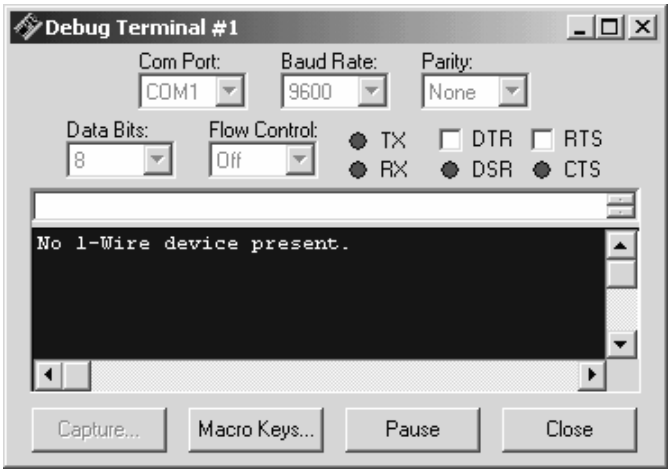


Figure 37 No iButton connected

If an iButton was found then the program reads it’s ROM content and displays it in the Debug Window as a hex dump. Afterwards a CRC-8 Check calculates the CRC over all read bytes except for the CRC byte. The data exchange was faultless if the calculated and read CRC are equal. Figure 38 shows the hex dump of the ROM content and the calculated CRC-8. For completeness only, the message “CRC OK” marks the verified CRC.

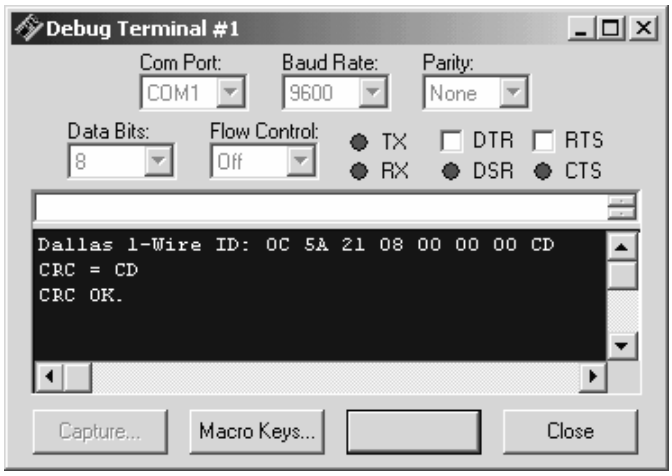


Figure 38 Hex dump of ROM content

```
' -----[ Title ]-----------------------------------------------
'
' File..... 1wire_id.bsp
' Purpose... Reading Family Code and Serial Number for
'            Identification
' Author.... Claus Kuehnel
' Started...
' Updated...

' -----[ Program Description ]-------------------------------
'

' -----[ Revision Hisory ]----------------------------------
'

' -----[ Directives ]----------------------------------------
'
'{$STAMP BS2p}              'specifies a BS2p

' -----[ Constants ]-----------------------------------------
'
OWpin         con 15        '1-wire device pin

OWFERst       con %0001     'Front-End Reset
OWBERst       con %0010     'Back-End reset
OWBitMode     con %0100

ReadROM       con $33       'Read ROM Command
SearchROM     con $F0       'Search ROM Command
```

```
NoDevice      con %11                   'No device present

' -----[ Variables ]----------------------------------------
'
idx           var byte
ROMData       var byte(8)
value         var byte
CRC           var byte
index         var byte
i             var byte
devcheck  var nib

' -----[ Initialization ]-------------------------------------
'
init:    pause 1000                     'open debug window

' -----[ Main Code ]-----------------------------------------
'
main:
     debug cls
     gosub devicecheck
     if (devcheck <> NoDevice) then displayROM

NoDeviceFoand:
     debug "No 1-Wire device present."
     pause 1000
     goto main

displayROM:
     debug "Dallas 1-Wire ID: "
     owout OWpin, OWFERst, [ReadROM]    'Read ROM Data
     owin  OWpin, OWBERst, [str ROMData\8]
     CRC = 0
     for idx = 0 to 7                   'Display ROM Data
          debug hex2 ROMData(idx), " "
     next
     for idx = 0 to 6                   'Calculate CRC-8
          value = ROMData(idx)
          gosub CRC8
     next
     debug cr, "CRC = ", hex2 CRC
     if CRC = ROMData(7) then nxt       'Check CRC-8
     debug cr, "CRC not OK." : goto nxt1
nxt: debug cr, "CRC OK."
nxt1:    pause 1000
     goto main

' -----[ Subroutines ]----------------------------------------
'
devicecheck:                    'Check if any 1-Wire Device is connected
     devcheck = 0
     owout OWpin, OWFERst, [SearchROM]
```

```
     owin  OWpin, OWBitMode, [devcheck.bit1, devcheck.bit0]
     return

CRC8:                           'Calculate next CRC-8 from table
     restore
     index = CRC ^ value
     for i = 0 to index
          read i, CRC
     next
     return

' -----[ Data ]-----------------------------------------------
'
data 0, 94, 188, 226, 97, 63, 221, 131, 194, 156, 126, 32, 163, 253, 31, 65
data 157, 195, 33, 127, 252, 162, 64, 30, 95, 1, 227, 189, 62, 96, 130, 220
data 35, 125, 159, 193, 66, 28, 254, 160, 225, 191, 93, 3, 128, 222, 60, 98
data 190, 224, 2, 92, 223, 129, 99, 61, 124, 34, 192, 158, 29, 67, 161, 255
data 70, 24, 250, 164, 39, 121, 155, 197, 132, 218, 56, 102, 229, 187, 89, 7
data 219, 133, 103, 57, 186, 228, 6, 88, 25, 71, 165, 251, 120, 38, 196, 154
data 101, 59, 217, 135, 4, 90, 184, 230, 167, 249, 27, 69, 198, 152, 122, 36
data 248, 166, 68, 26, 153, 199, 37, 123, 58, 100, 134, 216, 91, 5, 231, 185
data 140, 210, 48, 110, 237, 179, 81, 15, 78, 16, 242, 172, 47, 113, 147, 205
data 17, 79, 173, 243, 112, 46, 204, 146, 211, 141, 111, 49, 178, 236, 14, 80
data 175, 241, 19, 77, 206, 144, 114, 44, 109, 51, 209, 143, 12, 82, 176, 238
data 50, 108, 142, 208, 83, 13, 239, 177, 240, 174, 76, 18, 145, 207, 45, 115
data 202, 148, 118, 40, 171, 245, 23, 73, 8, 86, 180, 234, 105, 55, 213, 139
data 87, 9, 235, 181, 54, 104, 138, 212, 149, 203, 41, 119, 244, 170, 72, 22
data 233, 183, 85, 11, 136, 214, 52, 106, 43, 117, 151, 201, 74, 20, 246, 168
data 116, 42, 200, 150, 21, 75, 169, 247, 182, 232, 10, 84, 215, 137, 107, 53
```

Listing 7 1-Wire Identification (1WIRE_ID.BSP)

The program begins with some definitions of constants to make the source more readable. Among the different modes of operation you can find the used ROM commands there.

Other than some variables we have defined an array of eight bytes to save the read ROM content temporarily.

The device check is the first action with the iButton. This is the check for a connected iButton.

After a Reset we send the Search ROM command to the iButton and switch to Bit Mode after that. A connected iButton sends the True and False value of the LSB in ROM in succession. This means, if there is no iButton connected both bits will be high.

If no iButton is detected, the Debug Window displays the appropriate message.

If an iButton is detected, we send the Read ROM command and read the eight byte ROM content and save it in the array ROMData.

As mentioned already, next follows the hex dump, calculation of the CRC-8 value and display of the CRCs in the Debug Window.

The CRC-8 procedure is quite simple. In this program example we use a table-based procedure. The variable `value` hands over the byte to be processed by the subroutine `CRC8`. After passing all bytes to be considered to the subroutine `CRC8` the variable `CRC` holds the calculated CRC-8 value. A compare of this calculated value with the delivered CRC-8 value (Byte 7 of the ROM content) decides the validity of the read ROM content.

3.2.5 Access Control with iButtons

In the next program example we present a system for access control.

The iButtons we used can be assembled with mounting material for easy handling. Figure 39 shows an iButton in a key holder, while Figure 40 shows an iButton fixed in a printable identification card (Batch).

Figure 39 Key Holder

Figure 40 ID Card

To connect an iButtons we can use the hand-grip shown in Figure 41. You can connect this hand-grip directly to the Parallax BS2p Demo Board.

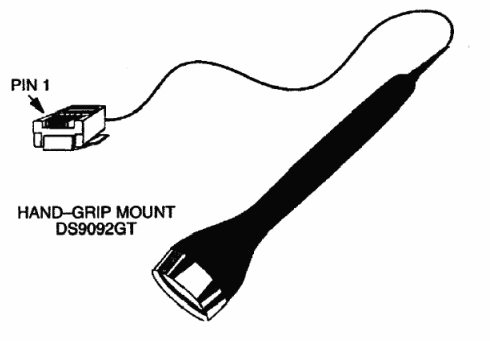


Figure 41 Hand-Grip

Before we explain the program itself have a look to the messages in the Debug Window. An LCD connected to the BS2p could handle the same display for product portability.

Figure 42 shows the request to connect the batch. As long as no batch is connected the request to connect is sent repeatedly.

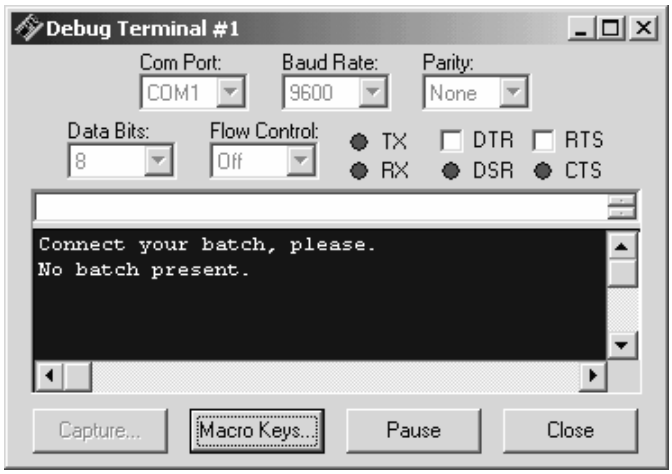


Figure 42 No Batch detected

Figure 43 shows that a batch is now connected. After reading the ROM content we output a hex dump to verify. Afterwards we search the database for the read identification. The displayed stars (*) visualize the search of the database. In this case the data base had no entry for the read iButton and the subject batch was not accepted.

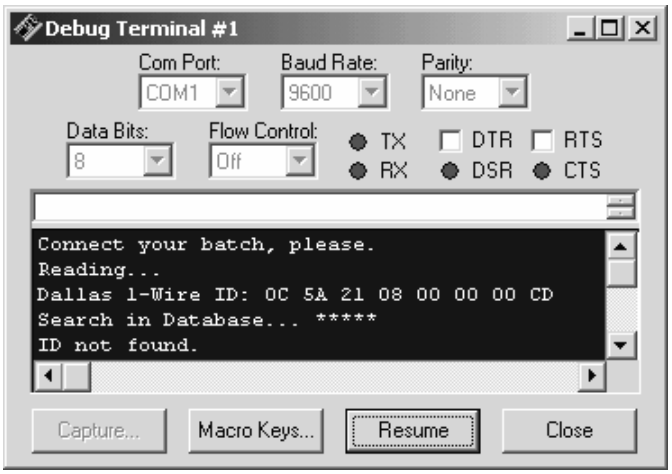


Figure 43 Batch read and not accepted

Figure 44 shows the reading of a further batch. After the hex dump the database search starts again. You can see a finished search after the output of the first star because an iButton was found.

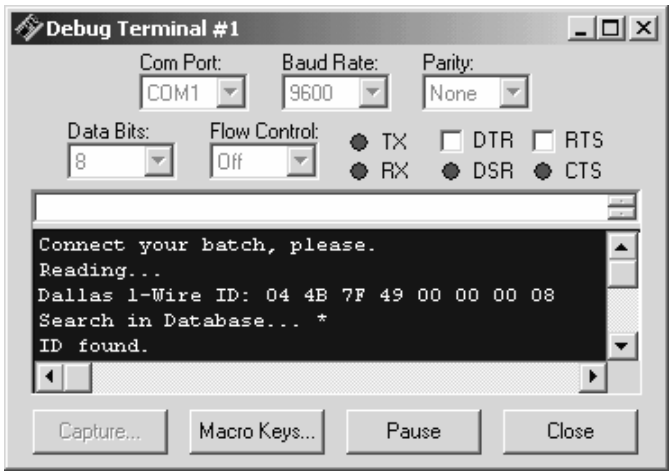


Figure 44 Batch read and accepted

A comparison of the outputs in the Debug Window with the source code in Listing 8 (ID_CHECK.BSP) makes the procedure clear.

We connect the iButton to I/O pin 15 as in the last program example and a LED with additional resistor from I/O pin 12 to GND.

```
' -----[ Title ]-----------------------------------------------
'
' File..... ID_check.bsp
' Purpose... Check a 1-wire identity against stored IDs
' Author.... Claus Kuehnel
' Started... 2001-07-24
' Updated...

' -----[ Program Description ]--------------------------------
'
' -----[ Revision Hisory ]-----------------------------------
'
' -----[ Directives ]-----------------------------------------
'
'{$STAMP BS2p}                  'specifies a BS2p

' -----[ constants ]------------------------------------------
```

```
'
OWpin          con 15         '1-wire device pin
LED            con 12         'LED

OWFERst        con %0001      'Front-End Reset
OWBERst        con %0010      'Back-End reset
OWBitMode      con %0100

ReadROM        con $33        'Read ROM Command
SearchROM      con $F0        'Search ROM Command

NoDevice       con %11        'No device present

IDs            con 5          'Number of stored IDs

' -----[ Variables ]----------------------------------------
'
idx            var byte       'Index variable
index          var byte
ROMData        var byte(8)    'ROM Data from iButton
ID             var byte       'Stored ID (0 to IDs-1)
value          var byte
devcheck       var nib        'Result of Device Check
IDtrue         var bit        'Result of ID Compare

' -----[ Initialization ]-----------------------------------
'
init:
     pause 1000               'Open debug window
     high LED                 'LED On

' -----[ Main Code ]----------------------------------------
'
main:
     IDtrue = 0
     debug cls, "Connect your batch, please.", cr
     gosub devicecheck
     if (devcheck <> NoDevice) then readiButton

NoDeviceFoand:
     debug "No batch present."
     pause 1000
     goto main

readiButton:
     gosub toggleLED
     debug "Reading...", cr
     debug "Dallas 1-Wire ID: "
     owout OWpin, OWFERst, [ReadROM]
     owin  OWpin, OWBERst, [str ROMData\8]
     gosub displayROMData
     debug cr, "Search in Database... "
     gosub searchID
```

```
     if IDtrue then checkOK
     pause 1000
     goto main

checkOK:
     debug cr, "ID foand."
     for idx = 0 to 10
          gosub toggleLED
     next
     pause 1000
     goto main

     end

' -----[ Subroutines ]----------------------------------------
'
devicecheck:
     devcheck = 0
     owout OWpin, OWFERst, [SearchROM]
     owin  OWpin, OWBitMode, [devcheck.bit1, devcheck.bit0]
     return

displayROMData:
     for idx = 0 to 7
          debug hex2 ROMData(idx), " "
     next
     return

searchID:
     restore
     ID = 0
nxt:for idx = 0 to 7
          read ID+idx, value
          if value <> ROMData(idx) then nextID
     next
     IDtrue = 1:return
nextID:
     ID = ID + 8   'Next stored ID
     if ID <= IDs * 8 then repeatcheck
     IDtrue = 0
     debug cr, "ID not foand."
     return
repeatcheck:
     debug "*"
     goto nxt

toggleLED:
     toggle LED
     pause 100
     toggle LED
     pause 20
     return
```

```
' -----[ Data ]-----------------------------------------------
'
data $10, $37, $84, $11, $00, $00, $00, $F0
data $04, $4B, $7F, $49, $00, $00, $00, $08
data $10, $37, $84, $11, $00, $00, $00, $F2
data $10, $37, $84, $11, $00, $00, $00, $F0
data $10, $37, $84, $11, $00, $00, $00, $F0
```

Listing 8 Access Control (ID_CHEK.BSP)

The program itself is quite similar to the last program example. The data exchange with the iButton is practically identical. We do not calculate the CRC-8 here.

The subroutine `searchID` is the new part of this program. We save the ROM content read from the iButton into the array `ROMData`. This ROM content is compared byte by byte with the identification numbers saved in EEPROM.

If there is a mismatch in one byte the next identification number is tested. The test repeats until a match is found or all identification numbers are checked against a match.

After a first match of the read ROM content (all eight bytes) with an entry in the database we exit the subroutine.

Figure 44 showed a successful identification of the connected iButton. Compare the displayed Hexdump of the ROM content with the entries of the data base and you see the match for the seconds entry of the data base and the second identification number.

You can enhance the number of identification number with respect to the available memory. The constant `IDs` must be reflect the number of data base entries.

3.2.6 Measuring of Temperature with DS1920

Using the DS1920 iButton temperature measurement is quite simple.

The following program example implements a periodic temperature measurement with a DS1920 iButton connected to the mounting clip installed on the BS2p Demo Boards.

The program itself consists of one single loop which the temperature measurement initializes and reads the result, stores the temperature measurement in Scratch Pad RAM and displays the value. An LED signalizes the moment of the temperature measurement. Listing 9 shows the source of the program DS1920.BSP.

```
' -----[ Title ]-----------------------------------------------
'
```

```
' File...... ds1920.bsp
' Purpose... Measuring of Temperature with DS1920 iButton
' Author.... Claus Kuehnel
' Started... 2001-08-11
' Updated...

' -----[ Program Description ]-------------------------------
'
' This program demonstrates the periodic measuring of ambient
' temperature with DS1920 iButton and calculation of high-
' resolution temperature value in degree C.

' -----[ Revision Hisory ]-----------------------------------
'
' 2001-08-11

' -----[ Directives ]----------------------------------------
'
'{$STAMP BS2p}              'specifies a BS2p

' -----[ Constants ]-----------------------------------------
'
OWpin         con 15        '1-wire device pin
LED           con 12        'Pin for LED

OWFERst       con %0001     'Front-End Reset
OWBERst       con %0010     'Back-End reset
OWBitMode     con %0100

SkipROM       con $CC       'Skip ROM Command
ReadScratch   con $BE       'Read Scratch Pad
ConverT       con $44       'Convert Temperature

' -----[ Variables ]-----------------------------------------
'
temp          var word      'Temperature Value
CRem          var byte      'Counts remaining value
CPerC         var byte      'Counts per degree C value

' -----[ Initialization ]------------------------------------
'
init:
     pause 1000             'open debug window
     low LED                'LED off

' -----[ Main Code ]-----------------------------------------
'
Start:
     'Send Convert Temperature command
     OWOUT OWpin, OWFERst, [SkipROM, ConverT]
     high LED
     debug cls
```

```
CheckForDone:
     'Wait until conversion is done
     pause 25
     OWIN OWpin, OWBitMode, [Temp]
     'debug bin temp            'uncomment to see conversion time
     IF Temp = 0 THEN CheckForDone

     'Send Read Scratch Pad command
     OWOUT OWpin, OWFERst, [SkipROM, ReadScratch]
     OWIN OWpin, OWBERst, [Temp.lowbyte,Temp.highbyte,CRem,CRem,CRem,CRem,CRem,CPerC]
     debug cr,"Temperature count (0.5°C) is ", dec Temp

     'Calculate temperature in degrees C
     Temp = Temp>>1*100-25+((CPerC*100-(CRem*100))/CPerC)
     debug cr, "Actual Temperature is ", DEC Temp/100, ".", DEC2 Temp-(Temp/100*100), "
C", CR
     low LED
     pause 5000    'next measurement in 5 sec
     goto Start
```

Listing 9 Periodic Temperature Measurement (DS1920.BSP)

The commands and I/O definitions are handled in the constants section. Because we are working with only one iButton, we can skip the SkipROM and the ROM Handling commands.

The Convert Command starts the temperature measurement. Afterwards the iButton is queried until the read bit goes from low to high. This edge signalizes the end of the temperature measurement. The result of this temperature measurement is saved in the Scratch Pad RAM and can be read now.

We read all eight memory cells of the Scratch Pad RAM. But, only the first and the last cells are of interest in this example.

To exploit the full accuracy of the temperature measurement we calculate the temperature value by the formula given in Chapter 3.2.2.3.

Figure 45 shows the read temperature count with a resolution of 0.5 °C and the calculated temperature value.

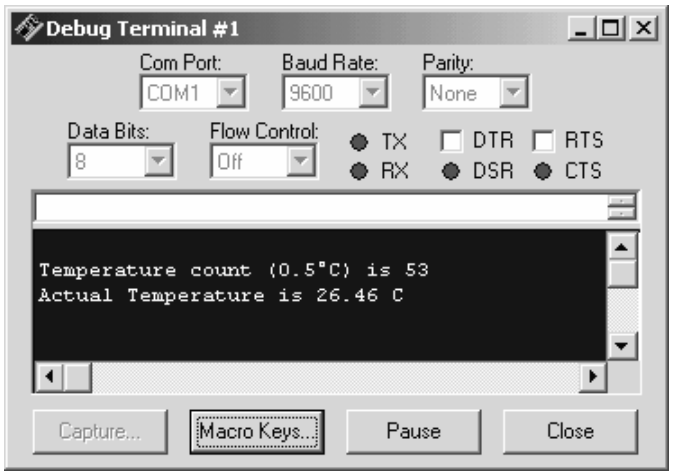


Figure 45 Result of Temperature Measurement

3.2.7 External Memory with DS1994

We explained the memory organization in Chapter 3.2.2.2. The following program example shows reading and writing the memory via Scratch Pad.

The construction of the program and the parts connected to the BS2p are identical to the last program examples. Listing 10 shows the source of the program DS1994_MEM.B2P.

```
' -----[ Title ]-----------------------------------------------
'
' File..... ds1994_mem.bsp
' Purpose... Storing data in Memory of DS1994 iButton
' Author.... Claus Kuehnel
' Started... 2001-08-11
' Updated...

' -----[ Program Description ]-------------------------------
'
' This program demonstrates the writing and reading DS1994
' memory via the internal Scratch Pad RAM

' -----[ Revision Hisory ]-----------------------------------
'
' 2001-08-11
```

```
' -----[ Directives ]----------------------------------------
'
'{$STAMP BS2p}          'specifies a BS2p

' -----[ Constants ]-----------------------------------------
'
OWpin          con 15        '1-wire device pin
LED            con 12        'Pin for LED

OWFERst        con %0001     'Front-End Reset
OWBERst        con %0010     'Back-End reset
OWBitMode      con %0100

SkipROM        con $CC       'Skip ROM Command
WriteScratch   con $0F       'Write Scratch Pad
ReadScratch    con $AA       'Read Scratch Pad
CopyScratch    con $55       'Copy Scratch Pad to Memory
ReadMemory     con $F0       'Read Memory

cTargetAddr    con $0077     'Target Address in Memory

' -----[ Variables ]-----------------------------------------
'
TargetAddr     var word
TA1            var TargetAddr.LowByte
TA2            var TargetAddr.HighByte
EndingAddr     var byte

temp           var byte(8)
idx            var nib

' -----[ Initialization ]-------------------------------------
'
init:
     pause 1000    'open debug window
     low LED       'LED off

' -----[ Main Code ]------------------------------------------
'
Start:
     'Initialize array
     for idx = 0 to 7
          temp(idx) = idx +$30
     next
     debug cls,"Array contains ", str temp\8, cr

     TargetAddr = cTargetAddr
     debug "TA:   ", hex2 TA2, hex2 TA1, cr

     'Write array to Scratch Pad RAM
     high LED
     debug "Write to Scratch Pad RAM...",cr
```

```
    OWOUT OWpin, OWFERst, [SkipROM, WriteScratch, TA1, TA2, str temp\8 ]
    low LED

    'Clear array
    debug "Clear ScrachPad RAM...",cr
    for idx = 0 to 7
         temp(idx) = $96
    next
    debug "Array contains ", str temp\8, cr

    'Read Scratch Pad RAM and save back into array
    debug "Read Scratch Pad RAM...",cr
    OWOUT OWpin, OWFERst, [SkipROM, ReadScratch]
    OWIN OWpin, OWBERst, [TA1, TA2, EndingAddr, str temp\8 ]

    debug "TA:   ", hex2 TA2, hex2 TA1, cr
    debug "EA:   ", hex2 EndingAddr, cr
    debug "Data: ", str temp\8, cr

    'Clear array again
    debug "Clear ScrachPad RAM...",cr
    for idx = 0 to 7
         temp(idx) = $96  ' any bit pattern
    next
    debug "Array contains ", str temp\8, cr

    'Copy Scratch Pad to Memory
    debug "Copy Scratch Pad to Memory...", cr
    OWOUT OWpin, OWFERst, [SkipROM, CopyScratch, TA1, TA2, EndingAddr]

    'Read Memory and save back into array
    debug "Read Memory...", cr
    OWOUT OWpin, OWFERst, [SkipROM, ReadMemory, TA1, TA2]
    OWIN OWpin, OWBERst, [str temp\8 ]
    debug "Array contains ", str temp\8, cr

    pause 5000
    goto start
```

Listing 10
Writing and Reading the DS1994 Memory (DS1994_MEM.B2P)

The program consists of one loop operating the following factions in succession:

Writing the characters “0” to “7” into an 8-Byte array and display

Defining a target address in the DS1994 Memory

Copying the arrays into the Scratch Pad RAM

Clearing of array and display

Reading the Scratch Pad RAM for verification, storing into the array and display

Clearing of array and display

Copying the Scratch Pad RAMs into memory (NVRAM)

Reading the memory, storing into the array and display

This procedure repeats after a short delay. Two images explain problems possible while addressing.

Figure 46 shows the numerous outputs of this program example to the Debug Window during the storage of the array beginning at target address $0077.

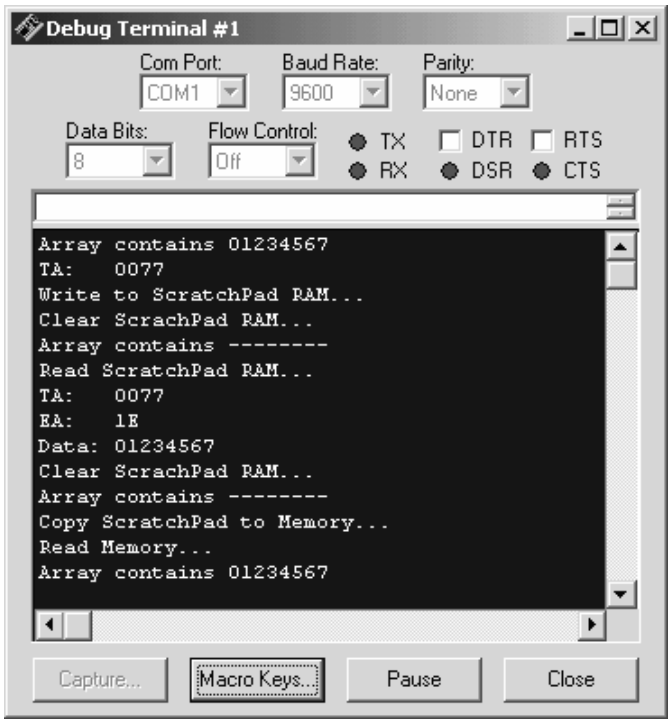


Figure 46 Target Address $0077

While reading data back from Scratch Pad RAM we get the target address and the ending address. The ending address contains the so-called ending offset (E4:E0) and the three flags AA, OF and PF.

B7	B6	B5	B4	B3	B2	B1	B0
AA	OF	PF	E4	E3	E2	E1	E0

The ending offset shows the last written memory cell. In a page of 32 bytes the ending offset is between $00 and $1F. Going beyond this limit will set the Overflow Flag (OF) and additional data will be ignored.

Figure 47 shows this situation. The target address is incremented to $0079. To the end of page 3 (target address $007F) there are only seven memory cells available.

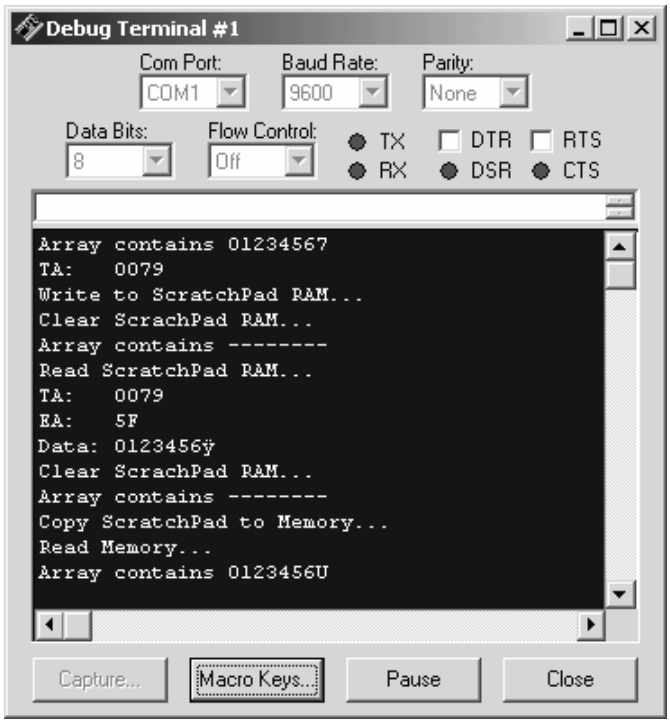


Figure 47 Target Address $0079

After reading the Scratch Pad RAM the ending address has a value of $5F. The set Overflow Flag signalizes that not all data fit into the addressed page.

Reading the data back we immediately see the wrong last byte. This situation does not change with the copying of the Scratch Pad RAM into the NVRAM. All data read back are corrupt in this case.

3.2.8 Timer with DS1994

The DS1994 contains three timers in addition to the NVRAM. In the following program example we experiment with the RTC and the interval timer.

The timer registers are located at the end of the NVRAMs beginning with the target address $0200

Status Regiser	$0200
Control Regiser	$0201
Real-Time Counter Registers	$0202
Interval Time Counter Registers	$0207
Cycle Counter Registers	$020C
Real-Time Alarm Registers	$0210
Interval Time Alarm Registers	$0215
Cycle Alarm Registers	$021A

The control register defines the mode of the timers. Chapter 3.2.2.2 explains the meaning of these bits.

In our simple example the interval timer is started at the beginning of the program. The RTC runs continuously. After a certain time the interval is stopped and the RTC runs on and on. Figure 48 shows the timer actions protocol in the Debug Window.

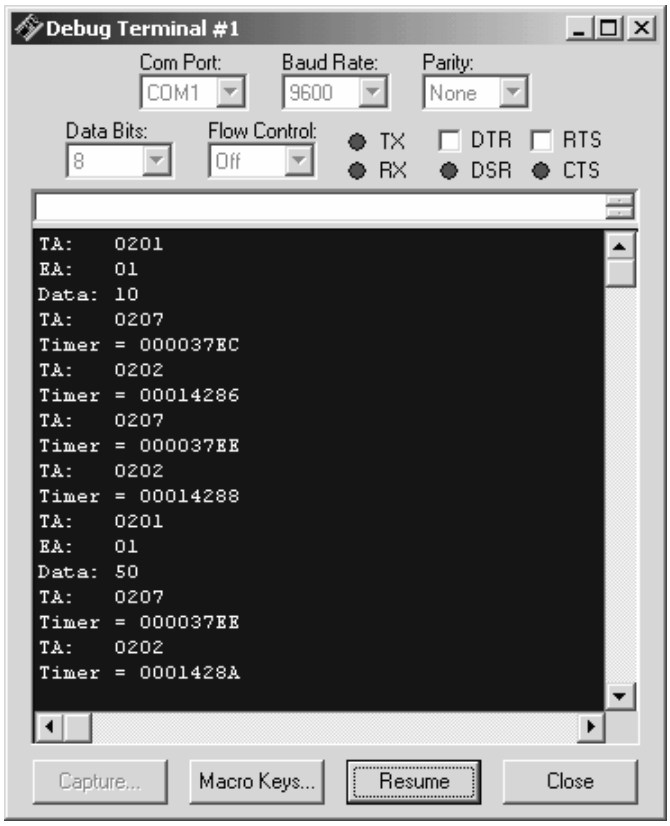


Figure 48 Controlling and Read-Out the DS1994 Timer

Writing the byte $10 into the control register starts the interval timer. The two read operations to the RTC ($0202) and to the interval timer ($0207) get different values as expected.

The writing of the $50 bytes into the control register stops the interval timer. The following operation shows a different value read from the RTC ($0202) but an unchanged value from the interval timer ($0207).

Listing 11 shows the source of the program DS1994_TIMER.BSP. Because we located the 1-Wire commands in subroutines the source is quite readable. The definition part is comparable to the last 1-Wire program examples.

```
' -----[ Title ]-----------------------------------------------
'
' File...... ds1994_timer.bsp
' Purpose... Reading and Writing the Timer of DS1994 iButton
' Author.... Claus Kuehnel
' Started... 2001-08-13
' Updated...

' -----[ Program Description ]-------------------------------
'
' This program demonstrates reading and writing DS1994 timer
' registers via the internal Scratch Pad RAM

' -----[ Revision Hisory ]-----------------------------------
'
' 2001-08-13

' -----[ Directives ]----------------------------------------
'
'{$STAMP BS2p}                  'specifies a BS2p

' -----[ Constants ]-----------------------------------------
'
OWpin         con 15            '1-wire device pin
LED           con 12            'Pin for LED

OWFERst       con %0001         'Front-End Reset
OWBERst       con %0010         'Back-End reset
OWBitMode     con %0100

SkipROM       con $CC           'Skip ROM Command
WriteScratch  con $0F           'Write Scratch Pad
ReadScratch   con $AA           'Read Scratch Pad
CopyScratch   con $55           'Copy Scratch Pad to Memory
ReadMemory    con $F0           'Read Memory

ControlReg    con $0201         'Timer Control Regiser
RTCReg        con $0202         'RTC Regiser   $0202-$0206
IntervReg     con $0207         'IntervallTimer $0207-$020B

StartTimer    con %00010000     'Start IntervalTimer
StopTimer     con %01010000     'Stop IntervalTimer


' -----[ Variables ]-----------------------------------------
'
TargetAddr    var word          'TargetAddress in NVRAM
TA1           var TargetAddr.LowByte
TA2           var TargetAddr.HighByte
EndingAddr    var byte          'EndingAddress
TargetCont    var byte          'Content to save
```

```
temp          var byte(8)
idx           var nib

' -----[ Initialization ]----------------------------------
'
init:
     pause 1000                'open debug window
     low LED                   'LED off


' -----[ Main Code ]-----------------------------------------
'
Start:
     high LED
     TargetAddr = ControlReg 'Start of IntervalTimer
     TargetCont = StartTimer
     gosub WriteDS1994Reg
     low LED

     debug cls
     debug "TA:   ", hex2 TA2, hex2 TA1, cr
     debug "EA:   ", hex2 EndingAddr, cr
     debug "Data: ", hex2 temp, cr

     TargetAddr = IntervReg     'Read IntervalTimer
     gosub ReadDS1994Timer
     gosub DisplayTimer

     TargetAddr = RTCReg               'Read RTC
     gosub ReadDS1994Timer
     gosub DisplayTimer

     pause 2000                'Wait 2 sec

     TargetAddr = IntervReg     'Read IntervalTimer
     gosub ReadDS1994Timer
     gosub DisplayTimer

     TargetAddr = RTCReg               'Read RTC
     gosub ReadDS1994Timer
     gosub DisplayTimer

     TargetAddr = ControlReg 'Stop IntervalTimer
     TargetCont = StopTimer
     gosub WriteDS1994Reg

     debug "TA:   ", hex2 TA2, hex2 TA1, cr
     debug "EA:   ", hex2 EndingAddr, cr
     debug "Data: ", hex2 temp, cr
```

```
     pause 2000                 'Wait 2 sec

     TargetAddr = IntervReg     'Read IntervalTimer
     gosub ReadDS1994Timer
     gosub DisplayTimer

     TargetAddr =RTCReg         'Read RTC
     gosub ReadDS1994Timer
     gosub DisplayTimer

     pause 5000
     goto start                 'Play it again


' -----[ Subroutines ]----------------------------------------
'
WriteDS1994Reg:
     OWOUT OWpin, OWFERst, [SkipROM, WriteScratch, TA1, TA2, TargetCont]
     OWOUT OWpin, OWFERst, [SkipROM, ReadScratch]
     OWIN OWpin, OWBERst, [TA1, TA2, EndingAddr, str temp\1 ]
     OWOUT OWpin, OWFERst, [SkipROM, CopyScratch, TA1, TA2, EndingAddr]
     return

ReadDS1994Timer:
     OWOUT OWpin, OWFERst, [SkipROM, ReadMemory, TA1, TA2]
     OWIN OWpin, OWBERst, [str temp\5]
     return

DisplayTimer
     debug "TA:   ", hex2 TA2, hex2 TA1, cr
     debug "Timer = ", hex2 temp(4), hex2 temp(3), hex2 temp(2), hex2 temp(1), cr
     return
```

Listing 11 DS1994 Timer Handling (DS1994_TIMER.BSP)

3.3 Controlling LCDs with the HD44780 Controller

Hitachi's HD44780 controller is the industry standard among the LCD controllers for alpha-numeric LCDs. Most alpha-numeric LCDs use this LCD controller or a compatible device.

3.3.1 LCD Module with the HD44780 LCD Controller

The HD44780 LCD controller can control LCD modules with a maximum of 80 displayable characters. Different manufacturers offer LCD modules with 1, 2 or 4 lines and 16 or 20 characters.

Figure 49 shows a BT 42005 LCD module with 4 lines and 20 characters.

Figure 49 LCD Module BT42005 (Batron)

On the left half of the top of the PCB there are 14 connectors to control the LCD. LCD modules with 2x8 connectors are also available.

Table 7 lists a description of the I/O connections. From this table one could believe that the number of connections is too many for a BASIC Stamp with its 16 I/O pins. Because the HD44780 LCD controller can work also with a 4-Bit data bus a microcontroller needs only seven I/O pins to control the LCD module in parallel.

Pin	*Name.*	*Level*	*Function*
1	VSS	GND	Ground
2	VDD	+5 V	Supply voltage
3	Vo	0 …15 V	Contrast control
4	RS	H/L	L: Command; H: Data
5	R//W	H/L	L: Write operation; H: Read operation
6	E	H/L	Enable
7	DB0	H/L	
8	DB1	H/L	
9	DB2	H/L	
10	DB3	H/L	8-Bit Data Bus
11	DB4	H/L	
12	DB5	H/L	
13	DB6	H/L	
14	DB7	H/L	

Table 7 Connectors on a LCD Module

Figure 50 shows the complete hardware required for parallel control of a LCD module. Of the 8-Bit data bus the four most significant bits are used in 4-Bit mode.

If the LCD will only be written to then the R//W lines can be fixed to GND and we only need the RS and E lines for control.

We use six lines if we use the LCD only as a display. If you want to read back data from the internal display memory then you must use the R//W line for Read/Write control.

A high/low edge on input E of the LCD controller reads the data from data bus to the LCD controller. The logic level on input RS distinguishes between commands and data.

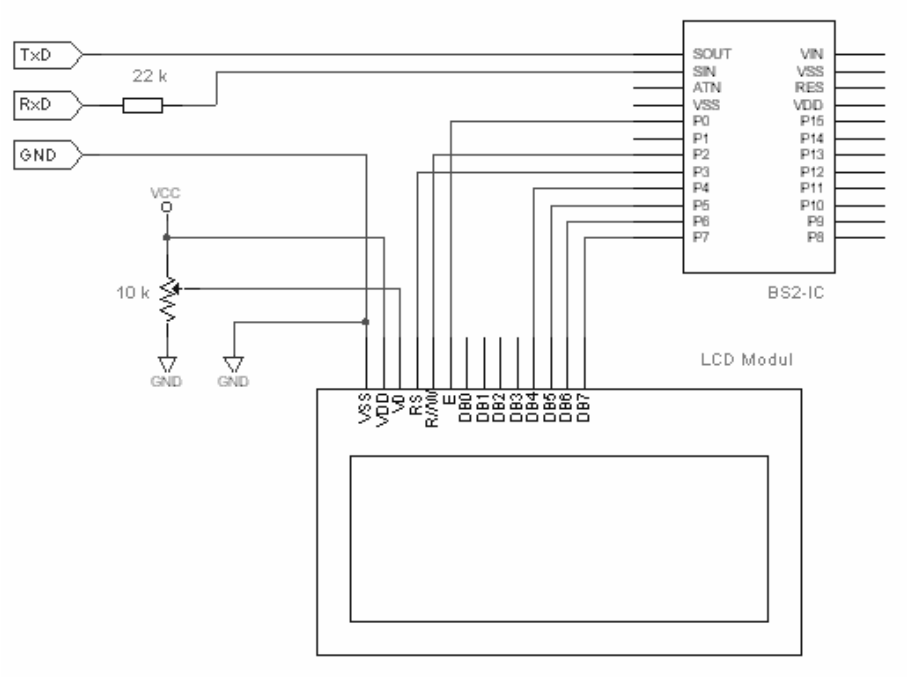


Figure 50 Parallel Control of a LCD Module

Before we turn to some program examples for LCD control we should have a closer look at the HD44780 controller itself. We'll describe only those details important for understanding our program examples. A link to an excellent description of the most aspects of the HD44780 is given in Chapter 9.

The HD44780 LCD controller contains internal two 8-Bit registers.

The instruction register (IR) saves the received command (RS = 0) while the data register (DR) saves the received data (RS = 1) before moving them into the Data Display RAM (DD RAM) or into the Character Generator RAM (CG RAM).

The DD RAM has a capacity of 80 Bytes. Therefore the maximum dimension of a connected LCD is limited to 4 lines and 20 characters. Bigger LCD modules work with several HD44780 controllers.

Table 8 shows the assignment of display position and memory location in DD RAM for a 4x16-LCD (LM041L, for example).

DDRAM	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*
1. Line	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
2. Line	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F
3. Line	10	11	12	14	13	15	16	17	18	19	1A	1B	1C	1D	1E	1F
4. Line	50	51	52	53	54	55	56	57	58	59	5A	5B	5C	5D	5E	5F

Table 8 Assignment of Display Position and Memory Location in DD RAM

As Table 8 shows, not all DD RAM locations are used to display characters. Unused memory cells are available as external RAM. But, in this case you must control the R/W line to change the data direction.

Table 9 shows the instruction set coding for the HD44780 while Table 10 shows some remarks for the used names.

Instruction	*RS*	*DB7*	*DB6*	*DB5*	*DB4*	*DB3*	*DB2*	*DB1*	*DB0*	*Description*
Clear Display	0	0	0	0	0	0	0	0	1	Clear display, cursor left top position
Cursor At Home	0	0	0	0	0	0	0	1	X	Cursor left top position
Set Entry Mode	0	0	0	0	0	0	1	I/D	S	Direction for movement of cursor and display content
Display On/Off	0	0	0	0	0	1	D	C	B	See Table 10 (D, C, B)
Cursor/ Display Shift	0	0	0	0	1	S/ C	R/L	X	X	See Table 10 (S/C, R/L)
Function Set	0	0	0	1	DL	N	F	X	X	See Table 10 (DL, N, F)
Set CG RAM Addr	0	0	1	ACG	>	>	>	>	>	Address of a CG RAM cell
Set DD RAM Addr	0	1	ADD	>	>	>	>	>	>	Address of a DD RAM cell
Data Write	1	Data	>	>	>	>	>	>	>	Write to DD RAM or CG RAM

Table 9 Coding of the Instruction Set of the LCD Controller HD44780

I/D	DD RAM resp. CG RAM address is incremented (I/D = 1) or decremented (I/D = 0) after writing.
S	Moves the display content (S = 1) or not (S = 0). The cursor does not move (Calculator). For I/D = 1 the display moves to left, for I/D = 0 to right.
D	Display on (D = 1) or off (D = 0). Data in DD RAM stay unchanged.
C	Cursor on (C = 1) or off (D = 0).
B	Cursor blinks (B = 1) or blinks not (B = 0).
S/C	Moves display content (S/C = 1) or the cursor (S/C = 0) one position according to R/L.
R/L	Moving to right (R/L = 1) or left (R/L = 0) without any change of data in DD RAM.
DL	Width of data bus: 8 Bit (DL = 1) or 4 Bit (DL = 0).
N	Number of lines in the display - one (N = 0) – several (N = 1).
F	Font - 5 x 7 Pixel (F = 0) - 5 x 10 Pixel (F = 1).
X	Don't care.

Table 10 Explanation of the HD44780 Instructions

With this explanation the following program examples can be interpreted and adapted to one's own requirements.

3.3.2 Parallel Control of an LCD Module

Figure 50 showed already the hardware needed for parallel control of a LCD module with the HD44780 LCD controller.

With the BS2p's special LCD commands the programming of an LCD control is very easy. With a little bit more code each BASIC Stamp can control such LCDs.

3.3.2.1 LCD Control by BS2p

The BS2p knows three very comfortable LCD commands LCDCMD, LCDOUT and LCDIN. The only limitations are the I/O pin assignments.

Listing 12 shows a program example for text output to a 4x16 LCD.

```
' -----[ Title ]-----------------------------------------------
'
' File...... LCD1.BSP
' Purpose... Stamp -> LCD (4-bit interface)
' Author.... Jon Williams (for BS1)
'            Claus Kuhnel (adaptions to BS2p & LM041)
' Started... 16 July 1994
' Updated... 03 Sept 2001

' -----[ Program Description ]-------------------------------
'
' This program demonstrates the various standard features of an LCD
' display that uses the Hitatchi HD44780 controller.
' The LCD used to test this program was the Hitachi LM041L (16x4).
'
' LCD Connections:
'
' LCD        (Function)         BS2p
' ----------------------------------
' pin 1         Vss             VSS
' pin 2         Vdd             VDD
' pin 3         Vo

' pin 4         RS              P3
' pin 5         R/W             P2
' pin 6         E               P0
' pin 7         DB0
' pin 8         DB1
' pin 9         DB2
' pin 10        DB3
' pin 11        DB4             P4
' pin 12        DB5             P5
' pin 13        DB6             P6
' pin 14        DB7             P7
'
' -----[ Revision Hisory ]---------------------------------
'
' 07-16-94 : Version 1.0 – Jon's original
' 09-20-94 : Version 2.0 - adaptation to LM041L
' 09-01-01 : Version 3.0 - adaptation to BS2
' 09-03-01 : Version 3.1 - adaptation to BS2p
'
' -----[ Directives ]--------------------------------------
'
'{$STAMP BS2p}                 'specifies a BS2p
'
' -----[ Constants ]---------------------------------------
'
E        con  0                ' LCD enable pin (1 = enabled)
RS       con 3             ' Regiser Select (1 = char)
```

```
' LCD control characters
'
WakeUp           con %00110000  'Wake-up
FourBitMode      con %00100000  'Set to 4-bit mode
OneLine5x8Font   con %00100000  'Set to 1 display line, 5x8 font
OneLine5x10Font  con %00100100  'Set to 1 display line, 5x10 font
TwoLine5x8Font   con %00101000  'Set to 2 display lines, 5x8 font
TwoLine5x10Font  con %00101100  'Set to 2 display lines, 5x10 font
DisplayOff       con %00001000  'Turn off display, data is retained
DisplayOn        con %00001100  'Turn on display, no cursor
DisplayOnULCrsr  con %00001110  'Turn on display, with anderline cursor
DisplayOnBLCrsr  con %00001101  'Turn on display, with blinking cursor
IncCrsr          con %00000110  'Auto-increment cursor, no display shift
IncCrsrShift     con %00000111  'Auto-increment cursor, shift display left
DecCrsr          con %00000100  'Auto-decrement cursor, no display shift
DecCrsrShift     con %00000101  'Auto-decrement cursor, shift display right
ClearDisplay     con %00000001  'Clear the display
HomeDisplay      con %00000010  'Move cursor and display to home position
ScrollLeft       con %00011000  'Scroll display to the left
ScrollRight      con %00011100  'Scroll display to the right
CrsrLeft         con %00010000  'Move cursor left
CrsrRight        con %00010100  'Move cursor right
MoveCrsr         con %10000000  'Move cursor to position (must add address)
MoveToCGRAM      con %01000000  'Move to CGRAM position (must add address)

Line1            con$80       ' addr line #1 ¦ 80H
Line2            con$C0       ' addr line #2 ¦ 80H
Line3            con$90       ' addr line #3 ¦ 80H
Line4            con$D0       ' addr line #4 ¦ 80H

' -----[ Variables ]----------------------------------------
'
char    var   byte           ' char sent to LCD
index   var   byte           ' loop counter


' -----[ Initialization ]-----------------------------------
'
data "THE BASIC STAMP!"      ' preload EEPROM
data "A PIC16C57 knows"      ' preload EEPROM
data "P B A S I C from"      ' preload EEPROM
data " Parallax Inc.  "      ' preload EEPROM

     outl = %00000000        ' clear the pins
     dirl = %11111111        ' set P0-P7 as outputs
     pause 1000              ' let the LCD settle

' Initialize the LCD (Hitatchi HD44780 controller)
'
LCDini:
     lcdcmd E,WakeUp                   ' 8-bit mode
     pause 10
```

```
     lcdcmd E,WakeUp
     pause 1
     lcdcmd E,WakeUp
     pause 1
     lcdcmd E,FourBitMode              ' 4-bit mode
     lcdcmd E,TwoLine5x8Font           ' set function for LM041
     lcdcmd E,DisplayOn                ' disp on, crsr off, blink off
     lcdcmd E,IncCrsr                  ' inc crsr, no disp shift
     lcdcmd E,ClearDisplay             ' clear LCD

' -----[ Main Code ]----------------------------------------
'
Start:
     lcdcmd E, ClearDisplay
     for index = 0 to 15
       read index, char                ' get char from EEPROM
       lcdout E, IncCrsr, [char]       ' write it
     next

     lcdcmd E, MoveCrsr+Line2          ' address second line
     for index = 16 to 31
       read index, char                ' get char from EEPROM
       lcdout E, IncCrsr, [char]       ' write it
     next

     lcdcmd E, MoveCrsr+Line3          ' address third line
     for index = 32 to 47
       read index, char                ' get char from EEPROM
       lcdout E, IncCrsr, [char]       ' write it
     next

     lcdcmd E, MoveCrsr+Line4          ' address forth line
     for index = 48 to 63
       read index, char                ' get char from EEPROM
       lcdout E, IncCrsr, [char]       ' write it
     next
     pause 2000                        ' wait 2 seconds

     goto Start                        ' do it all over
```

Listing 12 Text Output to an LCD (LCD1.BSP)

The program begins with a voluminous set of declarations. All LCD commands are defined as constants. Because such definitions use no memory it is the best to copy these declarations to the new source code. To develop readable source code always avoid direct usage of the assigned numbers in the LCD commands.

Handling the initialization of the LCD is quite similar. The first steps (WakeUp) are defined by the specifications of the HD44780 LCD controller. Adaptions can follow only after setting the 4-Bit mode.

In our program example we have four text strings saved in EEPROM. The main loop of the program reads these strings and displays them afterwards.

Before writing the first line the display is cleared and the cursor is moved to the home position (top left).

Before writing additional lines we have to set the start address. After that we read the string character by character from EEPROM and send it to the display.

The next program example is built quite similar. We write user-defined characters into the CGRAM.

In addition to the 192 defined characters the user can define eight characters of their own.

Figure 51 shows the HD44780 character set and the coding of the characters.

Figure 51 LCD Character Set

The user-defined characters are composed from bit patterns and will be saved in the CG RAM. The addressing of the CG RAMs is according to Table 11.

DB5	DB4	DB3	DB2	DB1	DB0
ASCII Code			Pixel Zeile		

Table 11 Content of a CG RAM Cell

Eight pixel lines build a user-defined character. The lowest line is identical to the cursor line and is normally empty.

In the next program example we define three user-defined characters. Maybe you know them from some LCD games (**Figure 52**). In a running sequence from left to right these characters eat the content of a LCD line.

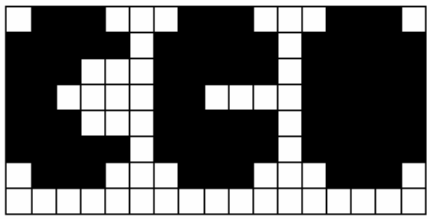


Figure 52 User-defined Characters

To simplify the generation a user-defined characters Parallax offers the ***LCD Character Creator*** for a free download. Generated user-defined characters can be saved. The resulting DATA instructions can be copied to the application program. Figure 53 shows the first of the three characters shown in **Figure 52** during its definition with the ***LCD Character Creator***.

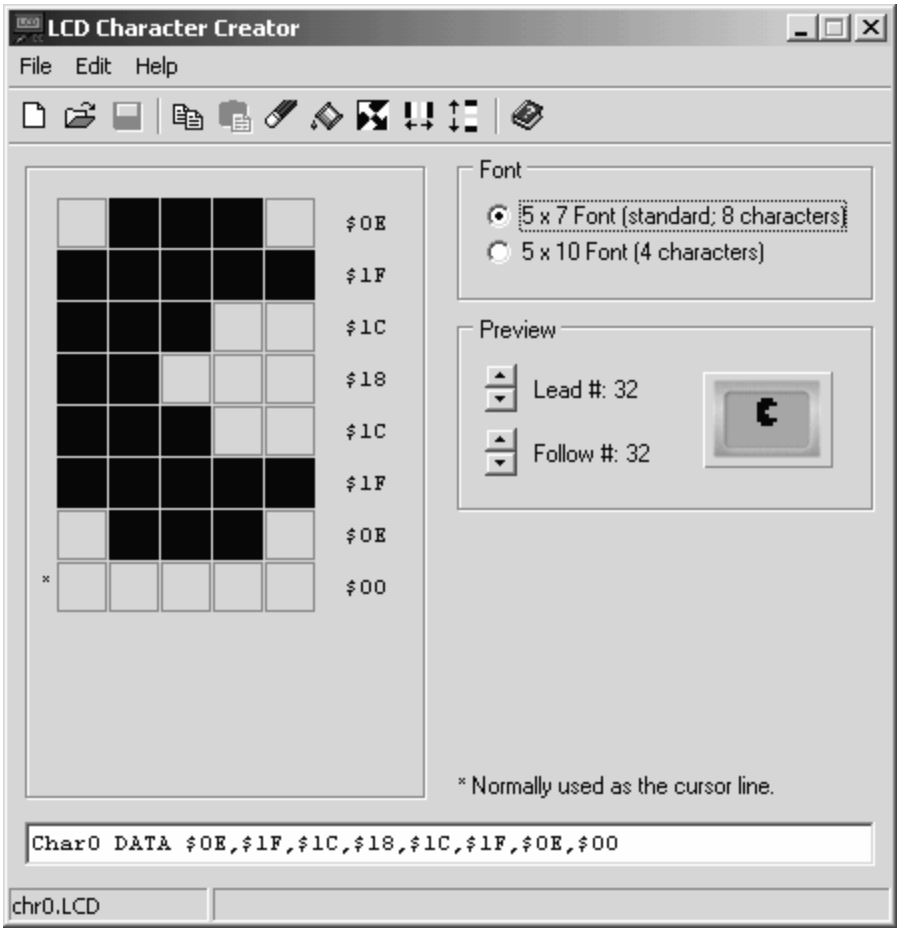


Figure 53 User-defined Character in LCD Character Creator

You can copy the DATA instruction output in the text field to your application program. The label (here `Char0`) must be adapted.

```
' -----[ Title ]-----------------------------------------------
'
' File...... LCD2.BSP
' Purpose... Stamp -> LCD (4-bit interface)
' Author.... Jon Williams (for BS1)
'            Claus Kuhnel (adaptions to BS2p & LM041)
' Started... 16 July 1994
' Updated... 03 Sept 2001
'
' -----[ Program Description ]-------------------------------
'
' This program demonstrates the generation of custom characters for
' an LCD display that uses the Hitatchi HD44780 controller. The LCD
' used to test this program was the Hitachi LM041L (16x4).
'
' LCD Connections:
'
' LCD        (Function)         Stamp
' ---------------------------------
' pin 1         Vss             VSS
' pin 2         Vdd             VDD
' pin 3         Vo
' pin 4         RS              P3
' pin 5         R/W             P2
' pin 6         E               P0
' pin 7         DB0
' pin 8         DB1
' pin 9         DB2
' pin 10        DB3
' pin 11        DB4             P4
' pin 12        DB5             P5
' pin 13        DB6             P6
' pin 14        DB7             P7
'
' -----[ Revision Hisory ]---------------------------------
'
' 07-16-94 : Version 1.0
' 09-20-94 : Version 2.0 - adaptation to LM041L
' 09-06-01 : Version 3.0 - adaptation to BS2p
'
' -----[ Directives ]--------------------------------------
'
'{$STAMP BS2p}              'specifies a BS2p
'
' -----[ Constants ]---------------------------------------
'
E         con  0             ' LCD enable pin (1 = enabled)
RS        con 3              ' Regiser Select (1 = char)

' LCD control characters
'
WakeUp           con %00110000 'wake-up
```

```
FourBitMode       con %00100000 'Set to 4-bit mode
OneLine5x8Font    con %00100000 'Set to 1 display line, 5x8 font
OneLine5x10Font   con %00100100 'Set to 1 display line, 5x10 font
TwoLine5x8Font    con %00101000 'Set to 2 display lines, 5x8 font
TwoLine5x10Font   con %00101100 'Set to 2 display lines, 5x10 font
DisplayOff        con %00001000 'Turn off display, data is retained
DisplayOn         con %00001100 'Turn on display, no cursor
DisplayOnULCrsr   con %00001110 'Turn on display, with anderline cursor
DisplayOnBLCrsr   con %00001101 'Turn on display, with blinking cursor
IncCrsr           con %00000110 'Auto-increment cursor, no display shift
IncCrsrShift      con %00000111 'Auto-increment cursor, shift display left
DecCrsr           con %00000100 'Auto-decrement cursor, no display shift
DecCrsrShift      con %00000101 'Auto-decrement cursor, shift display right
ClearDisplay      con %00000001 'Clear the display
HomeDisplay       con %00000010 'Move cursor and display to home position
ScrollLeft        con %00011000 'Scroll display to the left
ScrollRight       con %00011100 'Scroll display to the right
CrsrLeft          con %00010000 'Move cursor left
CrsrRight         con %00010100 'Move cursor right
MoveCrsr          con %10000000 'Move cursor to position (must add address)
MoveToCGRAM       con %01000000 'Move to CGRAM position (must add address)

Line1             con $80      ' addr line #1 ¦ 80H
Line2             con $C0      ' addr line #2 ¦ 80H
Line3             con $90      ' addr line #3 ¦ 80H
Line4             con $D0      ' addr line #4 ¦ 80H
'
' -----[ Variables ]----------------------------------------
'
char      var  byte           ' char sent to LCD
index1    var  byte           ' loop counter
index2    var  byte           ' loop counter
'
' -----[ Initialization ]-----------------------------------
'
Char0 DATA $0E,$1F,$1C,$18,$1C,$1F,$0E,$00     ' char 0
Char1 DATA $0E,$1F,$1F,$18,$1F,$1F,$0E,$00     ' char 1
Char2 DATA $0E,$1F,$1F,$1F,$1F,$1F,$0E,$00     ' char 2

data "THE BASIC STAMP!"      ' display string

     outl = %00000000        ' clear the pins
     dirl = %11111111        ' set P0-P7 as outputs
     pause 1000              ' let the LCD settle
'
' Initialize the LCD (Hitatchi HD44780 controller)
'
LCDini:
     lcdcmd E,WakeUp         ' 8-bit mode
     pause 10
     lcdcmd E,WakeUp
     pause 1
     lcdcmd E,WakeUp
```

```
     pause 1
     lcdcmd E,FourBitMode       ' 4-bit mode
     lcdcmd E,TwoLine5x8Font    ' set function for LM041
     lcdcmd E,DisplayOn         ' disp on, crsr off, blink off

     lcdcmd E,IncCrsr           ' inc crsr, no disp shift
     lcdcmd E,ClearDisplay      ' clear LCD
     lcdcmd E,MoveToCGRAM       ' set CG RAM addr to 0
    for index1 = 0 to 23        ' build 3 custom chars
     READ index1, char          ' get byte from data
        lcdout E, IncCrsr, [char] ' put into LCD CG RAM
    next
'
' -----[ Main Code ]----------------------------------------
'
start:
     lcdcmd E,ClearDisplay      ' clear LCD
    for index1 = 24 to 39       ' write the character string
     READ index1,char           ' get char from data
        lcdout E, IncCrsr, [char] ' write it
        pause 50                ' delay 50 ms--for fun only
    next
    pause 1000                  ' pause 1 second
    for index1 = 0 to 15        ' cover 16 characters
     for index2 = 0 to 4        ' 5 characters for animation
               ' set DD RAM address and move cursor to new addr
               lcdcmd E, MoveCrsr | index1
          LOOKUP index2,[0,1,2,1," "],char
               ' write animation character
               lcdout E, IncCrsr, [char]
               pause 75         ' delay between chars
          next
    next
    pause 1000
    goto Start                  ' replay
```

Listing 13 Definition of user-defined characters (LCD2.BSP)

The initialization part of this example program is particular to the BS2p. Each user-defined character is built in eight bytes and saved in EEPROM. After the initialization these bytes are saved in the beginning at address 0 in CGRAM. The *IncCrsr* command automatically increments the address. This works properly under the condition of saving the pixel lines in the their defined order and address 0 of CGRAM is selected before any saving.

The main part of the program displays the text saved in EEPROM on the LCD which are eaten from the user-defined Pac-Man characters. For that process the inner loop calls the user-defined characters and the space.

3.3.2.2 LCD Control by BS2 Modules

Using any other BASIC Stamp 2 module (BASIC Stamp 2, 2sx, 2e) you won't have the BS2p's easy-to-use LCD commands. However, we can program the needed signals in a parallel step-by-step fashion.

As a user you have to ask for each project: "Which performance and features am I willing to pay for when choosing a BASIC Stamp?".

Now you can see what is takes to use the less expensive BASIC Stamp 2 for the same project.

The following program example has the same function as the program in the chapter before. It is used to explain the procedure.

```
' -----[ Title ]-----------------------------------------------
'
' File...... LCD.BS2
' Purpose... Stamp -> LCD (4-bit interface)
' Author.... Jon Williams (for BS1)
'            Claus Kuhnel (adaptions to BS2 & LM041)
' Started... 16 July 1994
' Updated... 01 Sept 2001

' -----[ Program Description ]-------------------------------
'
' This program demonstrates the various standard features of
' an LCD display that uses the Hitatchi HD44780 controller.
'
' The LCD used to test this program was the Hitachi LM041L
' (16x4).
'
' LCD Connections:
'
' LCD        (Function)         BS2
' ---------------------------------
' pin 1         Vss             VSS
' pin 2         Vdd             VDD
' pin 3         Vo
' pin 4         RS              P3
' pin 5         R/W             P2
' pin 6         E               P0
' pin 7         DB0
' pin 8         DB1
' pin 9         DB2
' pin 10        DB3
' pin 11        DB4             P4
' pin 12        DB5             P5
' pin 13        DB6             P6
' pin 14        DB7             P7
```

```
'
' -----[ Revision Hisory ]---------------------------------
'
' 07-16-94 : Version 1.0 - Jon's original
' 09-20-94 : Version 2.0 - adaptation to LM041L
' 09-01-01 : Version 3.0 - adaptation to BS2
'
' -----[ Directives ]----------------------------------------
'
'{$STAMP BS2}                   'specifies a BS2
'
' -----[ Constants ]-----------------------------------------
'
E         con  0                ' LCD enable pin (1 = enabled)
RS        con 3                 ' Regiser Select (1 = char)

' LCD control characters
'
ClrLCD    con $01               ' clear the LCD
CrsrHm    con $02               ' move cursor to home position
CrsrLf    con $10               ' move cursor left
CrsrRt    con $14               ' move cursor right
DispLf    con $18               ' shift displayed chars left
DispRt    con  $1C              ' shift displayed chars right

Line1     con  $80              ' addr line #1 ¦ 80H
Line2     con $C0               ' addr line #2 ¦ 80H
Line3     con $90               ' addr line #3 ¦ 80H
Line4     con $D0               ' addr line #4 ¦ 80H

' -----[ Variables ]-----------------------------------------
'
outp    var byte                ' output workspace
char    var   byte              ' char sent to LCD
index   var   byte              ' loop counter

' -----[ Initialization ]-----------------------------------
'
data "THE BASIC STAMP!"         ' preload EEPROM
data "A PIC16C57 knows"         ' preload EEPROM
data "P B A S I C from"         ' preload EEPROM
data " Parallax Inc.  "         ' preload EEPROM

     outl = %00000000           ' clear the pins
     dirl = %11111111           ' set 0-5 as outputs
     pause 1000                 ' let the LCD settle

' Initialize the LCD (Hitatchi HD44780 controller)
'
LCDini:
     outl = %00110000           ' 8-bit mode
     PULSOUT E, 10
```

```
    pause 10
    PULSOUT E, 10
    pause 1
    PULSOUT E, 10
    pause 1
    outl = %00100000          ' 4-bit mode
    PULSOUT E, 10
    char = %00101000          ' set function for LM041
    gosub WrLCD
    char = %00001100          ' disp on, crsr off, blink off
    gosub WrLCD
    char = %00000110          ' inc crsr, no disp shift
    gosub WrLCD
    char = %00000001          ' clear LCD
    gosub WrLCD
    high RS                   ' LCD to character mode


' -----[ Main Code ]----------------------------------------
'
Start:
    for index = 0 to 15
      read index, char                    ' get char from EEPROM
      gosub WrLCD                   ' write it
    next

    char = Line2                    ' address second line
    gosub LCDcmd
    for index = 16 to 31
      read index, char              ' get char from EEPROM
      gosub WrLCD                   ' write it
    next

    char = Line3                    ' address third line
    gosub LCDcmd
    for index = 32 to 47
      read index, char              ' get char from EEPROM
      gosub WrLCD                   ' write it
    next

    char = Line4                    ' address forth line
    gosub LCDcmd
    for index = 48 to 63
      read index, char              ' get char from EEPROM
      gosub WrLCD                   ' write it
    next
    pause 1000                      ' wait 2 seconds

    char = ClrLCD                   ' clear the LCD
    gosub LCDcmd

    pause 500
```

```
    goto Start                      ' do it all over

' -----[ Subroutines ]------------------------------------------
'
' Send command to the LCD
'
' Load char with command value, then call
'
'   Clear the LCD.............. $01, %00000001
'   Home the cursor........... $02, %00000010
'   Display control........... (see below)
'   Entry mode................ (see below)
'   Cursor left............... $10, %00010000
'   Cursor right.............. $14, %00010100
'   Scroll display left....... $18, %00011000
'   Scroll display right...... $1C, %00011100
'   Set CG RAM address........      %01aaaaaa (Character Generator)
'   Set DD RAM address........      %1aaaaaaa (Display Data)
'
' Display control byte:
'
'   % 0 0 0 0 1 D C B
'               | | -- blink character ander cursor (1=blink)
'               | ---- cursor on/off (1=on)
'               ------ display on/off (1=on)
'
' Entry mode byte:
'
'   % 0 0 0 0 0 1 X S
'                 | --- shift display (S=1), left (X=1), right (X=0)
'                 ---- cursor move: right (X=1), left (X=0)
'
LCDcmd:
    low RS
    gosub WrLCD
    high RS
    return

' Write ASCII char to LCD
'
WrLCD:
    outl = outl & %00001000         ' RS = 1, data bus clear
    outp = char & %11110000         ' mask the high nibble
    outl = outl | outp              ' output the nibble
    PULSOUT E, 10                   ' strobe the Enable line
    outl = outl & %00001000
    outp = char & %00001111         ' get low nibble
    outp = outp << 4
    outl = outl | outp
    PULSOUT E, 10
    return
```

Listing 14 Text Output to an LCD (LCD.BS2)

3.3.3 Serial Control of an LCD Module

If the minimum six I/O lines are not available for LCD control you can use a serially controlled LCD module.

Scott Edwards Electronics [www.seetron.com] offers a Serial Backpack and completely equipped LCD modules. Figure 54 shows a printed circuit board able to connect different LCDs. To connect a BASIC Stamp there is a serial data line and +5 V and GND.

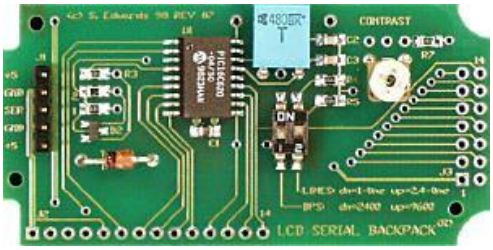


Figure 54 LCD Serial Backpack

The Serial Backpack converts the serially received data into parallel transmitted LCD commands.

The RS line differs between commands and data. On the serial side a command is marked by a leading code &FE. If you want to send the command Clear Display to the LCD the Serial Backpack must receive the sequence <&FE> <&01> first.

See Listing 15 for the details of serial LCD control. There are no special features.

```
' -----[ Title ]-----------------------------------------------
'
' File...... SERIALLCD1.BS2
' Purpose... Stamp -> Serial LCD
' Author.... Claus Kuhnel
' Started... 01 Sept 2001
' Updated...

' -----[ Program Description ]--------------------------------
'
```

```
' This program demonstrates the various standard features of an LCD
' display that uses the Hitatchi HD44780 controller.
' The LCD used to test this program was 20x4 Serial LCD from Seetron.
'
' -----[ Revision Hisory ]---------------------------------
'
' 09-01-01 : Version 1.0
'
' -----[ Directives ]-----------------------------------------
'
'{$STAMP BS2}               'specifies a BS2
'
' -----[ Constants ]-----------------------------------------
'
TxD       con  1             ' Serial Data to LCD
N9600     con $4054          ' Baudmode-9600 bps inverted
N2400     con $418c          ' Baudmode-2400 bps inverted
I         con $FE            ' Instruction prefix value

' LCD control characters
'
ClrLCD    con $01            ' clear the LCD
CrsrHm    con $02            ' move cursor to home position
CrsrLf    con $10            ' move cursor left
CrsrRt    con $14            ' move cursor right
DispLf    con $18            ' shift displayed chars left
DispRt    con  $1C           ' shift displayed chars right

Line1     con  $80           ' addr line #1 ¦ 80H
Line2     con $C0            ' addr line #2 ¦ 80H
Line3     con $94            ' addr line #3 ¦ 80H
Line4     con $D4            ' addr line #4 ¦ 80H

' -----[ Variables ]-----------------------------------------
'
char    var   byte                    ' char sent to LCD
index   var   byte                    ' loop counter

' -----[ Initialization ]-----------------------------------
'
data "THE BASIC STAMP!"               ' Preload EEPROM
data "A PIC16C57 knows"               ' Preload EEPROM
data "P B A S I C from"               ' Preload EEPROM
data "Parallax Inc."                  ' Preload EEPROM

' Initialize the Serial LCD (HD44780 controller & Serial Backpack)
'
LCDini:
     ' Now clear the screen in case there's text left from a previous
     ' run of the program. Note that there's a 1-second pause prior to
     ' sending any data to the Backpack. This gives the Backpack plenty
     ' of time to initialize the LCD after power up.
```

```
    low TxD                          ' Make the serial output low
    pause 1000                       ' Let the LCD wake-up

' -----[ Main Code ]----------------------------------------
'
start:
    serout TxD,n2400,[I,ClrLCD]      ' Clear the LCD screen
    serout TxD,n2400,[I,Line1+2]

    for index = 0 to 15
      read index, char               ' Get char from EEPROM
      serout TxD,n2400,[char]        ' and print it.
    next

    serout TxD,n2400,[I,Line2+2]     ' Move to line 2
    for index = 16 to 31
      read index, char               ' Get char from EEPROM
      serout TxD,n2400,[char]        ' and print it.
    next

    serout TxD,n2400,[I,Line3+2]     ' Move to line 3
    for index = 32 to 47
      read index, char               ' Get char from EEPROM
      serout TxD,n2400,[char]        ' and print it.
    next

    serout TxD,n2400,[I,Line4+3]     ' Move to line 4
    for index = 48 to 60
      read index, char               ' Get char from EEPROM
      serout TxD,n2400,[char]        ' and print it.
    next
    pause 2000                       ' Wait 2 seconds

    goto Start                       ' Do it all over
```

Listing 15 Serial LCD Control (SERIALLCD1.BS2)

3.4 Interface to the PC Keyboard

When interfacing to unique circuits we often look into using Al Williams' different products [www.al-williams.com/awce]. Al Williams designed several PAK Co-Processors. Table 12 shows the actual products:

Type	*Application*
PAK-I	Mathematic Co-Processor
PAK-II	Enhanced Mathematic Co-Processor
PAK-III	Port-Expander 8 I/O Lines
PAK-IV	Port-Expander 16 I/O Lines
PAK-V	PWM Co-Processor
PAK-VI	Keyboard Co-Processor
PAK-VII	Pulse-In Co-Processor
PAK-VIII	Pulse-Out Co-Processor
PAK-IX	Combines PAK-II with 8 digital I/O Lines and 5-Channel 10-Bit Analog-to-Digital Converter

Table 12 PAK-Co-Processors of AWC

We'll use the PAK-VI in this application example to connect a PC keyboard to a BS2.

Keyboards are tossed aside by computer users so they are offered cheap by second hand shops or even for free in the corner of your office. They are able to produced alphanumeric messages and control codes. It is useful to connect these keyboards to BASIC Stamps.

PAK-VI is a PICmicro specially programmed for that purpose. The keyboard signals are changed to serial data for working with the BASIC Stamp. The difficulties of trying to directly interface with the keyboard are avoided. This co-processor is also available for interfacing to a serial mouse instead of a keyboard, but we aren't using it in this application.

Let's take a short look at the bi-directional signals of a keyboard. It's normal use is connected to a PC with its cable. See to the books describing the PC connection in detail if you're interested in more details.

The synchronous serial data transfer is controlled by a clock signal from PC. In our case the PAK-VI generates this clock instead of the PC. Synchronous to the clock the data is transmitted bit by bit. The eight bits of a byte are enhanced with a frame of pulses. So we essentially get eleven bits for one byte.

This data transmission is hidden to the user with a PAK-VI. The BASIC Stamp is free of the trouble of handling the keyboard data transmission protocol on it's own.

The BASIC Stamp is connected to the PAK-VI with two-way RS232 in an asynchronous connection. With the SERIN and SEROUT commands the BASIC Stamp has control over the keyboard. In the following program example an additional line (I/O pin 15 of the BASIC Stamp) is used to reset the PAK-VI.

Sending commands to the keyboard and Pak-VI uses special control words. For both we'll use standard presettings. For PAK-VI these presettings come with the humorous name "Cook-Mode".

The keyboard is basically fixed to scan code 3. In this mode each key generates a so-called "make-code". Each key is marked with the number of its position. The task for the connected device is to align a specific character or control to this numbered event. Once a key is pressed there is no repeat function in this mode.

Figure 55 shows the connection diagram to the PS2 keyboard.

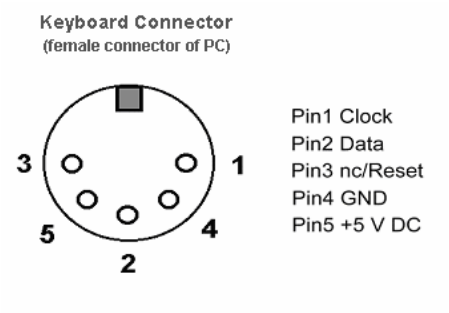


Figure 55 Connection Diagram to a PS/2 Keyboard

Table 13 shows the pin declarations of the PAK-VI co-processor.

Pin	*Name*	*I/O Type*	*Description*
1	RX	Input	TTL-level RS232 input
2	TX	Output	TTL-level RS232 output
17	Enable	Input	If this pin is not connected or high, the PAK transmits code it receives from the keyboard; the PAK always responds to commands.
18	Enable2	Input	If this pin is low, the PAK transmits code it receives from the keyboard; the PAK always responds to commands.
13	DAvil	Output	High when data is available.
4	RESET	Input	Hardware resets the PAK when low. Must be high for normal operation
3, 5	VSS	Power	GND (ground both pins)
14	VDD	Power	+ 5 V DC
15	RES1	Clock	connects to resonator
16	RES2	Clock	connects to resonator
11	DATA	I/O	Keyboard data line
12	CLOCK	I/O	Keyboard clock line
6…10	N/C	N/C	Not used

Table 13 Pin connections PAK-VI

Figure 56 shows the connection of a PC keyboard via keyboard co-processor PAK-VI to a BS2.

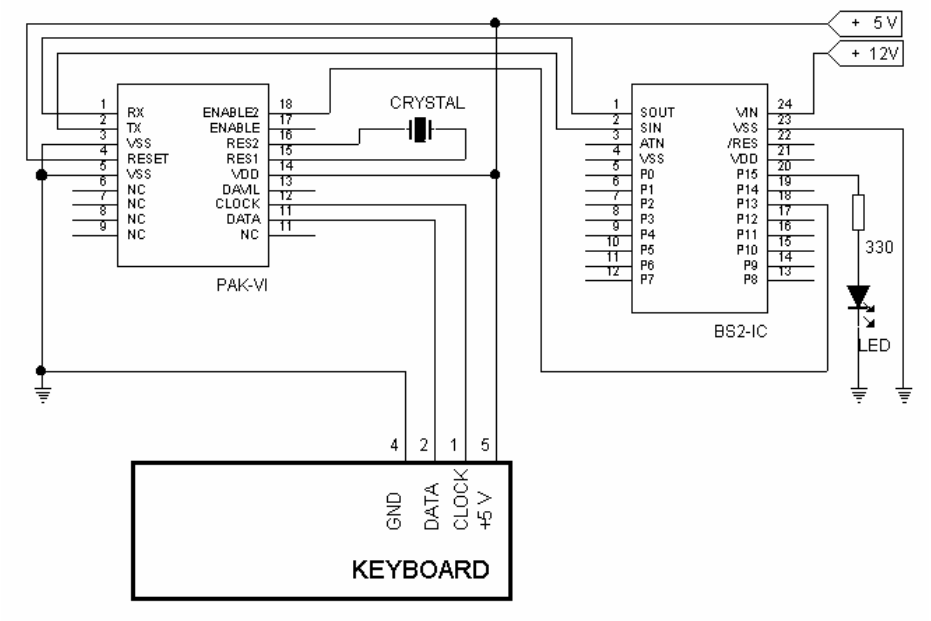


Figure 56 PC Keyboard with PAK-VI to BS2

One notable fact is that when interfacing via PAK-VI the keyboard appears to the BS2 as a serial asynchronous RS232 coupled device. Because we have no data transmission from BS2 to the keyboard in our example you will find no SEROUT command in the example program.

The connection from the pin SOUT of the BS2 to the PAK-VI input RX can be cancelled. This connection is shown here to prepare the reader for possible modifications.

As a simple example we control BS2 outputs with signals from the PC keyboard with an LED connected to I/O pin15. The flashing period of this LED is controlled by a keyed cipher. The keyed cipher is shown on the Debug Window.

Listing 16 shows the program used for the BS2.

```
' -----[ Title ]-----------------------------------------------
'
' File......  PAK_NUM2.BS2
' Purpose...  Using PAK VI for recognizing codes from
'             keyboard to control the period of a flashing LED
' Author....  Klaus Zahnert
' Started...  06/16/01
' Updated...
'
' -----[ Program Description ]--------------------------------
'
' BASIC Stamp 2 is connected to PC's keyboard by using the PAK VI
' keyboard-controller to make input for codes 0...9. Timing-value
' for blinking the LED is aligned to key pressed codes with schedules.
' this program runs for demonstration PAK VI with polling from BS2.
'
' -----[ Revision Hisory ]-------------------------------------
'
'
' -----[ Constants ]-------------------------------------------
'
LED       con 15             'pin to drive LED
'baudval  con 16624          'BS2SX/9600/8/n/1 direct con.
baudval   con 16468          'BS2  /9600/8/n/1 direct con.
tout      con 100            'wait for SERIN-response (ms)
datinpin  con 12
fpin      con 13
'
' -----[ Variables ]-------------------------------------------
'
border    var word           'generated with precalc.schedule
n         var word           'Loopindex
datinbyt  var byte
datoldbyt var byte
code var byte
'
' -----[ Initialization ]--------------------------------------
'
'{$STAMP BS2}

high LED
low  fpin                    'PAK 6 enable 2
border    =   10             'Startvalue

' -----[ Main Code ]------------------------------------------
'
start:
     For n = 1 to border        'toggle LED with period(keypr)
     serin datinpin,baudval,tout,goon,[datinbyt]
goon:    If datinbyt <> datoldbyt then calc 'need new period?
     next
```

```
     toggle LED
     goto start

calc:
     lookdown datinbyt,[6,103,51,102,25,101,50,100,12,99],code
    'makes code :    0,  1, 2,  3, 4,  5, 6,  7, 8  9

     lookup   code  ,[1,2,4,7,12,22,42,75,135,255],border
    'makes value for bord1 from schedules position, given by code
     debug "Ziffer = ", dec code, TAB,TAB,"Zeitkonst.= ",dec border,cr

     'store datinbyt in datoldbyt to compare again
     datoldbyt = datinbyt
     goto start
```

Listing 16 Keyboard input using PAK-VI (PAK_NUM2.BS2)

In this program example the restriction is that we're only using ciphers as a reference. By enhancing the scan code you can receive more inputs from the keyboard. The limit is only determined by the memory space of the BASIC Stamp.

The program permanently polls the state of the PAK-VI output register. If any key was pressed the output value changes. Recognizing that, the polling loop is left and the new output is used for identifying the pressed key.

In the program example the main code of the codes (from keyboard documentation) is stored in a lookdown table. The returned value of the LOOKDOWN command is the codes value. Comparing this value with one in the table is the same task as the PC's keyboard controller does but with more elegance for the whole set of characters.

The DEBUG command sends the value of recognized key to the Debug Window.

In this simple example it is not practical the flashing period of a LED it is not practical to set the flashing time immediately from the keyed values 0…9. For more visible effects we use a LOOKUP table to align blinking values in steady increments to the variable *Border*. This value is used as an index for a counting loop.

After finishing the counting by ending the loop, the LED output is toggled.

3.5 Port Enhancement with Shift Registers

With 16 I/O pins the BASIC Stamps (BS2, BS2sx, BS2e and BS2p-24) are armed with almost enough resources to connect different peripheral components. With BS2p-40 the number of I/O pins is doubled to 32. But, for enhancing the number of I/O pins often there is an inexpensive solution with separate integrated circuits.

Output registers store values from serial input with a few I/O lines. The bits are shifted from one register to the next while input occurs. After that, the message is available on parallel outputs.

This kind of port enhancement is useful in cases where the loss of speed in data transmission can be tolerated by the connected peripheral components. This is often the reality with common examples using keyboards and displays. It is often possible to design mechanical control systems with such hardware.

There are different types of serial protocols. With one line at a minimum you can have functional RS232 communication.

Synchronous serial protocols need a clock line in addition to the data line. A sample was already given in Chapter 3.1.1 with the I²C coupled port enhancement using the PCF8574A. This example developed a parallel printer control with only two I/O lines.

The cheapest solution is to use shift registers from the standard line of TTL circuits.

In the following example the D-Flip-Flop 74HCT174 is used. The 74HCT174 contains six single D-Flip-Flops with a common clock line. Each of the six Flip-Flops has an input and two outputs with inverse levels. A D-Flip- Flop moves the state of the input line to outputs at the positive edge of the clock line.

To build a shift register, the flip-flops must be cascaded in chain. For a 74HCT174 the data output Q(n) must be connected to the data input D(n+1) by external wiring. With the positive edge of the clock the state of the former Flip-Flop is moved to the following Flip-Flop. Using one 74HCT174 we gain six digital outputs for two I/O pins of the BASIC Stamp.

There is no strict advantage with this small number of wired outputs. But in the following example it is easy to recognize how it works as we demonstrate using a chain of 74HCT174s. By cascading the 74HCT174s we get n*6 outputs by using only 2 BASIC Stamp I/O pins.

Figure 57 shows the three 74HCT174 cascaded to get a 18-Bit shift register. The transition from one Flop-Flop to the next is drawn by arrows in a symbolic manner.

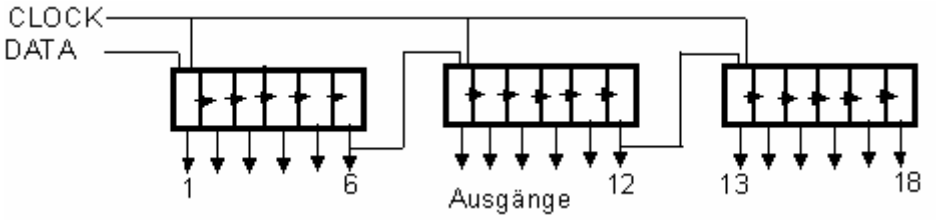


Figure 57 Cascading three 74HCT174

Figure 58 shows the circuit diagram for the port enhancement using one 74HCT174.

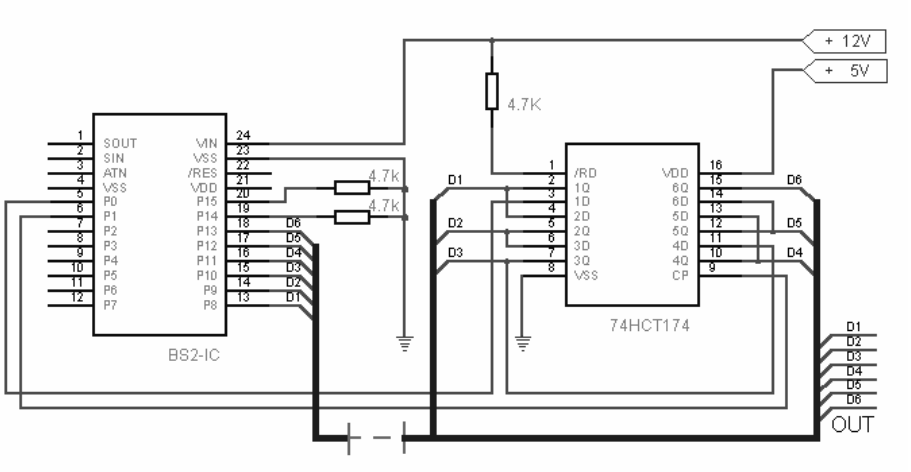


Figure 58 Port Enhancement with 74HCT174

The BASIC Stamp 2's P0 is used for data output and I/O pin 1 is for the clock. The /RD input of 74HCT174 is connected high using a pull-up resistor of 4.7 kOhm. In this configuration the shift register is setup for shifting in data serially. Pay careful attention to the wiring connection between the six flip-flops to get a properly functioning shift register.

The six outputs of the shift register, shown on the right site of the diagram on the bus are available for your project. But why reconnect the six output lines to the BASIC Stamp? This seems to negate the additional outputs we've created!

This connection, denoted by a dashed line, is only for demonstration. By connecting the parallel output to the BASIC Stamp, we can re-read it to see the result in the Debug Window. While running the program we can verify the equality of transmitted and received message.

Listing 17 shows the source of the program SHIFTREG.BS2.

```
'{$STAMP BS2}

' -----[ Title ]-----------------------------------------------
'
' File......  SHIFTREG.BS2
' Purpose...  Port Enhancement with serial connected
'             shift regiser
' Author....  Klaus Zahnert
' Started...  06.06.01
' Updated...

' -----[ Program Description ]-------------------------------
'
' Two output lines for clock and data to drive a connected
' shiftregiser are used for port enhancing up to 6 output-lines.

'For demonstration these outputs are inputs of BS2-IC. So the
'serial transmitted states of that 6 lines are shown on debug-
'window to see the same contents.

' -----[ Revision Hisory ]--------------------------------
'
' -----[ Constants ]-------------------------------------------
'
clkpin   con 1
datapin con 0

' -----[ Variables ]-------------------------------------------
'
outbyt   var byte

' -----[ Initialization ]--------------------------------------
'
     DIRH = $00             'I/O Port.highbyte for input
     low clkpin
     low datapin
```

```
' -----[ Main Code ]----------------------------------------
'
start:
     For outbyt = 0 to 31
          shiftout datapin, clkpin, msbfirst, [outbyt\6]
          debug dec2 outbyt,tab,tab,bin8 outbyt,tab,bin8 INH ,cr
          pause 500
     next
     end
```

Listing 17 Port Enhancement with Shift Register (SHIFTREG.BS2)

In a loop the SHIFTOUT command sends the values 0 to 31 to the shift register. The DEBUG is used to display the value sent with the value read back.

4 BASIC Stamps on the Net

TCP/IP is the standard for a platform-independent data exchange of different components via intranet or internet. A device connected with TCP/IP to the internet can accessed from any point in the internet. The infrastructure needed for this type of networking uses Ethernet networks, telephone lines, or even wireless. The device that should be integrated to a network only needs a TCP/IP stack.

Depending on the application there are very different solutions to make this work.

The resources required for implementing a TCP/IP stack are not available in small microcontrollers like the BASIC Stamp. A simple way out is to use the PC as a gateway.

4.1 MondoMini Webserver

The MondoMini Webserver (www.mondomini.com) is a gateway installed on a PC and can connect any microcontroller with the internet (Figure 59).

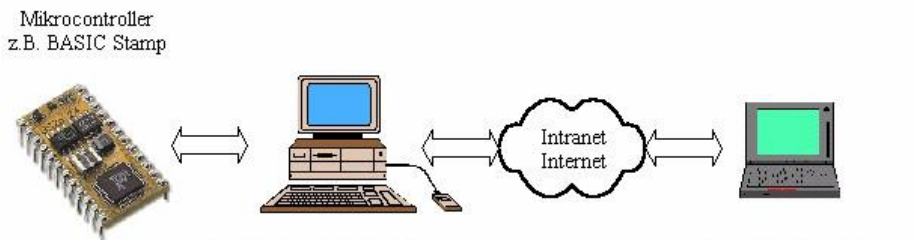


Figure 59 MondoMini Webserver as a Gateway

MondoMini Webserver has the following features:

Windows 98/NT/2000 compatibility

Usage of the PC to display web sites which were updated by the microcontroller

Update of websites via FTP

Transmission of control data to the microcontroller

Display of messages generated by the microcontrollers on a web site

Sending of an E-Mail triggered by event of the microcontroller application

The MondoMini Webserver runs on a PC using Windows 98/NT/2000. Simple commands over the serial interface build the communication between the microcontroller and the MondoMini Webserver. The MondoMini Webserver includes the received data into HTML pages accessible by a Web browser.

4.2 BASIC Stamp connected to the MondoMini Webserver

In the next program examples we can use any BS2 connected to the MondoMini Webserver. We do not need the special features of the BS2p for this application.

The hardware base for the following examples is the circuit diagram shown in Figure 60. If you use the BASIC Stamp Activity Board then you already have the complete circuitry.

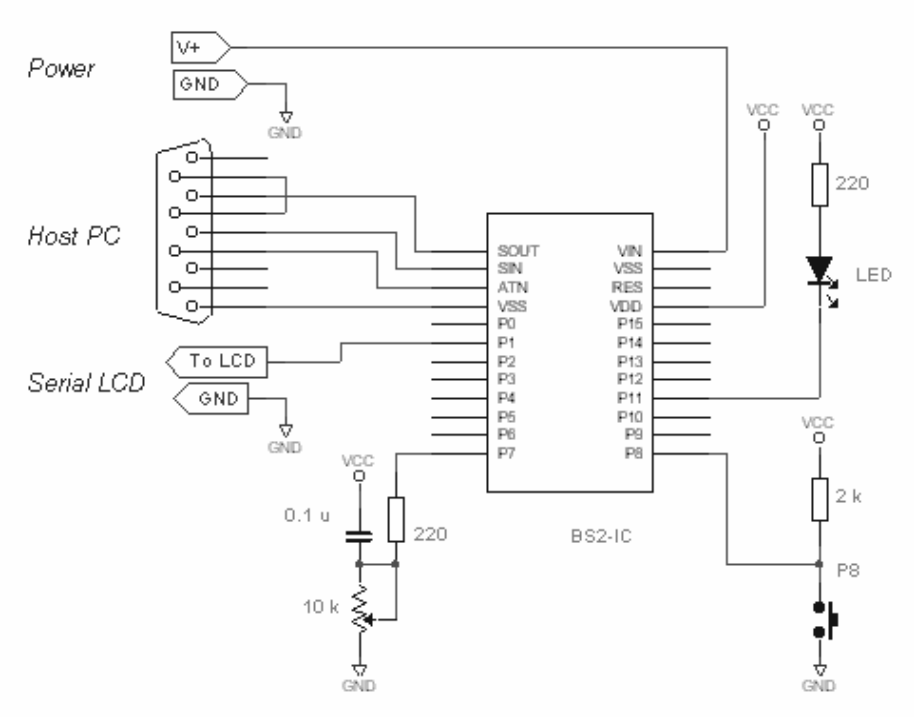


Figure 60 BS2 with MondoMini Webserver

To run the program examples the BS2 must be connected serial to the PC according to Figure 60 and the MondoMini Webserver must be running on the Host PC.

Because the MondoMini Webserver is connected to the programming interface of the BS2 we have to use the following commands for the communication between MondoMini Webserver and the BS2:

BS2 transmits:

```
SEROUT 16,84,1,[...]
```

BS2 receives:

```
        SERIN 16,84,1000,nocommand,[STR string\7\";"]
```

4.2.1 Sending E-Mails

Sending an E-Mail can be initiated at a selected time.

In our first program example the BS2 sends an E-Mail after the pushbutton on P8 is pressed.

In an endless loop we'll check the pushbutton on P8. A blinking LED signalizes that the program is running.

After detecting that the pushbutton is pressed we query the potentiometer connected to P7. You can put any hardware on this pin for any procedure or data acquisition. We display the result on a serial connected LCD for verification before the E-Mail is built and sent. Listing 18 shows the program responsible for sending E-Mails.

```
' This program demonstrates sending email from BASIC Stamp 2
' to one email address.
' BASIC Stamp Activity Board was used as target hardware.
' After pressing a key (P8) the pot meter connected to P7 is
' read and the pot meter value is sent afterwards.
' The serial link is connected to a PC's COM Port running
' MondoMini webserver.

' Created: 28.02.2001 Claus Kuhnel

LCD      con 1                        ' Serial LCD on P1
LED      con 10
POTIN    con 7                        ' Potentiometer on P7
N2400    con $418c                    ' Baudrate for serial LCD
I        con 254                      ' Instruction prefix value.
CLR      con 1                        ' LCD clear-screen instruction

adc  var word                         ' Word variable for ADC

     pause 1000
     SEROUT LCD,n2400,[I,CLR]         ' Clear the LCD screen.
     pause 1
     SEROUT LCD,n2400,[I,128]
     SEROUT LCD,n2400,["Value:"]      ' Print message.

start:
     high LED: pause 200              ' Blink LED
     low LED :pause 10
```

```
     IF In8 <> 0 THEN pass      ' Pass if key is not pressed
     gosub readpot              ' Read pot meter value
     SEROUT LCD,n2400,[I,135]   ' Print message on LCD
     SEROUT LCD,N2400,[DEC2 adc.lowbyte]
     gosub sendmail
pass:    goto start

readpot:
     high potin
     pause 1
     RCTIME POTIN, 1, adc
     adc=adc/2
     adc=adc.nib2
     return

sendmail:
     SEROUT 16,84+$4000,1,["EM=info@ckuehnel.ch;"]
     SEROUT 16,84+$4000,1,["This is an email alert generated by\n;"]
     SEROUT 16,84+$4000,1,["BASIC Stamp and MondoMini Webserver.\n;"]
     SEROUT 16,84+$4000,1,["P8 key was pressed on BS Activity Board.\n;"]
     SEROUT 16,84+$4000,1,["Read Pot value is ",dec adc,".;"]
     SEROUT 16,84+$4000,1,["EM;"]
     return
```

Listing 18 Sending E-Mails (EMAIL.BS2)

Building the E-Mail is quite simple. The tag `EM=...` marks the E-Mail address of the receiver. The text to be sent follows this address and the E-Mail is closed by the tag `EM`.

Now you can click the “Events” tab on MondoMini Webserver to show all activities of the MondoMini Webserver. Figure 61 shows this screen.

Figure 61 Webserver Protocol

At 12:44:19 an e-Mail was sent triggered by the pushbutton P8 on the BASIC Stamp Activity Board. Compare the protocol and program Listing 18 to verify.

The E-Mail received is shown below. Figure 62 shows the received E-Mail in the mail program Eudora Light. The appearance of this e-Mail depends of the e-Mail client you are using, of course. The content would be the same.

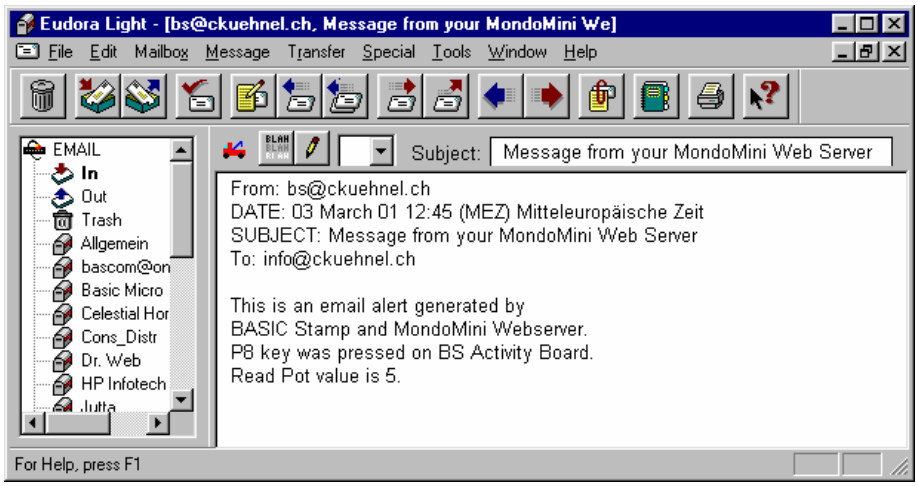


Figure 62 Received E-Mail

4.2.2 Query of Variables

The e-Mail in the last chapter showed the value of the potentiometer (5) as a variable inserted in the text of the message.

If one wants to query a variable in the BS2 using the web browser then MondoMini must know this variable's name. The BS2 application program would transmit the content of the variable to MondoMini so it can be queried by the web browser. The following program example explains how the BS2 transfers a variable to the MondoMini Webserver.

Listing 19 contains an endless loop which queries the potentiometer and transmits the value to the LCD and to MondoMini Webserver. The command SEROUT `16,84+$4000,1,["P1=",DEC adc,";"]` sends the variable `adc` to MondoMini. The web server can identify this variable by the tag `P1`. The blinking LED flashes to demonstrate the program is operating.

```
' This program demonstrates sending a variable from BS2
' to MondoMini web server.
' Your web browser can read this value and display.
' BASIC Stamp Activity Board was used as target hardware.
' After pressing a key (P8) the pot meter connected to P7
' is read and the pot meter value is sent afterwards.
' The serial link is connected to a PC's COM Port running
```

```
' MondoMini webserver.

' Created: 28.02.2001 Claus Kuhnel

LCD      con 1                          ' Serial LCD on P1
LED      con 10
POTIN    con 7                          ' Potentiometer on P7
N2400    con $418c                      ' Baudrate for serial LCD
I        con 254                        ' Instruction prefix value.
CLR      con 1                          ' LCD clear-screen instruction

adc      var word                       ' Word variable for ADC

     pause 1000
     SEROUT LCD,n2400,[I,CLR]           ' Clear the LCD screen.
     pause 1
     SEROUT LCD,n2400,[I,128]
     SEROUT LCD,n2400,["Value:"]        ' Print message.

start:
     high LED: pause 500:               ' Blink LED
     low LED :pause 10
     gosub readpot
     SEROUT LCD,n2400,[I,135]
     SEROUT LCD,N2400,[DEC2 adc.lowbyte] ' Print value on LCD
     SEROUT 16,84+$4000,1,["P1=",DEC adc,";"]
     goto start

readpot:
     high potin
     pause 1
     RCTIME POTIN, 1, adc
     adc=adc/2
     adc=adc.nib2
     return
```

Listing 19 Sending a Variable (PUTVAR.BS2)

For querying a variable using a web browser you need to install an HTML program. We need no special features here and can use any text editor to write this HTML program. Figure 63 shows the presentation of this file in the Internet Explorer before we have a look to the HTML text itself.

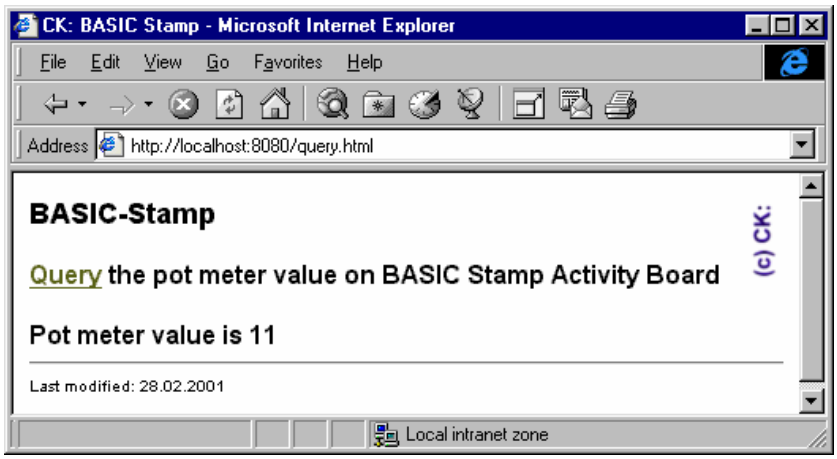


Figure 63 Query a Variable (QUERY.HTML)

A click to the hyperlink "Query" starts a query to the MondoMini Webserver and it transmits the subject value. Listing 20 shows the HTML text of the page shown in Figure 63.

```
<HTML>
<HEAD>
   <TITLE>CK: BASIC Stamp</TITLE>
</HEAD>
<BODY BGCOLOR="#FFFFFF">
<H3><FONT SIZE="+3" FACE="Arial"><B><IMG SRC="ck.gif" WIDTH=20 HEIGHT=55
ALIGN=right></B></FONT><FONT FACE="Arial">BASIC-Stamp</FONT></H3>

<P><A HREF="query.html"><FONT FACE="Arial"><B>Query</B></FONT></A>
<FONT FACE="Arial"><B>the pot meter value on BASIC Stamp Activity Board</B></FONT></P>

<P><FONT FACE="Arial"><B>Pot meter value is 'P1</B></FONT>

<HR>

<FONT SIZE="-2" FACE="Arial">Last modified: 28.02.2001</FONT></P>
</BODY>
</HTML>
```

Listing 20 Query a Variable (QUERY.HTML)

By clicking the links the program QUERY.HTML refreshes the web page. The variable P1, linked to the variable `adc` in the BS2 application program will be displayed by the Web browser.

4.2.3 Changing of Variables

If you want to initialize or modify a variable in the BS2 application program using the web browser you must know whether the webserver has access to the client. In our case the BS2 is the client (Client-Server-Model).

The BS2 application program has to ask the Webserver if there are new commands or data for it. Listing 21 shows an example BS2 program. The new commands here are marked in bold.

```
' This program demonstrates sending a variable
' from MondoMini web server to BASIC Stamp 2.
' A click in your web browser sets a flag on BS2.
' BASIC Stamp Activity Board was used as target hardware.
' The serial link is connected to a PC's COM Port running
' Mondo Mini webserver.

' Created: 28.02.2001 Claus Kuhnel

string   var byte(8)
flag     var bit

LCD      con 1                          ' Serial LCD on P1
LED      con 10
POTIN    con 7                          ' Potentiometer on P7
N2400    con $418c                      ' Baudrate for serial LCD
I        con 254                        ' Instruction prefix value.
CLR      con 1                          ' LCD clear-screen instruction


     DIRS =%0000111100000000
     OUTC =%1111

     pause 1000
     SEROUT LCD,n2400,[I,CLR]           ' Clear the LCD screen.
     pause 1
     SEROUT LCD,n2400,[I,128]
     SEROUT LCD,n2400,["Value:"]        ' Print message.

     'Tell MondoMini to clear all commands queued up
     SEROUT 16,84+$4000,1,["CC;"]

start:
```

```
     high LED: pause 500:              ' Blink LED
     low LED :pause 10


     'Query the MondoMini for a command.
     SEROUT 16,84+$4000,1,["QC;"]

     'Wait 1000 ms for a command from MondoMini
     SERIN 16,84+$4000,1000,nocommand,[STR string\7\";"]

     'Test for "P1=1"
     IF (string(0)<>"P" OR string(1)<>"1" OR string(3)<>"1") THEN nextcommand
          flag = 1 : OUT8 = flag
          SEROUT LCD,n2400,[I,135]
          SEROUT LCD,N2400,[BIN1 flag] ' Print value on LCD
          SEROUT 16,84+$4000,1,["P1=", BIN1 flag, ";"]

nextcommand:
     'Test for "P1=0"
     IF (string(0)<>"P" OR string(1)<>"1" OR string(3)<>"0") THEN nextcommand1
          flag = 0 : OUT8 = flag
          SEROUT LCD,n2400,[I,135]
          SEROUT LCD,N2400,[BIN1 flag] ' Print value on LCD
          SEROUT 16,84+$4000,1,["P1=", BIN1 flag, ";"]

nextcommand1:

nocommand:
     goto start
```

Listing 21 Receiving a Command (GETVAR.BS2)

Before running the endless loop the program clears everything in MondoMini Webserver by the command "CC". Inside the endless loop the program queries the MondoMini Webserver periodically for received commands.

The command `SERIN 16,84+$4000,1000,nocommand,[STR string\7\";"]` reads a command with maximum of 7 characters or until the character ";" in the variable `string` is encountered. If no command is received from MondoMini Webserver the BS2 goes into timeout and runs the loop again.

If a command is received then we have to decode it. Valid are the commands P1=0 and P1=1 only; all other commands are ignored. Depending on the flag variable the BS2 application program controls several displays (LED, LCD).

For setting or resetting this flag from the web browser an appropriate HTML program was installed on the Webserver. Figure 64 shows the web side in the Internet Explorer belonging to it.

Figure 64 Setting a Flag (SET.HTML)

A click on one of the hyperlinks sends a variable to MondoMini Webserver where it is saved for queries by the BS2. Listing 22 shows the HTML text of that web side shown in Figure 64. Both hyperlinks were marked bold later.

```
<HTML>
<HEAD>
   <TITLE>CK: BASIC Stamp</TITLE>
</HEAD>
<BODY BGCOLOR="#FFFFFF">
<H3><FONT SIZE="+3" FACE="Arial"><B><IMG SRC="ck.gif" WIDTH=20 HEIGHT=55
ALIGN=right></B></FONT><FONT FACE="Arial">BASIC-Stamp</FONT></H3>

<P><FONT FACE="Arial"><B>Set a flag on BASIC Stamp Activity
Board</B></FONT></P>

<P><A HREF="$P1=0;$set.html"><FONT FACE="Arial">Press here to reset the
flag</FONT></A></P>

<P><A HREF="$P1=1;$set.html"><FONT FACE="Arial">Press here to set the
flag</FONT></A></P>

<HR>

<FONT SIZE="-2" FACE="Arial">Last modified:
28.02.2001</FONT></P>
</BODY>
```

```
</HTML>
```

Listing 22 Setting a Flag (SET.HTML)

4.2.4 BASIC Stamp Monitoring System

Based on the explanations we've provided we can build a monitoring system with the following features:

- Query of a measuring value
- Alarm after exceeding one of the defined limits by E-Mail
- Periodic query of a measured value and limit by a Web browser
- Signalization of the Alarm on a web site displayed using a web browser
- Setting the limit via a web page

The measuring procedure is simulated by a query of the potentiometer as before. Listing 23 shows the BS2 application program. All commands important for the communication with the MondoMini Webserver were marked bold later again.

```
' This program demonstrates setting a limit in a BS2
' application program by your web browser.
' BASIC Stamp Activity Board with a serial LCD was
' used as target hardware.
' The BS2 application reads periodically the pot meter value,
' displays them on LCD and sends an email when the value is
' over the limit.
' The serial link is connected to a PC's COM Port running
' Mondo Mini webserver.

' Created: 28.02.2001 Claus Kuhnel


LCD      con 1                        ' Serial LCD on P1
LED      con 10
POTIN    con 7                        ' Potentiometer on P7
N2400    con $418c                    ' Baudrate for serial LCD
I        con 254                      ' Instruction prefix value.
CLR      con 1                        ' LCD clear-screen instruction

string   var byte(8)                  ' Command string
limit    var byte                     ' Byte variable for limit
adc      var word                     ' Word variable for ADC
number   var word                     ' Word variable for number conversion
```

```
ii       var nib                        ' Index
emailsent var bit                       ' Flag


     DIRS =%0000111100000000
     OUTC = %1111

     limit = 15                         ' Initialize limit with maximum

     'Initialize the LCD
     pause 1000
     SEROUT LCD,n2400,[I,CLR]           ' Clear the LCD screen.
     pause 1
     SEROUT LCD,n2400,[I,128]
     SEROUT LCD,n2400,["Value:"]        ' Print message.
     SEROUT LCD,n2400,[I,138]
     SEROUT LCD,n2400,["Limit:"]        ' Print message.


     'Tell MondoMini to clear all commands queued up
     SEROUT 16,84+$4000,1,["CC;"]

start:
     high LED: pause 100:               ' Blink LED
     low LED

     gosub readpot                      ' read pot meter value

     IF adc > limit THEN alert          ' Alert if adc exceeds limit
     emailsent = 0
     SEROUT 16,84+$4000,1,["M1= ;"]

displayvalues:
     SEROUT LCD,n2400,[I,135]           ' Print values on LCD
     SEROUT LCD,N2400,[DEC2 adc.lowbyte]
     SEROUT LCD,n2400,[I,145]
     SEROUT LCD,N2400,[DEC2 limit]

     'Report the value of pot meter and limit
     SEROUT 16,84+$4000,1,["P1=",DEC adc.lowbyte,";"]
     SEROUT 16,84+$4000,1,["P2=",DEC limit,";"]


     'Query the MondoMini for a command.
     SEROUT 16,84+$4000,1,["QC;"]

     'Wait 1000 ms for a command from MondoMini
     SERIN 16,84+$4000,1000,nocommand,[STR string\7\";"]

     'Test for "P2=xxx"
     IF (string(0)<>"P" OR string(1)<>"2" OR string(2)<>"=") THEN nextcommand
     number = 0
     ii = 3
```

```
m1:  IF string(ii) = 0 THEN numbered
     number = number * 10 + string(ii) - 48
     ii = ii + 1
     goto m1
numberend:
     limit = number.lowbyte

nextcommand:
     ' Insert further commands if needed

nocommand:
     goto start

readpot:
     high potin
     pause 1
     RCTIME POTIN, 1, adc
     adc=adc/2
     adc=adc.nib2
     return

alert:
     IF emailsent = 1 THEN displayvalues
     SEROUT 16,84+$4000,1,["EM=info@ckuehnel.ch;"]
     SEROUT 16,84+$4000,1,["BASIC Stamp indicates ;"]
     SEROUT 16,84+$4000,1,["an alert situation!\n;"]
     SEROUT 16,84+$4000,1,["Please check!;"]
     SEROUT 16,84+$4000,1,["EM;"]
     SEROUT 16,84+$4000,1,["M1=BASIC Stamp just sent an email alert!;"]

     emailsent = 1             ' EMail was sent
     goto displayvalues
```

Listing 23 Monitoring System (MONITOR.BS2)

At the first activity the BS2 sends a variable without content “M1= ;“ to the MondoMini Webserver. M1 is a message and will be filled with some text after exceeding the limit. After that the BS2 sends the value of the potentiometer and the limit to the MondoMini Webserver before querying for commands.

If no command is received by the MondoMini Webserver, a new pass through the loop begins after one second and the value on the Webserver is utilized.

If a command is received by MondoMini Webserver then it will be read and analyzed. MondoMini expects a command in the form P2=xxx with xxx describing the new limit value. The number string must be converted into a text string before we can update the variable and repeat the process again.

The next figures show the web site in the web browser. Figure 65 shows the output if the measured value is below the limit. However, Figure 66 shows the value exceeding the limit.

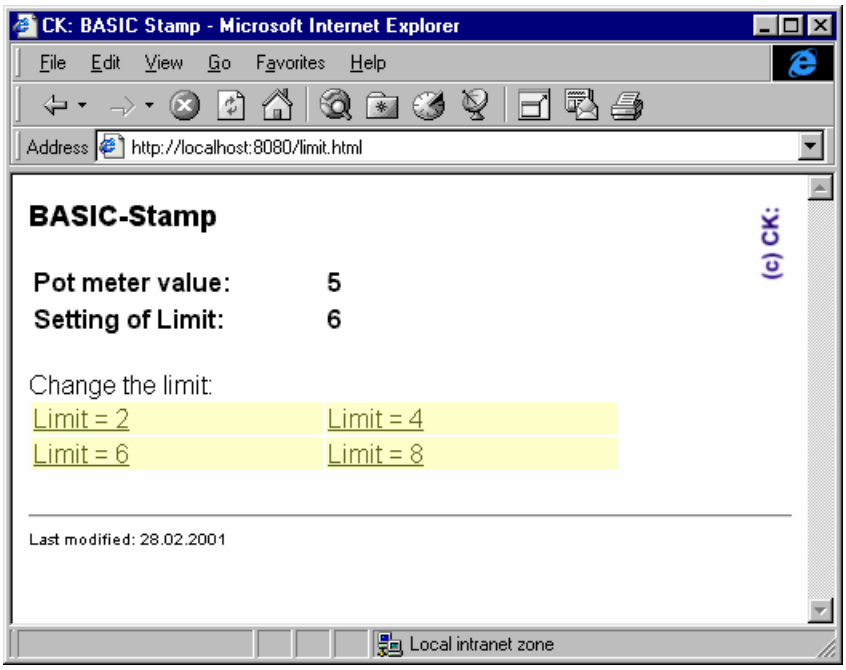


Figure 65 Measuring value below the limit

Before the e-Mail is sent the M1 variable receives the text as shown in Figure 66 and the web browser can display this message. The e-mail corresponds to the representation shown in Figure 67.

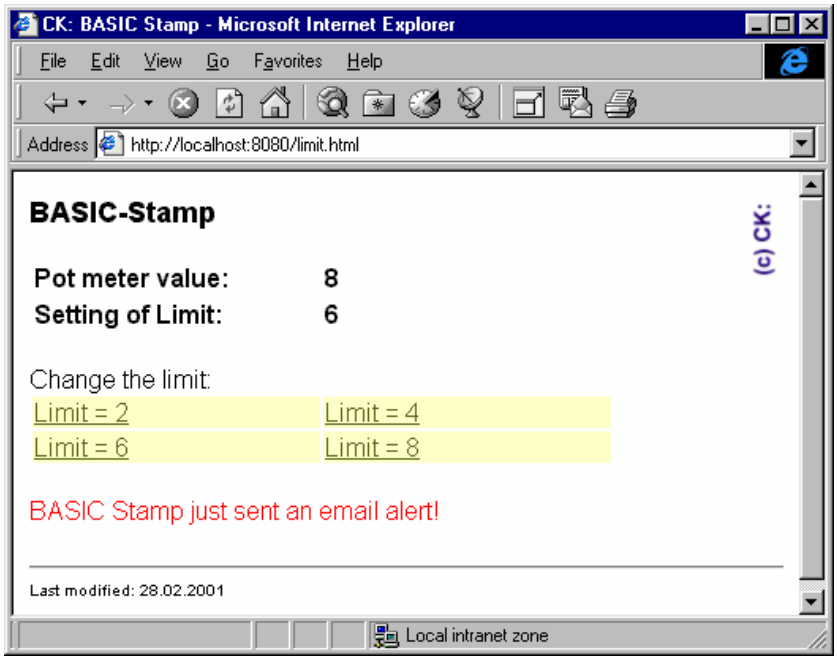


Figure 66 Measuring value above the limit

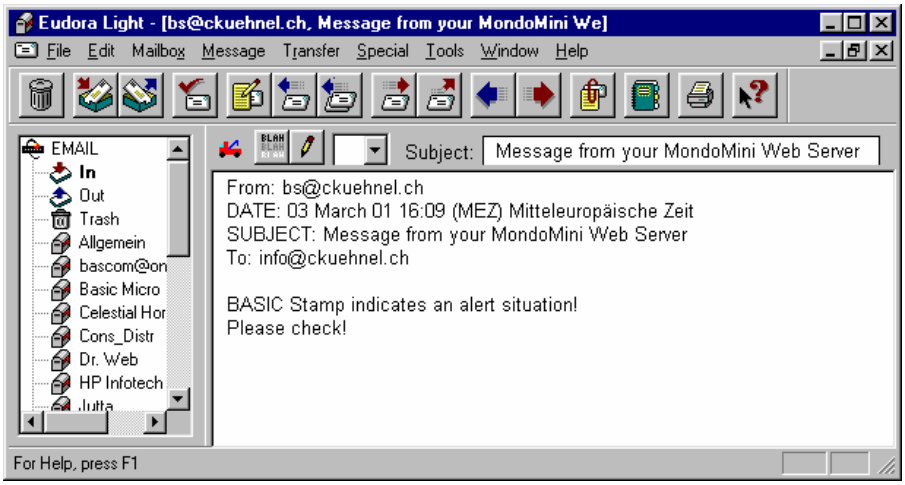


Figure 67 Sent E-Mail after limit is exceeded

The HTML text can be briefly explained (Listing 24). Locations where code was changed are marked bold. At the beginning we place a META-tag to re-load the page LIMIT.HTML every 5 seconds.

With a periodic refresh the web browser can display the actual data saved on the webserver. If the web browser does not support automatic refreshing then you have to reload the page manually.

```
<HTML>
<HEAD>
   <TITLE>CK: BASIC Stamp</TITLE>
<META HTTP-EQUIV="refresh" CONTENT="5; URL=limit.html">
</HEAD>
<BODY BGCOLOR="#FFFFFF">
<H3><FONT SIZE="+3" FACE="Arial"><B><IMG SRC="ck.gif" WIDTH=20 HEIGHT=55
ALIGN=right></B></FONT><FONT FACE="Arial">BASIC-Stamp</FONT></H3>

<P><FONT FACE="Arial"><TABLE BORDER=0>
   <TR>
      <TD WIDTH=180>
         <P><FONT FACE="Arial"><B>Pot meter value:</B></FONT></P>
      </TD>
      <TD>
         <P><B>'P1</B></P>
      </TD>
   </TR>
   <TR>
      <TD WIDTH=180>
         <P><B>Setting of Limit:</B></P>
      </TD>
      <TD>
         <P><B>'P2</B></P>
      </TD>
   </TR>
</TABLE>
 </FONT></P>

<P><FONT FACE="Arial">Change the limit:<B><BR>
</B><TABLE BORDER=0 BGCOLOR="#FFFFFF">
   <TR BGCOLOR="#FFFFCC">
      <TD WIDTH=180>
         <P><A HREF="$P2=2;$limit.html">Limit = 2</A></P>
      </TD>
      <TD WIDTH=180>
         <P><A HREF="$P2=4;$limit.html">Limit = 4</A></P>
      </TD>
   </TR>
   <TR BGCOLOR="#FFFFCC">
```

```
      <TD WIDTH=180>
         <P><A HREF="$P2=6;$limit.html">Limit = 6</A></P>
      </TD>
      <TD WIDTH=180 BGCOLOR="#FFFFCC">
         <P><A HREF="$P2=8;$limit.html">Limit = 8</A></P>
      </TD>
   </TR>
</TABLE>
 </FONT></P>

<P><FONT FACE="Arial" COLOR="#FF0000">'M1</FONT></P>

<P>

<HR>

<FONT SIZE="-2" FACE="Arial">Last modified: 28.02.2001</FONT></P>
</BODY>
</HTML>
```

Listing 24 User Program (LIMIT.HTML)

5 Using a Modem

In the past few decades the classic text-orientated approach to transmit messages underwent astounding enhancements. Now that we have a worldwide network with the internet, the transmission of all kinds of messages like text, pictures, music as well as data for control and measuring of remote devices is as easy as can be.

The modem is the connection link between the an analog phone line and a data processing device using a RS-232 connection.

Most PC users are connected to the internet with an analog modem. As time process more connections are made with direct digital transmitting using ISDN instead of a modem connection.

As the trend continues more modems are tossed aside in favor of digital connections.

It is useful and inexpensive to continue using these modems for transmission of controls with the BASIC Stamp. Modems are suitable for long distance communication with phone-line connections and analog lines are available in many places. Even modems with a baudrate of 2400 bps are useful for microcontroller-based connections.

5.1 Basic Functions of a Modem

The modem achieves the following for it's user:

As a standard device to connect with the lines of phone company in a correct manner.

Connect and disconnect for a specific transmission.

Synchronization of baud rates

Evaluation of data stream with security codes for correctness

Temporary storage of data as a buffer between data stream on-line and user behavior (not all modems)

Data compression

Modems are also in kind of integrated circuits, supplemented with an assortment of external components like crystals, resistors and so on for using in measuring and control. One of these is the CH1786 Modem from Cermetec [www.cermetec.com], offered from PARALLAX as an application kit and in an "AppMod format".

5.2 Remote Alarm via Modem

In the following program example the BS2 sends an alarm message using an external modem. A baudrate of 9600 bps is used.

For receiving the alarm message a PC is connected to a modem, ready to receive messages. The evaluation of what to do with the alarms isn't discussed here. For demonstration using a common terminal program like Hyperterminal or RS232MON is perfectly suitable.

In our program sample a short string is generated from BS2 to send between the modems.

The alarm event is given by interruption in an alarm line, opening one or more of the switches.

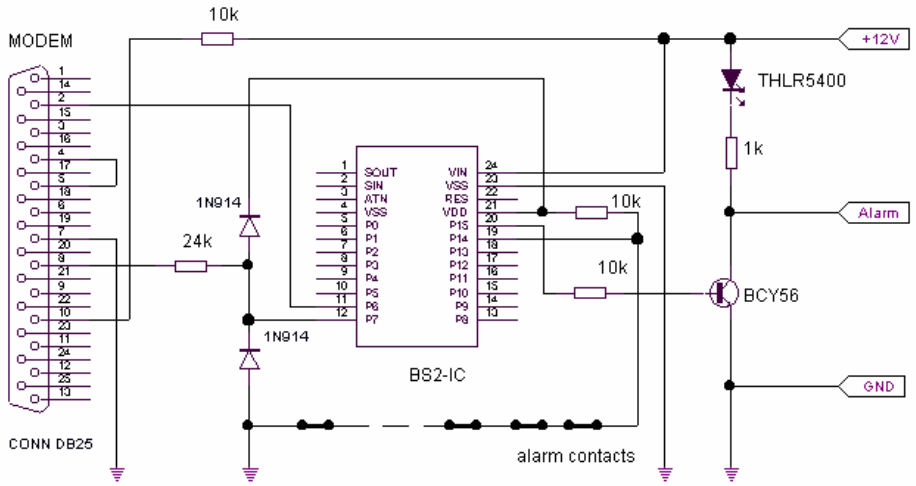


Figure 68 Alarm Circuitry

On the left site we see the DSUB25 connector for modem connection. Pin 6 of the BS2 sends the data to the modem. A successfully tuned connection between modems has the Data Carrier Detect (DCT) signal used from Modem output DCD (pin 8 of DSUB25).

This requires a special initializing string: there must be a " &C1". This means that "DCD follows the Carrier". It is a default part of a standard string.

Once the connection is started, the DCD signals change from –10V (no connection) to +10 Volts (stable connection). This DCD line is connected to I/O pin 7 of the BS2 with a protection resistor and protection diodes.

With the SEROUT command the level of the DCD line is tested because an output of an alarm string is sensed only if the modems are successfully connected. In this case both modems are connected with the carrier-frequency.

If the connection is not stable after a programmed time (named delay) the directive for the modem alarm is lost and a local alarm is started.

Listing 25 shows the program ALARM.BS2

```
' -----[ Title ]-----------------------------------------------
'
' File......alarm.BS2
' Purpose...alarm-message by phone with BS2 and modem
' Author....Klaus Zahnert
' Started...95-12-10
' Updated...01-09-20

' -----[ Program Description ]--------------------------------
' alarm or any other important event is marked with one input-pin
' going  high. In this case modem is initialized, makes connection with
' dial and gives alarm to remote station. The connection between sta-
' tions is tested by DCD-signal. Alternative alarm is giving for
' connection is  failured  after dial.

' -----[ Revision Hisory ]-------------------------------------
'
'
' -----[ Constants ]-------------------------------------------
'
txdpin    con 6         'direct connect. (without line-drv.)
dcdpin    con 7         'connected with modem/DCD-output
sirenpin  con 15        'output to siren/LED
alarm     con 4711      'typical number of alarm-message
bdmd      con 84+$4000  '9600 Baud,8n1, inverted mode
delay     con 1000      'wait for branch

' -----[ Variables ]-------------------------------------------
'
contact   var   IN14    'pullup to Vcc with chain of closed
                        'contacts to GND. Going high with
                        'opening one ore more contacts

' -----[ Initialization ]--------------------------------------
```

```
'
     DIRS = $8000                       'output sirenpin
     low sirenpin                       'no siren

' -----[ Main Code ]----------------------------------------
'
start:
     if contact = 1 then message        'alarm if any switch off
     goto start

message:
     serout txdpin,bdmd, ["ATZ",10,13]  'modem standard init
     pause 2000
     'dial remote station (change the number)
     serout txdpin,bdmd, ["ATDT 1234567",10,13]
     pause 2000

     'output alarm-message
     serout txdpin\dcdpin,bdmd,delay,siren,["ALARM = ",dec alarm,10,13]
     pause 3000

     serout txdpin,bdmd,10,["+++"]      'return to command-mode
     pause 3000

     serout txdpin,bdmd,["ATH",10,13]   'hang up

     goto start

siren:
     high 15       'siren/LED on
     goto start    'try again alarm-message if contact = high

end
```

Listing 25 Alarm transmitted by a Modem (ALARM.BS2)

The message loop is for new or repeated calling for continuation of the alarm. A local alarm is for manual resetting only (I/O pin 15 goes low). This is possible with a new start of the program or with the reset-key.

Note that the RS232 signals have a level of about +/-10 Volt. The signals on the DSUB25 connector are a negative voltage in the in-active state, aligned with the TTL level Hi.

The hardware exchange from TTL to RS232 is made by changing the voltage and inverting the signals. The RS232 signals are active-low, the corresponding TTL signals are active-high and the aligned RS232 voltage is about -10 Volt.

A commonly utilized integrated circuit between TTL and RS232 (such as the MAX232) is not required for the BASIC Stamp because the SERIN and SEROUT commands are

programmable for a “inner inversion”. The input signal of the BASIC Stamp is determined using a 24 kΩ resistor to limit the current.

Although input ports of the BASIC Stamps have inner clamping diodes to protect against dangerous voltages, additional diodes for limiting the voltage were added to I/O pin 7.

The output of the BASIC Stamp to the modem is TTL and the RS232 switching threshold requires only some positive voltage.

In a direct connection, the signal-polarity inversion is solved by the BASIC Stamp, programmed in its software in the SEROUT command.

In addition the modem needs two initializations with hardware connections. With this the BASIC Stamp is ready for data input and output without using the handshake signals.

This connections are:

Connection from CTS (modem output “Clear To Send”) to RTS (modem input “Ready to send”), shown with pin 5 and pin 4 of the DSUB25 connector.

With a connection from DTR (modem input “Data Terminal Ready”) to the positive power supply voltage the modem gets the message that a device is connected, ready for data-transfer (DSUB25 pin 20 with resistor 10 kΩ to + 12 Volt)

The alarm input of the BS2 is I/O pin 14. A Pull-Up resistor of 10 kΩ provides high level on this pin if one of the alarm-contacts is opened. The alarm contacts work in chain from this pin to GND. These alarm contacts can be mounted on different devices that should be protected with them. The principle of closed contacts in connection with current in the alarm line in case of no alarm is a reasonable solution to identify a cut alarm line.

If the dialing process is not successful, the local alarm starts by switching I/O pin15 to high.

An LED is used for simulating the local alarm.

Some additional discussion about communicating with the modem would be helpful.

Compared to other transmission devices on the end of phone lines (phone, fax, modem) signals are interchanged in followed order:

Starting the connection (dialing a remote station with a phone number)

Verifying/confirmation of the connection

Transmission of messages and/or data

Disconnecting from remote station

For the modem control signals to be transmitted they must be in agreement with the remote station about baud rates and format.

Shown here are examples for baudrate and format of the bytes. There is more full-scale information to develop a more specialized transmission protocol. These controls are mostly used by the modem. Nearly all these controls are ASCII-strings, marked with “AT” on the beginning. These “Command Mode” signals are from the standard Hayes command set.

We use the following commands in the program ALARM.BS2:

ATZ	Modem standard initialization. State is based from initializing string, which is programmed and stored before.
ATD <dial-number>	Tone dialing the remote station
+++	Escape Code (modem switches from transparent mode back to command mode)
ATH	Disconnects the modem (similar to hang up the phone)

If all settings used for the modem operation are preset and if the dialing was successful then the modem switches from “command mode” to “transparent mode”.

This means that all following data will transmit through the modem without any obstacles. So it is “transparent” to these signals. The transmitted data is transformed to different tones sent to the phone line and which are re-transformed to digital signals by the receiving modem.

These are only the basics to understand the example. RS232 is a very old serial transmission protocol, but having some understanding of it helps comprehend the example.

6 Additional Applications

6.1 Switching High Currents and Voltages

The electrical features of the BASIC Stamp's I/O pins limit the direct control of higher currents and/or voltages.

The BS2p's SX28AC/48AC can source/drive a current of 30 mA. The PIC16C57 of the BS2 can source 25 mA but only drive 20 mA.

The voltage on an I/O pin may not exceed the supply voltage.

If you want to switch higher currents and/or voltages you have to look for other solutions.

Beside relays and transistors we can control HEXFETs directly from the BASIC Stamp. Relays and transistors must be driven from the I/O pin. To drive HEXFETs from an I/O pin it needs almost no power. Data from two different HEXFETs are listed in the following table as an example:

	IRL7601	*IRL2203*
Max. Drain-Source-Voltage	20 V	30 V
Max. Draincurrent	4,6 A	82 A
Min. Gate-Source-Threshold-Voltage	0,7 V	1 V

Figure 69 shows driver using n-channel MOSFET (HEXFET), NPN-bipolar transistors and relays. We use a protection diode to prevent damage to the BASIC Stamp. If there is no inductivity we need no protection diode.

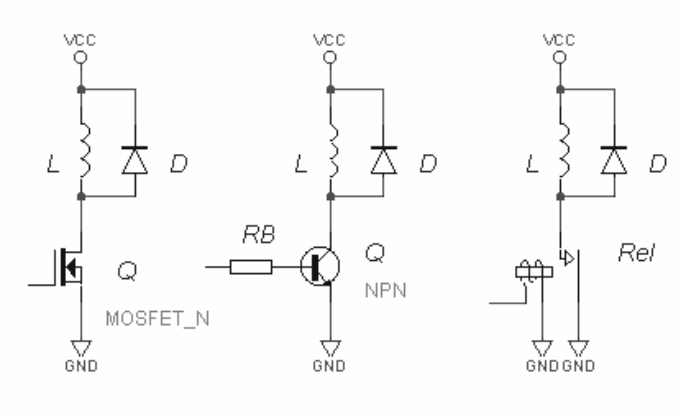


Figure 69 Driver

Pay attention to the following:

n-channel MOSFET

The gate-source-threshold-voltage of the n-channel MOSFET must be between the low and high level on an I/O pin of the BASIC Stamp.

Do not exceed the maximum drain current.

The supply voltage Vcc may not exceed the maximum allowed drain-source voltage.

npn-bipolar transistor

The current amplification factor of the transistor must be high enough so that the I/O pin can drive the required current.

The purpose of the resistor RB is to limit the current from switching the transistor, but limiting the current to avoid exceeding the maximum allowed basis current.

Do not exceed the maximum collector current.

The supply voltage Vcc may not exceed the maximum allowed collector-emitter voltage.

Relays

The I/O pin can drive the current switching on the relays.

The contact of the relays must be suited for the current of the voltage to be switched.

The supply voltage of the relays can differ from the supply voltage of the driver.

Pay attention to the isolation between the driver and the load circuit. Relays serve as potential separator but the required isolation resistance should be ignored.

Consider these guidelines for further combinations you might want to arrange. Figure 70 shows some enhanced drivers.

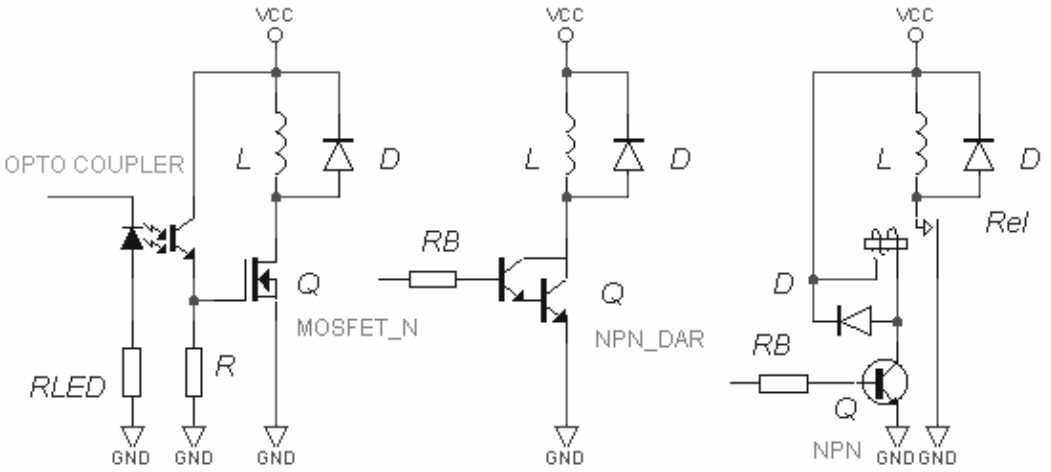


Figure 70 Enhanced Driver Circuits

The n-channel MOSFET was supplemented with an optocoupler. The load circuit is separated from the microcontroller this way.

To drive higher currents with a bipolar transistor and to avoid an overload on the I/O pin we can replace the simple transistor with a Darlington type.

If the relays need more current than the I/O pin can drive or source a transistor can help.

6.2 Networking of BASIC Stamps using RS-232 and RS-485

6.2.1 Point-to-Point Connection

If the BASIC Stamp can communicate with a PC via RS-232 then it should be possible to exchange data between several BASIC Stamps. The required connections are very simple. Figure 71 shows two serial connected BS2.

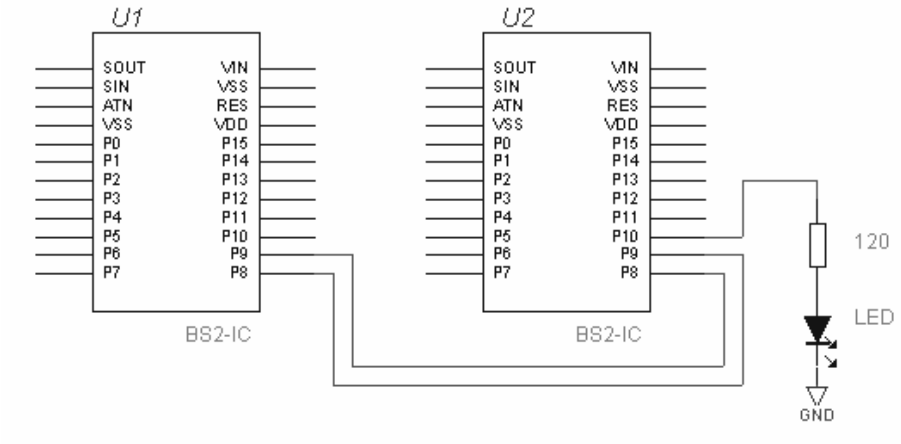


Figure 71 Serial Communication between two BS2

In both BS2s I/O pin 8 is setup as a serial input RxD and I/O pin 9 as serial output TxD. The BS2 marked U1 serves as the master, while that BS2 marked as U2 is the slave. The LED connected to I/O pin 10 signalizes the activity of the slave.

A simple example explains the communication between the two BASIC Stamps.

In our example the master can send only two commands (“1” or “0”) to the slave. The slave interprets these commands and set I/O pin 10 after receiving a “1”. If the received command was “0” it resets this I/O pin. The LED connected signalizes the state immediately.

The slave sends a return value back to the master to get an acknowledge of the action. After receiving a “not interpretable” command the slave sends $FF as return value back.

Listing 26 shows the communication program of the master, while Listing 27 shows those of the slave.

```
' -----[ Title ]-----------------------------------------------
'
' File......  COMM_M.BAS
' Purpose...  Communication between two BASIC Stamps
' Author....  Claus Kühnel
' Started...  30.09.94
' Updated...  24.09.01
'
' -----[ Program Description ]-------------------------------
'
' Two BASIC Stamps are connected over RS232.
' The master stamp sends a command to switch an output of the slave.
' After switching the slave reads this output an gives the state
' back to the master.
'
' -----[ Revision Hisory ]------------------------------------
'
' 30.09.94: Version 1.0 for BS1
' 24.09.01: Version 2.0 for BS2
'
' -----[ Directives ]-----------------------------------------
'
'{$STAMP BS2p}                  'specifies a BS2p
'
' -----[ Constants ]------------------------------------------
'
RxD       con 8
TxD       con 9

baud      con 396               'T2400 for BS2
'
' -----[ Variables ]------------------------------------------
'
retval    var byte  'return byte from the slave
command   var byte  'command byte for the slave
'
' -----[ Initialization ]-------------------------------------
'
' -----[ Main Code ]------------------------------------------
'
          pause 100
start:    command = "1"         'send command byte
          gosub send
          command = "0"         'send another command byte
          gosub send
          goto start            'repeat endless
'
' -----[ Subroutines ]----------------------------------------
'
send:     'send the command to the slave
          serout TxD, baud, [command]
          serin  RxD, baud, [retval]    'look for return byte
```

```
         'debug  ? retval
         pause 1000                    'wait a little bit
         return
```

Listing 26 Serial Communication by the Master (COMM_M.BS2)

```
' -----[ Title ]-----------------------------------------------
'
' File......  COMM_S.BAS
' Purpose...  Communication between two BASIC Stamps
' Author....  Claus Kühnel
' Started...  30.09.94
' Updated...  24.09.01
'
' -----[ Program Description]---------------------------------
'
' Two BASIC Stamps are connected over RS232.
' A master stamp sends a command to switch an output of the slave.
' After switching the slave reads this output an gives the state
' back to the master.
'
' -----[ Revision Hisory ]-------------------------------------
'
' 30.09.94: Version 1.0 for BS1
' 24.09.01: Version 2.0 for BS2
'
' -----[ Directives ]------------------------------------------
'
'{$STAMP BS2p}                         'specifies a BS2p
'
' -----[ Constants ]-------------------------------------------
'
RxD  con 8
TxD  con 9
OUT  con 10

Baud con 396                           'T2400 for BS2
'
' -----[ Variables ]-------------------------------------------
'
retval   var byte  'return byte to the slave
command var byte   'command byte from the slave
'
' -----[ Initialization ]--------------------------------------
'
' -----[ Main Code ]-------------------------------------------
'
start:   'look for a command from the master
         serin RxD, baud, [command]
         if command = "1" then PinHi   'set pin Hi
         if command = "0" then PinLo   'set pin Lo
```

```
         goto error                    'command invalid

PinLo:   low OUT                       'set pin Lo
         retval = in10                 'read pin
         goto send                     'send state back

PinHi:   high OUT                      'set pin Hi
         retval = in10                 'read pin
         goto send                     'send state back

error:   retval = $FF                  'write errorcode to return byte

send:    serout TxD, baud, [retval]
         goto start                    'repeat endless
'
' -----[ Subroutines ]------------------------------------------
'
```

Listing 27 Serial Communication by the Slave (COMM_S.BS2)

You can see from the listings that the serial communication of the BASIC Stamps works half-duplex in principle – it will be sent or received.

If there are two BASIC Stamps connected via two lines then only one is active at a time. That suggests the desire to save the inactive serial line.

The following program example shows how that it is possible. Only slight changes to the two listings are necessary:

Changing RxD con 8 to RTxD con 8

Comment the line `TxD con 9` (I/O pin 9 is not used for communication now)

Changing `RxD` and `TxD` to `RTxD` in all instructions

Adding a Pull-Up resistor from I/O pin 8 to VCC

Here are some hints for a reliable data exchange. Figure 72 shows the timing of the one-line serial connection.

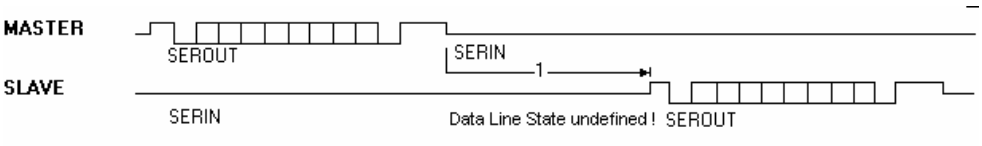


Figure 72 Timing of the one-line serial connection

Immediately after sending the command byte the master switches to receive mode to avoid loosing the byte being sent back from the slave.

The slave sends it's returned value only after `high out` and `low out` are finished. This time is shown in Figure 72 marked with (1) when the I/O pins of both BS2s are switched to input and the data line has an undefined state – it floats. Each disturbance on this data line could be interpreted as start bit for a new data transmission and the synchronization of the both BS2 is lost - the system hangs.

The problem described is solved for the inverting mode of the serial interface by the pull-up resistance between data line and Vcc. During the time in which the I/O pins of both BS2s are switched to input, the pull-up resistors produce a defined high state on the data line.

6.2.2 BASIC Stamp Network

All BASIC Stamps can be networked.

The BASIC Stamp network described as next consists of one master and two slaves. Figure 73 shows our small network consisting of three BS2.

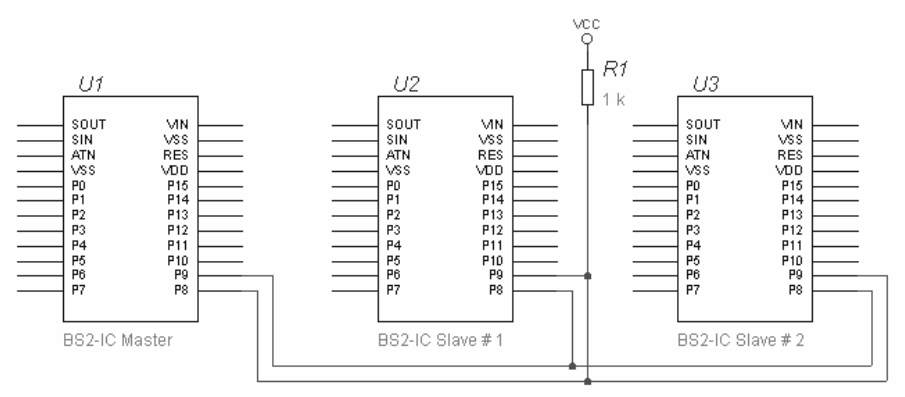


Figure 73 BASIC Stamp Network

I/O pin 8 serves as the receive line (RxD), while I/O pin 9 is transmit line (TxD) for all BASIC Stamps in the network.

The serial inputs of the slaves are driven by the serial output of the master without any problems. The serial input of the master is driven by the two serial outputs of the slaves.

In order to prevent electrical problems, these serial outputs must be operated as open-drain outputs with pull-up resistors. That represents an OR operation (wired-OR) of the serial outputs of both slaves. Our next program example proceeds from the following conditions:

The master sends addressed commands to the slaves.

The addressed slave sends a command. The other slave ignores this command.

After execution of the command the slave sends an answer to the master.

The answer can be evaluated only by the master.

Listing 28 shows the source of the program for our BS2 network master. The addresses “A_Node” and “B_Node” were assigned to the two attached Slaves. A command byte activates an assigned function into the slaves. In our example the commands “0” and “1” are permitted exclusively. After sending a command to a slave the master waits for a response from the addressed slave. A timeout of one second prevents endless waiting if the slaves

send no answer. The end of the program loop is marked by the character “>” in the Debug Window.

```
' -----[ Title ]-----------------------------------------------
'
' File......  MASTER.BS2
' Purpose...  Demonstration of a BASIC Stamp network
' Author....  Claus Kühnel
' Started...   8.10.94
' Updated...  24.09.01
'
' -----[ Program Description ]-------------------------------
'
' In this demonstration three BASIC Stamps build a network.
' The master sends commands to the two slaves named A_Node and
' B_Node. After execution the required function the slave sends a
' return value back to the master. This return value is displayed
' by DEBUG in this demonstration.
'
' -----[ Revision Hisory ]----------------------------------
'
'  8.10.94: Version 1.0 for BS1
' 24.09.01: Version 2.0 for BS2
'
' -----[ Directives ]---------------------------------------
'
'{$STAMP BS2}               'specifies a BS2
'
' -----[ Constants ]----------------------------------------
'
RxD       con 8
TxD       con 9
LED       con 10

baud      con 396            'T2400
tout      con 1000           'Timeout 1 sec
'
' -----[ Variables ]----------------------------------------
'
retval    var byte
command   var byte
'
' -----[ Initialization ]-----------------------------------
'
' -----[ Main Code ]----------------------------------------
'
start:
     command = "1"
    serout TxD,baud, ["A_Node",command] 'send "1" to A_Node
    serin RxD, baud, tout,tlabel, [retval] 'wait for ret value
    debug ? retval
```

```
    pause 500
    serout TxD, baud, ["B_Node",command]'send "1" to B_Node
    serin RxD, baud, tout,tlabel, [retval] 'wait for ret value
    debug ? retval
    command = "0"
    serout TxD, baud, ["A_Node",command]'send "0" to A_Node
     serin RxD, baud, tout,tlabel, [retval]     'wait for ret value
    debug ? retval
    serout TxD, baud, ["B_Node",command]       'send "0" to B_Node
    serin RxD, baud, tout,tlabel, [retval]     'wait for ret value
    debug ? retval
    debug ">"
    goto start                                 'repeat endless

tlabel:
     debug "Timeout - no answer from any slave!", CR
     goto start
```

Listing 28 BS2 Master (MASTER.BS2)

In the next two programs all the slaves have to do is to set or reset an I/O pin based on the command they receive. The LED connected to I/O pin 10 signalizes the state.

Both slave programs (Listing 29 and Listing 30) are unique. Serial input and serial output work with different baud modes. The serial outputs operate as Open-Drain (OT2400) to allow the wired-OR of both outputs with the external pull-up resistor.

```
' -----[ Title ]-----------------------------------------------
'
' File......  SLAVE1.BS2
' Purpose...  Communication in BASIC Stamp Network - Slave #1
' Author....  Claus Kühnel
' Started...   8.10.94
' Updated...  24.09.01
'
' -----[ Program Description ]--------------------------------
'
' In this demonstration three BASIC Stamps build a network.
' The master sends commands to the two slaves named A_Node and
' B_Node. After execution the required function the slave sends a
' return value back to the master.
'
' -----[ Revision Hisory ]-------------------------------------
'
'  8.10.94: Version 1.0 for BS1
' 24.09.01: Version 2.0 for BS2
'
```

```
' -----[ Directives ]------------------------------------------
'
'{$STAMP BS2}      'specifies a BS2
'
' -----[ Constants ]-------------------------------------------
'
RxD  con 8
TxD  con 9
LED  con 10

baudin   con 396            'T2400
baudout con 396+$8000       'OT2400
'
' -----[ Variables ]-------------------------------------------
'
command var byte
'
' -----[ Initialization ]--------------------------------------
'
     low LED
'
' -----[ Main Code ]-------------------------------------------
'
start:
     serin RxD, baudin, [wait("A_Node"), command]
     if command = "1" then LEDON
     if command = "0" then LEDOFF
     goto start

LEDON:
     high LED
     serout TxD, baudout, [command]
     goto start

LEDOFF:
     low LED
     serout TxD, baudout, [command]
     goto start
```

Listing 29 BS2 Slave # 1 (SLAVE1.BS2)

```
' -----[ Title ]-----------------------------------------------
'
' File......  SLAVE2.BS2
' Purpose...  Communication in BASIC Stamp Network - Slave #2
' Author....  Claus Kühnel
' Started...   8.10.94
' Updated...  24.09.01
'
' -----[ Program Description ]---------------------------------
'
' In this demonstration three BASIC Stamps build a network.
' The master sends commands to the two slaves named A_Node and
' B_Node. After execution the required function the slave sends a
' return value back to the master.
'
' -----[ Revision Hisory ]--------------------------------
'
'  8.10.94: Version 1.0 for BS1
' 24.09.01: Version 2.0 for BS2
'
' -----[ Directives ]-------------------------------------------
'
'{$STAMP BS2}               'specifies a BS2
'
' -----[ Constants ]-------------------------------------------
'
RxD  con 8
TxD  con 9
LED  con 10

baudin   con 396            'T2400
baudout  con 396+$8000      'OT2400
'
' -----[ Variables ]-------------------------------------------
'
command var byte
'
' -----[ Initialization ]--------------------------------------
'
    low LED
'
' -----[ Main Code ]-------------------------------------------
'
start:
    serin RxD, baudin, [wait("B_Node"), command]
    if command = "1" then LEDON
    if command = "0" then LEDOFF
    goto start

LEDON:
```

```
     high LED
     serout TxD, baudout, [command]
     goto start

LEDOFF:low LED
     serout TxD, baudout, [command]
     goto start
```

Listing 30 BS2 Slave # 2 (SLAVE2.BS2)

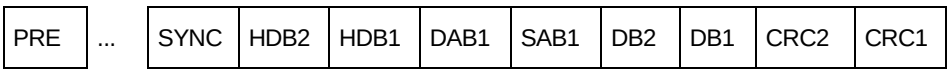
6.2.3 Scalable Node Address Protocol S.N.A.P.

High Tech Horizon from Sweden [www.hth.com] developed S.N.A.P. (Scalable Node Address Protocol), a communication protocol for their Powerline Modems PLM-24. The goal was to build a protocol that could be implemented in smaller microcontrollers without a large overhead.

The High Tech Horizon web site offers much information about S.N.A.P. and the PLM-24 Powerline Modems. Here we are limited to the necessary setup for letting BASIC Stamps communicate over a network.

Data exchange between network nodes takes place in the form of packages. These data packages may have a different length (number of bytes). The length of a data package depends of the number of address and data bytes, the error detection method and some special bytes.

The following example shows a S.N.A.P. data package with 1-Byte addresses, 2-Byte data and a CRC-16 error detection. These data packages are used in our program example later.



Each data package begins with preamble bytes before the synchronization byte. The number of preamble bytes is not important. The header definition bytes (HDBx) define the structure of the data package. Table 14 explains the meaning of the bytes in the data package.

Byte	*Name*
PRE	Preamble Byte
SYNC	Synchronization Byte
HDB2	Header Definition Byte 2
HDB1	Header Definition Byte 1
DAB1	Destination Address Byte
SAB1	Source Address Byte
DB2	Data Byte 2
DB1	Data Byte 1
CRC2	High byte of CRC-16
CRC1	Low byte of CRC-16

Table 14 S.N.A.P. Data Package

The data package described in Table 14 has a length of eight bytes (without preamble and synchronization bytes). The bytes have their LSB in the right most position (Bit7...Bit0).

Synchronization Byte - SYNC

The byte **SYNC** is predefined and marks the begin of each data package.

Bit	7	6	5	4	3	2	1	0	HEX	DEC
	0	1	0	1	0	1	0	0	54	84

Header Definition Bytes (HDB2 and HDB1)

Both the header definition bytes **HDB2** and **HDB1** define the structure of the data package. The definition here is used in our program example later.

Bit	7	6	5	4	3	2	1	0	HEX	DEC
HDB2	DAB		SAB		PFB		ACK			
	0	1	0	1	0	0	0	1	51	81
HDB1	C	EDM			NDB					
	0	1	0	0	0	0	1	0	42	66

Table 15 describes the meaning of the bits in HDB2 and HDB1.

Bits	*Meaning*	*Code*	*Definition*
DAB	Number of Destination Address Bytes	01	1 Byte
SAB	Number of Source Address Bytes	01	1 Byte
PFB	Number of Protocol specific Flag Bytes	00	no
ACK	ACK/NAK Bits	01	yes
C	Command Mode Bit	0	no
EDM	Error Detection Method	100	CRC-16
NDB	Number of Date Bytes	0010	2 Byte

Table 15 Meaning of the Bits in HDB2 and HDB1

If the transmitter sets ACK = 01 then it expects ACK (10) or NAK (11) in response to the receiver. This acknowledge mechanism is implemented in the program example `SNAP-IO.BS2`.

The command mode is a feature in bigger networks and is not used here.

In our program example we used the BS2 as a slave. This means the BS2 waits for a command executes it and send a response back to the master.

To detect errors during data exchange we use a CRC-16 algorithm for error detection. In the response to the receiver ACK/NAK tells the transmitter the state of the data exchange.

In case of an data transmission error the package can be sent again, for example.

Listing 31 shows the source of an BS2 Slave working according to S.N.A.P.

```
' -----[ Title ]-----------------------------------------------
'
' File......: SNAP-IO.BS2
' Purpose...: Turns LEDs on and off
' Author....: Chriser Johansson
' Version...: 1.01
' Stamp.....: BS2-IC
' Started...: 98-05-03
' Updated...: 98-09-18
' Modified..: 99-06-13 by Claus Kuhnel
'
' -----[ Program Description ]-------------------------------
'
' This program shows how to implement the S.N.A.P protocol in a BS2-IC
' and is an simple example to turn LEDs ON or OFF.
' This example uses 16-bit CRC-CCITT as error detection method which
' gives secure data transfer.
'
' If the node is addressed by another node (PC or another MCU) then
' the lower 4 bit of DB1 value are displayed by OUTC (Pin11-8) for
' BASIC Stamp Activity Board.
'
' The packet structure is defined in the received packets first two
' bytes (HDB2 and HDB1). The following packet structure is used.
'
' DD=01     - 1 Byte destination address
' SS=01     - 1 Byte source address
' PP=00     - No protocol specific flags
' AA=01     - Acknowledge is required
' D=0       - No Command Mode
' EEE=100   - 16-bit CRC-CCITT
' NNNN=0010 - 2 Byte data
'
' Overview of header definition bytes (HDB2 and HDB1)
'
'         HDB2             HDB1
' +-----------------+-----------------+
' | D D S S P P A A | D E E E N N N N |
' +-----------------+-----------------+
'
'
' -----[ Constants ]-----------------------------------------
'
TxD           con 16        ' Serial output pin
RxD           con 16        ' Serial input pin
LEDS          con %1111     ' LED outputs
Baud          con 80        ' Baudrate (9600 bps 8N1)
Preamble      con %01010101 ' Preamble byte
SYNC          con %01010100 ' Synchronization byte
CRCPOLY       con $1021     ' CRC-CCITT
cHDB2         con $51
cHDB1         con $42
```

```
MyAddress     con 123          ' Address for this node (1-255)

' -----[ Variables ]----------------------------------------
'
CRC           var word         ' CRC Word
LoCRC         var CRC.lowbyte  ' CRC Lo Byte
HiCRC         var CRC.highbyte ' CRC Hi Byte
HDB1          var byte         ' Header Definition Byte 1
HDB2          var byte         ' Header Definition Byte 2
DAB1          var byte         ' What node should have this paket
SAB1          var byte         ' What node sent this packet
DB1           var byte         ' Packet Data Byte 1
DB2           var byte         ' Packet Data Byte 2
CRC2          var byte         ' Packet CRC Hi_Byte
CRC1          var byte         ' Packet CRC Lo_Byte
TmpByte1      var byte         ' Temporary Variable
TmpByte2      var byte         ' Temporary Variable


' -----[ Initialization ]----------------------------------
'
Dirc = LEDS                    ' Set pin 11 to 8 as outputs

DB1 = 0                     ' Clear Data variable
DB2 = 0

'------[ Program ]-----------------------------------------
'
Start:
     ' Wait for SYNC byte, if received read next eight bytes from master
     serin RxD, Baud, [wait(SYNC),HDB2,HDB1,DAB1,SAB1,DB2,DB1,CRC2,CRC1]

     debug cls, "Received data:",cr
     debug dec4 HDB2, dec4 HDB1
     debug dec4 DAB1, dec4 SAB1, cr
     debug dec4 DB2,  dec4 DB1
     debug dec4 CRC2, dec4 CRC1,cr

     ' Packet header check routine
    '
     ' Check HDB2 to see if STAMP II are capable to use the packet
    ' structure, if not goto Start
    if HDB2 <> cHDB2 then Start

     ' Check HDB1 to see if STAMP II are capable to use the packet
     ' structure, if not goto Start
     if HDB1 <> cHDB1 then Start

    ' Address check routine
    '
    ' Check if this is the node addressed, if not goto Start
    if DAB1 <> MyAddress then Start
```

```
    ' Check CRC for all the received bytes
    gosub checkcrc

    ' Check if there was any CRC errors, if so send NAK
    if CRC <> 0 then NAK

    ' No CRC errors in packet so check what to do.
     ' Mask the low nibble of DB1 and switch LEDs
'
' Associated Function (place it between +++ lines)
'
'++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
      outc = db1 & $0F
'++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
'
ACK:
     ' Send ACK (i.e tell master that packet was OK)
    ' Set ACKs bit in HDB2 (xxxxxx10)
    HDB2 = HDB2 | %00000010
     HDB2 = HDB2 & %11111110
    goto send

NAK:
    ' Send NAK (i.e tell master that packet was bad)
    ' Set ACK bits in HDB2 (xxxxxx11)
    HDB2 = HDB2 | %00000011
    goto send

Send:
    ' Swap SAB1 <-> DAB1 address bytes
    TmpByte2 = SAB1
    SAB1 = DAB1
    DAB1 = TmpByte2

    ' Clear CRC variable
    CRC = 0
    ' Put HDB2 in variable TmpByte1
    TmpByte1 = HDB2
    ' Calculate CRC
    gosub calccrc

    ' Put HDB1 in variable TmpByte1
    TmpByte1 = HDB1
    ' Calculate CRC
    gosub calccrc

    ' Put DAB1 in variable TmpByte1
    TmpByte1 = DAB1
    ' Calculate CRC
    gosub calccrc

    ' Put SAB1 in variable TmpByte1
```

```
    TmpByte1 = SAB1
    ' Calculate CRC
    gosub calccrc

     ' Put Data in variable TmpByte1
    TmpByte1 = DB2
    ' Calculate CRC


     Gosub calccrc

    ' Put Data in variable TmpByte1
    TmpByte1 = DB1
    ' Calculate CRC
    Gosub calccrc

    ' Move calculated HiCRC value to outgoing packet
    CRC2 = HiCRC
    ' Move calculated LoCRC value to outgoing packet
    CRC1 = LoCRC

     debug cr,"Sent data:",cr
     debug dec4 HDB2, dec4 HDB1
     debug dec4 DAB1, dec4 SAB1, cr
     debug dec4 DB2,  dec4 DB1
     debug dec4 CRC2, dec4 CRC1,cr

Tx:
     ' Send packet to master, including the preamble and SYNC byte
     serout TxD, baud, [Preamble,SYNC,HDB2,HDB1,DAB1,SAB1,DB2,DB1,CRC2,CRC1]

     ' Give STAMP time to shift out all bits before setting to Rx
     Pause 50

    ' Done, go back to Start and wait for a new packet
    goto start
```

```
' -----[ Subroutines ]------------------------------------------
'
'Subroutine for checking all received bytes in packet
checkcrc:
     CRC = 0
    TmpByte1 = HDB2
    gosub calcCRC
    TmpByte1 = HDB1
    gosub calcCRC
    TmpByte1 = DAB1
    gosub calcCRC
    TmpByte1 = SAB1
    gosub calcCRC
    TmpByte1 = DB2
    gosub calcCRC
     TmpByte1 = DB1
    gosub calcCRC
    TmpByte1 = CRC2
    gosub calcCRC
    TmpByte1 = CRC1
     gosub calcCRC
    return

' Subroutine for calculating CRC value in variable TmpByte1
calccrc:
    CRC = TmpByte1 * 256 ^ CRC
    for TmpByte2 = 0 to 7
     if CRC.Bit15 = 0 then shiftonly
        CRC = CRC * 2 ^ CRCPOLY
        goto nxt
shiftonly:
     CRC = CRC * 2
nxt:
     next
    return

' -----[ End ]----------------------------------------------------
```

Listing 31 S.N.A.P. Slave (SNAP-IO.BS2)

The program shown in Listing 31 waits to receive a **SYNC** byte to read the following eight bytes according to the protocol definition.

Once the network node is addressed (`MyAdress con 123`) all bytes received are checked with the CRC-16 algorithm for correct data exchange.

If the CRC-16 Check detects no error then the Acknowledge Bits in **HDB2** are set to 10 and the associated function is executed by the network node.

This function is marked by comment lines (++++) in the listing. In our case the LEDs on the BASIC Stamp Activity Board display the Low Nibble of the data byte **DB1**.

If the CRC-16 Check detects an error then the Acknowledge Bits in **HDB2** are set to 11 and the associated function is not executed.

At the end the slave sends a data package back to the master for it's analyzing.

For test purposes the program SNAP-IO.BS2 contains several DEBUG commands. They're for a later example and can be commented.

This S.N.A.P. implementation was tested with the BASIC Stamp Editor StampW.

To send two data bytes with the values $AA and $55 to the BS2 we have to send the following data package:

SYNC	HDB2	HDB1	DAD	SAD	DB2	DB1	CRC2	CRC1
84	81	66	123	1	170	85	243	96

Figure 74 and Figure 75 show the outputs in the Debug Window (overlayed by the output of the SEROUT command).

Each byte is represented by a four-digit decimal number because the simplest way for number input is the number block of the PC's keyboard. For example, to input the decimal number 81 you have to key the 0 8 1 with an Alt-Key. Releasing the Alt-Key finishes the number input.

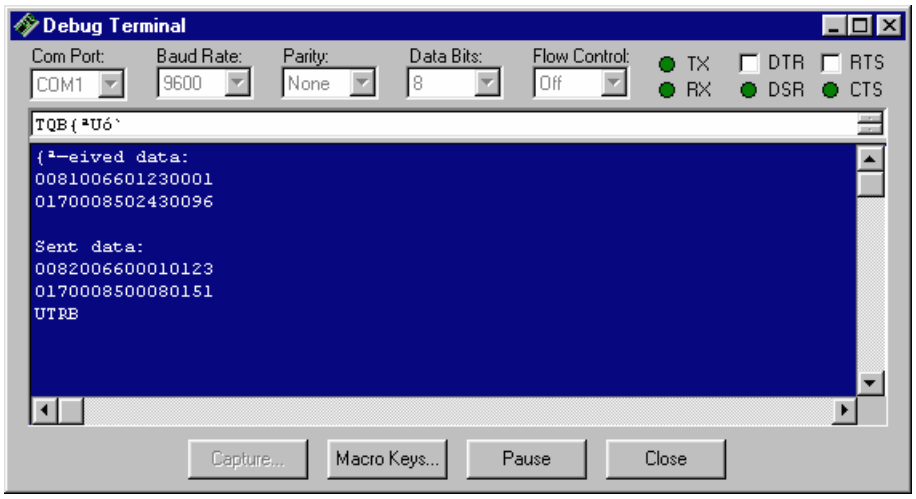


Figure 74 Data Package with correct CRC

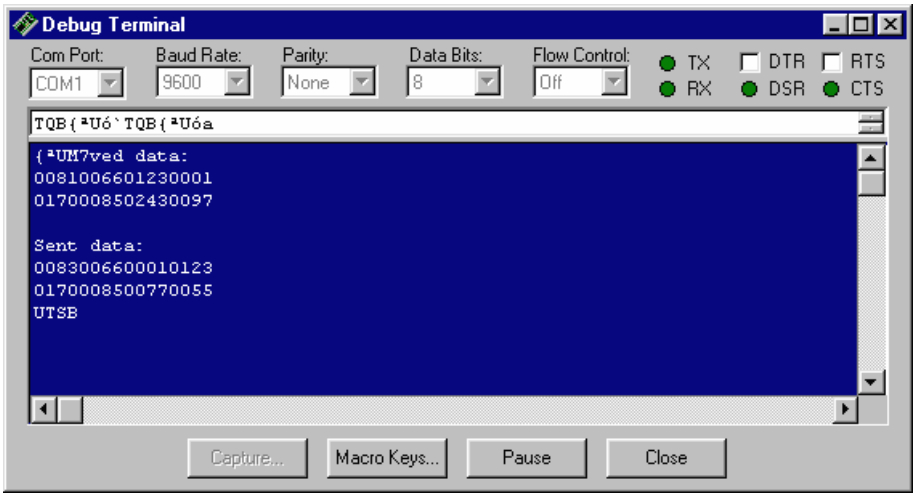


Figure 75 Data Package with Incorrect CRC

Figure 74 shows the error free transmission of a correctly entered data package. The Acknowledge Bits in **HDB2** of the response data package are 10 so the value of **HDB2** is 82 (=$ 52). The data bytes are unchanged here.

To test a faulty data transmission (Figure 75) the byte **CRC1** was incorrectly entered. The correct value is 96 but 97 was entered. The CRC Check detects this error as a transmission error and sets the Acknowledge Bits to 11, so **HDB2** has the value 83 ($ 53) now.

6.2.4 Data Transmission According to RS-422 and RS-485

The data communication between a PC and BASIC Stamp and/or between several BASIC Stamps, regarded so far, took place according to RS-232. However this kind of the data communication is suitable only for short distances.

For a wired data communication over longer distances the electrical conditions of the connection are increasingly important. Twisted two-wire lines with a correct termination can serve as an electrical connection between transmitters and receivers.

RS-422 and RS-485 are symmetrical interfaces for transmission and enhance the traditional possibilities while overcoming the restrictions of the RS-232 interfaces.

RS-422 or RS-485 have the following characteristics:

- Maximum data rate of 10 Mbit/s.
- Maximum length of connection 1200 m.
- Symmetric data transmission to avoid electrical disturbances.

Not all characteristics of these interfaces are relevant for BASIC Stamp applications. But, if data can be transferred several hundred meters over a twisted two-wire, that is already an outstanding result.

RS-422A is a special version of RS-422 and enables data transmission in one direction (simplex). There is one transmitter which can interface to several receivers.

RS-485 is an enhancement of the RS-422 interface and makes a bi-directional data exchange (half-duplex) possible via a twisted two-wire line.

Table 16 and Figure 76 show several features of the interfaces according to RS-422A and RS-485.

In the upper half of Figure 76 a transmitter drives a symmetrical line terminated by a 100 ohm resistor. According to the RS-422A standard up to ten receivers can be connected.

The lower half of Figure 76 shows a bi-directional bus system according to RS-485. Up to 32 Transceiver (that means transmitter and receiver) can be networked.

	RS-422A	*RS-485*
Common Mode Input Voltage	-7 V ... +7 V	-7 V ... +12 V
Receiver Input Resistance	4 kΩ min.	12 kΩ min.
Driver Load (line termination)	100 Ω	60 Ω
Short Circuit Output Current	150 mA to GND	150 mA to GND 250 mA to -7 V or +12 V

Table 16 Several Features of Interfaces according to RS-422A and RS-485

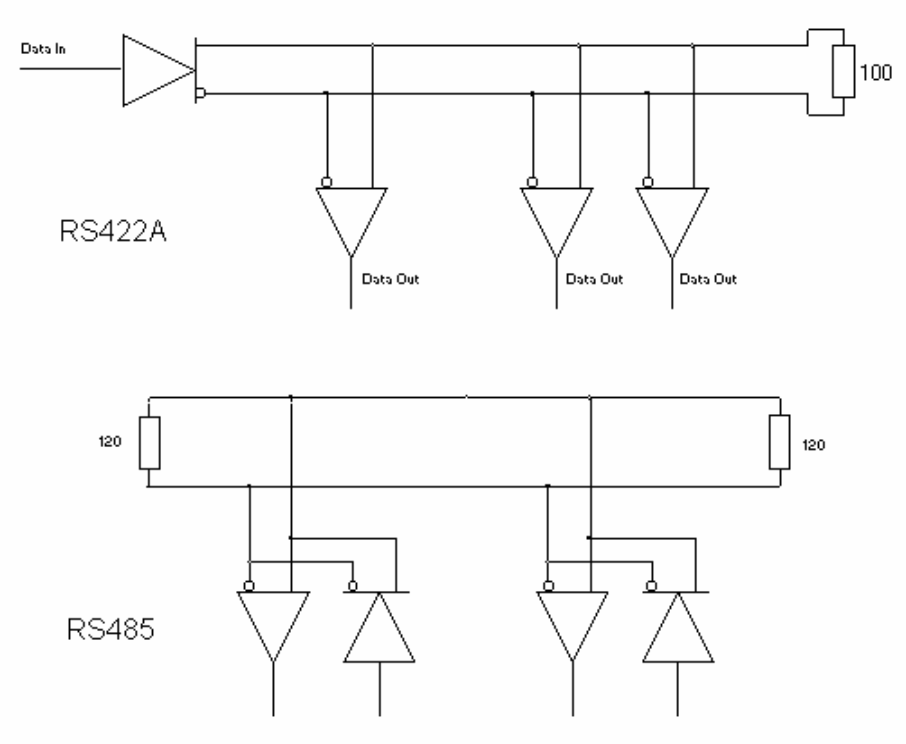


Figure 76 Interfaces according to RS-422 and RS-485

Almost any semiconductor manufacturer offers transceiver devices. Figure 77 shows the inner circuit of the SN75176A from Texas Instruments as an example.

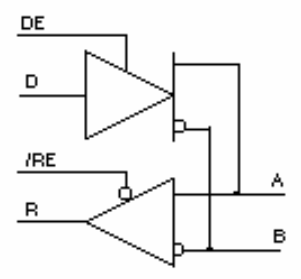


Figure 77 Inner Circuit of SN75176A

The connectors A and B on the right side denote the symmetrical bus lines. The left side shows input and output and the associated Enable pins.

For the half-duplex operation according to RS-485 either the transmitter or the receiver works. Therefore we can connect the Enable pins DE and /RE.

For the BASIC Stamp only small changes are required from the RS-232 protocol. The BASIC Stamp must control the wired Enable pins by an additional I/O pin. Before a SEROUT command we have to set DE = /RE = high and must reset afterwards by DE = /RE = low to activate the receiver.

6.3 Evaluation of GPS Information

The GPS receiver makes numerous information available from satellites. A quite simple but powerful GPS receiver is the GPS-Mouse from Garmin.

The Garmin GPS-Mouse is a 12-channel receiver with integrated antenna and therefore an ideal receiver for PC-based navigation or as sensor for microcontroller applications.

The package is completely closed, waterproof and suited for outside applications in a wide temperature range. The communication is RS-232. Different versions (land, see) offer appropriate flexibility for various areas of application. Figure 78 shows a picture of the GPS Mouse.

Figure 78 GPS Mouse

The GPS Mouse sends the data according to the NMEA protocol.

NMEA is the standard protocol which GPS units use for data exchange. The NMEA interface can be directly connected with an RS-232 Port of a PCs or a microcontroller. The NMEA interface works normally with 4800 Baud, eight data bits, one stop bit and without parity (4800-8N1).

The NMEA protocol consists of sentences of ASCII characters.

A NMEA sentences begins with the character $, followed by the address field (transmitter, sentence name) and the parameters separated by comma. The NMEA sentence is terminated by CR/LF. The check sum at the end of the NMEA sentence (before CR/LF) is optional. Not all GPS receivers send this parameter!

Two important NMEA strings are RMC and VTG.

RMC = Recommended minimum specific GPS/TRANSIT data

VTG = Actual track made good and speed over ground

Both NMEA sentences are given in the common form and as an example. Additional information to the NMEA protocol can be found at http://vancouver-webpages.com/peter/idx_faq.html, for example.

The RMC sentence contains date and time (marked bold) and the VTG sentence among others the speed over ground in km/h (marked bold again).

$GPRMC,**hhmmss.ss**,A,llll.ll,a,yyyyy.yy,a,x.x,x.x,**ddmmyy**,x.x,a*hh $GPRMC,**162715**,A,0000.673,N,00000.673,W,031.0,315.0,**270701**,000.0,E*64
$GPVTG,t,T,,,s.ss,N,**s.ss,K***hh $GPVTG,315.0,T,315.0,M,031.0,N,**057.4,K***4A

The following program example evaluates the speed in the VTG sentence and signalizes the exceeding of given speed limits.

The BASIC Stamp together with a GPS Mouse can inform a driver when speed limit is exceeded by flashing a LED or generating an acoustic signal.

```
' -----[ Title ]-----------------------------------------------
'
' File...... gps.bs2
' Purpose... Reading of GPS information
' Author.... Claus Kuehnel
' Started... 2001-07-27
' Updated...
'
' -----[ Program Description ]-------------------------------
'
' The program reads the VTG sentence from NMEA protocol sent
' by a GPS receiver each second.
' An LED signalizes a speed over four different
' speed limits.
'
' -----[ Revision Hisory ]-----------------------------------
'
' -----[ Directives ]----------------------------------------
'
'{$STAMP BS2}           'specifies a BS2
'
' -----[ Constants ]-----------------------------------------
'
B4800    con 188       '4800 Baud for NMEA default
RxD      con 16        'Serial I/O Pin 16

SpeedLimit1 con 50     'Speed limits
SpeedLimit2 con 80
SpeedLimit3 con 100
SpeedLimit4 con 120

LED1     con 8
LED2     con 9
LED3     con 10
LED4     con 11

' -----[ Variables ]-----------------------------------------
'
speed    var word
value    var word

' -----[ Initialization ]------------------------------------
'
' -----[ Main Code ]-----------------------------------------
'
loop:
     'Read the VTG sentence from GPS
     '$GPVTG,315.0,T,315.0,M,040.0,N,074.1,K*48
```

```
     serin RxD, B4800, [wait("VTG,"), wait(","),
wait(","),wait(","),wait(","),wait(","),wait(","),dec speed]
     value = 999
     lookdown speed, > [SpeedLimit4, SpeedLimit3, SpeedLimit2, SpeedLimit1], value
     branch value, [mSL4, mSL3, mSL2, mSL1]
     goto loop

' -----[ Subroutines ]----------------------------------------
'
mSL1:
     outc = $F  'LEDs off
     low LED1   'LED1 on
     goto loop

mSL2:
     outc = $F  'LEDs off
     low LED2   'LED2 on
     goto loop

mSL3:
     outc = $F  'LEDs off
     low LED3   'LED3 on
     goto loop

mSL4:
     outc = $F  'LEDs off
     low LED4   'LED4 on
     goto loop
```

Listing 32 GPS Speed Display (GPS.BS2)

To test such a GPS program it's not necessary to go for a drive. Here a GPS simulator can help.

Under www.sailsoft.nl there is a free GPS simulator well suited for this purpose. Figure 79 shows the subject data and the NMEA sentences in the Trace Window.

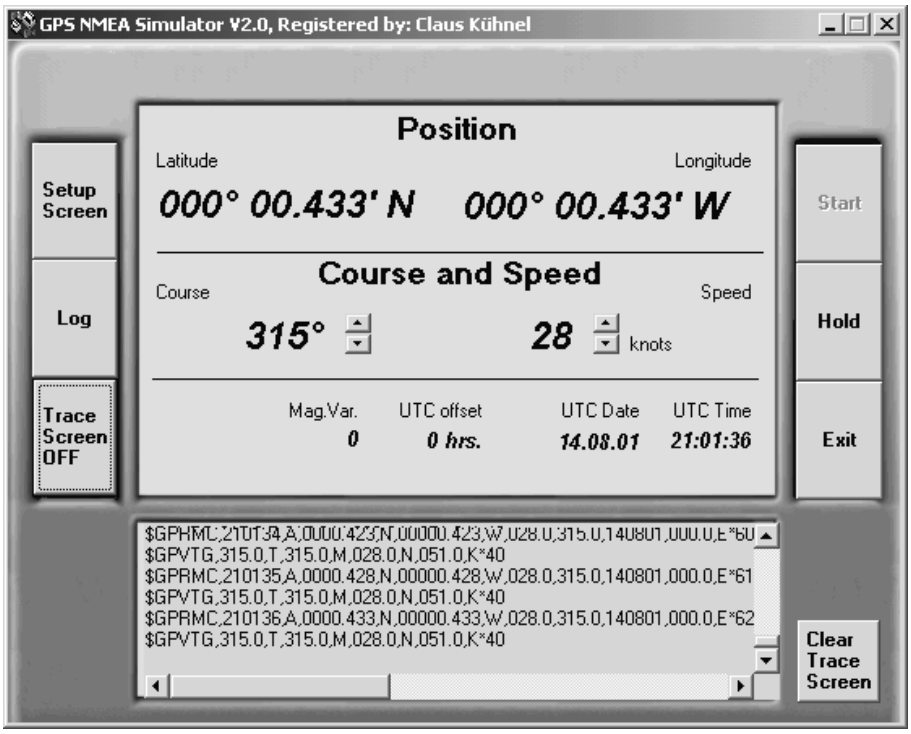


Figure 79 GPS Simulator

The generated NMEA sentences can be sent by a serial interface of the PC to the BASIC Stamp or can be stored in a log file.

You can send this log file at any time with a Terminal program to the BASIC Stamp.

The BS2p can save whole data sentences in the Scratch Pad RAM. This way we can work with higher baud rates. Listing 33 shows the program example adapted to the BS2p. In addition a serial LCD was attached to display the velocity values.

```
' -----[ Title ]-----------------------------------------------
'
' File..... gps1.bsp
' Purpose... Reading of GPS information
' Author.... Claus Kuehnel
' Started... 2001-09-15
' Updated...

' -----[ Program Description ]-------------------------------
'
' The program reads the VTG sentence from NMEA protocol sent
' by a GPS receiver each second.
' An LED signalizes a speed over four different
' speed limits.
'
' -----[ Revision Hisory ]-----------------------------------
'
' -----[ Directives ]----------------------------------------
'
'{$STAMP BS2p}              'specifies a BS2p

' -----[ Constants ]-----------------------------------------
'
B4800     con 500           '4800 Baud for NMEA default
N2400     con 17405         'Baudmode-2400 bps inverted

RxD       con 16            'Serial Data from GPS
LCDpin    con 0             'Serial Data to LCD

SpeedLimit1 con 50          'Speedlimits
SpeedLimit2 con 80
SpeedLimit3 con 100
SpeedLimit4 con 120

LED1      con 8             'Speed LEDs
LED2      con 9
LED3      con 10
LED4      con 11

offset    con 24            'Position of speed value in VTG string

I         con 254           'Instruction prefix value

' LCD control characters
'
ClrLCD    con $01           'clear the LCD
Line1     con $80           'addr line #1 ¦ 80H

' -----[ Variables ]-----------------------------------------
'
speed     var word
value     var word
```

```
idx      var byte
temp var byte(3)

' -----[ Initialization ]----------------------------------
'
' Initialize the Serial LCD (HD44780 controller & Serial Backpack)
'
LCDini:

     low LCDpin                         'Make the serial output low
     pause 1000                         'Let the LCD wake-up
     serout LCDpin,n2400,[I,ClrLCD]     'Clear the LCD screen

' -----[ Main Code ]----------------------------------------
'
loop:
     'Read the VTG sentence from GPS
     '$GPVTG,315.0,T,315.0,M,040.0,N,074.1,K*48
     'puts serin data in Scratch Pad RAM
     serin RxD, B4800, [wait("GPVTG,"), spstr 35]

     'put offset, 4 : put offset+1, 5 : put offset+2, 6
                                        'for test

     for idx = 0 to 2                   'read data from Scratch Pad
          get idx+offset, temp(idx)
     next
     speed = (temp(0)-48)*100           'convert to number
     speed = (temp(1)-48)*10 + speed
     speed = (temp(2)-48) + speed

     serout LCDpin,n2400,[I,Line1]      'display speed value
     serout LCDpin,n2400,["Speed = ", dec3 speed, " km/h"]

     value = 999
     lookdown speed, > [SpeedLimit4, SpeedLimit3, SpeedLimit2, SpeedLimit1], value
     branch value, [mSL4, mSL3, mSL2, mSL1]
     goto loop

' -----[ Subroutines ]--------------------------------------
'
mSL1:outc = $F  'LEDs off
     low LED1   'LED1 on
     goto loop

mSL2:outc = $F  'LEDs off
     low LED2   'LED2 on
     goto loop

mSL3:outc = $F  'LEDs off
     low LED3   'LED3 on
     goto loop
```

```
mSL4:outc = $F  'LEDs off
     low LED4   'LED4 on
     goto loop
```

Listing 33 GPS Speed Display (GPS1.BSP)

In both GPS program examples the GPS is connected to I/O pin 16. If a connected GPS will be disturbed by the echoes from this I/O pin then change the constant RxD in Listing 32 and Listing 33 to another free I/O pin.

6.4 Measuring Tilt and Acceleration

Several years ago sensors with micro-mechanical components improved dramatically. Analog Devices offers the ADXLxxx family of acceleration/tilt accelerometers at a low price.

The sensor's output is based on balance of capacitors made with micro-mechanical silicon arms as a bridge structure. This structure is assembled with some other electronic components through an etching process. Disturbing this balance with external causes a continuous PWM signal, whose duty is aligned to external tilt or acceleration.

The result of a external force is the duty T1 of a continuous generated pulse with the period T2.

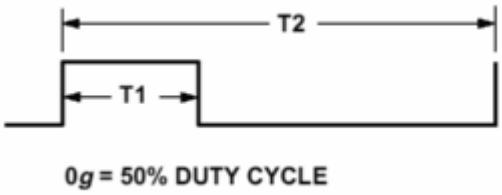


Figure 80 Pulse on ADXL202 output

Without influence of an external force the ratio of low to high is 0.5. The frequency of the following pulses is tuned by the size of two external capacitors chosen by the user.

Integrated within the ADXL202 are filters and amplifiers for producing an aligned analog output-voltage.

The ADXL202 sensor generates signals for tilt and acceleration in two directions with two separate sensors systems structured in a rectangular position on the sensor-chip.

It's an advantage that the ADXL202's pulse outputs are contacted directly to pins. With the PULSIN command the pulse-width can be measured immediately by counting. By this way an A/D converter is not needed.

Some more information from ADXL202 are given with the data sheet from Analog Devices [www.analog.com/iMEMS/products/ADXL202.html]. With this link are some information and application examples.

In our example we enhance the circuit with a parallel driven LCD module with the HD44780-compatible controller. For this kind of LCD the BS2P has the powerful LCD-statements that we'll use once again.

The LCD module NLC 16x2x06 is offered by the German warehouse CONRAD, specializing in electronic components.

In contrast to the RS232- serial controlled LCD-Modules this module is very inexpensive and we have a comfortable programming interface courtesy of the BS2p.

The disadvantage is the use of more pins for connecting to the LCD module.

So this approach works well for this project, where we have free this pins leftover. In some cases it is possible to use a single pin for multi-purpose in different parts of the program.

Figure 81 shows the scheme of two dimensional measuring tilt/acceleration with an alphanumeric display.

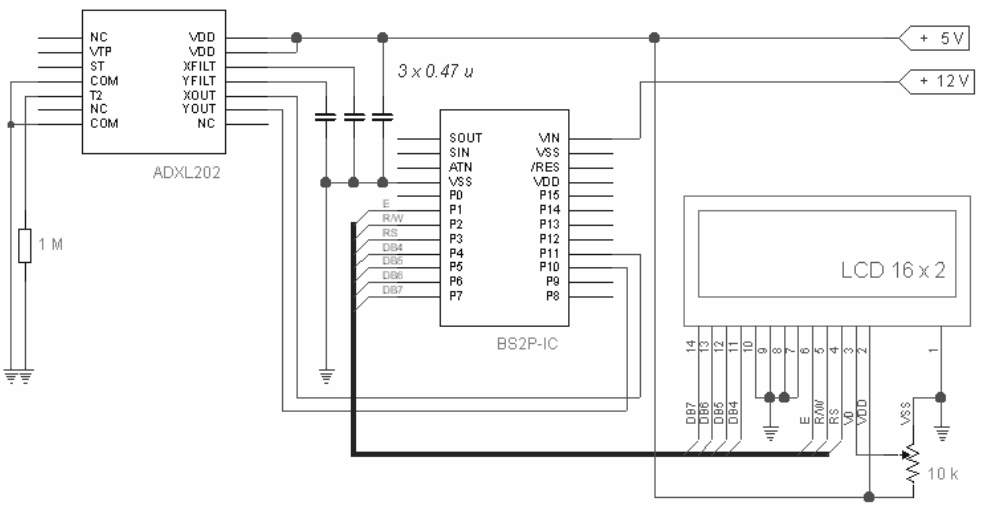


Figure 81 Two-dimensional sensor for tilt and acceleration

The 1 MΩ resistor on I/O pin T2 of ADXL202 generates a pulse-period of about 8 ms for the two output signals XOUT and YOUT. Each low/high response on I/O pins 10 and 11 starts counting, which is stopped by the high/low response. The resolution for this counting is two microseconds. Three MP-condensers on ADXL202-pins are for avoiding spikes and are needed to have a stable result in display. A variable resistor on the LCD is used for calibrating the contrast.

The program ADXL_2P1.BSP shows the reading of the ADXL202 output and the display of these values in an endless loop. Because there are no additional tasks done within this example, a delay of 100 milliseconds is added to the loop.

```
' -----[ Title ]-----------------------------------------------
'
' File......  ADXL_2P1.BSP
' Purpose...  Demonstration einiger LCD-Befehle der BS2P
'             in Verbindung Beschleunigungssensor ADXL202
' Author....  Klaus Zahnert
' Started...  10.6.01
' Updated...

' -----[ Program Description ]-------------------------------
```

```
'
' Use PULSIN to measure two-axis tilt from an ADXL202 and display
' on the LCD.
'' -----[ Revision Hisory ]----------------------------------
'
' 10.6.01 : Version 1.0

' -----[ Constants ]-----------------------------------------
'
p         con 1
xpin      con  11
ypin      con  10

' -----[ Variables ]-----------------------------------------
'
n         var byte
x         var word
y         var  word
sumx      var word
sumy      var word

' -----[ Initialization ]------------------------------------
'

' -----[ Main Code ]-----------------------------------------
'
start:
     sumx = 0
    sumy = 0
    for n = 1 to 8
     pulsin xpin,1,x    : sumx = sumx + x
     pulsin ypin,1,y    : sumy = sumy + y
    next
    x =sumx/8
     y =sumy/8

     debug "x = ",dec5 x,TAB,TAB,"y = ",dec5 y,cr
    gosub dispLCD
     pause 100
     goto start


' -----[ Subroutines ]---------------------------------------
'
init:
     lcdcmd p,48
     pause 5
     lcdcmd p,48
     pause 0
     lcdcmd p,48
     pause 0

     lcdcmd p,32
```

```
     pause 0
     lcdcmd p,44
     pause 0
     lcdcmd p,8
     lcdcmd p,12
     lcdcmd p,6
    return

dispLCD:
     gosub init

     lcdout p,1,  [" X-pos = ",dec5 x]
     lcdout p,192,[" Y-pos = ",dec5 y]
     return
```

Listing 34 Evaluation of ADXL202-Informations (ADXL_2P1.BSP)

To initialize the LCD the Parallax approach is used. Because it is to our advantage to introduce some delays in timing of the loop by using the PAUSE statement, there is enough time for a new initialization of the LCD-display. So the first character to display comes to first position. With this we have the x-message on the upper line.

One problem by using micro-mechanical sensors is to generate a stable signal.

This depends from on the necessary bandwidth and the sampling frequency.

To stabilize the display (see the randomized variations in the display on lowest position of characters) an average value is calculated from 8 measured values. It should be done with attention when accumulating these numbers in a word-sized variable. The capacity is limited by 65535.

For changing the horizontal position to a tilt of 30 degrees, there is a difference in pulses of about 700 on each channel.

There is some difficulty in using the ADXL202 sensor. Without a special socket or printed circuit board the sensor is designed as a surface-mounted device (SMD) and it’s not easy to handle with a usual soldering iron. The ADXL202 Evaluation Board is a small PCB that fixes this problem, available directly from Parallax.

6.5 Data Display with Stamp Plot Lite

In Chapter 6.4 the measured values are sent to an alphanumeric LCD.

Synchronous to this, the data is displayed in the PC's Debug Window. This is made by inserting the program line

```
debug "X =  ",dec5 x, TAB,TAB, "Y = ", dec5 y,cr
```

The DEBUG command provides the output of the designated information from BASIC Stamp's SOUT I/O pin.

Remember that this I/O pin is used for programming during download and during runtime this transmission line is used for the DEBUG commands.

While testing programs, the Debug Window of the StampW Editor is an indispensable help, but it's not a pleasure for visualization of data. Watching numbers scroll down the blue Debug window lacks visual imagery.

Selmaware Solutions has designed the program "Stamp Plot" for the PC, based on Visual Basic 6. The Lite version is with free for download from their website [www.selmaware.com/stampplotlite/home.htm].

Stamp Plot receives data from a serial port of the PC, transmitted from BASIC Stamp with DEBUG commands. With this data it generates a window for graphical display of the incoming numeric data as well as the display of ASCII characters like the Debug Window.

With the basic features of this versatile program we'll see the display of the measured results from tilt sensor ADXL202. We use the program ADXL_2P1 from Chapter 6.4.

Include a second DEBUG command as follows:

```
debug "X =  ",dec5 x, TAB,TAB, "Y = ", dec5 y,cr
debug dec y , cr
```

This second line without a part of printed text is an additional output. It is used for graphical display in a diagram.

Figure 82 shows the opened window of Stamp Plot Lite. All settings done with mouse clicks in this window. The alternative way is to configure presettings from software with modified DEBUG commands. Table 17 shows these commands.

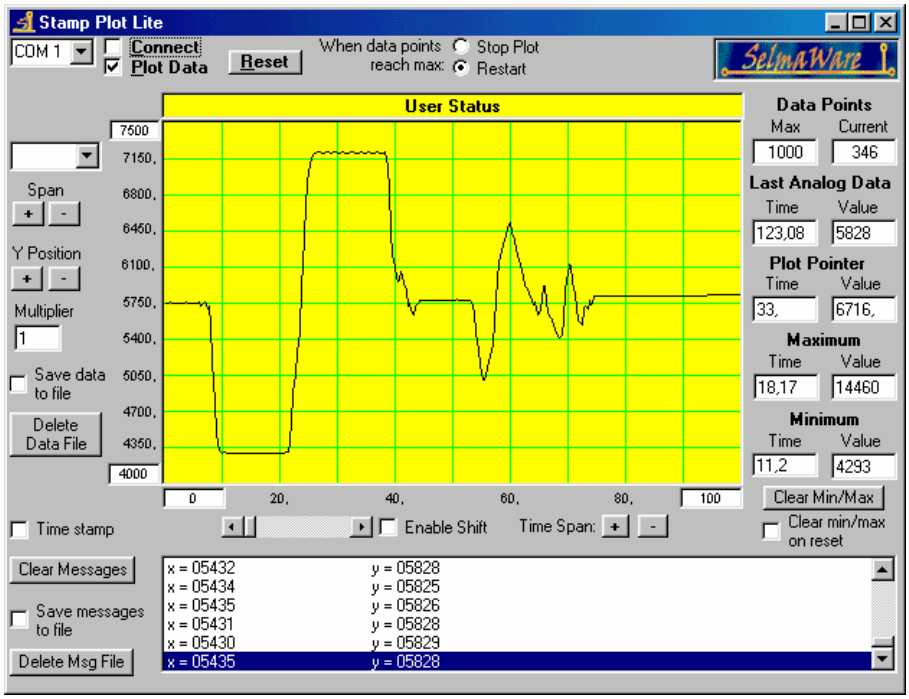


Figure 82 Display of Measured Data in Stamp Plot Lite

The parameters for serial transmission (9600 bps, 8 data bits, no parity (9600,8,N,1) and no handshake were already established by the DEBUG command. However, the PC must be configured for the appropriate COM-Port and in the program window the connect state must be activated by clicking it. Be careful when opening the COM-Port (with one device); working with Stamp Plot Lite means the active Debug Window in StampW must be closed.

Now you can see how a DEBUG command can be used to provide a visual depiction of sensors connected to your BASIC Stamp.

Based on preset values clicking the “Reset” button starts a new diplay. A very advantageous feature of the program is saving the stored values by clicking “Save messages to file”. This way we have the features of a data-logger. Saving to a file we can do a post processing with other programs like EXCEL or other table-processing program. It is possible to add a time stamp to each saved value.

Figure 82 shows the actual stored values until breaking the transmission with “disconnect”.

The high magnitudes on the left site of the diagram are made with a tilt of +/-90 degree, moving the ADXL202. They are equal to the result of +/- 1 g, working to the tilt-sensor from the earth acceleration force.

All StampPlot Lite settings can be set from BASIC Stamp with the DEBUG command. Setting a “!” before the argument defines a control for Stamp Plot. Different parameters to this enhanced DEBUG commands are for controls.

Table 17 shows the controls and how they work for Stamp Plot Lite.

Nr.	*Control*	*Argument*	*Explanation*
1	!TITL	message	Sets to title of the form to the message
2	!USRS	message	Sets the User Status box to display the message
3	!BELL		Sounds the bell on the PC
4	!AMAX	value	Sets the plot maximum analog value
5	!AMIN	value	Sets the plot minimum analog value
6	!SPAN	MinValue, MaxValue	Sets the plots analog maximum and minimum as above but also adds the range to the Range Drop-Down box
7	!AMUL	value	Sets the value to multiply incoming data by
8	!TMAX	value	Sets the plot maximum time (seconds)
9	!TMIN	value	Sets the plot minimum time (seconds)
10	!PNTS	value	Sets the number of data points to collect
11	!PLOT	ON/OFF	Enables/disables the plotting of data
12	!RSET		Resets the plot and all data
13	!CLRM		Clears the message field
14	!CLMM		Clears the min/max recorded values
15	!CMMR	ON/OFF	Enables/Disables clearing of Min/Max recorded values on reset
16	!MAXS		Sets the plot to STOP when data points are full
17	!MAXR		Sets the plot to RESET when data points are full
18	!SHFT	ON/OFF	Enables/disables the plot from shifting when recording data (may cause a loss of data accuracy if enabled)
19	!TSMP	ON/OFF	Enables time stamping of list messages, messages and data saved to files
20	!SAVD	ON/OFF	Enables saving of analog and digital data to files (stampdat.txt)
21	!SAVM	ON/OFF	Enables saving of messages to a file (stampmsg.txt)
22	!DELD		Deletes the saved data file (stampdat.txt)
23	!DELM		Deletes the saved message file (stampmsg.txt)

Table 17 Commands to Control Stamp Plot Lite

For example, the following command starts the plotting process:

```
DEBUG "!PLOT ON", CR
```

Because Stamp Plot is designed with Visual Basic 6, it is for Windows 95 and higher versions only. For studying applications with Stamp Plot Lite take a look to the experiments from “Stamps in Class” from Parallax. Selmaware Solutions has also created tutorials [www.selmaware.com/tutorials].

7 Appendix

7.1 Examples to Wiring the I/O Pins

7.1.1 Keys

Figure 83 shows how to connect a pushbutton to an I/O pin. Beside the respective key the parameter *Downstate* for the BUTTON command is determined by the circuit.

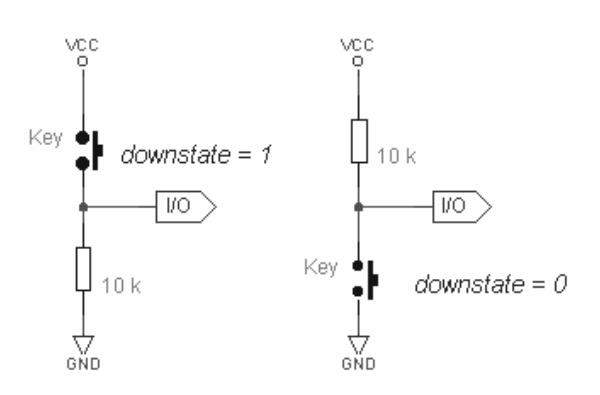
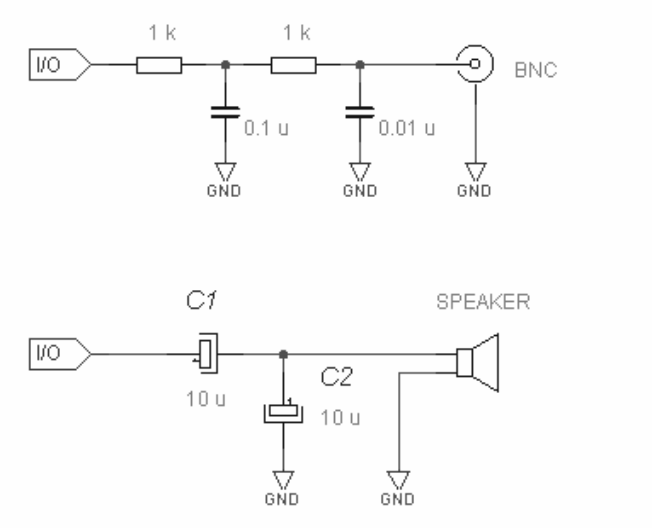


Figure 83 Connecting a Pushbutton

7.1.2 Tone Output

For tone output one can connect an amplifier or a speaker to the BASIC Stamp. C1 capacitor is not required
C2 capacitor is optional

Figure 84 shows different possibilities for connecting an external amplifier or speaker.



C1 capacitor is not required
C2 capacitor is optional

Figure 84 BS2 Tone Output

When connecting an external amplifier inserting a filter circuit is recommended. At the same time such a filter circuit protects the I/O pin against a short-circuit.

7.2 Baudmode Parameter in SERIN and SEROUT

The following tables list the parameter *Baudmode* for standard baud rates.

BS2 BS2e

Baud Rate	8-bit no-parity inverted	8-bit no-parity true	7-bit even-parity inverted	7-bit even-parity true
300	19697	3313	27889	11505
600	18030	1646	26222	9838
1200	17197	813	25389	9005
2400	16780	396	24972	8588
4800	16572	188	24764	8380
9600	16468	84	24660	8276

BS2sx BS2p

Baud Rate	8-bit no-parity inverted	8-bit no-parity true	7-bit even-parity inverted	7-bit even-parity true
1200	18447	2063	26639	10255
2400	17405	1021	25597	9213
4800	16884	500	25076	8692
9600	16624	240	24816	8432

Remark:

Has SEROUT to work with Open Collector, then add 32768 to the concerning *Baudmode* value.

If I/O pin 16 (Rpin = 16 and/or Tpin=16) is used for serial communication, then independently of the assigned *Baudmode* the polarity is always inverted and the transmitter has an active output.

7.3 Hayes Command Set

The Hayes Command Set is a common standard for Modems.

It acts using special initialization strings and commands to set the modem in a defined state of operation (for example, data compression on (V42bis), error correction on (V42), speaker off, etc.).

Such a command begins usually with AT (AT means Attention) and tells the modem that command follow.

Some important AT commands are listed now. A claim on completeness is not laid.

AT	Begin of a command
A/	Repeat last executed command
A	Answer Call
\Bn	Transmits break n (1-9) * 100 ms
&C0	DCD always on
&C1	DCD follows the carrier
\C0	No buffering of data
\C1	Buffers all data after calling the modem
\C2	No buffering of data after calling the modem
Dn	Dial command – allowed characters in the dial string: 0-9,- Phone numbers ',' : Pause, length defined in S8 'W' : Waits for second dial tone 'Nn', '\n' or 'S=n': Dials the saved phone number (dependant of the modem) '@' : Waits for a silent line (no dial pulses anymore) 'P' : IWV (Pulse dial) 'T' : MFV (Tone dial) '!' : the modem breaks for a half second (Flash Function) 'R' : calls in Answer Mode
&D0	Ignore DTR
&D1	Switches to common mode, after DTR is Lo
&D2	DTE Controls DTR
&D3	Reset after DTR is Lo
E0	No Command Echo
E1	Echo Command Chars
%E0	Disable auto retain
%E1	Enable auto retain
&Fn	Load Factory Setting n
H	On Hook (Hang Up)
H1	Off Hook
L/L0	Low speaker volume
L1	Low speaker volume
L2	Medium speaker volume
L3	High speaker volume
M/M0	Speaker Off
M1	Speaker On until CD
M2	Speaker always on
M3	Speaker Off during dial
O	Changes from command to data mode
O1	Changes from command to data mode and optimizes the connection (MNP/V.42)

&R0	CTS follows RTS
&R1	CTS on during Connects Hi
%R	Displays all S-Register
Sn=X	Sets the S-Register n to X
Sn?	Reads the S-Register n
&S0	DSR always Hi
&S1	DSR according to the RS-232 Specifications
\S	Displays the modem status
\Tn	Number of minutes n, nach denen Modem auflegt, wenn keine Daten übertragen werden
V0	Output messages as NUMBER
V1	Output messages as WORD
&V	Displays the important register and flags set by commands
%V	Displays modem firmware version (EPROM)
X0	Modem sends the message 'CONNECT' only
X1	Full Connect messages
X2	'X1' + dial tone detection ('No Dialtone')
X3	'X1' + busy detection ('BUSY')
X4	'X1' + 'X2' + 'X3'
+++	Escape Code (Modem changes from data to command mode)

8 Reference

[1] Zahnert, K.; Kühnel, C.:
BASIC Stamp.
Franzis: München, 1995

[2] Kühnel, C.; Zahnert, K.:
BASIC Stamp 2nd Edition.
Newnes: Boston, u.a., 2000

[3] Edwards, S.:
Programming and Customizing the BASIC Stamp Computer.
Mc-Graw Hill: New York, u.a., 2001

[4] Williams, A.:
Microcontroller Projects with Basic Stamps.
R&D Books: Lawrence, Kansas, 2000

[5] Held, Gilbert:
The Complete Modem Reference.
The Technician's Guide to Installation, Testing and Trouble-Free Communications. 3rd Ed.
Wiley: New York, 1996

9 Links

Author's website
http://www.ckuehnel.ch

Parallax's website
http://www.parallax.com

Information to the BASIC Stamp
http://www.nutsvolts.com/stmpindx.htm
http://www.nutsvolts.com/ftpindex.htm
http://www.emesystems.com/BS2index.htm
http://www.al-williams.com/awce/som.htm

How to control HD44780-based Character-LCD
http://home.iae.nl/users/pouweha/lcd/lcd.shtml

HD44780-Based LCD Modules
http://www.doc.ic.ac.uk/~ih/doc/lcd/
http://www.electronic-assembly.de/deu/pdf/doma/4_20.pdf

StampPlot Pro (S-Plot Pro) Tutorial For the BASIC Stamp 2
http://www.selmaware.com/

Gate Drive Characteristics and Requirements for HEXFET®s

http://www.powerdesigners.com/InfoWeb/design_center/Appnotes_Archive/an-937.pdf

Complete 2400 bps Modem Modules with built-in DAA
http://www.cermetek.com

MODEM - Basic Hayes Modem AT strings
http://www.computerhope.com/atcom.htm