

JBug11 - Version 5

Manual

This page intentionally left blank

CONTENTS

1	INTRODUCTION	7
1.1	The Program	7
1.2	This Manual	7
1.3	Hardware	7
1.4	JBug11 Source Code	7
1.5	Philosophy	8
1.5.1	Keyboard Input	8
1.5.2	Default Hexadecimal Number Entry	8
1.5.3	Screen Appearance	8
1.5.4	Command Syntax	8
1.6	Development History	8
1.7	Getting Started	9
1.8	Note on the Author	9
1.9	An Appeal	9
2	MISCELLANEOUS	10
2.1	Installing	10
2.2	Uninstalling	10
2.3	Registry Entries	10
2.4	Project Files	10
2.5	Supporting Files	11
2.6	Remote Reset	11
2.7	General-Purpose Switching	12
2.8	Local and MCU-controlled Memory	12
2.9	Accessing Expansion Memory	12
2.10	Importing Configurations from Version 4.x	12
2.11	USB-to-Serial Adapters	13
3	TALKERS	14
3.1	Introduction	14
3.2	Talker Overlay Files	15
3.3	Activating the Talker	16
3.4	Baud Rates	17
3.5	Talker Map Files	18
4	MEMORY MANAGEMENT	20
4.1	Writing to MCU-Controlled RAM	20
4.2	Writing On-chip EPROM and EEPROM	20
4.3	Writing to EEPROM	20
4.4	Managing the BPROT Register	21
4.5	EEPROM mapping	21
4.6	Writing to EPROM	22
4.7	Writing External Memory	23
4.8	Accessing Indirect Memory	23
4.9	Memory Map Display	23
5	DEBUGGING	24
5.1	Breakpoints	24

5.2	Setting and Clearing Breakpoints	24
5.3	Tracing in EEPROM	25
5.4	SWI in user code	25
5.5	Illegal opcodes in user code	26
6	PROGRAM FEATURES	27
6.1	Screen Layout	27
6.2	Main Menu	28
6.3	Speedbuttons	28
6.4	Output Window	28
6.5	Command History Window	29
6.6	Command Edit Box	29
6.7	Status Line	30
6.8	Information Sidebar	30
	6.8.1 CPU Registers Display	30
	6.8.2 T/G and L/U addresses	30
	6.8.3 Start / Break Points	31
	6.8.4 Watch Window	32
6.9	Add to Watch	32
6.10	File Menu	33
6.11	View Menu	33
6.12	Actions Menu	34
6.13	Macro Menu	35
6.14	Settings Menu	36
6.15	Help Menu	36
6.16	Keyboard Shortcuts	36
6.17	Symbol Table/Register Display	37
6.18	Base Converter	37
7	CONFIGURATION	38
7.1	Settings Dialog	38
7.2	Settings>General	38
7.3	Settings>COM Port	40
7.4	Settings>Macros	42
7.5	Settings>Debug	43
7.6	Settings>Talkers	45
7.7	Settings>Overlays	47
7.8	Settings>Memory	48
7.9	Settings>Notes	50
7.10	Settings>Ind Mem	50
8	COMMANDS	52
8.1	General information	52
8.2	Path Tokens	53
8.3	Labels instead of Addresses	54
8.4	PCbug11-style Alternative Commands	55
8.5	Set Breakpoints	56
8.6	Set Pass Breakpoint	56
8.7	Clear Breakpoints	57
8.8	Clear Output Window	57

8.9	Clear Local Memory	57
8.10	Compare Memory	58
8.11	Alter CONFIG	59
8.12	Connect / Disconnect	60
8.13	Cyclic Redundancy Check	61
8.14	Duplicate Memory	63
8.15	Bulk Erase EEPROM	63
8.16	Fill Memory	64
8.17	Find Bytes	65
8.18	Find Next	66
8.19	Go (Run)	67
8.20	List Memory	68
8.21	Load Memory	69
8.22	List Macros	70
8.23	Modify Memory	71
8.24	Pause & Wait	72
8.25	Register Display and Change	73
8.26	Reset	74
8.27	Reset Pass Count	74
8.28	Save Memory	75
8.29	Stop	76
8.30	Launch Terminal Window	76
8.31	Trace	77
8.32	Step Over	78
8.33	Switch	79
8.34	Unassemble	80
8.35	Using a Symbol Table	81
8.36	Verify	83
8.37	Verify Erase	84
8.38	Indirect Memory Commands	85
9	AUTOMATION	86
9.1	Introduction	86
9.2	Macros	86
9.3	Boot Script	87
9.4	Autostart Macro	88
9.5	Replaceable Parameters	88
9.6	Playing and Recording Macros	89
9.6.1	Playing	89
9.6.2	Recording	89
9.6.3	Stopping Playing or Recording	90
9.7	Macro Editor	90
10	TERMINAL WINDOW	92
10.1	Introduction	92
10.2	Window Layout and Features	92
10.3	Interaction with the Monitoring/Debugging Functions	93
10.4	Terminal Window Menu Items	94
10.5	Terminal Window Settings	95
10.6	Terminal Demonstration Program	96

11	ERRORS	97
11.1	Communication and Echo Errors	97
11.1.1	Comms Error	97
11.1.2	Echo Error	97
11.2	Diagnostics	98
11.3	Error Report	98
11.4	Command Line Errors	98
APPENDIX A	- ACKNOWLEDGMENTS	99
APPENDIX B	- HARDWARE	100
APPENDIX C	- COMMAND SUMMARY	102
APPENDIX D	- SETUP FOR DIFFERENT MCU's	104
APPENDIX E	- COMMAND-LINE ERROR MESSAGE SUMMARY	108
APPENDIX F	- RS232 COMMUNICATIONS	112
APPENDIX G	- GETTING STARTED and QUICK TOUR	116
APPENDIX H	- DISTRIBUTION FILES	121
APPENDIX J	- JBUG11 REVISION HISTORY & KNOWN BUGS	125
APPENDIX K	- CHANGES: Version 4 to Version 5	126

<p>Note: All Trade Marks and Trade Names mentioned in this manual are, and remain, the property of their respective owners.</p>

1 INTRODUCTION

1.1 The Program

JBug11 is a monitor and debugger for developing assembly language programs on the Freescale (Motorola) 68HC11 series of micro-controllers (MCU). It is designed to be used on a personal computer running one of the Microsoft Win32 operating systems, and to communicate with the micro-controller under test by means of an RS232 serial link from one of the PC's serial communications ports. This program owes a lot to PCbug11 which was written by Motorola in the days of MS-DOS computers. Its development was originally inspired by the difficulty of running PCbug11 on modern fast processors; at the same time it provides some of the features expected in a modern GUI environment, and some additional de-bugging tools.

It has been tested with MC68HC11D0, MC68HC11E1 and MC68HC811E2 chips, running at an E-clock rate of 2MHz, and with the MC68HC11F1 chip running at 4MHz. It has been tested with the Windows 98SE, Windows 2000 and Windows XP operating systems. I believe that there should be no difficulty with other Microsoft Win32 OS, or with most other chips in the 68HC11 series, but all feedback on these aspects would be very welcome.

1.2 This Manual

The Manual is laid out in the following chapters:

- Introduction
- Miscellaneous
- Talkers
- Memory Management
- Debugging
- Program Features
- Configuration
- Commands
- Automation
- The Terminal Window
- Errors
- Appendices

1.3 Hardware

Development of this program was carried out with the micro-controller mounted in a Finger Board II. This is a small printed circuit support board produced by Embedded Acquisition Systems of Michigan. It has also been tested with Marvin Green's BotBoard+, the MicroStamp11 by Technological Arts, and the F1 Controller Board by Pete Dunster. See [Appendix B](#) for a typical hardware layout.

1.4 JBug11 Source Code

JBug11 is written in Object Pascal as implemented by Borland, in my case in Delphi version 4. Two additional components are used in JBug11 which are not natively present in my Standard copy of Delphi 4: The chief one is the comms port component and for this I've used the one developed by Dejan Crnila which is available as freeware on the SourceForge site. The other is an extended combo

box available from the Tory's Delphi site. The JBug11 source code is available from my web site if you wish to tinker.

1.5 Philosophy

1.5.1 Keyboard Input

JBug11, although using the usual graphical interface, is not designed to be primarily mouse-driven. This is because my own experience of assembly language programming and debugging is that it calls for continuous use of the keyboard.

Default Hexadecimal Number Entry

I have also made the decision that, with very few exceptions, it is much easier to work all the time in hexadecimal. For this reason all commands default to expecting their data in hexadecimal (there is one exception where specifying a number of repeats - see the [Unassemble](#) command on page 80 for an example). Values may, however, also be input in decimal and in binary by the use of prefixes.

1.5.3 Screen Appearance

The actual GUI appearance of JBug11 is loosely based on the on-screen appearance of Motorola's PCbug11. The extra resolution available with a modern graphical user interface has meant that certain features such as the breakpoint display may be permanently on view. The main form may be resized by dragging, and restored to one of three 'standard' sizes by clicking the menu item under View. Whatever size and layout the form has when it is closed will be remembered in the Windows Registry until next time. The Registry also stores the position and size of subsidiary windows opened as part of the program.

1.5.4 Command Syntax

Some of you may wonder why I didn't stick to exactly the same command syntax that Motorola uses in PCbug11. I've no one single answer to this: I used to use a debugger on CPM machines which had much shorter commands, DEBUG.COM for the PC under MSDOS has some neat syntactical features (and one-letter commands). I've tried to make a combination of the best features of the debuggers that I'm familiar with. In the end it comes down to personal pride and the 'not invented here' syndrome. If you download the source code, you can modify the program to accept any command mnemonics you like. Most PCbug11 commands are, in fact, accepted.

1.6 Development History

Development of JBug11 began around June 2000, with the first public release of version 1.0 in November of that year. This early version used different talkers for writing to RAM, EEPROM and EPROM and the correct talker had to be booted for each type of memory. Version 2, introduced in March 2002, only used the one talker, and wrote to EEPROM and EPROM by laboriously writing the various bits in PPROG using the read and write routines in the standard talker. It was very slow! Talker overlay files first appeared in version 3 which was introduced in May 2002. Users, so far, had had to manage without on-line help, this was introduced in version 4 in December of 2002, with the last revision in July 2004, bringing the version numbering to 4.5.

Way back in 2002, a user had suggested that it would be more sensible to keep configuration information in files rather than in the Windows Registry. This good idea has had to wait until 2006 to become incorporated in version 5.0, along with a host of other changes, which I hope will be regarded as improvements!

There have been so many changes between versions 4xx and 5 that I have put a summary of the more important ones in Appendix K.

1.7 Getting Started

For those simply wishing to get started as soon as possible, I've put the basic installation and operating instructions in [Appendix G](#).

1.8 Note on the Author

Before you, the reader, complain that the program doesn't appear to behave exactly as you would like, or lacks some essential feature, please bear in mind that I'm only a hobby programmer with an interest in the 68HC11. I'm not working in a commercial environment. The upside of this is that the program and its source code are freely available, although I would of course much appreciate being told of any modifications you may make, or criticisms you may have. Let me know at john.beatty@virgin.net

1.9 An Appeal

JBug11 has a large number of features, and may take a little time to set up, particularly if you have not previously used Motorola's PCbug11. New users are advised to read the Getting Started section in [Appendix G](#), and to follow the Quick Tour so as to get a feel for the program.

2 MISCELLANEOUS

2.1 Installing

Double click on the self-installing exe file, named something like 'Install-JBug11-xyz.exe' which you have downloaded from the web site. This is a console application, so will start in a DOS box. You will be given the chance to change the default installation folder (C:\Program files\JBug11) if you wish to install it somewhere else.

The installation utility does not also place an icon on the desktop; if want one, you can right click on JBug11.exe and use the context menu to Send To>Desktop (create shortcut).

2.2 Uninstalling

To uninstall JBug11, use the Windows Add/Remove Programs utility (in the Control Panel). This will remove the sub-directories of ..\JBug11\ where they have no files in them other than those originally installed. If you have your own files in, for example, the \Project\ sub-folder, then these will need to be removed manually.

Uninstalling will not remove the Windows Registry entries, which are managed by JBug11 itself, so if you need a completely clean machine, and are feeling brave, back up the Windows Registry and then delete the subkeys:

- HKEY_CURRENT_USER\Software\BeattSoft\Jbug11, and, possibly
- HKEY_USERS\Software\BeattSoft\Jbug11

2.3 Registry Entries

When JBug11 is run for the first time on a new machine it will make an entry in the Windows Registry under HKEY_CURRENT_USER as follows:

HKEY_CURRENT_USER\Software\BeattSoft\Jbug11\5.x

This key will contain sub-keys that store information on the on-screen layout of the program, and on general user preferences. The value names correspond to the variables which may be altered in the Settings dialog.

2.4 Project Files

JBug11 stores the settings peculiar to each project in Project Files. These are plain-text files somewhat similar to old-fashioned Windows 'INI' files. They have the extension '.jbp', and this extension may be associated with JBug11 so that double-clicking a project file will open it in JBug11.

2.5 Supporting Files

To be fully operational, JBug11 needs access to various supporting files of information. These are:

1. The binary data for the talker (unless using an EEPROM-resident talker). Example name: Talk_E.BOO This file must be accessible each time the MCU is re-booted with a RAM talker.
2. A map file of addresses within the talker. Example name: Talk_A.MAP This file must be available whatever kind of talker (RAM or EEPROM-resident) is in use.
3. Talker overlay files in standard Motorola S19 format to enable writing to EEPROM and programming EPROM or external Flash PROM. Example name: Ovly_Eeprom_A.rec
4. A file of information on all the MCU opcodes. Default name: HC11_Opcodes.csv

This file is used by the [Unassemble](#), [Go \(Run\)](#) and [Trace](#) commands to establish and process break points. It is a text file in comma separated value format prepared directly from Motorola's published information on the binary codes for each mnemonic of their instruction set. It may be examined with any text editor (and changed at your own risk!).

5. A file of information on the MCU control registers. Example name: Regs_HC11E9.csv

This file is used by the [Register Display and Change](#) command, as well as internally by JBug11. These files have been prepared from the published Motorola documentation and may be examined in any text editor.

6. A file of default data on the various MCU's in the HC11 series. It includes default memory map information, talker file names and control register file names. The file has the name: MCUData.cfg. The file may be edited, or added to, in any text editor, provided that the expected format is adhered to.

A full listing of the files in the distribution is in [Appendix H](#).

2.6 Remote Reset

Some target development boards allow the MCU to be reset remotely from the host PC running JBug11, and this can save some time and frustration in program development. To make this possible, it is necessary to run a wire from one of the COM port control pins, either DTR (Data Terminal Ready) or RTS (Ready To Send) via some simple interface logic to the MCU reset pin; details are shown in [Appendix B - Hardware](#). When JBug11 requires to reset the MCU, it toggles the polarity on the appropriate pin. See [Settings>COM Port](#) for information on setting up the toggles. A remote reset capability is a luxury - JBug11 works perfectly well without it.

Note that Motorola's PC'bug11 had a 'RESET' instruction which could do a reset via the clock monitor fail detector, without the need for additional wiring. But to succeed, the talker and communication link had still to be synchronized, and as the usual reason for a reset is because of trouble with just these two things, it appeared to me more trouble than it was worth.

2.7 General-Purpose Switching

With suitable circuitry on the target board, JBug11 is able to make use of the spare COM port output line (either DTR or RTS, whichever is not in use by remote reset) as a general-purpose logic output. If the circuitry is available (see [Appendix B - Hardware](#)), then this option may be set up in [Settings>COMPort](#), and activated using the right-hand speedbutton, or by the [Switch](#) command (page 79). For an example of the use of this facility, see [Writing to EPROM](#) on page 22.

2.8 Local and MCU-controlled Memory

The whole 64KB memory space of the MCU has its counterpart in a 64KB array of bytes within JBug11. I refer to the latter array as the local version of MCU memory. It may or may not be an exact copy of the MCU memory, but is used internally whenever communication of memory contents with the MCU is required. For example, a command to list MCU memory will initiate a request for the talker to transmit the required bytes of MCU memory to JBug11 where they will be stored in the corresponding address locations in the local memory array. From there they will be displayed in the output window.

It should be noted that it is certainly not the case that the local memory is always a faithful copy of the MCU controlled memory. In fact, at start-up, the JBug11 local memory is initially filled with zeros. Only as a result of commands such as L (List) will the local memory become a copy of the MCU memory. Also, commands which load the MCU with data, such as the command to load a Motorola S19 file, do so via the local memory, so making the local and MCU memories identical over the range of addresses covered by the loaded file.

Many commands have a version that affects only the local memory, for example 'LDL' - see [Load Memory](#) on page 69.

2.9 Accessing Expansion Memory

JBug11 relies on the chip being reset in Special Bootstrap Mode. Many actual MCU circuits and programs make use of memory external to the MCU, and to access this memory it is usually convenient to switch the chip to Special Test Mode immediately after activating the talker. This is done by writing the MDA bit in HPRIO to 1, and, for safety, writing the IRV bit to zero; that is to say, writing \$E5 to HPRIO. To do this automatically after the talker is boot-loaded, add the line:

```
R HPRIO=E5
```

to the Boot Script in [Settings>Macros](#).

2.10 Importing Configurations from Version 4.x

From the File menu, a dialog is available to import configurations from JBug11 version 4xx into Version 5. In version 4, configurations were stored in the Windows Registry, while in version 5 they are stored as plain-text [project files](#) (page 10). This dialog is grayed-out if a version 4 of JBug11 was never installed, or if the Registry entries relating to such a version have been deleted.

Using the Import dialog

In Version 5, it is necessary to store the MCU type in the project file and therefore this dialog is arranged in steps so that the MCU must be selected before the configuration can be saved.

- Step 1** The list box on the left of the dialog lists all the stored version 4 configurations. The top-most item is the configuration with which JBug11 version 4 was last closed; the other items, if any, are those that were stored in the Registry using the version 4 'Save/Recall' dialog. Select a Version 4xx configuration to save. Preview this configuration, if necessary, by clicking the 'Preview...' button
- Step 2** Choose the appropriate MCU for this configuration from the drop-down list. The save button is not available until an MCU has been selected.
- Step 3** A default project file name will be generated and placed in the edit box. If you are happy with this, click on the 'Save' button (the file will be saved in the '\Projects\' sub-folder of the folder containing the JBug11 executable); or select a different file name or folder with the 'Save As...' button.

Select another configuration to save, or close the Import dialog.

Projects saved via the Import dialog have an automatically-generated note to this effect added to the project file - see [Settings>Notes](#).

2.11 USB-to-Serial Adapters

Many USB-to-serial adaptors will not work satisfactorily with JBug11 (or with other PC host programs). This is because they do not recognize the 'break' character emitted by the MCU when about to boot load a talker. As recognition of this break is critical to the operation of the host PC, such adaptors are useless.

However, adaptors based on the FTDI chipset appear to work just fine. The author uses a

'US232B/LC USB to RS232 Laptop Companion' from EasySync in the UK:

<http://www.easysync.co.uk/products.html>

or Saelig in North America:

<http://www.saelig.com>

FTDI's website is at: <http://www.ftdichip.com/index.html>

3 TALKERS

3.1 Introduction

As with Motorola's own PCbug11, this program uses a *talker* - a small program which remains resident on the micro-controller and which provides basic communication with the PC. This allows all the sophistication to be built into the PC host program, where storage is effectively unlimited, and keeps to a minimum the usage of precious memory resources on the MCU.

The talker is interrupt driven in such a way that it takes over control of the MCU whenever the host PC sends a byte to the MCU. This allows the host to inspect or alter programs even while they are running on the MCU. The talker may make use of either the SCI or XIRQ interrupt, the latter method requires a hardware connection between the PD0 and XIRQ\ pins (see [Appendix B - Hardware](#)). Different talkers are provided for the two interrupt methods, ones with the extension .BOO are designed to work with the SCI interrupt, ones with .XOO are for XIRQ\.

The talker may be located either in internal RAM or in EEPROM or even in external (expansion) memory provided that the reset vector can be pointed to the start of the talker. In the case of a RAM based talker, it is uploaded to the MCU each time the MCU is re-booted. A talker in external memory may require a code fragment to be boot-loaded to get it running, or it may always be present after a reset. The type of talker should be selected on the [Settings>Talkers](#) tab.

If the talker is to work correctly, some or all of the RESET, SCI, XIRQ and SWI vectors will need to point to talker code. The writing of these vectors is handled by JBug11 using information in [talker map files](#) - see page 18.

The talker resident on the MCU provides five basic communication functions:

1. Reading MCU memory
2. Writing MCU memory
3. Reading the CPU inherent registers (A, B, IX, IY, SP, PC, CCR)
4. Writing the CPU inherent registers
5. Servicing a software interrupt (SWI)

All the features of the monitor/debugger are provided by using one or more of these five functions. For example, the command to run a program in MCU memory at a particular address is implemented using the 'Write CPU Register' function with the program counter altered to the desired start address. The SWI Service function is used in tracing and debugging to signal to JBug11 that a break point has been reached.

The talkers supplied with JBug11 are identical to the ones supplied by Motorola for use with their PCbug11. The file Jbug_Talk.asm included in the distribution .zip file has some additional comments and background information on the working of the talkers, and may be re-assembled for different base addresses. **Note:** The talker files all have \$FF as their first byte - this is required by the MCU bootloading firmware to determine which baud rate is being used (see [Activating the Talker](#) below). If you re-assemble a talker and generate an object output, don't forget to add the \$FF byte if your assembler cannot do this automatically.

3.2 Talker Overlay Files

The basic RAM-resident talker is able to read all types of memory, both on-chip and external, but can only write on-chip or external (expansion) RAM.

To write to on-chip EEPROM, to program on-chip EPROM (OTP ROM) and to write to external memory which requires special writing procedures, such as Flash PROM, supplementary files are used to overlay the standard talker. These overlays are loaded and unloaded automatically by JBug11 as and when necessary. Note that writing which requires overlays is slower than writing to RAM, so external memory should always be declared as RAM if at all possible.

When loaded, they overwrite the functions of the talker which read and write the inherent registers and which handle breakpoints; none of these functions are needed simultaneously with the reading and writing of memory. A separate talker is needed for each of the following kinds of memory, which correspond to the available memory types on the Settings>Memory tab:

- On-chip EEPROM (\$02)
- On-chip EPROM (One-time programmable ROM) (\$20)
- External Byte-programmable memory (maybe EEPROM) (\$42)
(Note that external RAM does NOT need an overlay.)
- External Page-programmable memory (for example: FLASH)
and certain EEPROM where writes are constrained to lie within
one page at a time (\$22)
- Indirect Memory Access:
 - Read Indirect Memory Register (\$31)
 - Write Indirect Memory Register (\$32)
 - Read Indirect Memory (\$33)
 - Write Indirect Memory (\$34)

Each of these overlays has a unique JBug11 command byte, which is shown in brackets in the above list. Note that a 'talker in other memory' (see [Talkers](#) above) is expected to incorporate all the code which, in a boot-loaded talker, has to go into overlays; within such a talker the various sections of code may be selected by the unique control bytes.

Overlay files are in standard Motorola S19 record format. In the distribution they are called something like 'Ovly_Eeprom_A.rec'. The source files are also included, and the comments in these provide an insight into their operation. Select appropriate overlays on the [Settings>Overlays](#) tab.

If you wish to try your hand at adding a new MCU to JBug11's repertoire, you may need to write your own overlays: if you use external memory with a special write requirement, you will certainly have to modify the basic overlay source code provided and re-assemble it.

Overlays need to follow the format shown below in order to interact correctly with JBug11.

1. The S19 file containing the overlay has first to alter the bytes at \$003E..\$0041 in the talker (see the source file JBug_Talk.asm), to jump to the start of the overlay, and then to fill the space from \$0075 onwards with overlay code. The basic sending and receiving routines in the standard talker are not overwritten by the overlay, and these routines remain responsible for certain initial actions each time the overlay is called:

2. Reception of the command byte (see above). The one's complement of this byte is then echoed by the talker back to the PC.
3. Reception of the byte containing the count of the number of bytes to be written. A value of \$00 will expect 256 bytes. This byte is not echoed.
4. Reception of the address in memory at which to begin writing bytes, high byte first, followed by low byte. These bytes are not echoed.

On entry to the overlay routine, ACCA holds the one's complement of the command byte, ACCB holds the count of bytes to be written, and IX the address of the first memory location to be written. So the overlay code has then to perform the following:

1. Check that the one's complement of the command byte is correct
2. Do any pre-processing necessary
3. Wait for the first byte (new memory value) to be sent from the host
4. Echo the written byte back to the host.
5. The host PC will then send the remaining bytes, optionally waiting for an echo before sending the next.
6. Do any post-processing necessary

The overlay for page-written memory can cope with two cases, how it is used depends upon the code in JBug11:

- Memory such as FLASH where complete pages must be written each time a write takes place, and
- Memory such as EEPROM where bytes can be written individually, but if more than one byte is written in any one write operation, then all the bytes must lie within a single memory page.

The overlay must store the bytes to be written in a RAM buffer, and echo them back to the host immediately. This means that the echo is only of the buffer contents, not of the memory as programmed, so a subsequent [Verify](#) operation is desirable.

As noted near the beginning of this section, JBug11 automatically handles the substitution of the overlay, and the reinstatement of the talker afterwards. The overlay, being an S19 record format file, may alter any bytes, not necessarily at contiguous addresses. When about to substitute an overlay, JBug11 uses the information in the overlay S19 file to read in from the MCU the bytes at the addresses which will be overlain. These bytes it stores internally (actually as another S19 format file) so that when the overlay is no longer needed the original memory may be restored.

3.3 Activating the Talker

The talker will usually be located either in RAM or in EEPROM. In both cases the MCU must be wired to come out of reset in Special Bootstrap Mode, i.e. with the MODA and MODB pins at logic

zero. In this mode, the CPU of the MCU runs a bootloader program, located in mask ROM, which sets up the MCU serial communications interface (SCI) and then emits a break character on the Transmit Data line (pin PD1). What happens next depends on how the host PC replies.

Host sends \$FF character

This is the signal that some form of boot-loading talker is about to be sent, which the MCU will store in its onboard RAM. The \$FF character establishes the bootstrap communication baud rate (see Freescale documentation, particularly Application Note AN1060, for a description of how this happens). The talker program may be of any length, up to 512 bytes, but note that for 'A' series (and the E2) chips it needs to be exactly 257 bytes long (including the \$FF). After sending the talker, the MCU bootloading firmware executes a jump to address \$0000 and the talker takes over.

Host replies with a break character

Sending the break character, instead of \$FF, causes the MCU to branch to the first byte of EEPROM. Provided a suitable talker has been loaded starting at this address, it will assume control.

Activating a Talker Without Break Hand-shake

It is possible to send a boot-loading talker, or activate an Eeprom-resident talker, without relying on the MCU to output a 'break' character. This option may be selected in [Settings>General](#). The disadvantage of this option is that the user must remember to reset the MCU and then re-boot the talker in the right sequence, although if remote reset circuitry is in place, single-click rebooting is still possible.

Note

You may find that it is difficult to upload a boot-loading talker if your computer is busy doing something else which is CPU-intensive. For example, it may be impossible to get the talker to upload if your PC is simultaneously downloading a file from the internet. This happens when the pre-emptive multitasking in Windows takes control away from JBug11 for long enough that the MCU thinks that the talker uploading is complete before it really is, because more than 4 byte times (at the upload baud rate) have been allowed to elapse while the operating system has assigned CPU time elsewhere. This effect is more noticeable at higher upload baud rates.

Talker in external (expansion) memory

If the talker is located in external memory, it may be present immediately after reset, or it may require a small boot-loaded RAM-resident code fragment to start it. Such a fragment would be activated as noted above and would then jump to the start of the talker in external memory, possibly carrying out some initialization first.

3.4 Baud Rates

As a consequence of the boot loading firmware design in the HC11 series microprocessors, two different baud rates are required to communicate with an MCU, one to boot load the talker, and one to use subsequently for transfer of bytes to and from the MCU. These rates are selected on the [Settings>COM Port](#) tab.

For MCUs using an 8 MHz crystal (E-clock rate of 2 MHz), the default rate is 7812 baud for talker boot loading and the subsequent general communication rate is 9600 baud. Because 7812 is a non-standard rate (see below), the boot load firmware on the chip also allows a boot loading rate of 1200 baud. It is the function of the initial \$FF character in the talker to set which rate is adopted by the MCU - see [Activating the Talker](#) on page 16. At 1200 baud the talker takes a little over 5 seconds to load, so the default rate of 7812 would be preferable, if obtainable. The good news is that 7680 baud appears to be quite acceptable as a substitute for the 7812 baud specified. It demonstrates the extent to which Motorola went to make the SCI speed-tolerant.

If other crystal frequencies are used, these rates may be pro-rated, for example for a 7.3728 MHz crystal, the default upload speed would be 7200 baud - which is an exact integer division of 115200 ($\div 16$), and the communication speed would be 8861 baud ($115200 \div 13$). See also [Appendix D](#).

Note: not all baud rates are obtainable. On most PC systems the UART is a 16550 compatible device, and the COM port driver is the standard one that comes with Windows. The result is that the only baud rates obtainable in practice are those which are integer divisions of 115200 baud, for example 7680 baud is obtainable because $7680 = 115200 \div 15$. If a rate somewhere between the available rates is requested, then the 16550 defaults to the first available rate **above** the rate entered.

Note that the communication baud rate must be greater than, or (theoretically) equal to, the talker boot loading baud rate. This is because JBug11 opens the PC's COM port expecting reception at the general communication rate, and when the MCU emits the break character to initiate loading of the talker, it does so at the default upload rate - if this is less than the general communication rate then the host PC will definitely recognize it as a break signal. When about to send the talker, JBug11 switches to the slower boot loading rate, sends the talker, and then switches back to the general communication rate.

3.5 Talker Map Files

A suitable map file is always needed when tracing or running programs on an MCU, whatever kind of talker is in use, as it provides information to JBug11 on certain key addresses. Map files are plain text ASCII files, but they must be formatted in accordance with certain rules to work properly. A sample is shown below:

Listing of map file: Talk_A.MAP:

```
* Name of constant must be spelled correctly
* At least one space or tab must separate the name from the value
* Comments must begin with an asterisk in column 1, or a semicolon
*
talker_start      $0000 ; Talker code start address
talker_idle       $0012 ; Talker code idle loop address
swi_srv           $0094 ; Talker's SWI service address for break points
swi_jump          $00F5 ; SWI vector
illop_jump        $00F8 ; Illegal opcode vector
```

If you use your own talker, for example the one by Al Williams (see [Tracing in Eeprom](#) on page 25), or an EEPROM-resident one, then it is important to ensure that these constants have the correct values:

```
talker_start      Address to which the bootloader jumps after downloading the talker, if in
                  RAM, or on receipt of the break character, if in EEPROM.
```

<code>talker_idle</code>	Address of the idle loop in the talker which "does nothing" while waiting for the next instruction. The S (Stop) command writes this value to the program counter to force program execution into a known state.
<code>swi_srv</code>	Address of the SWI service routine within the talker, which is written to the SWI jump vector at <code>swi_jmp</code> before breakpoints can be handled.
<code>swi_jmp</code>	Pseudo vector location in RAM which stores the address of the SWI service routine. During running or tracing, the two bytes at this and the following address point to a location within the talker; otherwise they may point to a user's SWI service routine. If substitution of jumps is enabled (see Settings>Debug), JBug11 dynamically manages these bytes so that tracing through a user-placed SWI service routine is possible.
<code>illop_jmp</code>	Pseudo vector location in RAM which stores the address of an illegal opcode service routine. As part of the normal talker loading and initialization, these bytes are written to point to the start of the talker, so that an illegal opcode fetched by the CPU will cause the talker to re-start. However, if the user's code on the MCU incorporates a suitable service routine, this jump location may be overwritten by the user with the routine entry point, and then JBug11 will allow tracing through illegal opcodes. See the section Illegal opcodes in user code on page 26.

Future versions of JBug11 may introduce other constants for particular purposes.

4 MEMORY MANAGEMENT

4.1 Writing to MCU-Controlled RAM

The talker resident on the MCU is always able to write directly to on-chip RAM, which includes the control registers. It can also write to external RAM when in ‘Special Test Mode’; to put the MCU into this mode it is necessary to set the MDA bit in the HPRIO control register to 1 - see [Accessing Expansion Memory](#) on page 12.

Some forms of external EEPROM can also qualify as RAM, that is their bytes may be randomly accessed for writing, but a relatively long delay may be needed after each write cycle. This delay may be achieved by modifying and re-assembling the talker - see the assembly source file Jbug_Talk.asm included in the distribution and the example Part 1 Talker for the D0 MCU: ‘Talk1_D_MS11.asm’.

Important note: if your talker incorporates this delay, you must tick the ‘Wait for echo on write’ checkbox on the [Settings>Talkers](#) tab.

4.2 Writing On-chip EPROM and EEPROM

Writing to EEPROM and programming of EPROM memory provided in the 68HC11 series chips is carried out transparently by JBug11, subject to certain general conditions, as follows:

- A RAM based talker is running. Writing to EEPROM and EPROM cannot be carried out from a talker that is already resident in EEPROM or EPROM, although certain talkers, such as the one by Al Williams, which are mainly EEPROM-resident, can be used to write EEPROM (they have a small RAM-resident portion, see the section on [Tracing in EEPROM](#) on page 25)
- The appropriate talker overlay files are available
- The correct ranges are specified in [Settings>Memory](#)

EEPROM and EPROM memory writing is discussed further below.

4.3 Writing to EEPROM

Writing to EEPROM is controlled by the value in the BPROT register, and this must be set appropriately by the user in advance of any commands which alter EEPROM. JBug11 does not alter BPROT automatically, although it can warn the user if it detects an inappropriate value. See [Managing the BPROT Register](#) on page 21.

Because writing to EEPROM is carried out transparently by JBug11, all commands which involve altering MCU memory work in EEPROM the same as they do in RAM (but slower - see below). Even tracing is possible in EEPROM. The only exception to this is altering CONFIG, see the description of the [Alter CONFIG](#) command on page 59.

Writing to EEPROM is slower than writing to RAM because:

- The talker overlay file has to be loaded over the standard talker, and the standard talker has to be reinstated when writing to EEPROM is complete.

- To reduce the actual writing time as much as possible, and to keep wear and tear on the EEPROM memory to a minimum, the overlay implements the following strategy:
 - If the byte to be written to EEPROM is the same as the byte already at that address, no action is taken (no write is necessary)
 - If the byte then to be written is not currently \$FF, the byte will be erased first.
- Each byte written to previously-erased EEPROM takes approximately 11ms (the actual time for which the internal programming voltage is present is 10ms for an E-clock rate of 2MHz), and each byte written to an EEPROM address containing a value other than \$FF takes double this time. For clock crystals other than 8 MHz, the correct programming delay can be achieved by amending the overlay source code file and re-assembling it.

4.4 Managing the BPROT Register

On most chips in the 68HC11 series, a BPROT register provides some protection against accidental erasure or change of EEPROM contents. It is the user's responsibility to write the BPROT register to an appropriate value before attempting a command to modify EEPROM. As an example, the following macro will un-protect EEPROM, erase all the EEPROM locations, re-protect it, verify that the memory is erased, and finally list the contents to allow a visual check:

```
DEFM      EraseAll
BEGIN
    R BPROT=10
    EBULK
    R BPROT=1F
    VE B600 B7FF      ; Address range to suit E1 and E9 chips
    L  B600 B7FF
END
```

Provided that 'Show warning on BPROT bits set' is checked on the [Settings>General](#) tab, JBug11 will warn the user if it is about to write to CONFIG or EEPROM and it appears that the value in BPROT will not allow this to happen. JBug11 checks the PTCON bit in BPROT when about to write to CONFIG. When about to write to the main block of EEPROM, the check is that all four low-order bits of BPROT are zero (i.e. BPROT = \$10 or \$00). This may be unnecessarily restrictive, as these bits control different parts of the block of EEPROM - when the warning appears, the user is given the option of aborting the operation or carrying on anyway. If the warning is turned off on the Settings tab, then JBug11 will attempt to carry out the specified operation without checking BPROT at all. Where no BPROT register is provided, as in the A series chips, no warning message is shown.

4.5 EEPROM mapping

On most chips in the 68HC11 series, the EEPROM is at a fixed location (\$B600 - \$B7FF), but on the 68HC811E2 there is more EEPROM, and it may be located at the top end of any one of the sixteen 4 KB pages in the 64 KB address space, depending on the values programmed into the upper four bits of the CONFIG register. It is the user's responsibility to ensure that a valid address range is specified in the EEPROM window of the Settings>Memory dialog. This address range is used by JBug11 to validate addresses supplied as arguments to commands; and to provide the address required internally by the [Bulk Erase EEPROM](#) command (page 63).

4.6 Writing to EPROM

Programming EPROM memory, where present, is carried out transparently by JBug11 provided that the correct EPROM range is defined in Settings>Memory and that a suitable programming voltage is available. The LD(Load) command works in EPROM the same as it does in RAM, but only provided that the locations to be written are \$FF before writing. [D\(Duplicate\)](#), [F\(Fill\)](#), [M\(Modify Memory\)](#) and [T\(Trace\)](#) commands will not work in EPROM.

Programming EPROM memory areas is slower than writing to RAM. The actual time for which Vpp is present is 3ms per byte with a 2MHz E-clock rate (8 MHz crystal), although this can be altered if you re-assemble the appropriate talker overlay file: Ovly_Eprom_X.asm.

JBug11 can optionally display a warning when about to program EPROM so that the operator can check that the programming voltage, Vpp, is present on the chip. At the end of the programming operation, another warning dialog alerts the operator that the voltage may be disconnected. These warnings may be enabled or disabled on the [Settings>General](#) tab.

The general-purpose switch output may be used to automatically switch the programming voltage, Vpp, on and off, using a macro, and provided suitable hardware is available on the target board - see [Appendix B - Hardware](#). For example, the following macro, which takes a filename as the single replaceable parameter, will control the application of Vpp:

```
DEFM ProgEprom @1 ; @1 is an S19 file covering EPROM addresses
BEGIN
    Switch ON
    Pause 100      ; Wait for Vpp to stabilize
    LD @1
    Switch OFF
END
```

Most chips in the 68HC11 series that have an EPROM area use the ELAT bit in the PPROG register to switch EPROM in and out of the notional on-board programming socket, but the 711E20 and K series chips have a separate control register, EPROG, for this job. If you are using one of the latter chips, you will need to make sure that you have nominated the correct talker overlay file in Settings>Overlays>'On-chip OTP ROM' overlay.

Note: the EEPROM and EPROM overlay files may need re-assembling if MCU crystal frequencies other than 8 MHz are used, so that the time delays specified in the data sheets are maintained.

MC68HC711D3

The MC68HC711D3 has too little RAM to be able to use a full talker that can run or trace programs, but it will accommodate a cut-down talker and overlay so that the EPROM may be programmed even when expansion memory is not available. There is, in fact, a ROM-resident routine on this chip for programming EPROM, but it never returns to the talker, the only way to regain control being to reset the MCU when programming is finished. The JBug11 overlay does not make use of this built-in routine, and so is able to exit gracefully back to the talker when programming is finished. As supplied, the overlay is suitable for an MCU running off an 8 MHz crystal and it will need reassembling with a different time delay factor if another frequency is used.

4.7 Writing External Memory

JBug11 can write to most types of external (expansion) memory. Writing to external RAM is as straightforward as writing to on-chip RAM and is carried out by the routines built into the basic talker. Writing to some other types can be more involved, and in these cases, JBug11 uses talker overlays which broadly serve two types of memory:

- External Byte-written Example: Certain kinds of EEPROM, and
- External Page-written Example: FLASH or EEPROM

Because of the wide number of possible memory architectures, the overlays for writing to these kinds of memory will have to be written and assembled by the user, although two example overlay source files are included in the \Overlays\ subdirectory. Guidelines are also provided in the section [Talker Overlay Files](#) on page 15.

The JBug11 interface to the overlay that handles page-written memory is 'smart' to the extent that if the data to be written does not coincide with the data page boundaries, then the current content of the memory is read and re-written as necessary to make up a full data page. Only data page sizes between \$10 (16 decimal) and \$100 (256 decimal) are available.

4.8 Accessing Indirect Memory

As of Version 5.1.0, JBug11 can also read and write some kinds of indirectly-connected memory such as serial Eeprom. Only a limited set of commands is provided - see [Indirect Memory Commands](#) on page 85. Both reading and writing this kind of memory requires the use of overlays, and provision has been made for two of these, one for accessing serial memory registers, and one for the serial memory itself.

Because of the wide number of possible memory architectures, the overlays for reading and writing these kinds of memory will have to be written and assembled by the user, although two example overlay source files are included in the \Overlays\ subdirectory, suitable for accessing the Microchip 25LC640 64K SPI Bus serial eeprom. See also the section [Talker Overlay Files](#) on page 15.

Writing to serial Eeprom memory is usually constrained to a maximum of a 'page' at a time. Only data page sizes between \$08 (8 decimal) and \$100 (256 decimal) are available.

All settings for accessing indirect memory are grouped together on the [Settings>Ind Mem](#) tab.

4.9 Memory Map Display

This window displays the memory map for the currently selected MCU. It reflects the data supplied in [Settings>Memory](#).

The map may be sorted with the 'sort' button either in ascending or descending order.

This window is launched from the View menu, or by the keyboard shortcut Ctrl+M.

5 DEBUGGING

5.1 Breakpoints

Tracing, and running programs to breakpoints, requires that breakpoints can be established at the right places in the MCU program code. The 68HC11 series of microcontrollers do not have a dedicated breakpoint register, so break points are implemented by substituting an SWI instruction (\$3F) for the opcode at which a break is desired. This means that breakpoints can only be used where memory is modifiable i.e. in RAM or EEPROM. Tracing proceeds a bit slower in EEPROM because of the overhead needed to handle EEPROM writing. It should be noted that breakpoints cannot be placed, nor tracing carried out, in ROM or external Flash memory.

Three types of breakpoint may be set:

- **Transient:** Transient breakpoints only exist until the program reaches them, after which they are deleted.
- **Fixed:** Fixed break points are permanent in the sense that if the MCU program halts at one of them, and is then resumed from that point, the break point is restored 'behind' the continued program. This requires some processing overhead and will cause a slight delay in execution time as additional temporary breakpoints are automatically inserted and deleted.
- **Pass:** Pass breakpoints are similar to fixed ones except that MCU program execution does not halt until the breakpoint address has been reached the number of times specified in a 'trigger' value. Pass points take a considerable processing overhead, for example, inserting one in a timing loop would render the loop useless for its original timing purpose, although it may well help to debug it. To reset the trigger value to zero, use the [Reset Pass Count](#) command (page 74) or the context menu in the [Start/Break Points](#) display window (page 31).

The [Trace](#) and [Step Over](#) commands automatically place a breakpoint as necessary to 'catch' the program after execution of the current instruction. If this happens to be a branch instruction, JBug11 places the next breakpoint by an internal analysis of the CCR register or by additional reads of memory.

Breakpoints remain set in MCU memory while running or tracing, except at the current stopping point (otherwise the program would break again as soon as it were set running). All breakpoints are cleared from memory when the [Stop](#) command is given.

5.2 Setting and Clearing Breakpoints

A [transient](#) breakpoint or a [fixed](#) breakpoint may be set:

- as part of the [Go \(Run\)](#) command, or
- by the [Set Breakpoints](#) command.
- by using a right-click context menu within the break point display window, (part of the [Information Sidebar](#), see page 30).

A [pass](#) breakpoint can be set via the context menu or the [Set Pass Breakpoint](#) command (page 56).

Breakpoints may be cleared by using the right-click context menu in the [breakpoint](#) window (page 31), or by the [Clear Breakpoints](#) command (page 57). The [Trace](#) command automatically sets and clears transient breakpoints as necessary.

To reset the pass counts of Fixed and Pass breakpoints, use the [Reset Pass Count](#) command (page 74), or the context menu in the 'Start/Break Points' display.

Note that setting a breakpoint so that it appears in the breakpoint display does not mean that the breakpoint has been set in MCU-controlled memory as the \$3F SWI character. Modification of memory only happens when a Go (Run), Trace or Step Over command is issued, and memory is restored to its original contents when the Stop command is given.

If an attempt is made to close the JBug11 program while breakpoints possibly remain set in memory, a warning dialog is shown.

5.3 Tracing in EEPROM

If breakpoints are to be set in EEPROM, make sure that the BPROT register, where one is provided, contains a suitable value. Tracing cannot normally be carried out in EEPROM unless a RAM based talker is running. However, users of the 811E2 chip might like to check out the talker by Al Williams (go to: <http://www.wd5gnr.com/hc11.htm>). This talker is mainly in EEPROM, but copies a small section of code to RAM on start-up which allows it to write EEPROM space transparently to the user as though it was all RAM. Using this talker, what is actually EEPROM in the chip must be nominated as RAM in [Settings>Memory](#).

5.4 SWI in user code

The Software Interrupt instruction is allowed in user's own code on the MCU. Tracing through such an SWI is allowed in the right circumstances:

1. The user code has a suitable service routine, and
2. The entry point of this routine is written to the SWI pseudo-vector jump at \$00F5/6, and
3. Vector substitution is enabled in [Settings>Debug](#).

If the above conditions are not met, and a user-placed SWI opcode is encountered while running or tracing, JBug11 reports with an error message.

Note that the vector at \$00F5/6 must be written to point to the user's service routine in some way before running or tracing is begun, for example by writing it as part of the loading of the program. As part of the Run and Trace commands, JBug11 reads the value of the SWI vector (this is why the constant `swi_jmp` is needed in the map file) and notes this value internally before substituting its own SWI service routine (the constant `swi_srv`). Executing the Stop command restores the user's SWI vector address. Breakpoints may be placed at user-programmed SWI's, and tracing through such SWI's is possible. Tracing through user-placed SWI's only works reliably with an .XOO type talker and the PD0-XIRQ\ connection (see [Appendix B - Hardware](#)).

5.5 Illegal opcodes in user code

Illegal opcodes are sometimes inserted deliberately in user code, for example in some real-time operating systems. Tracing through an illegal opcode is allowed in the right circumstances:

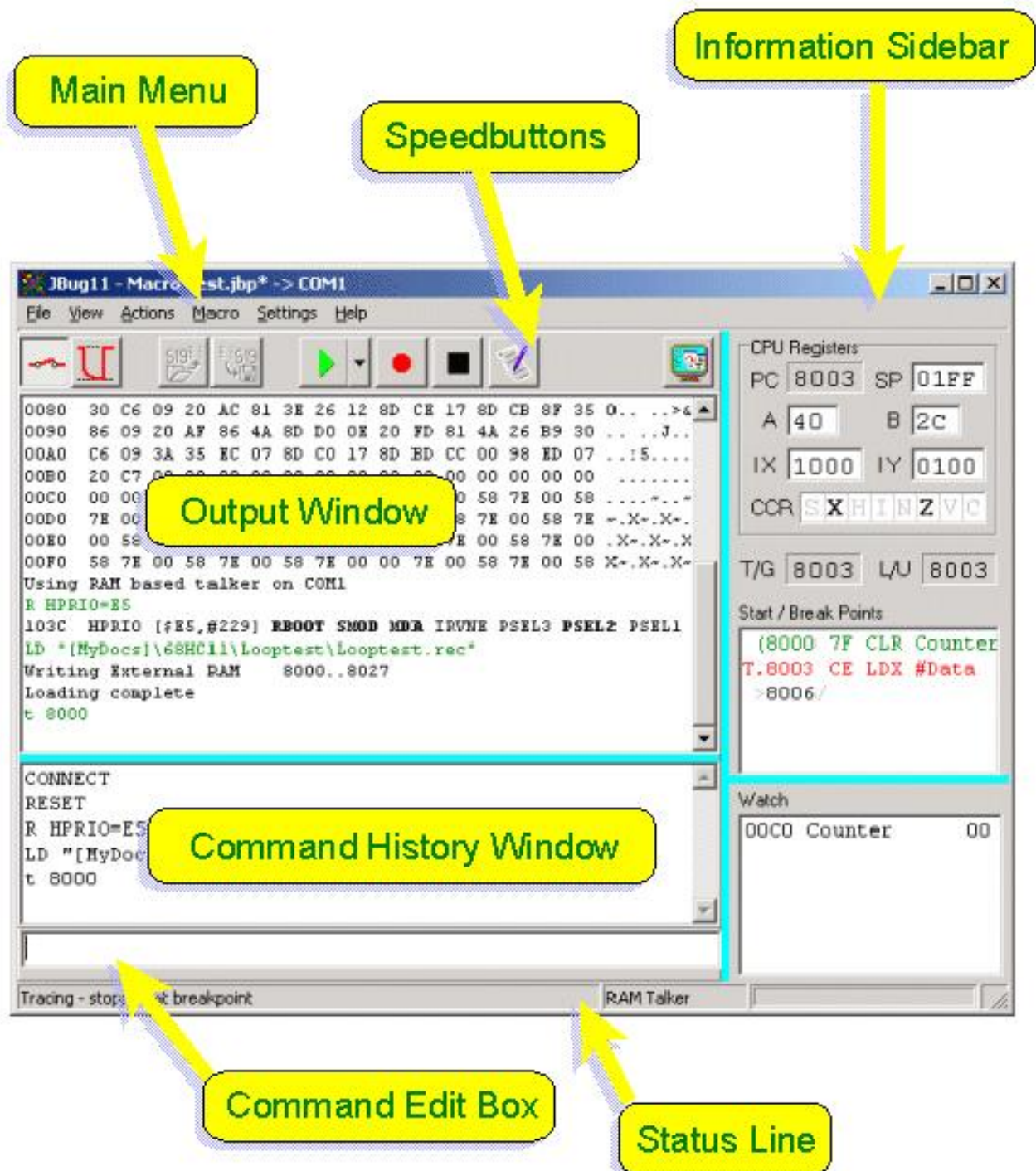
1. The user code has a suitable service routine, and
2. The entry point of this routine is written to the illegal opcode trap pseudo-vector jump at \$00F8/9, and
3. Vector substitution is enabled in [Settings>Debug](#).

As part of the Run and Trace commands, JBug11 will read the value of the vector (this is why the constant `ilop_jump` is in the map file) and notes this value internally so that when an illegal opcode is encountered during tracing, the action of the MCU on fetching an illegal opcode can be simulated. Transient, Fixed and Pass type breakpoints may all be set at an illegal opcode.

As with the SWI in User Code described above, the pseudo-vector jump to the user's service routine must be written before running or tracing begins.

6 PROGRAM FEATURES

6.1 Screen Layout



6.2 Main Menu

The main menu provides access the various facilities in JBug11. For more details see the individual menu descriptions later in this chapter.

6.3 Speedbuttons

Nine speedbuttons are provided along the top of the main form to carry out common tasks. From left to right these are:

Connect	Click to carry out the Connect / Disconnect command (page 60), i.e. open or close the serial (COM) port.
Reset	Click to carry out the Reset command (page 74). This button will be grayed-out if remote resetting is not available.
Load S19 File	Click to carry out the Load Memory command (page 69). This button is grayed-out unless a talker is loaded, and the status line reads 'Stopped'.
Save MCU Memory	Click to carry out the Save Memory command (page 75). Grayed-out as above.
Play Macro	Click to play the last macro selected from the drop-down list. If a macro is already playing, click this button again to stop it (or use the Stop button).
Record Macro	Click to begin recording a macro. If a macro is being recorded, click this button again to stop recording (or use the Stop button).
Stop	Click to stop a macro playing or recording.
Edit Macro	Click to open the Macro Editor . (See page 90)
Switch	Click to operate the remote switch, see General-Purpose Switching on page 12. This button is only available if remote mode switching is enabled - and this requires the necessary hardware on the target board.

6.4 Output Window

Displays commands and their results. Also miscellaneous information, such as diagnostic information following communication errors.

The division between the output and command history windows is a moveable splitter bar, allowing one window to be enlarged or reduced at the expense of the other. Word wrap may be enabled in this window.

A right-click context menu is available within the Output Window. This provides:

- Select All
- Copy
- Clear All
- Word Wrap

6.5 Command History Window

Displays a list of all the commands issued during the use of the program.

A command shown here may be re-used by:

- clicking on it, whereupon it will replace whatever currently appears in the [Command Edit Box](#), or by
- double clicking, when it will be executed immediately, or by
- using the up and down arrow keys while the Command Edit Box has focus; this will recall a previous command in the same way that was possible in PCbug11.

Note that if a previously issued command is selected, it will be transferred to the Command Edit Box without any comment that may have been added (see [Commands](#) on page 52 for an explanation of the addition of comments).

A right-click context menu is available within the Command History Window. This provides:

Select All

Copy

Copy to Macro Editor Click to append any selected text to the current macro library, and open the [Macro Editor](#) (see page 90).

Help on Error Message If a previous command has resulted in an error message in the Command History Window, of the type that begins with an arrow: <--, and this line has been highlighted by left-clicking on it, then this menu item will open on-line help at the relevant topic. This menu item is not available if more than one line has been highlighted or the line does not have an error.

6.6 Command Edit Box

This is a standard Windows edit box for the typing-in of [commands](#). It is designed to be the main point of user interaction with JBug11. Pressing the Enter or Return key while this box has focus initiates the following events:

1. The command is parsed and checked for syntactical correctness. If incorrect, an explanatory message is appended to the command.
2. The command is copied to the command history list and to the output window, where it appears in green type.
3. If the command is correctly formatted, it is executed and the results, if any, will appear in the Output Window and in the [Information Sidebar](#) displays, as appropriate.

Pressing the escape key while this box has focus will clear its contents, and stop any macro playing or recording.

6.7 Status Line

This is divided into three panels as follows (from the left):

1. General program state

This panel will display one of the following messages depending on the currently executing command:

Invalid project data (check "Settings")
Disconnected
Connected
Stopped
Running
Running - stopped at breakpoint
Tracing
Tracing - stopped at breakpoint

2. Info

This panel shows which type of talker is currently running, RAM-resident, EEPROM-resident or in ROM/expansion memory

3. Progress Bar

This is a standard windows progress bar which shows the progress of operations that involve the transfer of bytes to or from the MCU.

6.8 Information Sidebar

The right-hand side of the JBug11 main form is given over to a display of various items of information useful when tracing and running programs on the MCU. Resize this area using the vertical splitter bar. From top to bottom, information is displayed as follows:

6.8.1 CPU Registers Display

When stopped, or stopped at a breakpoint, this panel shows the values in the CPU registers. The values for SP, A, B, IX and IY may be edited directly in their respective boxes, and the new value will take effect when running or tracing recommences. A flag in the CCR may be edited by double-clicking it, when it will change sign. The program counter PC is not editable; the only way to change PC is to give a Go(Run) or Trace command with the desired starting value.

6.8.2 T/G and L/U addresses

These are two addresses maintained internally by JBug11:

- The T/G address show where Tracing and Running will next begin in the absence of a specific address.
- The L/U address shows where Listing or Unassembling will begin in the absence of a specific address.

Suppose a file to be loaded at \$8000 and the following Trace command given:

T 8000

then tracing will commence at 8000 (hex), and when tracing stops at the next breakpoint, the value of the T/G address will be updated to show the break point address. Simply pressing the <Enter> key will then trace the next instruction. The L/U address also tracks the breakpoint, so if the program were paused at a breakpoint, say \$8003, then issuing the command:

U3

would unassemble the next three instructions, altering L/U to the address immediately after the operands of the third instruction, but leaving T/G unchanged, so that typing

T

would resume tracing where it was previously left off.

6.8.3 Start / Break Points

Displays breakpoint information. When running or tracing, the top line in this window shows the instruction at the starting address in brackets and in green, followed by a list of the currently set break points. Transient, Fixed and Pass breakpoints are prefixed with T, X and P respectively. If the MCU is halted at one of the listed break points, then that line is displayed in red type, followed by a line showing where the program would go next if another trace command is given. If halted at a branch instruction, then this line will display two addresses, the sequential one first and then the branch address. Whichever is the current destination will be highlighted in bold.

Following **fixed** breakpoints is a figure in brackets which shows the number of times program execution has passed this point. Following **pass** breakpoints, two figures appear, the first as for fixed breakpoints, and the second, after the slash, shows the 'trigger' value.

A right-click context menu is available in this display, with the following items:

Add...	Opens a dialog for the addition of breakpoints.
Delete	Deletes a selected breakpoint
Clear All	Deletes all breakpoints
Reset Pass Counts	Resets all the pass counts of Fixed and Pass breakpoints to zero.
Sorted By	Submenu allows the display to be sorted by: <ul style="list-style-type: none">• Type (Transient, Fixed and Pass),• Address• the order in which they were generated and added to the list.

This sorting order is remembered in the Windows Registry between sessions.

6.8.4 Watch Window

Displays the value of selected memory locations during the running and tracing of programs. The items in the watch window are automatically updated when a running program stops at a breakpoint, or when the [Stop](#) command (page 76) is issued.

Items in this display may be re-ordered by dragging and dropping.

A right-click context menu is available in this display, with the following items:

Refresh	Click to read the current value(s) at the address(es) and update the display in the watch window.
Add...	Click to open the Add to Watch dialog for adding an address to watch (see below).
Delete	Deletes an item selected in the Watch display.
Clear All	Deletes all the items in the display.
Undelete	Restores deleted items

Adjust the relative sizes of the Start/Break Point display and the Watch window with the moveable splitter bar. The addresses and labels in the Watch display are remembered in the project file between sessions.

6.9 Add to Watch

Dialog for adding items to the watch window, displayed from the right-click context menu in the Watch Window. This dialog allows you to add addresses in different categories, as follows:

User Defined

Type the address in MCU memory that you wish to watch in the 'Address' edit box (hexadecimal characters only), and add an optional description. Click 'Add' to add this item to the watch window, or 'Add & Close' to add the item and simultaneously close the 'Add to Watch' dialog.

IX Relative

Addresses relative to the IX index register may be watched. Type in an offset or use the drop-down list. The default offset number base is decimal, hex numbers may be entered with a \$ prefix. Zero is an acceptable offset, but not negative amounts.

IY Relative

As for IX Relative above, but in this case the offset is with respect to the IY register.

Go to Symbols/Registers

Click to close the dialog and open the [Symbol Table/Register Display](#). From this display, symbolic labels may be added to the watch window.

6.10 File Menu

Menu used to manage [project files](#).

New Project...	Begins a fresh project. If the current project has changed, the user is prompted to save it first, then the Settings Dialog opens with default values
Open Project...	Opens, and makes current, an existing project configuration. Keyboard shortcut: Ctrl+O
Save Project...	Saves the current configuration to a project file. If the file exists, it will be silently updated, if not, a file save dialog will appear. Keyboard shortcut: Ctrl+S
Save Project As...	Opens a file save dialog for the user to save the current configuration to a file with a new name. Keyboard shortcut: F12
Reopen...	The sub-menu displays a list of up to ten of the most recently used project files. The top entry in the list is grayed-out if it is the one currently open.
Import...	Open a dialog to Import configurations from a JBug11 version 4xx. This item is grayed out (unavailable) if no version 4xx configuration information is found in the Windows Registry.
Exit	Closes JBug11. If the current project configuration information has changed, the user is prompted to save it before closing. Similarly, if the Macro Editor has been in use, and the results not saved, the user will be reminded to save them.
Terminate	Terminates the program in circumstances where exit does not appear to work.

6.11 View Menu

Base Converter...	Opens the number Base Converter (page 37). Shortcut: Ctrl+K
Macro Editor...	Opens the Macro Editor (page 90). Shortcut: Ctrl+E
Symbol Table...	Opens the Symbol Table/Register Display (page 37) with the currently-loaded symbol table on display. Shortcut: Ctrl+L
MCU Registers...	Opens the Symbol Table/Register Display (page 37) with the MCU register list on display. Shortcut: Ctrl+R
Terminal...	Brings up the Terminal Window (page 92). Shortcut is Ctrl+T.
Memory Map...	Opens the Memory Map Display (page 23). Shortcut: Ctrl+M
Base Layout	Use the sub-menu to select a starting point layout for various screen resolutions. The actual layout is remembered in the Windows registry at the

end of each session, so the program should have the same on-screen appearance when you next start it up.

Font Sizes Chose a font size for the Output and Command History windows.

Colors Chose a color to enhance the display of bits that are 'set' in the output of the [Register Display and Change](#) command (page 73) .

6.12 Actions Menu

Menu to simplify the issuing of commands that involve the loading and saving of files to and from the MCU. Such commands all need a file name typed on the command line which is tiresome to do accurately. Using these menu items brings up a 'file open' or 'file save' dialog as appropriate, in which the user may select a file name in the usual way. Closing the dialog then fills in the command line and executes the command. Actions involving transfers to and from MCU-controlled memory will be grayed out (unavailable) unless a talker is loaded and the status line reads 'Stopped'.

Connect Executes the [Connect / Disconnect](#) command (page 60) to open the serial port (COM port).

Reset Executes the [Reset](#) command (page 74) to send a reset signal to the MCU. Note that this is only available if your hardware supports it, and the COM port is connected.

Load S19 MCU... Opens a dialog for the user to select an S19 format file for loading to the MCU. The [Load Memory](#) (page 69) command is then executed via the command line.

Save MCU to S19... Saves a block of MCU memory to a file in S19 format. An address input dialog will appear for the user to define the starting and ending addresses of the block to be saved, followed by a file-save dialog. The [Save Memory](#) command (page 75) is then executed via the command line.

Verify S19... Opens a dialog for the user to select an S19 format file for verifying MCU memory. The [Verify](#) command (page 83) is then executed via the command line.

CRC-16 MCU v. File.. Opens a dialog for the user to select an S19 format file against which to perform a CRC-16 checksum of MCU memory. The [Cyclic Redundancy Check](#) command (page 61) is then executed via the command line.

Load Binary to MCU... Opens a dialog for the user to select a binary image file (usual extension: .obj or .bin). A second dialog box appears to request a loading address. If a valid file is selected, the file is opened and loaded to MCU controlled memory via the Command edit box, as though the [Load Memory](#) command (page 69) had been issued.

Save MCU to Binary...	Saves a block of MCU-controlled memory to a binary image file. An address input dialog will appear for the user to define the starting and ending addresses of the block to be saved, followed by a file-save dialog. The Save Memory command (page 75) is then executed via the command line.
------------------------------	--

Local

Sub-menu allows the user to load, save and verify S19 files against [local memory](#). These commands do not require an MCU to be connected.

Load S19 to Local...	As though the 'LDL' command had been issued - see the Load Memory command
Save Local to S19...	As though the 'SVL' command had been issued - see the Save Memory command
Verify S19 v. Local...	As though the 'VL' command had been issued - see the Verify command
CRC-16 for File...	As though the 'CRCL' command had been issued - see the Cyclic Redundancy Check command
Load Binary to Local...	As though the 'LDL' command had been issued - see the Load Memory command
Save Local to Binary...	As though the 'SVL' command had been issued - see the Save Memory command

6.13 Macro Menu

Boot Script...	Open the Settings>Macros tab for editing the Boot Script (see page 87).
New...	Begin a new macro library script. If the currently open one has changed, the user will be prompted to save it first. The contents of the macro editor will be cleared and the editor window shown if not currently visible.
Open ...	Open a macro library file into the Macro Editor . The default file extension is '.mcr'. Shortcut is Ctrl+O.
Save	Save the contents of the macro editor to a macro library file. Shortcut is Ctrl+S.
Save As...	Save the contents of the macro editor to a new file, default extension '.mcr'.
Record	Start recording, adding new commands to the end of the current macro library.
Play	Choose a macro to play from the sub-menu.
Stop	Stop playing or recording.

Edit Open the [Macro Editor](#)

6.14 Settings Menu

Open a tab in the [Settings Dialog](#). Once the Settings dialog is open, any other tab may be selected.

6.15 Help Menu

Index	Opens Help Topics at the Index tab. Shortcut is F1.
Contents	Opens Help Topics at the Contents tab. Shortcut is Shift+F1.
Getting Started	Opens Help at 'Getting Started' - see Appendix G .
Error Report...	Opens the Error Report dialog for compiling a text file report on a problem encountered while using JBug11, see page 98 . Useful if email support is required.
About	Brings up the usual 'About' box.

6.16 Keyboard Shortcuts

Keyboard Shortcuts - Main Window Active

Ctrl+B	Reset the MCU. Useful if you have fitted the remote resetting hardware.
Ctrl+E	Open the Macro Editor
Ctrl+K	Opens the number Base Converter
Ctrl+L	Open the Symbol Table/Register Display with the currently-loaded symbol table on display.
Ctrl+M	Opens the Memory Map Display .
Ctrl+O	Open a project file
Ctrl+R	Opens the Symbol Table/Register Display with the MCU register list on display
Ctrl+S	Save the current project file
Ctrl+T	Open the Terminal Window .
F12	Save the current project file under a new name
Esc	Clear the command line (if it has focus). If a macro is playing, it will stop.
F1	Open on-line help at the 'Index' tab

Shift+F1 Open on-line help at the 'Contents' tab

Keyboard Shortcuts - Macro Editing Window Active

Ctrl+O Open a macro library file into the macro editor

Ctrl+S Save the current macro library file

Ctrl+C Copy to Clipboard

Ctrl+V Paste from Clipboard

Ctrl+X Cut to Clipboard

Ctrl+Y Delete the line containing the insertion point (caret).

Keyboard Shortcuts - Terminal Window Active

Ctrl+I Send a binary image file

Ctrl+Q Close the terminal window

6.17 Symbol Table/Register Display

Window to display the currently-loaded symbol table (see [Using a Symbol Table](#) on page 81), or the current MCU control registers.

Display this window by clicking the item 'Symbol Table' or 'MCU Registers' in the View menu, or use the keyboard shortcuts Ctrl+L or Ctrl+R.

- Select which kind of display you want with the buttons at the top of the form.
- Sort the list by address or name using the radio buttons at the foot of the window.
- Select one or more items in the list and click 'Add to Watch' to transfer them to the Watch window.

6.18 Base Converter

This window allows rapid conversion between hexadecimal, decimal and binary notations. Launch it from the view menu, or by using the keyboard shortcut Ctrl+K.

Entering a valid number in one of the boxes and pressing the enter key, or clicking the '=' button, will fill in the other two boxes accordingly. If the 'Auto completion' check box is ticked, the current hexadecimal number in the calculator will be appended to the text in the Command edit box, and focus will be transferred to that box. The converter form will not close, but will become inactive. If 'Auto completion' is unchecked, then clicking '=' will not affect the command line, and will leave the converter open and active.

The maximum value that the converter will handle is \$FFFF or 65535 decimal. Only valid hexadecimal characters may be entered in the 'Hexadecimal' edit box, similarly for the other two boxes.

Clicking the 'ASCII' button brings up a quick-reference table of ASCII characters (in the Arial font face). Double-clicking on a character in this table will transfer its hexadecimal value to the converter.

7 CONFIGURATION

7.1 Settings Dialog

The eight tabs of the Settings Dialog contain various editable fields for the customization of JBug11. All the configuration data on the tabs is stored in [project files](#) when the program closes, and the file name of the project file is itself stored in the Windows Registry, so that the settings are restored when JBug11 is started again. At the bottom of the dialog are three buttons:

OK Button

Click to close and save the data entered in the various tabs. The data is checked for consistency, and if incorrect or missing data is found, an error message will appear with diagnostic information.

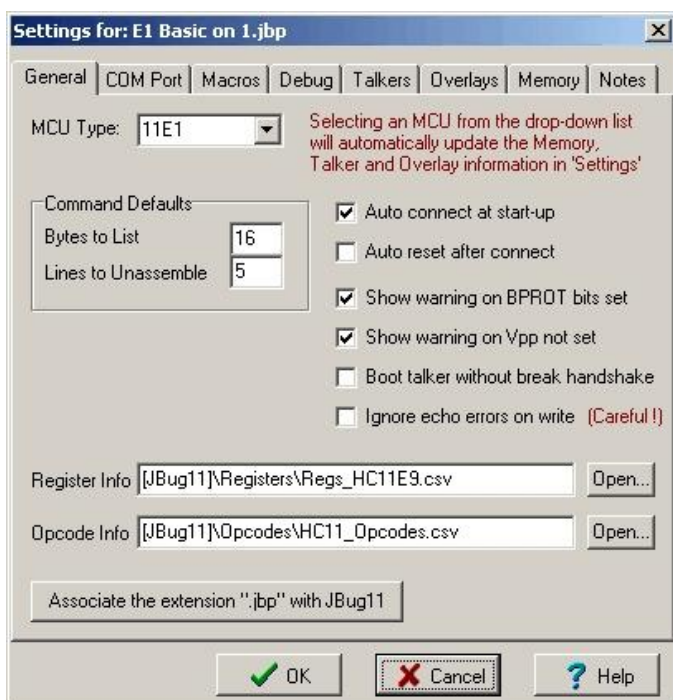
Cancel Button

Click to close Settings and discard any changes

Help Button

Click to open the on-line help topic corresponding to the currently visible tab.

7.2 Settings>General



Tab for miscellaneous, general settings.

MCU Type

Select the MCU that you are using from the drop-down list. This action will fill in all the relevant edit boxes in 'Settings' with the default information for this MCU. This information is taken from the file MCUData.cfg in the 'MCU' subdirectory of the installation folder. If the particular MCU that you are using is not in the drop-down list, then it may be added to this file - see [Appendix D](#).

Command Defaults

Bytes to List Sets the default value for the number of bytes to list if the [List](#) command (page 68) is issued with only a starting address.

Lines to Unassemble Sets the number of instructions to unassemble if the [Unassemble](#) command (page 80) is issued with only a starting address.

Auto Connect at start-up

Tick this box so that JBug11 automatically issues the Connect command when the program is launched - see the [Connect / Disconnect](#) command on page 60.

Auto Reset after connect

Tick this box so that JBug11 automatically issues the Reset command after connecting.

Show warning on BPROT bits set

Check this box to show a warning message if the user tries to program on-chip EEPROM (or the CONFIG register) while one or more of the BPRT bits remain set in the BPROT register, where such a register is provided.

Show warning on Vpp not set

Check this box to show a warning message when JBug11 is about to program EPROM so that the user is reminded to ensure that the programming voltage, Vpp, is present.

Boot talker without break handshake

Checking this box will by-pass the handshaking process whereby JBug11 waits for the MCU to transmit the 'break' character on its serial TxD line after a reset in special bootstrap mode, as the signal for a boot-loading program to be transmitted. Unless really necessary, I advise you to leave this box unchecked. It is provided because certain [USB-to-Serial adaptors](#) (page 13) appear to be unable to handle the 'break' character.

Ignore echo errors on write

Check this box to have JBug11 write MCU memory without checking that the bytes returned by the talker match the bytes that were sent. This might be useful, for example, if a device is wired to the MCU in such a way that its control registers occupy part of the MCU address space, but these registers have a different behavior when written and when read. I strongly advise you to leave this box unchecked unless you are certain it is necessary, as valuable feedback on the operation of JBug11 will be lost if there is no check on bytes written.

Register Info

Edit box for the file containing information on the MCU control registers. If necessary, use the 'Open...' button to browse for the correct file - see [Appendix D](#). This box is automatically updated when the drop-down list is used to select an MCU. File names may appear in this and the following edit boxes with a token replacing part of the path - see [Path Tokens](#) on page 53.

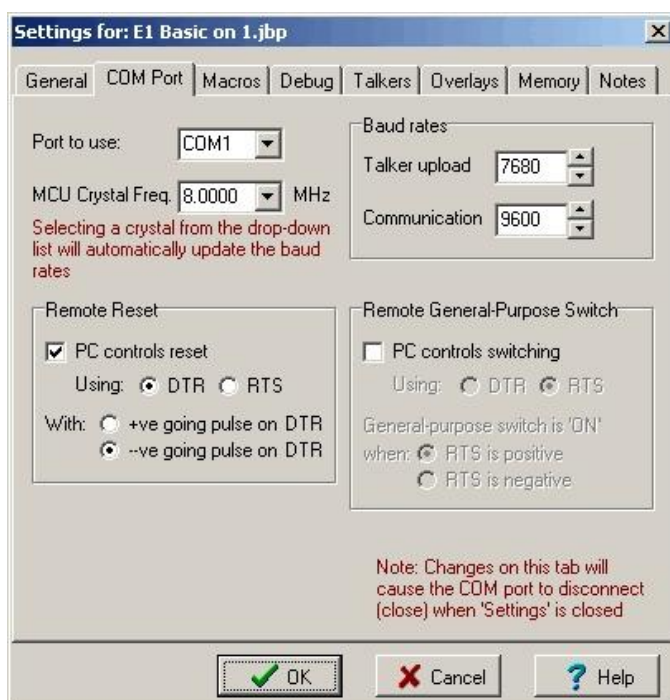
Opcode Info

Edit box for the file containing information on the MCU opcodes. Actually, there is only one opcode file for all varieties of HC11: 'HC11_Opcodes.csv'.

Associate the extension ".jbp" with JBug11

Click this button to do as its caption says. When JBug11 is first run, it automatically makes this association, but if there is subsequently a problem with file associations, the link between the .jbp file type and the JBug11 executable may be restored with this button.

7.3 Settings>COM Port



Tab for configuring the RS232 port which will be used to communicate with the target board. Note the caveats in [USB-to-Serial Adapters](#) on page 13.

Port to use:

Select an available COM port from the drop-down list. This combo box displays all the COM ports returned by the Windows 'Enumerate COM Ports' function.

MCU Crystal Freq.

Select the frequency of the crystal which controls the MCU. When the drop-down list closes the Talker Upload and Communication baud rates will be automatically updated. Because of the design of the bootloading firmware in the MCU, alternative rates are available for uploading the talker. Where such a rate is achievable by the PC host, this may be selected from the dialog that appears when the drop-down list closes.

Baud Rates

The two check boxes in this panel select the baud rate which JBug11 uses to boot load the talker (where you are using a RAM talker), and the rate used for subsequent general communication with the MCU thereafter. The only baud rates obtainable in practice are those which are integer divisions of 115200 baud, for example 7680 baud is obtainable because $7680 = 115200 \div 15$. If a rate somewhere between the available rates is entered in one of the edit boxes, then the UART in the PC defaults to the first available rate **above** the rate entered. Beside each edit box is a spin control which will automatically select the next higher or lower integer division of 115200.

For a general discussion of the necessary baud rates, see [Baud Rates](#) on page 17.

To set these rates automatically for a number of common crystal frequencies, select a crystal from the combo box - see previous item.

Remote Reset

The options on this panel control the signaling of a reset between the host PC and the target board. The RS232 communication ports on a PC have two available output pins (besides TxD): DTR (Data Terminal Ready) and RTS (Ready To Send). Either of these may be used to perform a reset remotely, provided the necessary hardware logic is in place (see [Appendix B - Hardware](#)).

PC controls reset

Tick this box if the host PC is able to control the reset cycle on the target board.

Using DTR/RTS

Select whichever signal is used for the remote re-setting.

With +ve/-ve going pulse

Select the pulse polarity. For the avoidance of doubt about the meaning of RS232 'asserted', etc. the descriptions +ve/-ve refer to the actual polarity of the pulse with respect to the common ground pin (pin 5 on the DB9 connector). Note that most RS232 level shifter chips on target boards invert this polarity, and also that RS232 line receivers usually bias their inputs so that a disconnected input appears to be at a negative voltage.

Remote General-Purpose Switch

With the necessary logic hardware on the target board, the PC can be used to activate a function of the user's choice, using whichever RS232 output signal is not already being used by the remote reset function. See [General-Purpose Switching](#) on page 12 and [Appendix B - Hardware](#) for more details.

PC controls switching

Tick this box if the host PC is able to toggle the output.

Using:

Select whichever RS232 output pin is to control the switching. If DTR is selected then the labels in this box will change to DTR rather than RTS.

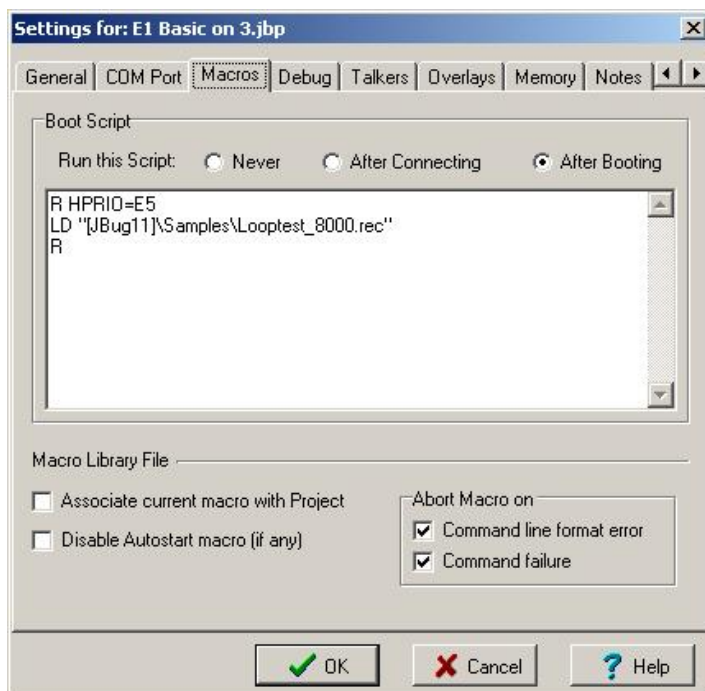
General-purpose switch is 'ON' when RTS is positive/RTS is negative

Choose the pin voltage level which corresponds to the 'ON' state.

Note

Note that changes made on this tab will cause the COM port to close (disconnect) if the Settings dialog is closed by clicking the 'OK' button.

7.4 Settings>Macros



Tab for editing the boot script and for settings affecting macro library files.

Boot Script

This box is provided to enter commands which can be executed every time a talker is loaded, or the COM port is opened. Select an appropriate radio button and enter the desired commands in the edit box below. See [Boot Script](#) on page 87 for more details.

Associate current macro with Project

Check this box to cause the current macro library file to be reloaded when the current project is reloaded. The current macro library file is whatever is currently open in the [Macro Editor](#) (page 90) when the project file is saved.

Disable Autostart macro (if any)

Check this box to prevent any macro named 'AUTOSTART' from running - see [Autostart Macro](#) on page 88 for details.

Abort Macro on

A playing macro can be arranged to stop automatically under certain conditions. These are:

Command line format error

Tick this box to cause a playing macro to stop if a command within the macro is formatted incorrectly, causing an error message beginning with the <-- symbol to appear in the Command History window.

Command failure

Tick this box to cause a playing macro to stop if a command within the macro has an unsuccessful result; for example the [Verify Erase](#) command (page 84) finds some un-erased memory.

7.5 Settings>Debug



Tab for configuring the behavior of JBug11 while running and tracing programs, including:

- How much information is displayed in the Output Window,

- How SWI instructions and illegal opcodes are treated, and
- The behavior when the [Step Over](#) command (page 78) is used.

UnAsm: add spaces at code breaks

Check this box to have JBug11 add a blank line in unassembled listings after the following instructions:

- Unconditional jumps, including the BRA (Branch Always) instruction. No space is left if the jump address is that of the immediately following instruction, i.e. the jump was included simply to use up processor cycles.
- Return from Interrupt, RTI
- Return from Subroutine, RTS

Such spaces can improve the readability of disassembled code by highlighting the places where sequential code execution is not possible.

Disable vector substitution

Check this box to prevent JBug11 substituting the SWI and Illegal Opcode pseudo-vectors prior to running or tracing programs. If this box is ticked, it is not possible to trace through an [SWI in user code](#) (page 25) or [Illegal opcodes in user code](#) (page 26) even where the user has provided his own service routines. The only occasion on which it is useful to check this box is when the vectors are in non-modifiable memory. This checkbox is grayed-out (unavailable) if you are in the middle of tracing or running a program.

Do not stop at breakpoints within 'stepped-over' routines

When using the [Step Over](#) command (page 78) to step over a subroutine which itself contains previously set breakpoints, JBug11 can be customized to halt at, or ignore, such breakpoints. Note that, irrespective of the selection in this checkbox, JBug11 will still update the pass counts of Fixed and Pass breakpoints in a 'stepped-over' routine. This checkbox is grayed-out (unavailable) if you are in the middle of tracing or running a program.

Run/Trace Display in O.W.

This group of checkboxes governs the extent of the information provided in the output window during running and tracing.

Instruction at Start : check this box to display a disassembly of the CPU instruction at the address at which running or tracing started.

Registers at Start: check this box to add a display of the CPU inherent registers at the start point.

Show starting info only once: with repetitive tracing there is no need to display the starting **and** break information, since the break information at one step will be the same as the starting information for the next step. Check this box to suppress the additional information.

- Instruction at Break:** check this box to display a disassembly of the CPU instruction at the breakpoint address.
- Registers at Break:** check this box to add a display of the CPU inherent registers at the breakpoint.

Symbol Table

When tracing or disassembling code, symbolic labels may optionally be displayed. This panel controls the display, and the location of the symbol table from which the information is taken. See [Using a Symbol Table](#) on page 81 for more details.

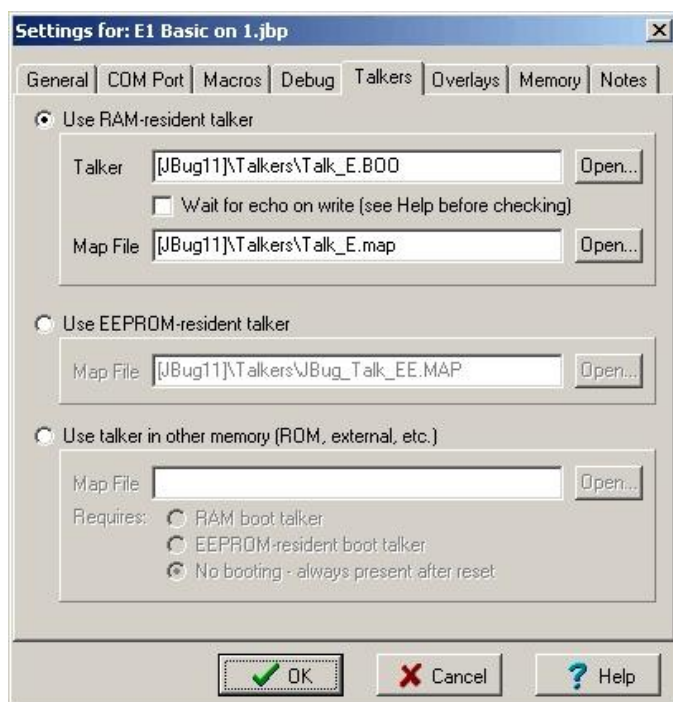
Do not use Symbolic labels will not be added to traced or disassembled code.

As last loaded S19 file (plus filename, if one is loaded)

Select this option if symbolic label information is to be taken from a symbol table generated alongside an S19 file. If selected, then every time an S19 record file is loaded to memory, JBug11 will look to load a symbol table of the same name but '.sym' extension.

From file: Select this option if symbolic label information is to be taken from a fixed location. If so, choose a file in the edit box. File names may appear in this edit box with a token replacing part of the path - see [Path Tokens](#) on page 53.

7.6 Settings>Talkers



Tab for selecting a *talker* for JBug11 to use for communicating with the target MCU. See [Talkers](#) on page 14. This talker may reside in RAM, and be loaded each time the MCU is reset, or it may be in MCU EEPROM, or in some other memory. Use this tab to select the talker type. Note that file names

may appear in these edit boxes with a token replacing part of the path - see the section on [Path Tokens](#) on page 53. The easy way to select the correct files is to use the combo box on the 'General' tab to select an MCU - when the drop-down list closes, these file names are updated.

Use RAM-resident talker

Check this radio button to select a talker which will be reloaded each time the MCU is reset. Then choose the appropriate files. As a minimum you will need to select a Talker and a Map File. If you wish to program memory other than RAM, for example on-chip EPROM, then you will also need to select overlay files - see [Settings>Overlays](#).

The checkbox 'Wait for echo on write' affects the way JBug11 interacts with the 'TWritMem' subroutine in the talker (see a typical talker .asm file such as JBug_Talk.asm in the \Talkers\ sub-folder). This routine is used to write on-chip or external RAM. When this checkbox is checked, JBug11 waits for the echoed byte before writing the next byte, when unchecked JBug11 sends all the bytes then reads the COM port received-data buffer to check that the bytes were correctly sent. The latter method is faster, but depends on the talker being able to write and read memory within the time it takes to transmit a byte. If you modify the talker to use the EEPROM write delay routine, then check this box.

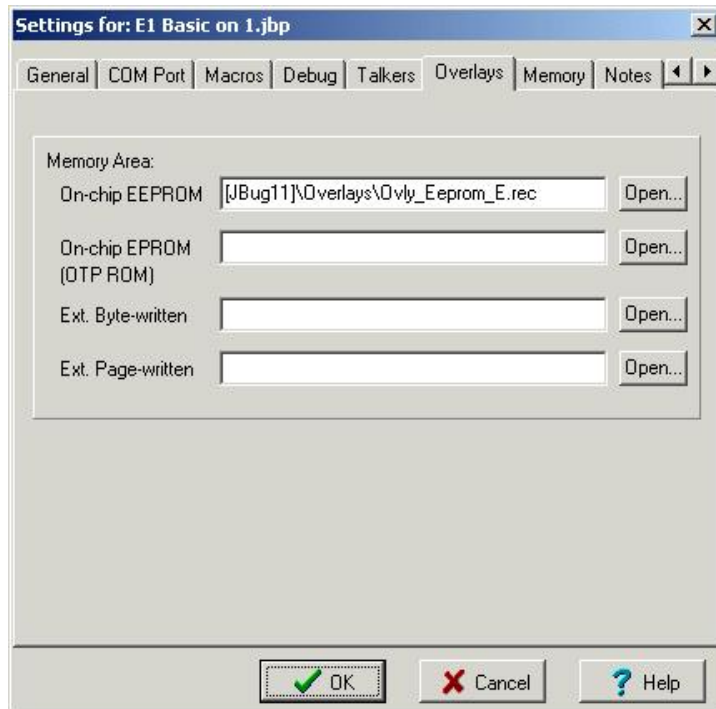
Use EEPROM-resident talker

Check this radio button to use a talker which has been previously programmed into EEPROM on the MCU. A Map File must be selected. See the JBug11 Manual for more details on talker options.

Use talker in other memory

This is for advanced users only who understand the implications of having a talker in, say, external memory. A Map File is required and additionally there are radio buttons to select a 'starting' talker if one is needed to get the main talker running.

7.7 Settings>Overlays



Tab for selecting [Talker Overlay Files](#) (see page 15) for use with a boot-loaded, RAM-resident talker. Four different overlays are available for different purposes, though in the case of the last two, you will need to write and assemble them yourself, based on the sample source code provided.

The easy way to select the correct files is to use the combo box on the 'General' tab to select an MCU - when the drop-down list closes, these file names are updated.

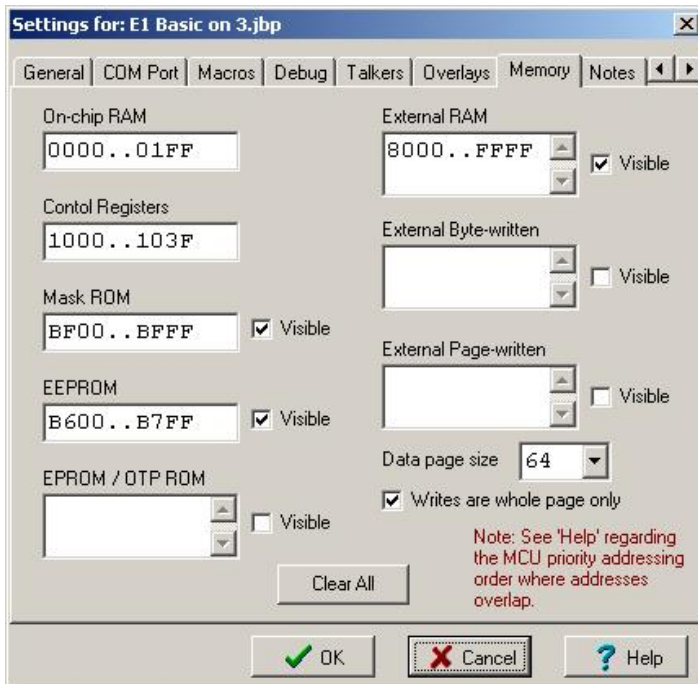
On-chip EEPROM Overlay for writing the on-chip EEPROM, usually from \$B600 to \$B7FF, but from \$F800 to \$FFFF on the 811E2.

On-chip EPROM (OTP ROM) Overlay for writing EPROM, usually only one-time-programmable, unless you have a windowed device allowing UV erasure of EPROM.

Ext. Byte-written Overlay for writing external (expansion) memory that may be written one byte at a time. Note that there is **NO NEED** to use a special overlay for most external RAM, which may be handled by the standard talker. This overlay is only needed if you have some form of EEPROM memory that needs additional operations to write.

Ext. Page-written Overlay for writing external (expansion) memory that has to be written a 'page' at a time, such as FLASH memory.

7.8 Settings>Memory



This tab defines to JBug11 the organization and types of memory available to the MCU on the target board. The format is two hexadecimal numbers separated by an ellipsis, e.g:

0000..00FF

It is not necessary to type dollar signs before the numbers - they will be assumed to be in hexadecimal anyway. Spaces may be included to improve readability (not within numbers, of course).

Memory ranges which are not entered in any of the boxes will be assumed to be undefined, and inaccessible to the CPU. The 'Visible' check boxes tell JBug11 that this memory is visible to the CPU. Depending on the mode and on bits in the CONFIG register, memory may be visible or not - it is up to you to keep track of this, as JBug11 has no way of determining for itself what memory is visible at what address.

The easy way to select the correct memory ranges is to use the combo box on the 'General' tab to select an MCU - when the drop-down list closes, this information is updated.

To get a quick overview of the whole memory map when the Settings dialog is closed, click 'Memory Map...' in the View menu, or use the keyboard shortcut Ctrl+M.

'Visible' check boxes

These allow a quick control of whether the ranges defined in the edit boxes are in fact visible to the MCU.

- **On-chip RAM**

This edit box defines to JBug11 the range of addresses which are on-chip RAM.

- **Control Registers**

This edit box defines the range of bytes covered by the on-chip control registers. For 'A' and 'E' series chips this will be \$1000 to \$103F (unless the INIT register is altered), while on other chips in the 68HC11 family there may be more or less control registers, and they may have different default locations in the memory map. For example, the 'K' series chips have more registers, and their default location is \$0000 to \$007F.

- **Mask ROM**

This edit box defines to JBug11 the range of addresses which are implemented as mask ROM - that is ROM provided on-chip but not including EPROM which may be programmed by the user. This ROM holds the boot loading code and Special Mode vectors. Format as for the RAM box.

- **EEPROM**

This edit box defines to JBug11 the range of addresses which are on-chip EEPROM. The 'On-chip EEPROM' overlay handles writing to this kind of memory. Format as for the RAM box.

If you are using Al William's 'takeree' (see [Tracing in EEPROM](#) on page 25), make sure that the EEPROM edit box is empty, and nominate the range which is actually EEPROM on the chip as External RAM.

- **EPROM / OTP ROM**

This multi-line edit box defines to JBug11 the range of addresses which are on-chip EPROM or one-time-programmable ROM. The 'On-chip OTP ROM' overlay handles writing to this kind of memory. Format each line as for the RAM box.

- **External RAM**

This multi-line edit box allows you to define additional RAM which may be available in one of the MCU expanded modes. Writing of this memory is handled by the standard talker, no overlay is needed. The format for each line is similar to the RAM edit box, for example:

```
8000..8FFF
E000..FFFF
```

- **External Byte-written**

This multi-line edit box allows you to define external memory, such as EEPROM, that requires special processing to write it. Memory defined here will be read as if it is external RAM, but written using the 'Ext. Byte-written' overlay - see [Talker Overlay Files](#) on page 15. Do **NOT** use this definition box for RAM type expansion memory which may be written satisfactorily by the standard talker.

- **External Page-written**

This multi-line edit box allows you to define external memory, such as FLASH which requires to be written a page at a time, or external EEPROM where writes are constrained to be within one page at a time. Memory defined here will be read as if it is external RAM, but written using the 'Ext. Page-written' overlay - see [Talker Overlay Files](#).

Select a data page size from the combo box. This must lie between \$10 (16 decimal) and \$100 (256 decimal). If a whole page must be written every time a write takes place, then check the box: 'Writes are whole page only'. If fewer bytes than a page's worth can be written at a time, but these bytes must all lie within a page boundary, then uncheck this box.

Clear All

Click to clear all the memory definition edit boxes.

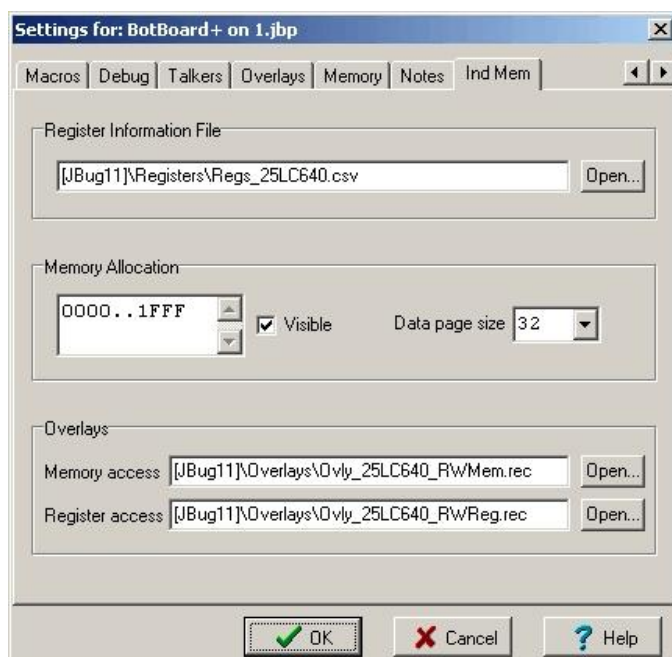
Memory Priority

Where there are different kinds of memory at overlapping addresses, the MCU prioritizes access to memory as follows: The Control Registers have top priority, followed by: Internal RAM, then Expansion RAM. If the MCU is in one of the Special modes, then the internal mask ROM has priority over expansion memory. The range of page-written memory (e.g. external FLASH) has the lowest priority for writing as far as JBug11 is concerned. For more information about which areas of memory are visible in which operating mode, see the data sheet for the MCU that you are using.

7.9 Settings>Notes

This tab has a single memo edit box. Use this memo edit to make notes about the current project. These will be stored with the project file.

7.10 Settings>Ind Mem



Tab for configuring indirectly-addressed memory (see page 23), that is memory which does not form part of the MCU's address space, but which the MCU can access in some other way, for example by serial communication using the SPI interface. Such memory commonly has one or more control registers beside its main memory space. This tab brings together in one place all the necessary configuration information.

Register Information File

Edit box for the file containing information on the indirect memory control register(s). If necessary, use the 'Open...' button to browse for the correct file. File names may appear in this and the following edit boxes with a token replacing part of the path - see [Path Tokens](#) on page 53.

Memory Allocation:

- Memory edit box** This multi-line edit box allows you to define the address range(s) of the indirectly-addressed memory. The following example shows the necessary format although it is unlikely that you will need more than one line with a single memory chip:

```
0000..1FFF
E000..FFFF
```
- Visible check box** Use this to quickly change the visibility of the data defined in the edit box above.
- Data page size** Writing to indirectly-addressed memory, particularly SPI serial eeprom, is usually limited to a page of data at a time. Use this box to specify the page size.

Overlays

Both reading and writing of indirectly-addressed memory require the use of overlay files to replace temporarily parts of the talker code, see [Overlays](#) on page 15.

- Memory access** Overlay for reading and writing indirectly-addressed memory (e.g. serial eeprom). When writing, the page size specified above is used.
- Register access** Overlay for reading and writing the registers in indirectly-addressed memory.

8 COMMANDS

8.1 General information

Commands to JBug11 to carry out monitoring, debugging and loading operations are typed in to the Command Edit Box at the lower left-hand corner of the main form. When the Enter or Return key is pressed, the command is copied to the command history list in the window above the edit box. It does not matter where the cursor is in the command line when the Enter key is struck, provided that the command line edit box has focus. Each of the available commands is allocated a separate page in this manual with a full description of its use and limitations, and a summary of all the commands is given in Appendix C. Note also these points:

- If a syntax error is made in typing the command, JBug11 echoes the command to the history window with the addition of an explanatory message. In the following example of a mis-typed List command, 008G is not a valid hexadecimal number:

```
L 0080 008G <-- Argument(s) not recognized
```

See the summary of Command Line Errors in [Appendix E](#) on page 98 for details of the possible messages.

- Previously issued commands may be recalled by using the up and down arrow keys, and then edited. Previous commands may also be reused by clicking or double-clicking on a line in the command history list. When commands are recalled, they are transferred to the command line edit box minus any comments, so that they may be corrected if necessary.
- Hitting the escape key clears the command line.
- The default number system for commands is hexadecimal (except for the specification of the number of repeats - see, for example, the [Unassemble](#) command, page 80). However, decimal and binary notation may also be used: precede a decimal number by a '#' symbol, and a binary number by '%'. There is no need to prefix hexadecimal numbers with the dollar sign, although it is not an error to do so. Leading zeros are optional. The following commands are identical:

```
L 0080 8F
L $0080 $008F
L #128 %10001111
```

- Commands may be typed in upper or lower case; it makes no difference as they are converted to upper case before being parsed. The following commands are identical:

```
L FFF0 FFFF
L fFf0 ffff
```

- Command elements may be separated by one or more spaces or by a single comma, or a mixture of the two. The following commands are identical:

```
L,0080,008F
L    80          8F
```

- Some commands can be limited to affecting only the local, JBug11, copy of the MCU memory space (see [local memory](#) on page 12). All the sample L(List) commands above would result in listing MCU memory. Typing the following command lists only the local copy of memory:

```
LL 80 8F
```

The addition of the letter L after the first L limits the command to act only on the local memory. Because the local commands do not have to read and write to the MCU they will appear to work faster.

- Where a command expects a range of memory to be specified by a starting and an ending address, then it is always possible to replace the second (ending) address by a '+' sign followed by a value to add to the starting address. Thus the following two commands are identical in their action:

```
L 80 8F
L 80 +F
```

- Label names, as listed in a loaded symbol table, or as found in the register information file, may be used wherever an address is expected. Label names are partially case sensitive, see [Labels instead of Addresses](#) on page 54 for an explanation of how JBug11 treats labels. The currently available symbolic labels may be viewed in the [Symbol Table/Register Display](#) (page 37).
- Certain commands conclude by pre-filling the command line with information so that the command may be repeated simply by pressing the <Enter> key. The following commands do this:

```
List Memory
Unassemble
Trace
Step Over
Go (Run)
```

In every case, only the first element of the command is written back to the command line. For example, if you wish to trace a program beginning at \$F800, and type: 'T3 F800' in order to trace the first three instructions, then after the command has executed, the command line will be filled in with 'T3', allowing you to trace the next three commands simply by pressing the <Enter> key.

8.2 Path Tokens

In order to save space in file name edit boxes, and on the command line, two common path names are presented in tokenized form:

- | | |
|-----------|---|
| [MyDocs]\ | This token replaces "My Documents\\"", itself an alias for a path such as: "C:\Documents and Settings\Current User\My Documents\" |
| [JBug11]\ | This token replaces the installation path of the JBug11 executable, e.g. "C:\Program Files\JBug11\" |

The tokenized forms are also used within project files to store file information. This simplifies the backup and transfer of projects to another computer, since file information is interpreted locally instead of by using absolute paths.

8.3 Labels instead of Addresses

Label names, as listed in a loaded symbol table, or as found in the register information file, may be used wherever an address is expected. Label names are partially case sensitive.

JBug11 looks up symbol names in an ordered fashion: first it checks for a name with exactly the case entered. If there is no case-sensitive match, then a case-insensitive search is done. If a single match is found, then this is used; however, if more than one label is found which matches the spelling (but not the case) of the label supplied, then the first matching spelling is used.

For example, suppose the following symbol file is loaded:

Main	8026
Dly1	802c
LongDly	8033
MAIN	8039
TestLoc	8042

- Entering 'Dly1' will find \$802C
- 'DLY1' will find \$802C (no other label has the spelling 'D L Y 1')
- 'Main' will uniquely find the value \$8026.
- 'MAIN' will uniquely find \$8039
- 'MaiN' will find \$8026 being the first instance of 'M A I N' in the list

The command:

```
L Main LongDly
```

would be the same as:

```
L 8026 8033
```

And the command:

```
F 8050 8050 TestLoc
```

would fill the two bytes at \$8050/1 with the address of TestLoc (\$8042)

All the control registers on an 'E' series chip could be listed by typing:

```
L PortA Config
```

this is equivalent to:

```
L 1000 103F (provided the registers are in their default location)
```

8.4 PCbug11-style Alternative Commands

The following command words, in the format expected by PCbug11, are also allowed. They are translated to the JBug11 format shown in the table below, and then executed:

PCbug11 command	JBug11 translation	Notes
ASM	A	Not supported in JBug11 Version 5xx
BF	F	The JBug11 command insists on having two addresses specified, so the PCbug11 form with only one address will fail or give an incorrect result
DASM	U	
DB, MD	L	
LOADS	LD	JBug11 only allows S19-record files to have the extension .rec or .s19; also, JBug11 does not allow the specification of an offset loading address
LSTM	LM	
MOVE	D	In JBug11, moves to EPROM are not allowed
QUIT	Q	Not supported in JBug11 Version 5xx
RD	R	The PCbug11 'T' option is not supported in JBug11 (it is unnecessary)
VERF	V	The PCbug11 option 'SET' is not supported.
VERF ERASE	VE	

The following PCbug11 commands have not been implemented: ASM, MS, QUIT, RM, RS, EEPROM

8.5 Set Breakpoints

BR

Format:

BR Breakpoint1[X] [Breakpoint2[X] [Breakpoint3...]]]

Command to set one or more breakpoints. A breakpoint address may be preceded or followed by an 'X' to indicate that a fixed breakpoint is to be set at that address. Two breakpoints may not be set at the same address.

It is the user's responsibility to see that breakpoint addresses are set at the first byte of instructions.

Examples:

`br 8010` Set a [transient](#) breakpoint at address \$8010

`BR 801ax` Set a [fixed](#) breakpoint at address \$801A

`br 8023 x804B 8027` Set transient breakpoints at \$8023 and \$8027 and a fixed breakpoint at \$804B

Breakpoints may also be set as part of the [Go \(Run\)](#) command (page 67), and by using the right-click context menu in the 'Start/Break Points' window. To delete breakpoints, use the [NOBR](#) (page 57) command or the context menu. The BR command will not set pass type breakpoints - see [Set Pass Breakpoint](#) (page 56). Breakpoints may be set in RAM or EEPROM but not in EPROM or external Flash PROM.

To reset the pass count of Fixed breakpoints, use the [Reset Pass Count](#) command (page 74).

8.6 Set Pass Breakpoint

BRP

Format:

BRP[Trigger] Breakpoint

or

BRP Breakpoint Trigger

Command to set a pass type breakpoint. If the first form is used, BRP should be followed by the pass trigger value as a decimal number without a space. If 'Trigger' is omitted it will default to 1. Only a single pass type breakpoint can be set at a time with this command. Pass breakpoints may also be set by using the right-click context menu in the breakpoint 'Start/Break Points' window. It is the user's responsibility to see that breakpoint addresses are set at the first byte of instructions.

Example:

`brp10 801B` Sets a pass breakpoint at address \$801B, with a pass count trigger of 10 (decimal)
or
`BRP 801B 10`

To reset the pass count, use the [Reset Pass Count](#) command (page 74).

8.7 Clear Breakpoints

NOBR

Format:

NOBR [Breakpoint1 [Breakpoint2 [Breakpoint3...]]]]

Command to clear breakpoints. If no argument is supplied then all breakpoints are removed. If one or more arguments are supplied, then the breakpoints at those addresses are removed.

Examples:

nobr Removes all breakpoints

NOBR 801a Removes the breakpoint at address \$801A

nobr 8023 804B Removes the breakpoints at \$8023 and \$804B

Breakpoints may also be removed, individually or collectively, by using the right-click context menu in the ['Start/Break Points'](#) window.

8.8 Clear Output Window

CLS

Format:

CLS

This command only clears the output window. It does not affect the CPU or its memory at all.

8.9 Clear Local Memory

CLM

Format:

CLM

This command only clears the [local memory](#) held within JBug11. It does not affect the CPU or its memory at all.

8.10 Compare Memory

CMP

Format:

CMP StartAddress EndAddress|+Length CompAddress

or

CMPL StartAddress EndAddress|+Length CompAddress

To compare a block of memory starting at one address with another starting at 'CompAddr'. The start address of the first block must be explicitly stated (an asterisk will not do). Either the end address, or a length must then be specified, followed by the address of the start of the block to be compared. Blocks of memory for comparison must not overlap.

The CMP form compares MCU-controlled memory, CMPL only compares blocks of [local memory](#).

If the blocks are identical the output window displays the message:

```
Blocks are identical
```

If the blocks are not identical, then the output window displays, in red, the memory addresses and the corresponding memory contents at the first locations which differ, for example the command:

```
CMPL 0 +ff 8000
```

might give:

```
Compare fails at  00F6 = 94      80F6 = 15
```

Format

CONFIG NewValue

Command to change the EEPROM value of the CONFIG register. NewValue is the desired new byte value. An advisory dialog appears when the command has completed.

The CONFIG register on most variants of the HC11 is a latched register whose value is stored in an EEPROM cell and is read from there to static latches every time the MCU is reset. Writes to CONFIG are actually writes to the EEPROM cell.

Using any other command which writes to memory, for example the R command ([Register Display and Change](#), page 73), will attempt a write to the static latch copy of CONFIG. However, some MCU's in the HC11 series do allow writes to the static latch copy of CONFIG (the HC11F1) and others only implement CONFIG as static latches (the HC11D0 and D3). For these MCU's the R command is useful.

The CONFIG EEPROM location is protected from accidental writes by the PTCN bit in the BPROT register (where provided). This bit defaults to 1 when the MCU is reset. If the CONFIG command is issued when the PTCN bit is set, the command will fail with an error message. It is the user's responsibility to change BPROT before using the CONFIG command.

A complete sequence might be:

1. Issue the following commands:

```
R BPROT=0F
CONFIG xy
R BPROT=1F
```

2. Reset the MCU
3. Check that the CONFIG register has changed:

```
R CONFIG
```

Notes:

- On certain chips, the F1 for example, the latch to which the CONFIG EEPROM cell is transferred at reset, is itself writable in Special Test Mode. In this case the [Register Display and Change](#) (page 73) or [Modify Memory](#) (page 71) commands may be used to write this latch (treated as RAM).
- The 'CONFIG' command first erases the value of the eeprom CONFIG location back to \$FF and then programs it with the new value. It appears that this does not work reliably on chips where the static latches may be written independently of the eeprom location, unless the static latches hold the value \$FF. This effect has been noticed on F1 chips. The workaround is to issue the command:

R CONFIG=FF

before issuing the CONFIG command itself.

8.12 Connect / Disconnect

Format

CONNECT | DISCONNECT

Command to open or close the RS232 serial COM port.

When disconnected, commands may still be given to JBug 11, but these will fail unless they affect only [local memory](#).

The 'Connect' speedbutton may be used as a quick way of issuing this command - see [Speedbuttons](#) on page 28.

Format:

CRC Path&FileName.rec|s19

or

CRCL Path&FileName.rec|s19

or

CRC StartAddress EndAddress|+Length

or

CRCL StartAddress EndAddress|+Length

Command to compute the CRC-16 cyclic redundancy check sum, either for a Motorola S19 format file or a range of bytes. *FileName* must include the path and either an 's19' or 'rec' extension. In the second form, the start address must be specified explicitly (an asterisk will not do), with either an end address or a length.

The checksum is done by calculation: $CRC = X^{16} + X^{15} + X^2 + 1$. It will protect 2^{16} bits or 8192 bytes against single bit errors. The version used in JBug11 was adapted from a Delphi routine posted on: <http://www.ibrtss.com/delphi/dcrc.html> and Copyright (99,2000) Ing.Büro R.Tschaggelar.

Finding the CRC-16 for a Motorola S19 format file

JBug11 may be used to find the checksum for an S19 format source file, or for finding the checksum of an S19 file as-loaded to MCU memory. Use the *CRCL filename* form of the command to get the checksum of the original file and *CRC filename* for the as-loaded data. The checksum is only carried out on the data bytes, not on all the bytes that make up S19 records. S19 files need not have contiguous loading addresses.

CRCL filename reads the S19 file into local memory and then performs the checksum calculation on the local data.

CRC filename opens the S19 file but only uses the loading information in the S19 records to read the corresponding data from the MCU into local memory. The checksum calculation is thus done on the MCU's image of the file data. In this way a check may be made that a previous loading operation was successful.

Finding the CRC-16 for a range of bytes

CRCL StartAddress EndAddress finds the checksum over the nominated range of bytes in local memory.

CRC StartAddress EndAddress finds the checksum over the nominated range of bytes in MCU-controlled memory.

As an example, the following command could be used to compute the CRC for the sample S19 file supplied with JBug11:

```
CRCL "[JBug11]\Samples\Looptest_8000.rec"
```

This should produce, in the Output Window:

```
CRC-16 for S19 file:  
    [JBug11]\Samples\Looptest_8000.rec  
    is $91BA (#37306)
```

Now loading the file to memory, the CRC-16 of the memory image may be found:

```
LD "[JBug11]\Samples\Looptest_8000.rec"  
CRC "[JBug11]\Samples\Looptest_8000.rec"
```

should produce:

```
CRC-16 for MCU memory, according to file:  
    [JBug11]\Samples\Looptest_8000.rec  
    is $91BA (#37306)
```

If this talker is loaded in memory, then typing

```
CRC 8000 8028
```

should produce:

```
CRC-16 for MCU memory from 8000 to 8028  
    is $91BA (#37306)
```

8.14 Duplicate Memory

D

Format:

D StartAddress EndAddress|+Length ToAddress

or

DL StartAddress EndAddress|+Length ToAddress

To duplicate a block of memory at some other address, the start address of the block to be copied must be explicitly stated (an asterisk will not do). Either the end address, or a length must then be specified, followed by the address to which the start of the block is to copied. Blocks of memory may be copied forwards or backwards in the memory space. If the resulting copy overlaps the original, then the original is obliterated by the copy within the range of addresses which overlap.

The D form writes to MCU-controlled memory, DL writes only to [local memory](#).

The following example shows the duplication of the interrupt vectors present in the boot ROM from BFD6 to BFFF, to the top end of expansion memory (an operation only possible in Special Test Mode and with external RAM or flash PROM):

```
D BFD6 BFFF FFD6
```

The 'DL' form of this command may be used to fill memory anywhere in the 64 KB address space, but the 'D' form will fail if the destination is not covered by one of the address ranges specified in [Settings>Memory](#).

This command will copy bytes to EEPROM and external memory subject to the conditions specified in [Writing to EEPROM](#) (page 20) and [Writing External Memory](#) (page 23). It will not work in EPROM.

8.15 Bulk Erase EEPROM

EBULK

Format:

EBULK

Command to erase the whole of EEPROM. The correct EEPROM address range should be specified on the [Settings>Memory](#) tab before this command will work.

Suppose that an HC811E2 was in use, with the CONFIG register having its upper four bits all set, so putting EEPROM at \$F800 - \$FFFF, and this range had been entered in Settings>Memory, then issuing the command:

```
EBULK
```

will show a confirmatory dialog, and then erase all the bytes in the range \$F800 - \$FFFF back to \$FF.
Notes:

1. This command will fail unless the BPROT register (where provided) is written to a suitable value before carrying it out.
2. This command cannot be used to erase CONFIG

8.16 Fill Memory

F

Format:

F StartAddress EndAddress|+Length ByteString|CharacterString

or

FL StartAddress EndAddress|+Length ByteString|CharacterString

To fill memory with a repeating sequence of bytes, the start address must be explicitly stated (an asterisk will not do). Either the end address, or a length must be specified, followed by the bytes or character string to be placed in memory. Character strings must be delimited by matching pairs of single or double quotation marks. The number of bytes or characters in the string is limited to 32. Byte strings must have two characters per byte.

The F form writes to MCU-controlled memory, FL writes only to [local memory](#).

Examples:

F 8000 800F 01	Fills MCU controlled memory from address \$8000 through \$800F with the byte \$01
F 8000 800F 414143	Fills MCU controlled memory from address \$8000 through \$800F with the byte sequence \$41, \$42 \$43 repeated as often as will fit.
FL 8000 +F 414243	As preceding example, but only fills JBug11 local memory .
FL 8000 801F 'DEF'	Fills local memory from \$8000 to \$801F with the repeating sequence of bytes \$44, \$45, \$46.

To fill memory with a single instance of a byte string or character string, the start and end addresses may be entered as the same value. Memory is then filled with one copy of the byte string, or ASCII bytes corresponding to the given character string, starting at StartAddress; as in the following examples:

F 8000 8000 "JBug11"	Fills MCU controlled memory from address \$8000 through \$8006 with the bytes \$4A, \$42, \$75, \$67, \$31, \$31.
F 8000 +0 'JBug11'	Identical with previous example.
F 8000 +0 4A4275673131	Identical in result with the previous example (bytes specified directly, rather than as a character string)

The 'FL' form of this command may be used to fill memory anywhere in the 64 KB address space, but the 'F' form will fail if the destination is not covered by one of the address ranges specified in [Settings>Memory](#).

This command will fill EEPROM and external Flash PROM subject to the conditions specified in [Writing to EEPROM](#) (page 20) and [Writing External Memory](#) (page 23). It will not work in EPROM.

8.17 Find Bytes

FIND

Format:

FIND StartAddress EndAddress|+Length ByteString|CharacterString

or

FINDL StartAddress EndAddress|+Length ByteString|CharacterString

To find a string of bytes within memory, the start address must be explicitly stated (an asterisk will not do). Either the end address, or a length must be specified, followed by the bytes or character string to be found. Character strings must be delimited by matching pairs of single or double quotation marks. The number of bytes or characters in the string is limited to 32.

The FIND form searches MCU-controlled memory, FINDL searches only [local memory](#).

The address returned by this command is that of the start of the first occurrence of the string. Further occurrences of the same search string may be found using the NEXT command.

Examples:

<code>FIND 8000 800F 414143</code>	Searches MCU-controlled memory from address \$8000 through \$800F for the byte sequence \$41, \$42 \$43.
------------------------------------	--

<code>FIND 8000 +1F 'DEF'</code>	Same effect as preceding example.
----------------------------------	-----------------------------------

<code>findl 0 ff 39</code>	Searches local memory for the byte \$39 between addresses \$0000 and \$00FF
----------------------------	---

See also the description of the NEXT command.

8.18 Find Next

NEXT

Format:

NEXT

To find the next occurrence of a string of bytes previously specified with a FIND command. This command actually searches only [local memory](#), but this is unimportant as local memory will have been updated by the FIND command. No commands other than FIND or NEXT can be interposed between one use of FIND or NEXT and the NEXT command - for example, issuing the FIND command followed by the L(List) command will generate an error if the NEXT command is then issued.

The address returned by this command is that of the start of the next occurrence of the string.

For example, suppose that memory contained the following bytes:

	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+A	+B	+C	+D	+E	+F	
8000	86	00	B7	10	08	86	38	B7	10	09	86	16	B7	10	28	B68.....(.
8010	10	28	8A	40	B7	10	28	44	45	46	44	45	46	44	45	46	.(.@(DEFDEFDEF
8020	A7	00	08	5A	26	FA	86	00	C6	07	CE	80	60	BD	80	86	...Z&.....'...

and that the command:

```
FIND 8000 802F 'DEF'
```

had found the first occurrence of the byte sequence \$41, \$42, \$43 at \$8017. Issuing the command

NEXT

would elicit the output:

```
String found at: $801A
```

and issuing NEXT again would produce:

```
String found at: $801D
```

Format:

G [StartAddress]* [Breakpoint1[X] [Breakpoint2[X]...]]]

Command to run a program on the MCU. If StartAddress is not specified, or an asterisk is used, then the current value of the 'T/G' address will be used. If StartAddress is specified, or an asterisk is used, then [Breakpoints](#) (page 24) may be specified at which execution will halt. Breakpoint addresses may optionally be preceded or followed by an 'X' to define a [fixed breakpoint](#).

Programs may be started at an address in any kind of memory (RAM, EPROM etc.), but breakpoints may be defined only in alterable memory; this also excludes flash memory because of its bulk writing nature.

The user is responsible for ensuring that breakpoint addresses are located at the first byte of actual instructions. The [Unassemble](#) command (page 80) may help to locate these.

The following example runs a program starting at address \$8100, with one breakpoint at \$810F and a second (fixed) breakpoint at \$813A:

```
G 8100 810F 813Ax
```

The instruction at the starting address is disassembled, and appears in brackets and in green in the breakpoint window, together with the breakpoints set as a result of issuing the G command. When the MCU program is executing but not yet halted at a breakpoint, the status bar will display the word 'Running'. When halted, the status display will change to 'Running - stopped at break point', and the relevant breakpoint will be highlighted in the breakpoint window in red. Issuing the G command without any arguments will then continue the program to the next breakpoint.

If the [Go \(Run\)](#) command (page 67) is issued while JBug11 is stopped at a breakpoint during tracing (status line reads: 'Tracing - stopped at breakpoint'), then any transient breakpoint set by the previous Trace command is deleted.

Issuing the [Stop](#) command (page 76) at any time will cause the running program to stop (unless the talker has become corrupted). But note that if you are running a program that has interrupt service routines (ISR), it is only possible to stop in the middle of a service routine if you are using the XIRQ\PD0 connection and a '.XOO' type talker, unless interrupts have been re-enabled within the routine. But it is always possible, in advance of running, to set a breakpoint at which to stop within the ISR.

If breakpoints are to be set in EEPROM, make sure that the BPROT register, where one is provided, contains a suitable value to allow writing to EEPROM.

8.20 List Memory

L

Format:

L [StartAddress]* [EndAddress|+Length]]
or
LL [StartAddress]* [EndAddress|+Length]]

Command to list memory. If no end address is specified, then 16 locations will be listed, beginning at StartAddress. If no start address is specified in the command, then 16 locations will be listed, beginning at the current value of the JBug11 'L/U' address - see the [Information Sidebar](#) on page 30. The default value of 16 locations may be changed in [Settings>General](#).

The L form lists MCU-controlled memory, LL lists only [local memory](#).

Examples:

```
L 8000 8018      List MCU controlled memory from $8000 to $8008
L 8000 +18      Same as preceding example
```

The above two commands will produce something looking like this (of course the actual bytes in memory depend upon the programming):

```
      +0 +1 +2 +3 +4 +5 +6 +7 +8 +9 +A +B +C +D +E +F
8000  86 00 B7 10 08 86 38 B7 10 09 86 16 B7 10 28 B6 .....8.....(.
8010  10 28 8A 40 B7 10 28 7E 00                      .(.@..(~.
```

The command:

```
LL 8008
```

will list the 16 locations in local memory, starting at address \$8008, and finishing with location \$8017, for example:

```
      +0 +1 +2 +3 +4 +5 +6 +7 +8 +9 +A +B +C +D +E +F
8000                                10 09 86 16 B7 10 28 B6 .....(.
8010  10 28 8A 40 B7 10 28 7E                      .(.@..(~.
```

If the current value of the 'L/U' address were \$8010, then typing the following:

```
L
```

would list the 16 bytes of MCU controlled memory from \$8010 to \$801F. The value of the 'L/U' address is updated by the Trace and Go commands so that it has the value of the latest breakpoint, making it easier to see where the program has halted.

Format:

LD Path&Filename.rec|.s19

LDL Path&Filename.rec|.s19

or

LD Path&Filename.obj|bin StartAddress

LDL Path&Filename.obj|bin StartAddress

Command to fill memory from a Motorola S19 file or from a binary image file. The LD form writes to MCU-controlled memory, LDL writes only to [local memory](#).

A complete path and filename, with extension, must be supplied. If a file in Motorola S19 record format is to be loaded, then the extension must be '.s19' or '.rec'. If a binary image file is to be loaded, it must have the extension '.obj' or '.bin' and a starting address for loading must be specified. This command will program EEPROM, EPROM and external memory under the right conditions, see [Writing to EEPROM](#) (page 20) and [Writing External Memory](#) (page 23), but binary image files cannot be loaded to EPROM.

The path part of the filename may be shortened by the use of tokens - see the section on [Path Tokens](#) on page 53. If the path and filename include spaces, then the whole argument string must be enclosed in single or double quotation marks, as in this example:

```
LD "C:\My Documents\68HC11\S19 Files\First.rec"
```

The following example loads a binary image file into MCU controlled memory, starting at address \$9000. The file will be looked for in the JBug11 installation folder, subfolder 'Samples':

```
LD "[JBug11]\Samples\Bin Example 1.obj" 9000
```

Note the use of quotation marks to enclose an argument which contains spaces and would otherwise be wrongly interpreted.

To save having to type the file name and path, this command may be executed automatically from the [Actions](#) menu with one of the four items: Load S19 to MCU, Load Binary to MCU, Load S19 to Local and Load Binary to Local. The word 'LD' or 'LDL', as appropriate, followed by the full file name, is then written to the command edit box, and the command is executed. A speedbutton is provided for the common operation of loading an S19 format file to MCU-controlled memory.

The 'LDL' form of this command may be used to fill memory anywhere in the 64 KB address space, but the 'LD' form will fail if the destination is not covered by one of the address ranges specified in [Settings>Memory](#).

During the execution of the Load command, the Output window displays information on the areas of memory being written.

8.22 List Macros

LM

Format:

LM

Command to list the names of the macros in the currently loaded macro library file. See [Macros](#) on page 86 for more information on the structure of macro files and commands.

The following example lists the names of the macros in the currently loaded macro library file.

LM

If the file "Sample 1.mcr", supplied with the program, was the currently loaded macro library, then the above command would produce the following in the Output Window:

```
MACRO1          {Macro to process an S19 file}
MACRO2          {Trace testing macro}
```

The names of the individual macros are also copied to the Command History window, followed by an arrow thus:

```
MACRO1 <--
MACRO2 <--
```

This is so that any one of the macros may then be executed by double-clicking on its name in the Command History window (or by using the up/down arrow keys on the keyboard to select it)

8.23 Modify Memory

M, MM

Format:

M StartAddress

or

MM StartAddress

Command to modify memory byte by byte. An explicit start address must be specified. When the command is given, JBug11 will reply with the address and the current value at that address, followed by a period. Typing a new byte value, followed by the enter key, will change that memory location, then advance the memory address by one and present the next byte for modification.

Pressing the space bar followed by the enter key will preserve the existing memory value. Pressing the enter key alone will terminate the modify memory command. Typing a '+' followed by an offset (default hexadecimal) will advance the location to be modified by that amount, similarly a '-' will retard the location. Typing '=' followed by an address will make this address the new location for modification. An omitted offset defaults to 1, omitting the address defaults to the current address.

For example, typing

```
M 8000
```

might give

```
8000 34.
```

Completing this line thus:

```
8000 34.4A
```

and pressing the enter key will change memory location \$8000 from \$34 to \$4A and present the next memory location for modification thus

```
8001 C3.
```

If the enter key is now pressed without typing a new byte value, the command will terminate, leaving location \$8001 with the value \$C3. If the space bar was pressed before the enter key, then the value \$C3 would also have been preserved, but the command would not terminate, and the next location, \$8002 would be presented for modification. Typing:

```
8001 C3.+3
```

will advance the location to \$8004 without changing the byte at \$8001, and present the byte for modification:

```
8004 5F.
```

Typing an '=' and a new address moves modification to that address:

```
8004 5F.=8010
```

leads to:

8010 30.

This command also writes equally well to EEPROM, see [Writing to EEPROM](#) (page 20), but it will not write to EPROM or external Flash PROM. It is not available within a macro.

8.24 Pause & Wait

PAUSE, WAIT

Format:

PAUSE [milliseconds]

or

WAIT [milliseconds]

The Pause (or Wait) command has two forms. If PAUSE is followed by a number of milliseconds, then JBug11 will pause macro execution for this length of time. 'Milliseconds' is a decimal number. 'Wait' is an exact synonym for 'Pause'. They both work in the same way as they do in PCbug11.

PAUSE is only available in macros, and following the G(Go) command.

If PAUSE is not followed by a number, then macro execution is paused until the user hits any key, or the character \$4B is received by the serial port. This lets the target MCU control the execution of JBug11 macro commands. In this form of the command, the message 'Pause end signaled' appears in the Output Window when the \$4B character is received or the user hits a key. Note that the program will still be running on the MCU, and if there are further commands in the macro, these will then be executed.

Examples:

WAIT	Suspends macro execution until the user presses a key or until the value of \$4B is received on the PC serial port
------	--

PAUSE 1000	Suspends macro execution for 1 second.
------------	--

8.25 Register Display and Change

R

Format:

R

or

R InhReg=NewValue

or

R CtrlReg

or

R CtrlReg=NewValue

Command to display and modify registers. InhReg stands for one of the HC11 CPU inherent registers: A, B, IX, IY, SP or CCR. CtrlReg stands for one of the HC11 control registers, e.g. HPRI0 or PORTA. When using this command to change a register, no spaces should be left either side of the equals sign. The R command may be used in four different ways:

1. Issuing R by itself causes JBug11 to read and display the current values of the CPU inherent registers. The results are displayed in the register display area of the [Information Sidebar](#) (page 30) and also in the output window. This command will work when stopped at a breakpoint, but not during the running of a program on the MCU.

2. Issuing R followed by the name of one of the inherent registers, an equals sign and a new value (in hexadecimal) will change the register to that value. For example:

```
R A=25
```

The R command cannot be used to set PC (the program counter). Use the [Go \(Run\)](#) command (page 67). Actually, this command only changes the value that appears in the register display (see [Information Sidebar](#)). The new value takes effect when running or tracing begins.

3. Issuing R followed by the name of an MCU control register displays the current state of that register, with the set bits highlighted in bold. For example:

```
R HPRI0
```

might produce the following line in the output window:

```
103C HPRI0 [E5=229] RBOOT SMOD MDA IRV PSEL3 PSEL2 PSEL1 PSEL0
```

The output line shows: the address of the register, its name, its current value in hexadecimal, an equals sign, its current value in decimal, the eight bits named, and highlighted in bold if set. Because bold type does not always show up well at small font sizes, the 'set' bits may also be colored - see [View menu](#) on page 34.

Note that there is one refinement to this version of the command. If the register is one of those that have a hi and lo part effectively making up a sixteen bit register, then typing the name without the last H or L will display the current sixteen bit value, for example:

```
R TCNT
```

might produce:

```
100E    TCNT    [$5C4A, #23626]
```

4. Issuing R followed by the name of an MCU control register, an equals sign and a new value, will set that control register to the new value (if the register is one that allows writing). Binary number representation may be useful here. Example:

```
R TMSK2=03 or  
R TMSK2=%00000011
```

sets the timer prescale factor to 16 (see Freescale documentation). This command cannot be used to change the eeprom implementation of the CONFIG register - see [Altering CONFIG](#) on page 59.

8.26 Reset

RESET

Format

RESET

This command, which is only available when 'PC controls reset' is checked on the [Settings>COM Port](#) tab, and the necessary hardware is in place, allows JBug11 to reset the MCU remotely by toggling an RS232 control pin. This is a convenience, but you can still use JBug11 without this facility.

This command is only available when connected, i.e. when the COM port is open.

The 'Reset' [speedbutton](#) may be used as a quick way of issuing this command.

8.27 Reset Pass Count

RPC

Format:

RPC [Breakpoint1 [Breakpoint2 [Breakpoint3...]]]

Command to reset the pass count of [fixed](#) or [pass](#) breakpoints. If no argument is supplied then all pass counts are set to zero. If one or more arguments are supplied (up to a maximum of 5), then the pass counts at those addresses are set to zero.

Examples:

```
nobr                                Resets all pass counts to zero
```

```
RPC  801a                           Resets the pass count of the breakpoint at address $801A
```

All pass counts may be reset by using the right-click context menu in the breakpoint [Start/Break Points](#) window (page 31).

8.28 Save Memory

SV

Format:

```
SV StartAddress EndAddress|+Length Path&Filename.rec|.s19
SVL StartAddress EndAddress|+Length Path&Filename.rec|.s19
or
SV StartAddress EndAddress|+Length Path&Filename.obj|.bin
SVL StartAddress EndAddress|+Length Path&Filename.obj|.bin
```

Command to save a block of memory to a file in Motorola S19 or binary image format. The SV form saves MCU-controlled memory, SVL saves only [local memory](#).

The start address of the block to be saved must be stated explicitly (an asterisk will not do). Either the end address, or a length must then be specified, followed by the path and name of the file. The extension determines whether the data is saved in Motorola S19 format or as a binary object file.

The path part of the filename may be shortened by the use of tokens - see [Path Tokens](#) on page 53. If path and filename include spaces, then the whole argument string must be enclosed in single or double quotation marks.

In the following example, the EEPROM-resident talker in an E1 type chip is saved to the file Test1.S19:

```
SV B600 B6CE "[MyDocs]\68HC11\Test1.s19"
```

If successful, a 'Saving complete' message will appear in the Output Window:

To save having to type the file name and path, this command may be executed automatically from the Actions menu with one of the four items: Save MCU to S19, Save MCU to Binary, Save Local to S19, Save Local to Binary. The word 'SV' or 'SVL', as appropriate, followed by the full file name, is then written to the command edit box, and the command is executed. A speedbutton is provided for the common operation of saving MCU-controlled memory to an S19 file.

8.29 Stop

S

Format:

S

This command stops JBug11 running or tracing a program. The status display will revert to the word 'Stopped', and the 'T/G' address will be updated to show the stopping point. The PC value in the inherent register display will be that of the talker idle loop address: 0012 for one of the supplied RAM talkers, or some other value if an EEPROM-resident talker is in use.

The Stop command deletes [transient](#) breakpoints if they were set by a previous [Trace](#) operation, otherwise all breakpoints are preserved in the list of breakpoints, but the corresponding temporary SWI opcodes (\$3F) are removed from the MCU memory.

Note: if you are running a program that has interrupt service routines, it is only possible to stop in the middle of a service routine if you are using the XIRQ\--PD0 connection and a '.XOO' type talker, unless interrupts have been re-enabled within the routine. But it is always possible to set a breakpoint in advance at which to stop, since the SWI instruction cannot be masked by the I bit in the CCR.

8.30 Launch Terminal Window

TERM

Format:

TERM

This command opens the [Terminal Window](#) (see page 92). It is included so that a macro can put the user in direct terminal communication with a target board. Suppose, for example, that the demonstration program included in the distribution, TermDemo.asm, has been compiled to run in expansion memory at \$8000. Then the following macro might be used to load the demonstration program, start it running and open the terminal window, whenever the MCU is re-booted:

```
DEFM Autostart
BEGIN
    LD "C:\Program Files\JBug11\Samples\TermDemo.s19"
    G 8000
    TERM
END
```

Format:

T[Repeats] [StartAddress]*]

Command to trace a program in MCU controlled memory. If no explicit start address is specified, then tracing will begin at the current value of the 'T/G' address. Repeats is an optional number which specifies how many times the command is to be repeated. Note that repeats is specified as a decimal number, unlike all other values which are entered in hexadecimal. The value of 'Repeats' follows the letter T without a gap. When the command is given, JBug11 carries out the following operations:

1. The instruction at the starting address is disassembled to discover the address of the following instruction (or instructions if there is the possibility of a branch);
2. JBug11 determines which is the effective following instruction, based on the opcode, the contents of the CCR, and on other information as appropriate.
3. A transient breakpoint is then written to MCU memory to 'catch' the program after execution of the current instruction. This breakpoint is displayed in the [Start/Break Points](#) window (page 31);
4. The MCU is set running at the current instruction, the status line is updated to read 'Tracing', and the instruction at the trace starting address appears in the breakpoint display in green;
5. As soon as the MCU stops at a breakpoint, that breakpoint is highlighted in red in the breakpoint display, the breakpoint information is copied to the output window, and the status line displays: 'Tracing - stopped at breakpoint'. If populated, the watch window display is also updated.
6. The above processes are repeated if 'Repeats' is specified as greater than 1.

The JBug11 internal trace starting address, T/G, and the default list and unassemble address, L/U, are also updated to show where the last trace instruction has stopped; so issuing the 'T' command alone will continue tracing from where the previous trace operation halted.

Care needs to be taken if tracing is begun at an RTS or RTI instruction if program execution has not arrived at that point by itself, e.g. as the result of issuing a [Go \(Run\)](#) command (page 67). This is so because the next address at which tracing (or stepping over - see the [Step Over](#) command on page 78) will halt is computed from the preceding value on the stack: this value will be quite arbitrary if tracing is begun 'out of the blue' at either of these instructions.

Examples:

T 800B	Begins tracing with the instruction at address \$800B
T3	Begins tracing at the current value of 'T/G' address and traces the next three instructions

This command is not available within EPROM or external Flash PROM.

8.32 Step Over

O

Format:

O[Repeats] [StartAddress]*]

Command to step over a program in MCU controlled memory, that is to jump over subroutine calls. If the next instruction is not a subroutine call, then this command behaves exactly like the [Trace](#) command. The 'T' and 'O' commands may be freely inter-mixed. If no explicit start address is specified, then stepping/tracing will begin at the current value of the 'T/G' address. When the 'O' command is given, JBug11 carries out exactly the same five operations listed under the 'T' command, with the exception that the 'following instruction' mentioned in operation no.2 will be adjusted in the case of subroutine calls to be the next sequential instruction rather than the instruction at the entry to the subroutine.

'Repeats' is an optional number which specifies how many times the command is to be repeated. Note that repeats is specified as a decimal number, unlike all other values which are entered in hexadecimal. The value of 'Repeats' follows the letter O without a gap.

The T/G address (see [Information Sidebar](#) on page 30) is also updated to show where the last trace instruction has stopped; so issuing the 'O' command alone will continue stepping from where the previous trace or step operation halted.

Supposing that the following fragments of code are in memory:

b600	ce1000		LDX	#\$1000
b603	a609		LDAA	9,X
b605	8d0c		BSR	Delay
b607	a72f		STAA	\$2F,X
....				
b613	18ce0064	Delay	LDY	#100
b617	1809	Dly1	DEY	
b619	26fc		BNE	Dly1
b61b	39		RTS	

Issuing the command:

T B600

will begin tracing with the instruction at address \$B600, and will halt at \$B603. Issuing either 'T' or 'O' by itself will then advance tracing by one instruction, halting at \$B605. Issuing the 'T' command alone would then trace into the subroutine, and halt at \$B613, whereas issuing the 'O' command would step over the subroutine, halting at \$B607.

The behavior when stepping over a routine may be customized to some extent by checking the box 'Do not stop at breakpoints within 'stepped-over' routines' on the [Settings>Debug](#) tab. When checked, this will cause JBug11 to silently process any breakpoint encountered within the 'stepped-over' routine. Transient breakpoints will be silently deleted, fixed and pass points will have their pass count updated, but tracing will not halt there.

Note also the caveat in the Trace command concerning beginning tracing or stepping at an RTS or RTI instruction.

This command is not available within EPROM or external Flash PROM. Stepping through a program in EEPROM requires that the BPROT register, where one is provided, contains a suitable value to allow modification of EEPROM.

Giving the 'Step Over' command when halted at an SWI instruction which the user has incorporated in his own code, and for which a service routine has also been written, will step over the service routine and halt next at the instruction following the SWI opcode.

8.33 Switch

SWITCH

Format

SWITCH ON | OFF

Command to toggle the polarity on the spare COM port output line (either DTR or RTS, whichever is not in use by remote reset). Only available if the necessary hardware is available on the target board, and the correct options have been chosen on the [Settings>COM Port](#) tab. This command may also be issued using the 'Switch' speedbutton.

8.34 Unassemble

U

Formats:

U[Repeats] [StartAddress]*

or

UL[Repeats] [StartAddress]*

or

U [StartAddress]* [EndAddress]+Length

or

UL [StartAddress]* [EndAddress]+Length

Command to unassemble (disassemble or reverse engineer) memory. If no start address is specified, or an asterisk is used, JBug11 will use the current value of its own starting address, in L/U. Either an ending address, or a number of repeats may be specified (but not both). Repeats is an optional number which specifies how many times the command is to be repeated. Note that repeats is specified as a decimal number, unlike other values which are entered in hexadecimal. The value of 'Repeats' follows the letters U or UL without a gap. Code cannot be unassembled between \$FFFC and \$FFFF.

The U form unassembles MCU-controlled memory, UL unassembles only [local memory](#).

The following example disassembles the 10 instructions beginning at \$0000:

```
U10 0000
```

This should produce something like the following output if one of the supplied RAM based boot talkers is loaded:

0001:	0000	8E00ED	LDS	#\$00ED
0002:	0003	CE1000	LDX	#\$1000
0003:	0006	6F2C	CLR	\$2C,X
0004:	0008	CC302C	LDD	#\$302C
0005:	000B	A72B	STAA	\$2B,X
0006:	000D	E72D	STAB	\$2D,X
0007:	000F	8640	LDAA	#\$40
0008:	0011	06	TAP	
0009:	0012	7E0012	JMP	\$0012
0010:	0015	B6102E	LDAA	\$102E

The information on each line is: line number, address, bytes at that address, instruction, operands. In certain instances a comment will be added, for example if an unrecognized opcode is encountered. A symbolic disassembly is also possible - see [Using a Symbol Table](#) on page 81.

The above result would also be achieved by issuing the command:

```
U 0 +17
```

It may be noticed in passing that the instruction at \$0012 is the idle loop which allows the talker to implement the 'Stopped' mode.

To reverse engineer a program available, for example, as a binary image file, it is not necessary to connect to an MCU at all; the program may simply be loaded into local memory at the correct starting address, and the 'UL' command used to disassemble it.

The readability of disassembly listings may be enhanced by checking 'UnAsm: add spaces at code breaks' on the [Settings>Debug](#) tab. This will add a blank line in unassembled listings after the following instructions:

- Unconditional jumps, including the BRA (Branch Always) instruction. No space is left if the jump address is that of the immediately following instruction, i.e. the jump was included simply to use up processor cycles.
- Return from Interrupt, RTI
- Return from Subroutine, RTS

8.35 Using a Symbol Table

A symbol file may be nominated to JBug11 so that when addresses are encountered during disassembly which match a symbol value, then the symbol is displayed instead of the address. The symbol file is a plain text file. JBug11 will cope with any file, provided that:

1. The symbol name appears first on the line;
2. The symbol value appears next, and is separated from the name by at least one space, comma or tab character. The value must be in hexadecimal but it is optional for it to be preceded by a '\$' sign
3. After the value any characters will be ignored, for example: comments, occurrences of the symbol, etc. (provided they are separated from the value by at least one space, comma or tab character)
4. Lines which cannot be interpreted as being a name followed by a value are ignored. This means that most heading lines will be discounted.
5. Some assemblers introduce the ASCII form feed character, \$0C, at the top of the listing - all control characters are ignored.

To make use a symbol file, select one of the radio buttons on the [Settings>Debug](#) tab. If the nominated file cannot be found, or cannot be interpreted, no symbol table will be loaded. The example given in the description of the [Unassemble](#) command (page 80) might look like this if a symbol table was in use:

```
0001: 0000 talker_start 8E00ED    LDS    #$00ED
0002: 0003                CE1000    LDX    #RegBase
0003: 0006                6F2C      CLR    $2C,X
0004: 0008                CC302C    LDD    #$302C
0005: 000B                A72B      STAA   $2B,X
0006: 000D                E72D      STAB   $2D,X
0007: 000F                8640      LDAA   #$40
0008: 0011                06        TAP
0009: 0012 talker_idle   7E0012    JMP    talker_idle
0010: 0015 sci_srv       B6102E    LDAA   SCSR
```

One or two factors have to be borne in mind when using a symbol table:

- Labels are partially case-sensitive. When looking-up the symbol table, JBug11 searches first for an exact case match, and returns the associated value if found, then makes a search ignoring case. This allows you to have two labels with the same spelling, but different capitalization (if your assembler can do it).
- In an assembly language program, more than one label may reference the same value. JBug11 has no way of knowing which label is the correct one if such a value is encountered - it simply picks the first label in the list which matches the given value.
- Labels longer than 8 characters will cause the output listing to become mis-aligned.
- Only 16-bit values are converted to labels, except in the case of the single byte representing an address in page-zero memory when a Direct Address mode opcode is encountered.

The currently-loaded symbol file may be viewed by clicking on 'Symbol Table' in the View menu, or using the keyboard shortcut: Ctrl+L

8.36 Verify

V

Format:

V Path&Filename.rec|.s19

or

VL Path&Filename.rec|.s19

Command to verify memory by comparing it with a Motorola S19 format file. The V form verifies MCU-controlled memory, VL verifies only [local memory](#). If the path and filename includes spaces, then the whole argument string must be enclosed in single or double quotation marks, as in this example:

```
V "C:\My Documents\68HC11\S19 Files\First.rec"
```

The path part of the filename may be shortened by the use of tokens - see [Path Tokens](#) on page 53. This command may also be activated from the 'Actions' menu.

If the command is successful, the message 'MCU memory verifies OK' will appear in the output window. If the memory does not verify correctly, then a message similar to this will appear:

```
Verify fails at 8043      MCU = 8D      S19 = 00
```

As another example of the use of the Verify command, the following macro might be used to program and verify EEPROM with an EEPROM-resident talker:

```
DEFM Load_EEPROM_Talker
BEGIN
    R bprot=10
    ebulk
    LD "[JBug11]\Talkers\JB_Talk_EE.rec"
    R bprot=1F
    V "[JBug11]\Talkers\JB_Talk_EE.rec"
END
```

Note that PCbug11 allowed an additional argument, an address in MCU memory at which to begin the verification process. This has not been implemented because it is meaningless in the case that the S19 file specifies non-contiguous address ranges. The JBug11 Verify command strictly compares the S19 file information with the bytes in MCU-controlled memory at the addresses specified in the file.

8.37 Verify Erase

VE

Format:

VE StartAddress [EndAddress|+Length]

Command to verify that MCU memory is erased (bytes = \$FF). StartAddress must be stated explicitly (an asterisk will not do). If no second address is given, only the location at StartAddress will be checked. If an explicit end address or an increment to add to StartAddress is specified, then the given range will be checked. The command simply reads the MCU-controlled memory over the specified range and compares the memory values with \$FF. For example, if the command:

```
ve b7f0 +f
```

were issued, then the program might reply with the following message in the Output window, showing that memory over this range is fully erased:

```
Memory erased over range B7F0 to B7FF
```

While the command:

```
VE B7F0 B800
```

might produce:

```
Memory NOT fully erased. At B800 MCU memory = 59
```

8.38 Indirect Memory Commands

Four commands are provided for interacting with Indirectly-addressed Memory (see page 23). These are:

- Read or Write an indirect register
- List indirect memory
- Load an S19 or binary file to indirect memory
- Save indirect memory to an S19 or binary file

They generally operate in the same way that the main memory commands do, and their syntax is the same, with the addition of the letter 'I' after the command letter. Note that all accesses of indirect memory require an overlay and separate ones are provided for accessing the indirect registers(s) and the indirect memory itself. These overlays must be specified on the Settings>Ind Mem tab.

Read an indirect register:	RI	RegName
Write an indirect register:	RI	RegName=NewValue

Same syntax as the Register command, but note that an indirect register definition file has to be specified in the Settings>Ind Mem tab.

List Indirect Memory	LI	[StartAddress]* [EndAddress +Length]
-----------------------------	-----------	---

Same syntax as the [List](#) command

Load a File		LDI	Path&Filename.rec .s19	
	or	LDI	Path&Filename.obj .bin	StartAddress

Same syntax as the [Load](#) command

Save a File		SVI	StartAddress	EndAddress +Length	Path&Filename.rec .s19
	or	SVI	StartAddress	EndAddress +Length	Path&Filename.obj .bin

Same syntax as the [Save](#) command

The Load and Save commands may be issued from the 'Actions' menu item, in which case file open and save dialogs will appear as necessary.

JBug11 maintains an internal 64K array of bytes to correspond with those in indirect memory, and which are used in data transfers, like [local memory](#) is used with transfers to MCU addressed memory. This array cannot be viewed directly by the user.

The overlays used to access indirect memory will need tailoring to suit your particular circumstances. The source code of two typical overlays is provided, and these may be used as a starting point.

9 AUTOMATION

9.1 Introduction

It is possible to automate the action of JBug11 in four ways:

1. Two commonly-used commands can be issued automatically at start-up:
 - Connect / Disconnect
 - ResetTo use this feature, tick the appropriate boxes in [Settings>General](#).
2. A group of commands may be executed automatically immediately after boot loading the talker - see [Boot Script](#) on page 87.
3. A group of commands in a macro script named 'AUTOSTART' may be executed automatically after booting, see [Autostart Macro](#) (page 88).
4. Any number of commands may be grouped into named macros, which can then be executed by typing their names into the command line, or selecting one from a drop-down list.

See the section on [Macros](#) below for an explanation of macro scripts. JBug11 includes a simple [Macro Editor](#) for writing macro script files, or macros can be 'learnt' as you type in commands.

9.2 Macros

Macros allow a succession of commands, grouped together under a named macro heading, to be executed sequentially, as though they were typed successively on the command line. This saves a lot of repetitive typing. Macros are stored as plain text in Macro Library Files. Macro library files follow the same general format as the ones used by Motorola's PCbug11. Each library file can contain an unlimited number of separate macros, and each macro can contain any number of statements. Macro library files have the extension '.mcr', and a typical one might look like this:

```
* Sample Macro library file (Sample1.mcr)
*
DEFM MACRO1          {Macro to process an S19 file}
BEGIN
    LD "[JBug11]\Samples\Looptest_8000.rec"
    L 8000 +30
    U6    8000
END
DEFM MACRO2          {Trace testing macro}
BEGIN
    F 8000 +3F 01
    LD "[JBug11]\Samples\Looptest_8000.rec"
    L 8000 +3F
    T 8000             {Trace}
    T                  ; NOTE: comments may also begin
    T                  // with a semicolon or 2 slashes
    S
    L 8000 +3F
END
```

Definition of a macro begins with the word 'DEFM' (Define Macro) followed by the name of the macro. The name may in turn be followed by up to 10 [Replaceable Parameters](#) (see page 88) . The next line must be 'BEGIN'. Thereafter the commands that make up the macro are written in the order in which they are to be performed. The last line of the macro must be 'END'.

You should note the following:

- Blank lines, and those beginning with an asterisk in the first character position, are ignored.
- The macro name following DEFM cannot contain spaces or commas.
- If one of the following is encountered on a command line:
 - an opening curly brace (the closing curly brace is merely treated as part of the comment),
 - a semicolon, or
 - a double forward slash.then everything that follows on that line will be treated as a comment.
- Upper and lower case may be freely mixed.
- Spaces, tabs or commas may be used as separators.

Macros may call other macros, but should not call themselves. When processing a command, JBug11 checks the command against the list of macro names first, before checking whether the command is one of the built-in ones. This has the side effect that if a macro is given the same name as one of JBug11's native commands, it will supersede the native command, which then becomes inaccessible.

9.3 Boot Script

Command script to run immediately after the talker is booted, or after the COM port is opened, and before any [Autostart Macro](#) (See following section). Any number of JBug11 commands may be used, including any macro in the current macro library. This script is entered on the [Settings>Macros](#) tab. It is independent of any macro script nominated elsewhere and is stored with the project data, not in a separate file.

Do **not** include the 'DEFM', 'BEGIN' and 'END' pseudo-commands which are used in macro library files - see Macros above.

A typical use might be to alter HPRI0 after booting so that Special Test Mode is available; if the following line is typed into this box:

```
R HPRI0=E5
```

then E5 will be written to HPRI0 immediately after booting - see [Register Display and Change](#) (page 73) for more details of the 'R' command.

9.4 Autostart Macro

If a macro named 'AUTOSTART' is part of a library file already loaded when the MCU is re-booted, then this macro will be played automatically after a talker is loaded (and after any boot script) . Even where a macro library file does contain an 'Autostart' macro, the automatic playing may be suppressed by checking the 'Disable Autostart macro (if any)' box on the [Settings>Macros](#) tab.

Here is an example of an auto start macro which will put the MCU into normal expanded mode on start-up. Of course, this will only work if writeable memory is present at the top of the memory map, where the interrupt vectors are located in normal modes:

```
DEFM Autostart          ; Macro to set up expanded mode
begin
    d bfd6 bfff ffd6    ; Duplicate the interrupt vectors
    r hprio=25          ; Set normal expanded mode
end
```

The need for an 'autostart' macro has been partially obviated in JBug11 Version 5 by the addition of the [Boot Script](#) facility (see page 87).

Note that, in PCbug11, it was possible to have a macro named TRACE which would run whenever a breakpoint was reached during tracing - this feature is not supported in JBug11.

9.5 Replaceable Parameters

Macros may contain replaceable parameters. Up to 10 parameters may be specified, from @0 to @9 or %0 to %9. @0 is the same parameter as %0, and reference to one is a reference to the other. Following the name of the macro on the DEFM (define Macro) line should be a list of the parameters that will be used in the macro.

Replaceable parameters may be defined (on the DEFM line) in any order, for example:

```
DEFM MyMacro @3 @5 @1
```

is valid. But note that when the macro is called as a command, then it will require three arguments, and these will be allocated in strict left-to-right order. So, calling MyMacro thus:

```
MYMACRO 8000 8001 8002
```

will assign \$8000 to @3, \$8001 to @5 and \$8002 to @1.

In the following (over-complicated) example MACRO3 is called by MACRO4, with one replaceable parameter:

```
DEFM MACRO3 @0          ; macro takes one replaceable parameter
BEGIN
    LD @0                ; (LD %0 would do the same thing)
END
DEFM MACRO4
BEGIN
    MACRO3 "[JBug11]\Samples\Looptest_8000.rec"
END
```


Playing MACRO4 in the above example thus:

```
Macro4
```

is equivalent to:

```
LD "[JBug11]\Samples\Looptest_8000.rec"
```

9.6 Playing and Recording Macros

9.6.1 Playing

Macros may be played by:

1. typing the macro name as a command on the command line, or
2. selecting a macro from the drop-down list adjacent to the 'Play' speedbutton, or
3. selecting a macro from the 'Play' item in the 'Macro' menu, or
4. clicking on the 'Play' speedbutton (provided a macro has been previously selected).

Only method 1. above allows the playing of macros that have [Replaceable Parameters](#) (see above), since these must be added after the name.

For example, to play the macro MACRO1 listed in the [Macros](#) section above, the following command would be issued:

```
MACRO1
```

As the macro plays, each line of the macro is copied to the command edit box and executed, just as if it had been typed in by hand and the enter key pressed.

Macros may also be played via the 'Macros' menu and via the 'Play Macro' speedbutton.

JBug11 may be configured automatically to stop playing a macro under certain conditions:

- If the macro contains an incorrectly formatted command, that is one that results in an error message beginning with the '<--' characters, or
- The command is one that can have an unsuccessful result, e.g. a Verify command might result in memory not being verified correctly.

These configurations are selected in [Settings>Macros](#).

9.6.2 Recording

Macros may be recorded, i.e. the commands typed into the command edit box will be added to a macro script. This is often referred to in other programs as 'learning' a macro. Access this via the Macro menu or the speedbuttons. Recorded command lines are appended to the end of whatever library script is open in the macro editor.

9.6.3 Stopping Playing or Recording

Use the 'Stop' speedbutton, or click the 'Record' or 'Play' button again, as appropriate, when it will return to its inactive state. While the command edit box has focus, hitting the <escape> key will also stop playing and recording.

9.7 Macro Editor

JBug11 includes a simple text editor for macros which may be launched in three ways:

- With the 'Edit Macro' speedbutton, or
- From the 'View' menu on the main form, or
- By using the shortcut Ctrl+E,

The editor's size and position will be remembered in the Windows registry on closure.

If changes have been made in the editor, but not saved, JBug11 will prompt the user to save the changes before finally closing down.

Macro Editor Menu Items

File

New	Clears the macro editor in preparation for writing or recording new macros.
Open ...	Opens a text file into the editor. The default file extension is .mcr, which is the standard extension for Motorola macro library files. The shortcut key is Ctrl+O.
Save ...	Saves the contents of the macro editor to a text file, default extension .mcr. The shortcut key is Ctrl+S.
Save As...	Saves the contents of the macro editor to a text file with a new name.
Close	Closes the macro editor.

Edit

Undo	Undoes the previous change. Shortcut: Ctrl+Z
Cut	Cut the current selection to the Windows clipboard. Shortcut: Ctrl+X
Copy	Copy the current selection to the Windows clipboard. Shortcut: Ctrl+C
Paste	Paste the current contents of the Windows clipboard into the editor at the current cursor position. Shortcut: Ctrl+V
Select All	Select all the text in the editor. Shortcut: Ctrl+A
Delete Line	Delete the line containing the insertion point. Shortcut: Ctrl+Y

Make into Macro	If a line or lines have been selected in the editor, then this menu item will insert the necessary 'DEFM', 'BEGIN' and 'END' lines to form a complete macro definition. There is no need to be precise about the selection, provided that the selected area begins somewhere in the first desired line and ends somewhere in the last desired line.
Check Syntax	Checks that 'DEFM', 'BEGIN' and 'END' lines appear in the right order and that the rules on replaceable parameters have been followed. It does not check that the commands forming the body of the macro are legal.

Macro

Play	Select a macro to play from the sub-menu.
Record	Begin recording. Commands typed into the command edit box are copied and appended to the text in the macro editor.
Stop	Stop recording or playing.

Options

Keep on Top	Click this menu item to keep the macro editor form on top
Font Size	Select a font size for the macro editor from the sub-menu

Help

Index	Opens Help Topics at the Index tab. Shortcut is F1.
Contents	Opens Help Topics at the Contents tab. Shortcut is Shift+F1.
Help on Macro Editor	Opens the 'Macro Editor' help topic.

Note:

The Edit menu is also duplicated as a right-click context menu while you are working in the editor. A single line may be executed by double clicking it; in which case it is transferred to the command edit box for execution.

10 TERMINAL WINDOW

10.1 Introduction

JBug11 has an integrated terminal window which allows the user to send and receive bytes with a connected MCU, provided a suitable program is running on the MCU. Bytes may be sent as ASCII text, or in their hexadecimal representation. A demonstration HC11 program is included in the distribution file; this is in assembly language and will need compiling to suit the memory architecture of the proposed target MCU - see the section on the [Terminal Demonstration Program](#) below.

The terminal window may be activated by:

- clicking the item in the View menu of the main form,
- typing the keyboard shortcut Ctrl+T, or
- issuing the command 'Term' (see [Launch Terminal Window](#)) on the command line.

10.2 Window Layout and Features

The window is divided into two panes. The left hand pane displays bytes sent (black) and received (red) in their alphabetic representation as far as possible. Non-printing characters display as hollow rectangles (this option may be suppressed - see [Terminal Window Settings](#) below). The right hand pane displays the same characters in their hexadecimal representation, as pairs of numbers in the range 00 to FF.

The division between the two panes is a moveable splitter bar, by default the right hand pane is twice as wide as the left hand one because the hex representation of bytes takes twice the width of their ASCII form.

Bytes may be entered in either pane at will. The <Tab> key will swap between panes. The currently active pane is highlighted in aqua. Only printable characters can be entered in the left hand pane.

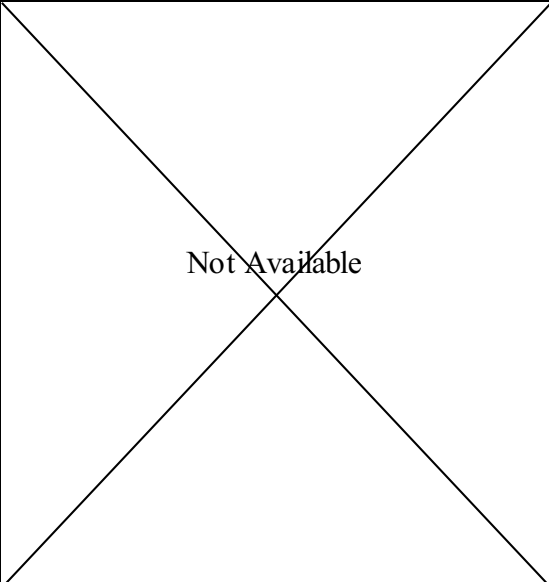
The action of the <Enter> key depends upon the terminal settings. The terminal can be configured either to send bytes immediately they are typed, or only upon pressing <Enter>. In the latter case, additional bytes can, optionally, be sent when <Enter> is pressed. These bytes may be chosen by the user.

The user may also chose whether or not bytes typed into the terminal are echoed before being sent. If the program running on the target board automatically echoes the bytes it receives, then disabling the local echo stops duplicate occurrences of sent bytes. Local echo is always enabled if 'Send only on <Enter>' is in force, and when sending bytes from the HEX edit on the right hand side of the terminal window.

Word wrapping may be enabled in the two panes, from the 'View' menu.

The terminal window uses the communication baud rate selected in 'Baud Rates' on the [Settings>COM Port](#) tab, and the COM port must be connected (opened) before the terminal can be used.

The <Backspace> key operates differently in the two panes, depending on whether bytes are sent immediately, or only when <Enter> is pressed, and partly on whether or not the echoing of characters is enabled. The action of the <Backspace> key is shown in the following table:

		Send only on <Enter>	
		<input checked="" type="checkbox"/> True - i.e. bytes are sent to the MCU only when the <Enter> key is pressed	<input type="checkbox"/> False - i.e. bytes are sent to the MCU immediately they are typed
Echo sent characters locally	<input checked="" type="checkbox"/>	<u>Working in the ASCII window</u> Deletes the last character in the buffer, unless the buffer is empty. Deletes the corresponding character in the Hex display. \$08 is not sent to the MCU <u>Working in the HEX window</u> Deletes the last character in the buffer, unless the buffer is empty. Deletes the corresponding character in the ASCII display. \$08 is not sent to the MCU Note: if the byte '08' is entered directly in the HEX edit window, then it will be sent to the MCU	<u>Working in the ASCII window</u> \$08 is sent to the MCU and the ASCII and HEX displays are updated accordingly. <u>Working in the HEX window</u> If only the first character of a two-character byte has been typed, then this character is erased. Otherwise there is no effect - no byte is sent to the MCU and there is no change in the display.
	<input type="checkbox"/>	<div style="text-align: center;">  Not Available </div>	<u>Working in the ASCII window</u> \$08 is sent to the MCU. No change to the ASCII or HEX displays. <u>Working in the HEX window</u> Local echo is always on when working in the HEX window, so if only the first character of a two-character byte has been typed, then this character is erased. Otherwise there is no effect - no byte is sent to the MCU and there is no change in the display

10.3 Interaction with the Monitoring/Debugging Functions

It is basically only possible to use either the main JBug11 monitor program or the terminal window at any one time, although it is quite possible to switch from one to the other provided the correct sequence is followed. This comes about because of the way that the monitor function acquires control

of the MCU - whenever the monitor transmits, it expects to seize control by using an interrupt mechanism. It can use either the XIRQ\ or the SCI 'On Receive' interrupt, and one or other of these must be unmasked while the monitor function is in use. The terminal window, on the other hand, cannot freely send bytes if doing so would immediately cause the talker interrupt service routine to take over. It is necessary, therefore, either to disable (mask) the interrupt mechanisms or to re-direct the SCI interrupt vector while using the terminal program for communication with an MCU. An .XOO type of talker cannot be used, and the XIRQ\ pin must not be connected to the PD0 pin. A study of the section on the demonstration terminal communication program provided in the distribution, and of the code in that program, will answer most questions about the mutual compatibility of the monitor and terminal.

10.4 Terminal Window Menu Items

File

Logging	Select 'On' or 'Off' from the sub-menu. If logging is on then an internal log is kept of the bytes sent and received by the terminal window, and this log is appended to the file specified in Terminal Window Settings (see below) whenever the terminal session is completed. If no file has been specified, then this menu item is unavailable.
Send Image File	Opens a File Open dialog for the user to select a file to be sent to the MCU as though it had been typed in at the terminal window. Any file nominated here is treated as a file of bytes which are sent sequentially. If 'Local Echo' (see below) is on, then the characters from the file will be echoed in the terminal windows as the file is sent, whether or not the MCU is itself echoing bytes received. Note that no handshaking is implemented, so the MCU has to be able to process the received bytes at the current communication baud rate, plus the nominated inter-character delay (see Terminal Window Settings). Keyboard shortcut is Ctrl+I
	Notes: 1. Files larger than 64 KB will not be sent. 2. Shortcut is Ctrl+I. 3. Sending of a file may be aborted by pressing the <Escape> key.
Close	Closes the terminal window and returns to the main form. Shortcut key is Ctrl+Q.

Edit

Clear Windows	Clears the two panes
---------------	----------------------

View

Default Layout	Provides a default size and position for the terminal window. This may be used as a starting point for the user's preferred size and position; whatever is subsequently chosen will be remembered in the Windows Registry until next time.
Keep on Top	Checking this menu item will force the terminal window to stay on top of other forms even when inactive.

Font Size	Allows the user to chose the font size for the Ascii and Hex edit windows. The as-supplied default size is 8pt.
Word Wrap	Check to wrap text in the display.

Settings

Terminal Settings...	Click to open the Terminal Window Settings dialog.
----------------------	--

Help

Index	Opens Help Topics at the Index tab. Shortcut is F1.
Contents	Opens Help Topics at the Contents tab. Shortcut is Shift+F1.
Help on Terminal	Opens the 'Terminal Window' help topic.

10.5 Terminal Window Settings

Dialog for customization of the Terminal Window. Access this dialog from the 'Settings' menu item.

Show non-displayable characters as hollow rectangles

Tick this check box to cause non-displayable Ascii codes, those from \$00 to \$1F and from \$7F to \$FF, to display as a hollow rectangle in the left hand, ASCII, pane. When this box is not checked, no character is displayed at all. The default is to have these characters displayed as hollow rectangles.

Send only on <Enter>

If this box is checked, then bytes typed into the window will not be sent until the <Enter> key is pressed. If unchecked, bytes are sent immediately they are typed.

Echo sent characters locally

Tick this check box to have characters typed at the keyboard echoed locally before being sent. If “Send only on <Enter>” is checked then “Echo sent characters locally” will also be checked, and grayed out. Note that a local echo is always operative in the Hex edit window when you are inputting bytes in that window.

Bytes to send when <Enter> is pressed

Enter in the combo box any byte sequence that it might be desirable to send, in addition to whatever has been typed, when the <Enter> key is pressed and 'Send only on <Enter>' is checked.

Bytes received which trigger a new line in the terminal windows

Enter in the combo box a byte sequence which, when received, will cause a newline in the terminal window.

Bytes received which trigger closure of the terminal window

Enter in the combo box a byte sequence which, when received, will cause focus to return to the main form. This is provided to speed up switching between the terminal and the de-bugging facilities.

Log File

Specify a name in the edit box, or choose a filename in the 'Open...' dialog, a file to become the terminal log file.

Delay (in milliseconds) between characters when sending a file

Enter a time in milliseconds which will be inserted between the sending of each character from a binary image file, or from the keyboard when 'Send only on <Enter>' is checked. Valid values are 0 ms to 100 ms.

10.6 Terminal Demonstration Program

In the distribution file is the program 'TermDemo.asm', which is installed by default in the '..\Samples\ sub-directory. This is designed to allow a quick demonstration of the capabilities of the terminal window. It will be necessary to compile it to suit the available memory of the target MCU - instructions are provided in the heading comments of the file. DO NOT USE an .XOO talker, and make sure there is no cross-connection between the XIRQ\ pin and the PD0 pin. On chips with only 256 bytes of RAM, locate the character stacking buffer at \$00B3 if using Talk_A.BOO or TalKE2.BOO, and put the body of the program in EEPROM or expansion memory. MCU's with more RAM pose less of a problem. After compilation, the program may be loaded with

```
LD "C:\...your path...\TermDemo.s19"
```

Execute a [Go \(Run\)](#) command (page 67) to the start of the terminal program body:

```
G xxxx
```

Now, without issuing any further JBug11 commands, type Ctrl+T or use the [View](#) menu to launch the terminal window. In [Terminal Window Settings](#) (page 92), make sure that the check box 'Send only on <Enter>' is ticked, and that the three edit boxes are blank

Typing any character in the left hand pane, or byte in the right hand one, and pressing <Enter> should produce an echo of the same byte from the program running on the MCU. Typing 'Hello' should produce the echo 'world'. Experiment with immediate sending by un-ticking 'Send only on <Enter>' in Terminal Window Settings. To exit, type 'Exit' (capital E, lowercase x, i & t). After send 'Exit', do not type any more into the terminal window, but continue in the main JBug11 window - the terminal demonstration program will still be running so a [S\(Stop\)](#) command (page 76) is probably the next one to issue.

If the following byte string is entered in the combo box 'Bytes received which trigger closure of the terminal window' in Terminal Window Settings (or is selected from the drop-down list):

```
45786974    (these are the ASCII values for "Exit")
```

then typing 'Exit' will automatically transfer control back to JBug11 and de-activate the terminal window.

11 ERRORS

11.1 Communication and Echo Errors

When JBug11 sends a byte to the talker it usually expects to receive an echoed byte. By comparing the transmitted and received bytes, the program is able to assure the integrity of the RS232 communications link. If the echo is wrong, JBug11 shows an error dialog. Two classes of byte error may be distinguished: Errors in the command byte (see the source file Jbug_Talk.asm in the distribution, and the section on [Talker Overlay Files](#), page 15) and errors in bytes being written to memory. These are referred to, for convenience, as ‘Comms Errors’ and ‘Echo Errors’.

11.1.1 Comms Error

This occurs when JBug11 loses synchronisation with the talker on board the MCU. The command byte sent to the MCU to initiate a command has been wrongly echoed back to JBug11. This happens when:

- the talker becomes corrupted, usually by being accidentally overwritten, or
- The stack has been accidentally overwritten, or
- while tracing or running, the SWI vector has become corrupted, or
- a mismatch has occurred between the communication baud rate settings of the MCU and JBug11, or
- an .XOO type talker is being used without the PD0--XIRQ\ connection, see [Talkers](#) (page 14) and [Appendix B - Hardware](#), or
- unexpected bytes have been sent to the MCU. This can happen when the terminal window has been in use, and control is regained by JBug11, or
- an operation has left the MCU in an unknown state.

It is usually necessary to reset the MCU and reboot the talker when this error appears. Where [remote reset](#) is available, the dialog appears with [Yes] and [No] buttons, allowing you to click [Yes] for an immediate re-boot.

When this error occurs, JBug11 writes diagnostic information to the Output Window - See [Diagnostics](#) on page 98.

11.1.2 Echo Error

This occurs when JBug11 is sending data and the MCU target echoes back a different character from the one that was sent. This happens when:

- the memory at an address could not be altered. It might, for example, be a location in ROM, or a location in EEPROM when the bits in the BPROT register do not allow writing to EEPROM, or
- the memory at an address is a control register which has one or more read-only bytes, or
- the address is that of an external memory-mapped I/O port which is read- or write- only, or
- the talker has become corrupted during a writing operation (seldom happens)

It should not be necessary to reboot the talker after this error as communication has not been lost.

Note that this error can occur part way through the execution of a command, so that some memory may have been correctly written, while some may not be written at all.

When this error occurs, JBug11 writes diagnostic information to the Output Window - See [Diagnostics](#) below.

11.2 Diagnostics

When an error dialog is displayed, diagnostic information is added to the Output Window.

In the following example, I added the line 7FF0..7FFF to the 'External RAM' box in [Settings>Memory](#). Now this area of memory is unimplemented on my system, so trying to use the [Fill Memory](#) command will fail:

```
F 7FF0 7FF0 12
```

produces:

```
Echo error writing RAM
  Addr 7FF0  Sent 12  Rcvd BE
  State:Stopped
  Reboot MCU? = No
```

The constants in the information above are:

Addr	Address of byte at which the echo occurred. In the case of a Comms error writing the CPU inherent registers, an address as such is meaningless, and will always be zero.
Sent	Byte sent from the PC to the talker
Rcvd	Byte as echoed by the talker
State:	The information in the left-hand panel of the status line.

The line 'Reboot MCU? = Yes|No' will only appear when the [remote reset](#) option is in use.

11.3 Error Report

Use this dialog (open in [Help](#) menu) to prepare a plain text report if you need to send information by email with a request for support. Fill in the edit boxes as you feel necessary, then press 'Make Text File...' This will make a plain text file called something like ErrRpt_20060706_1820.txt, or whatever is the current date and time. Then send it to me at john.beatty@virgin.net

11.4 Command Line Errors

These are the error messages generated by JBug11 when it cannot carry out a command. They appear in the Command History box, following a command in error and beginning with the symbol '<--'. From the right-click context menu in the Command History Window you can call up the relevant help topic on one of these errors. For a printable list of all the command line errors, see Appendix E of the Manual.

APPENDIX A - ACKNOWLEDGMENTS

Motorola/Freescale

For their flexible design of the HC11 micro controller, for their good documentation, and for their design of PCbug11 to which JBug11 obviously owes plenty

Borland

For their Delphi RAD tool - just the greatest

Dejan Crnila

For his comms port component for Delphi, v2.63. This is available from Delphi City:

<http://www.delphicity.net>

Embedded Acquisition Systems

For their neat and economical development board for the 68HC11E9:

<http://www.embeddedtronics.com>

Microsoft

For their graphical user interface and 32 bit operating system

I would like to thank Jean St-Pierre for his enthusiastic help during the early stages of program development, and continuing input. Recently, many others have contributed good ideas, among them: John Samperi (infinitely re-sizeable forms, a sensible start-up directory), Ken McCaughey (variable baud rates), Patrice Kadionik (the progress bar), and Sebastien Kramm (various suggestions). Bob Smith has been a constant source of encouragement and detailed suggestion, particularly for the return to the PCbug11 methods of managing EEPROM/EPROM writing and verifying memory. The appearance of the Terminal Window, with simultaneous display of bytes in ASCII and Hex was inspired by correspondence on the 68HC11 list server. Nico Mijster has spotted several bugs, and been the inspiration for extending support to 'K' series chips. Thomas Morgenstern has inspired me to add support for writing to Flash memory, and his work has been extended by a useful dialog with Bruce Elliott. My thanks go to Stijn de Witt for discovering how to get WinHelp to open the Topic Dialog with the tab of your choice. Håkan Nilsson discovered a serious bug in the disassembly of relative jumps and suggested the extension to handle tracing through illegal opcodes, and the storing of project configuration information in plain-text files. Mark Schultz made many useful suggestions, most of which I have incorporated!. Seralathan suggested the method of automating the application of Vpp when programming EPROM. Finally I should say that every single correspondent who has written to me over the last seven years of JBug11's development will have contributed something - usually their problems have stemmed from my inadequate documentation or an insufficiently intuitive interface.

Manual written in:

PDF document editing

Graphics edited in:

Hardware line diagram drawn in:

HC11 assembly language programs written with:

Programs compiled with:

Help file composed with:

Installation program:

WordPerfect 10

Acrobat 5

Paint Shop Pro 7

Isis 6.9 Lite (part of the Proteus suite)

UltraEdit 14 by Ian D Mead

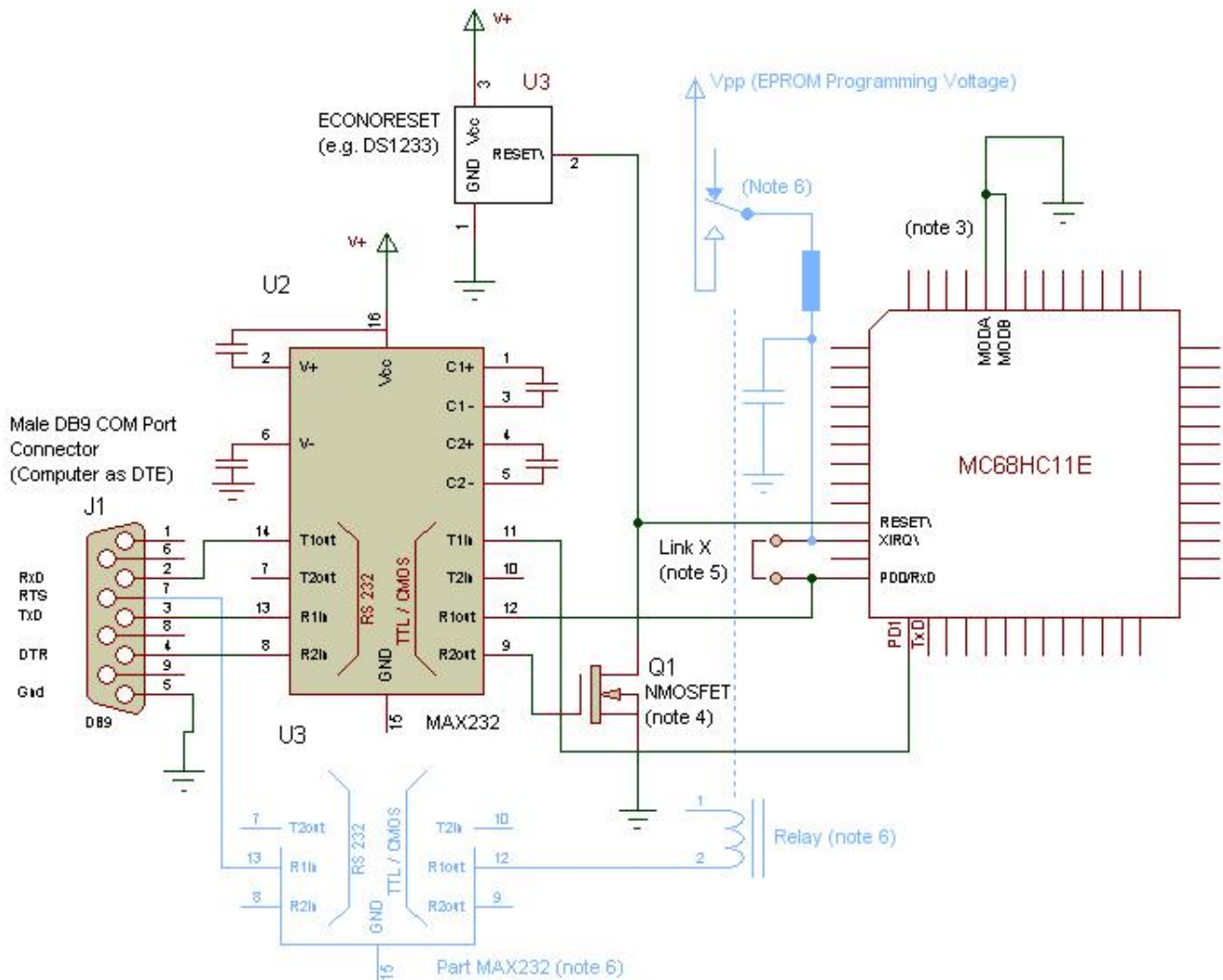
Pass11 by Pascal Niklaus

HelpHikes Pro by Sanjay Kanade

SIBuilder from PJSoft by Peter Johnson

APPENDIX B - HARDWARE

This diagram shows the basic set-up needed for JBug11 to communicate with the MCU, to allow JBug11 to reset the MCU remotely, and possibly make use of the general-purpose switch output.



NOTE: NOT TO BE USED FOR CONSTRUCTION. This diagram is a notional schematic only; intended to illustrate the intent of the various components. Please see the circuit diagrams and pin and voltage information published in the Freescale data sheet for the particular MCU that you are using. I am also not suggesting that electro-mechanical relays are actually used to realize the circuit - once again they are there to indicate an intent.

Notes:

1. The most basic setup is TxD connected via a level-shifting chip such as a MAX232 with Port D0, and Port D1 connected via a level shifter to RxD. A push switch is often provided on target boards instead of Q1, to allow the chip to be reset.
2. None of the other connections needed for a fully operational MCU are shown - any development board should provide the necessary supporting circuitry.

3. Note that the MODA and MODB pins are tied to ground - so that the MCU will reset in Special Bootstrap Mode.
4. Q1 allows JBug11 to perform a remote reset of the MCU. If this is not required, then the N-MOSFET and its connections may be omitted.
5. If the non-maskable interrupt will be used by JBug11 to gain control of the processor, using a *.XOO talker, it is necessary to connect XIRQ\ to PD0 as shown by 'Link X' in the diagram above (connecting the received-data pin to the non-maskable interrupt). It is quite possible to run JBug11 without this connection, if a *.BOO talker is used, as this gains control by using the SCI interrupt. Make sure this link is not in place if programming EPROM!
6. As an example of the General-Purpose Switching facility, a simple circuit is shown for switching Vpp. This is made up of the additional level shifting chip, part of a MAX232, and a relay. Vpp is the higher-than-Vdd voltage needed to program OTP EPROM. See the MCU data sheets for its value. Note also the comments in AN101 available from;

<http://www.smithmachineworks.com/embedprod.html>

APPENDIX C - COMMAND SUMMARY

BR Breakpoint1[X] [Breakpoint2[X] [Breakpoint3...]]	Set one or more transient or fixed breakpoints
BRP[PassCount] Breakpoint or BRP Breakpoint PassCount	Set Pass type breakpoint
CLS	Clear output window
CLM	Clear the local, JBug11, copy of memory
CMP StartAddr EndAddr +Length CompAddr	Compare two blocks of MCU memory
CMPL StartAddr EndAddr +Length CompAddr	Compare two blocks of local memory
CONFIG NewValue	Change the EEPROM implementation of the CONFIG control register
CONNECT / DISCONNECT	Open or close the COM port
CRC Path&FileName.rec s19 CRCL Path&FileName.rec s19 CRC StartAddr EndAddr +Length CRCL StartAddr EndAddr +Length	Compute a CRC-16 sum for a file or a block of MCU memory
D StartAddr EndAddr +Length ToAddr	Duplicate a block of MCU memory
DL StartAddr EndAddr +Length ToAddr	Duplicate a block of local memory
EBULK	Erase all of EEPROM
F StartAddr EndAddr +Length ByteString CharString	Fill MCU memory with bytes or a character string
FL StartAddr EndAddr +Length ByteString CharacterString	Fill local memory with bytes or a character string
FIND StartAddr EndAddr +Length ByteString CharacterString	Find a byte or character string in MCU controlled memory
FINDL StartAddr EndAddr +Length ByteString CharacterString	Find a byte or character string in local memory
G [StartAddr * [Breakpoint1 [Breakpoint2 [Breakpoint3...]]]]	Go - run a program, stopping at breakpoints if requested.
LD Path&Filename.rec .s19 LD Path&Filename.obj .bin StartAddr	Load MCU memory with a Motorola S19 format file, or with a binary image file.

LDL Path&Filename.rec .s19 LDL Path&Filename.obj .bin StartAddr	Load JBug11 local memory with a Motorola S19 format file, or with a binary image file.
L [StartAddr * [EndAddr +Length]]	List MCU controlled memory
LL [StartAddr * [EndAddr +Length]]	List JBug11 local memory
LM	List macro names
M StartAddr or MM StartAddr	Modify MCU controlled memory
NEXT	Search for further occurrences of string nominated in the FIND command
NOBR [Breakpoint1 [Breakpoint2 [Breakpoint3...]]]	Clear all or selected breakpoints
O[Repeats] [StartAddr *]	Trace over subroutines
PAUSE WAIT [milliseconds]	Pause macro execution (same as WAIT)
R [InhReg=NewValue CtrlReg CtrlReg=NewValue]	Display and modify registers
RESET	Remotely reset the target MCU
S	Stop a program running on the MCU
SV StartAddr EndAddr +Length Path&Filename.Ext or SVL StartAddr EndAddr +Length Path&Filename.Ext	Save a block of MCU or local memory to an S19 or binary image file
SWITCH	Toggle polarity of spare output pin
T[Repeats] [StartAddr *]	Trace a program in MCU controlled memory
TERM	Open the terminal window
U[Repeats] [StartAddr *] or U StartAddr * EndAddr +Length	Un-assemble (disassemble) a program in MCU controlled memory
UL[Repeats] [StartAddr *] or UL StartAddr * EndAddr +Length	Un-assemble a program in local memory
V Path&Filename.rec .s19 or VL Path&Filename.rec .s19	Verify a program in MCU or local memory against a Motorola S19 format file
VE StartAddr [EndAddr +Length]	Verify that memory is erased (all bytes \$FF)

APPENDIX D - SETUP FOR DIFFERENT MCU's

Using JBug11 with different members of the 68HC11 family.

Talker, Map and Overlay Files:

Chip	Default CONFIG register	Suitable RAM talker	Map file for RAM talker	EEPROM and EPROM programming overlay files
A0, A1, A8	\$0C, \$0D, \$0F	Talk_A.BOO or Talk_A.XOO	Talk_A.MAP	Ovly_Eeprom_A.rec <i>OTP Eprom is not available in 'A' chips</i>
D	N/A	Talk1_D.BOO and Talk2_D.rec <i>See note 1 below</i>	Talk_D.MAP	
E0, E1, E9	\$0C, \$0D, \$0F	Talk_E.BOO or Talk_E.XOO	Talk_E.MAP	Ovly_Eeprom_E.rec Ovly_Eprom_E.rec
811E2	\$FF	Talk_E2.BOO or Talk_E2.XOO	Talk_E2.MAP <i>If using Al Williams Talkeree:</i> Talk.AW.MAP	Ovly_Eeprom_E.rec
E20	\$0F ?	Talk_E.BOO or Talk_E.XOO	Talk_A.MAP	Ovly_Eeprom_E.rec Ovly_Eprom_E20.rec
F1	\$FF	Talk_F1.BOO	Talk_F1.MAP	Ovly_Eeprom_F1.rec
K series		Talk_K.BOO or Talk_K.XOO <i>See note 2 below</i>	Talk_K.MAP	Ovly_Eeprom_K.rec Ovly_Eprom_K4.rec

Notes:

1. The D series chips present something of a challenge to JBug11 because of the limited RAM available for the talker. I have experimentally overcome this limitation, but only for chips able to access external RAM memory, such as the MicroStamp11 development boards by Technological Arts. The talker is split into two parts: the first is loaded by the normal bootstrap loading operation; it contains the basic memory read and write routines. The first part is then able to load the second part, which has the inherent register and SWI service routines, as an S19 into any convenient bit of external RAM. Anyone proposing to use these talkers must study the assembly file listings, as they will probably need to re-assemble both parts to suit the available on-board memory.
2. The K series talkers should be treated as experimental at this stage, although from user feedback, they appear to be satisfactory.

3. The A, E, E2 and F1 talkers are identical, apart from differing initial locations for the stack.

Talker Source Files

The JBug11 distribution includes assembly language source files for all the above talkers. If you wish to re-assemble a talker, do not forget that the .BOO and .XOO files differ from the standard object file output by an assembler in having an additional byte at the beginning. This byte, usually \$FF, is there to select the bootstrap loading baud rate - see the Motorola/Freescale M68HC11 Reference Manual, and their application note AN1060.

Register information files

The correct register information file for the particular target chip needs to be nominated on the [Settings>General](#) tab. This will happen automatically if use the combo box to select an MCU from the drop-down list, otherwise the file may be selected from this table:

Chip Type	Register Information File
A0,A1, A8	Regs_HC11A.csv
D0, D3	Regs_HC11D3.csv
E0, E1, E9	Regs_HC11E9.csv
E2	Regs_HC11E2.csv
E20	Regs_HC11E20.csv
F1	Regs_HC11F1.csv
K4	Regs_HC11K4.csv
KS2	Regs_HC11KS2.csv

Using JBug11 with different crystal frequencies

A firmware bootloader is built into the HC11 chips; this runs when the chip is reset with the MODA and MODB pins at 0V and its function is to load the talker. This firmware expects the talker to be sent at a baud rate of: $(\text{Crystal Frequency} / 2^{10} = \text{Crystal Frequency}/1024)$. For an 8 MHz crystal, this is a baud rate of 7812.5 which is a non-standard rate (although 7680 is close enough in practice). Because 8 MHz is the commonest frequency, the firmware is arranged to 'fall back' to a rate of: $(\text{Crystal Frequency} / (2^9 * 13) = \text{Crystal Frequency}/6656)$. For an 8 MHz crystal this is a baud rate of 1201.9, close enough to 1200 to suit any UART. Whether or not the firmware falls back to the lower rate depends upon how it interprets the initial \$FF character sent by the host.

Because these rate ratios are fixed by the firmware coding, JBug11 must be set up to send the talker at a suitable rate. The following table shows the rates necessary for a range of crystal frequencies. The theoretical baud rates are those obtained by the ratios mentioned in the previous paragraph. The practicable rates are those which are an exact integer division of 115200, see [Baud Rates](#) on page 17.

Crystal Frequency MHz	E Clock rate MHz	Primary talker upload baud rate		'Fall back' talker upload baud rate		Default communication rate with talkers as-supplied	
		Theoretical	Practicable	Theoretical	Practicable	Theoretical	Practicable
4	1	3906	3840	601	600	4807	4800
4.1943	1.0486	4096	4096	630	629	5041	5008
7.3728	1.8432	7200	7200	1108	1107	8862	8861
8	2	7813	7680	1202	1200	9615	9600
9.8304	2.4576	9600	9600	1477	1476	11815	11520
12	3	11719	11520	1803	1800	14423	14400
16	4	15625	Not available	2404	2400	19231	19200
16.7772	4.1943	16384	16457	2521	2504	20165	Not available - see note 1

When the bootloading firmware finishes loading the talker, it executes a jump to the start of the talker. From now on, the serial communication parameters are under the control of the talker. With the supplied talkers, the communication baud rate is set to $(\text{Crystal Frequency} / (2^6 * 13) = \text{Crystal Frequency}/832)$ but this may be changed by altering the talker source code and re-assembling. The default communication baud rates are also shown in the above table. **NOTE** the communication baud rate must be greater than or equal to the talker upload rate - see [Baud Rates](#).

Note 1 With a 16.7772 MHz crystal, the rate of 20165 which would be expected by the standard talker is not achievable by most PC's. The first achievable rate is 16457, so the talker would have to be re-assembled for this rate.

A, E and F1 baud rate table

The following table is based on Table 9-3 in the HC11 Reference Manual (M68HC11RM.pdf). It shows the baud rates obtainable on an A, E or F1 chip for various common crystal frequencies. The shaded areas are baud rates that cannot be approximated to within 3% by the rates obtainable from the 16550 UART commonly fitted to personal computers. According to the Data sheet, the worst case allowable mis-match in baud rates is +/- 4.5% for the 68HC11, but I don't know what it is for a 16550.

BAUD register bits							Crystal Frequency in MHz						
SCP2	SCP1	SCP0	RC	BR2	BR1	BR0	16.7772	16.0000	9.8304	8.3886	8.0000	7.3728	4.1943
0	0	0		0	0	0	262144	250000	153600	131072	125000	115200	65536
0	0	0		0	0	1	131072	125000	76800	65536	62500	57600	32768
0	0	0		0	1	0	65536	62500	38400	32768	31250	28800	16384
0	0	0		0	1	1	32768	31250	19200	16384	15625	14400	8192
0	0	0		1	0	0	16384	15625	9600	8192	7813	7200	4096
0	0	0		1	0	1	8192	7813	4800	4096	3906	3600	2048
0	0	0		1	1	0	4096	3906	2400	2048	1953	1800	1024
0	0	0		1	1	1	2048	1953	1200	1024	977	900	512
0	0	1		0	0	0	87381	83333	51200	43691	41667	38400	21845
0	0	1		0	0	1	43691	41667	25600	21845	20833	19200	10923
0	0	1		0	1	0	21845	20833	12800	10923	10417	9600	5461
0	0	1		0	1	1	10923	10417	6400	5461	5208	4800	2731
0	0	1		1	0	0	5461	5208	3200	2731	2604	2400	1365
0	0	1		1	0	1	2731	2604	1600	1365	1302	1200	683
0	0	1		1	1	0	1365	1302	800	683	651	600	341
0	0	1		1	1	1	683	651	400	341	326	300	171
0	1	0		0	0	0	65536	62500	38400	32768	31250	28800	16384
0	1	0		0	0	1	32768	31250	19200	16384	15625	14400	8192
0	1	0		0	1	0	16384	15625	9600	8192	7813	7200	4096
0	1	0		0	1	1	8192	7813	4800	4096	3906	3600	2048
0	1	0		1	0	0	4096	3906	2400	2048	1953	1800	1024
0	1	0		1	0	1	2048	1953	1200	1024	977	900	512
0	1	0		1	1	0	1024	977	600	512	488	450	256
0	1	0		1	1	1	512	488	300	256	244	225	128
0	1	1		0	0	0	20165	19231	11815	10082	9615	8862	5041
0	1	1		0	0	1	10082	9615	5908	5041	4808	4431	2521
0	1	1		0	1	0	5041	4808	2954	2521	2404	2215	1260
0	1	1		0	1	1	2521	2404	1477	1260	1202	1108	630
0	1	1		1	0	0	1260	1202	738	630	601	554	315
0	1	1		1	0	1	630	601	369	315	300	277	158
0	1	1		1	1	0	315	300	185	158	150	138	79
0	1	1		1	1	1	158	150	92	79	75	69	39

Notes:

- 1** Default boot-loading baud rate. If the initial \$FF character of the talker is not received as \$FF, then the bootloading rate falls back to the following:
- 2** Fall-back bootloading baud rate.
- 3** Default communication baud rate. This may be changed by re-assembling the talker source file.

APPENDIX E - COMMAND-LINE ERROR MESSAGE SUMMARY

Command Line Errors

There are numerous ways in which what is typed on the command line may not be appropriate. If JBug11 detects an error, it replies by echoing the command line to the Command history window with a left-pointing arrow and an explanatory message. The following table lists all the errors that may occur. Most are self-explanatory; brief guidance notes are included where appropriate:

Command line error message	Notes
Unrecognized command	Basic error message
Project error - command not available	Project file has errors. Open 'Settings', then click 'OK' for a list of the errors
Not available when disconnected	COM port is closed
Requires talker initialization	MCU needs resetting, and the talker needs to be uploaded to the MCU
Not available when Stopped	
Not available when running	Command cannot be used while a program is being run on the MCU
Not available when tracing	Command cannot be used while a program is being traced on the MCU
Not available when stopped at breakpoint	Command cannot be used while a program on the MCU is stopped at a breakpoint
Already disconnected	
Already connected	
Switch not available	The General Purpose switching function has not been set up
Switch is already ON	
Switch is already OFF	
Address(es) overlap undefined memory	Command arguments, or the loading addresses in an S19 file, overlap undefined memory
Address(es) overlap RAM	(should never occur)
Address(es) overlap Control Registers	Command arguments, or the loading addresses in an S19 file, overlap the control registers
Address(es) overlap ROM	Command arguments, or the loading addresses in an S19 file, overlap undefined ROM

Address(es) overlap EEPROM	Command arguments, or the loading addresses in an S19 file, overlap EEPROM. Some commands cannot write to EEPROM
Address(es) overlap EPROM/OTP ROM	Command arguments, or the loading addresses in an S19 file, overlap EPROM. Some commands cannot write to EPROM
Address(es) overlap external RAM	
Address(es) overlap byte-written memory	Command arguments, or the loading addresses in an S19 file, overlap byte-written memory. Some commands cannot write to byte-written memory
Address(es) overlap page-written memory	Command arguments, or the loading addresses in an S19 file, overlap page-written memory. Some commands cannot write to page-written memory
Address span not allowed	Some commands can only write one type of memory at a time
Comparison address ranges overlap	
Must follow FIND or NEXT	The NEXT command must immediately follow a use of the FIND or NEXT command
Previous FIND failed	NEXT cannot be used if a previous FIND or NEXT was unsuccessful
Register not found	The register name supplied to the Register Display and Change command was not found.
Argument out of range	
Remote reset not available	The Remote Reset function has not been set up
Formatting error in S19 file	This message will be followed in the Output Window by a supplementary message in red type: !! <i>FileName</i> - Checksum error in line xx !! <i>FileName</i> - Length error in line xx !! <i>FileName</i> - Unknown record type in line xx !! <i>FileName</i> - Invalid character in line xx !! <i>FileName</i> - File has overlapping addresses
Cannot change PC	The Register command cannot be used to change the value of the Program Counter

Eeprom overlay error	<p>An error has occurred in the overlay for accessing Eeprom. This will be followed in the Output Window by a supplementary message in red type:</p> <p>!! <i>FileName</i> - File not found !! <i>FileName</i> - Checksum error in line xx !! <i>FileName</i> - Length error in line xx !! <i>FileName</i> - Unknown record type in line xx !! <i>FileName</i> - Invalid character in line xx !! <i>FileName</i> - File has overlapping addresses</p> <p>This overlay is specified in Settings>Overlays</p>
Eprom overlay error	Ditto for accessing Eprom
Byte-writing overlay error	Ditto for accessing byte-writable external memory
Page-writing overlay error	Ditto for accessing page-writable external memory
Indirect register overlay error	Ditto for accessing indirect memory registers. This overlay is specified in Settings>Ind Mem
Indirect memory overlay error	Ditto for accessing indirect memory This overlay is specified in Settings>Ind Mem
PTCON is set in BPROT	CONFIG cannot be altered because this bit is set
Talker in EEPROM cannot write EE/Eprom	
No EEPROM memory is defined	
BPROT value disallows erasing	One or more of the BPRT bits are set in the BPROT control register
BPROT value may disallow writing	One or more of the BPRT bits are set in the BPROT control register
Duplicate breakpoint(s) not allowed	
Breakpoint(s) not found	
Argument(s) not recognized	
Too few arguments	
Too many arguments	
Address range error	Usually because EndAddress is less than StartAddress
String exceeds 32 bytes	
Command not available in macros	

Command only available in macros	
BEGIN not found	The 'BEGIN' directive was not found in a macro invoked from the command line
No library macros found	
File not found	
Unrecognized file type	Files for loading or saving can only have the extensions: .rec, .s19, .bin, or .obj
Binary file is too large	The number of bytes in the file exceeds the available memory on the MCU
Invalid opcode at start address	No illegal opcode jump vector is defined
Invalid SWI at start address	No SWI service jump vector is defined
Command not implemented	

APPENDIX F - RS232 COMMUNICATIONS

Introduction

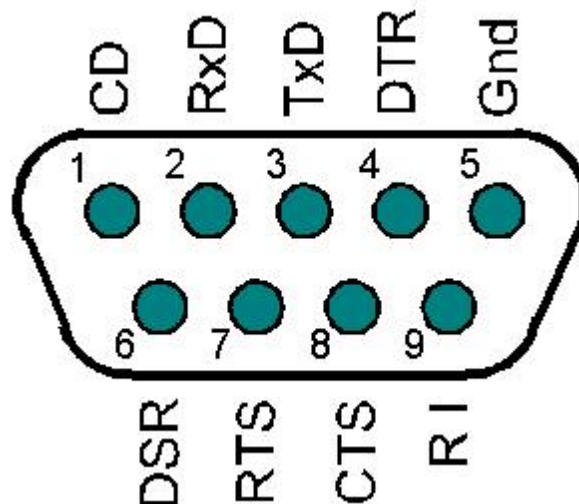
The RS232 standard, now more correctly the EIA232 standard, first appeared in the 1960's. It was originally written to standardize the interconnection of terminals and modems, in the days when one mainframe computer hosted many remote terminals. In the standard, a terminal is referred to as a DTE (*Data Terminal/Terminating Equipment*), and the modem as a DCE (*Data Communication Equipment*). This primer is a brief introduction to the standard, and to the signaling protocol usually employed, from the perspective of a JBug11 user who just wants to 'get it all to work'

Physical Connectors

The DTE is traditionally fitted with a male 'D' connector, and the modem (DCE) with a female one. When the IBM personal computer first appeared it had serial or 'COM' ports which were implemented as RS232 standard 25-pin male 'D' connectors on the back panel. When the AT version of the PC appeared, IBM began to adopt a nine pin version of the 'D' connector, and this is now universal on personal computers that still have hardware serial ports.

One pin is reserved as the common signal return or ground pin (5). Two pins are used for full duplex data transfer (2,3). The remaining six pins all carry handshaking information to control the flow of data on Tx/D and Rx/D.

The pinout, looking on the outside of the male DB9 connector as found on the back of a PC, is as follows:



The pins are labeled with a functional description as follows. 'IN' and 'OUT' refer to the signal directions as they apply to the DTE (personal computer):

Pin Number	Abbreviation	IN or OUT	Description
1	CD (or DCD)	IN	Carrier Detect. Used by a modem (DCE) to tell the DTE that it has detected a carrier signal
2	RxD	IN	Received Data. Data sent by the DTE to the DCE
3	TxD	OUT	Transmitted Data. Data sent by the DTE to the DCE
4	DTR	OUT	Data Terminal Ready. Used by the DTE to tell the DCE that it is operational.
5	GND		Ground. Common signal return.
6	DSR	IN	Data Set Ready. Used by the DCE to tell the DTE that it is operational.
7	RTS	OUT	Request to Send. Used by the DTE to signal the DCE that it may begin sending data.
8	CTS	IN	Clear to Send. Used by the DCE to signal to the DTE that it may begin sending data.
9	RI	IN	Ring Indicator. Used by a modem (DCE) to tell the DTE that an incoming call has been detected.

In the case of JBug11, the personal computer host is a DTE and the target board is a DCE. As explained in Appendix B, the only pins that must be connected are RxD, TxD and GND. Remote resetting will require a connection to DTR (or RTS) also. No other handshaking signals are needed; if connected, they will be ignored.

Voltages, Impedances and Biasing

RS232 is referred to as an 'unbalanced' signaling system in that all the signals are referred to a common ground conductor. This is inherently more susceptible to crosstalk and common-mode sources of interference than, for example, the balanced system described in RS422. Binary state signaling is used, with one state represented by positive voltage on the signaling pin of anywhere between +3V and +15V with respect to the common ground pin (pin 5); and the other state by a negative voltage between -3V and -15V. Voltages within the transition region from +3V to -3V are meaningless.

The source impedance of a signal driver should be such that accidental cross-connection with another output cannot damage either driver even where one is outputting a positive voltage and the other a negative one.

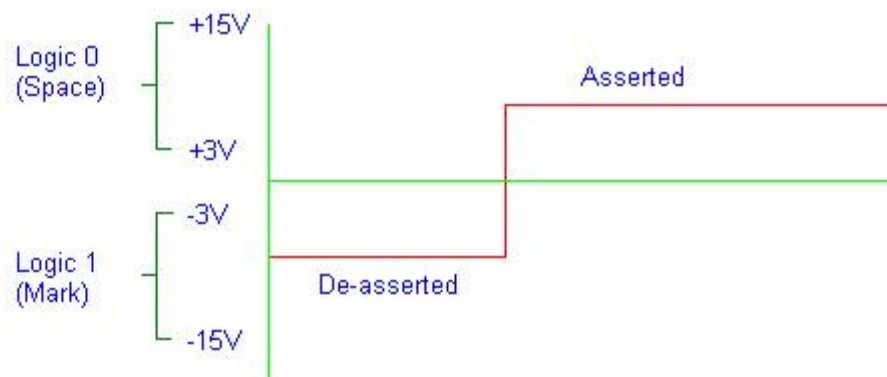
The default state of all signals is de-asserted, i.e. a negative signal voltage. The line receiving devices should be biased so that they 'see' a negative voltage if the cable is disconnected, or if the cable is connected but the transmitting end is unpowered. But note that biasing the RxD pin so that it 'sees' a positive voltage in the absence of a connecting cable might be used to give an automatic indication of a break in the cable - see the section on the *break signal* below.

Logic Levels

It is this aspect which causes the most bewilderment among users, largely because of the counter-intuitive naming conventions used. The negative voltage is referred to as a logic 1 or 'mark' condition and a positive voltage is referred to as logic 0 or 'space' condition.

A driver transmitting data on the TxD pin idles in the logic 1 or 'mark' state, i.e. with a negative voltage output. In the absence of data being actively sent from the remote device, the voltage on the RxD pin will also be negative, because that is the idling state of the driver at the 'far' end of the link.

In the case of the control signals, a de-asserted signal is one in the 'mark' or negative voltage state. For example, DTR is asserted if it goes from the negative (logic 1, mark) state to the positive (logic 0, space) state, as here:

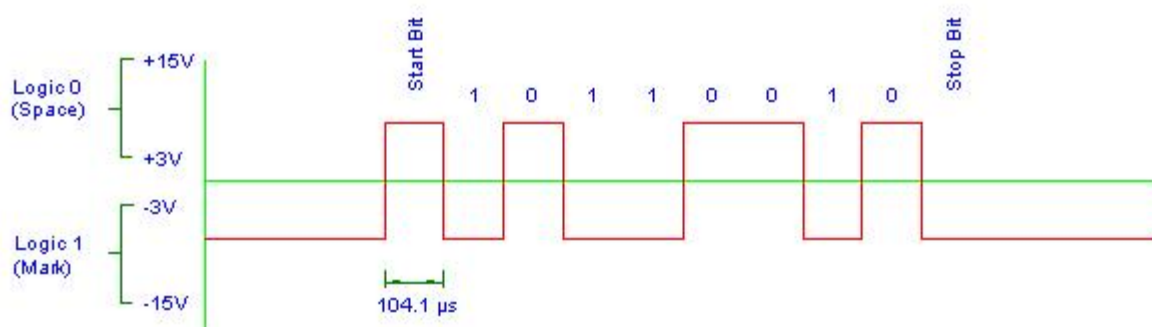


It may be noted that all of the common RS232-to-logic level converters (MAX232, 1488, 1489) invert the polarity of the transmitted and received signals. So a TTL 'high' level, logic 1, +5V, corresponds to a negative RS232 voltage (somewhere between -3V and -15V).

Data Protocols

Serial data on an RS232 link is transmitted asynchronously, i.e. a separate clock signal is not required and the start of a data byte can occur at any time. To achieve this, a logic 0 *start* bit is sent, followed by the data bits, and ending with a logic 1 *stop* bit. I will only consider here the commonest 8N1 format - a start bit followed by **Eight** data bits, **No** parity bit and **One** stop bit. The data bits are sent in least-significant-bit first order. All the bits (start, data and stop) have a duration equal to the reciprocal of the *baud rate*. Note that the technical definition of baud rate is more complex, and it only happens that baud rate and bit rate are equal for the simple binary signaling system used on RS232 links. To send a single byte takes 10 bit times (1.042 ms at 9600 baud). Since a start bit can immediately follow a stop bit, at 9600 baud the maximum throughput is 960 bytes per second.

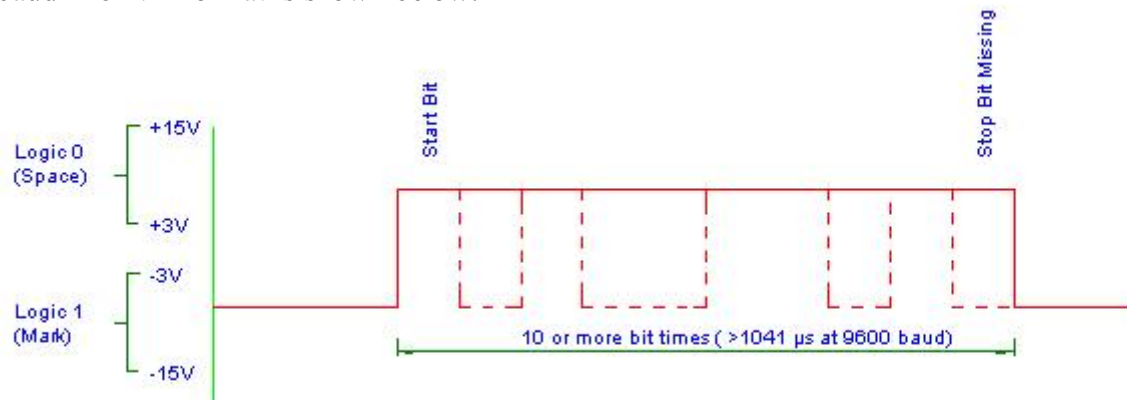
Here is the letter 'M' in ASCII coding (hex 4D, binary 01001101) sent according to the 8N1 format at 9600 baud, as it appears on the TxD pin:



Break Signal

The RS232 standard also specifies a signal called a Break, which is the sending of continuous Space values (no Start or Stop bits). I believe the name dates from the days when many news teleprinters used to be connected in series, and the break signal could be used to alert all machines. When there is no electricity present on the data circuit, the line is considered to be sending Break.

The Break signal must be of a duration longer than the time it takes to send a complete byte plus the Start and Stop (and Parity, if any) bits. The minimum signal which will be considered a break at 9600 baud in 8-N-1 format is shown below:



Not all UARTs will reliably detect a break signal of the minimum length. This applies particularly to [USB-to-Serial adaptors](#).

APPENDIX G - GETTING STARTED and QUICK TOUR

Installation

Double click on the self-installing exe file, named something like 'Install-JBug11-xyz.exe' which you have downloaded from the web site. This is a console application, so will start in a DOS box. You will be given the chance to change the default installation folder (C:\Program files\JBug11) if you wish to install it somewhere else.

The installation utility should place an icon on the desktop; if it does not, you can right-click on JBug11.exe and use the context menu to Send To>Desktop (create shortcut).

Launch

Launch JBug11. If this is the first time that the program is run, you will receive an error message, which may be ignored at this stage (answer OK). If JBug11 has been previously installed, and a newer release of the program is being started for the first time, a dialog will appear to allow the smooth transfer of configuration information from the previous version.

Adjusting the Displayed Form

As supplied, the main form of the program has quite a small size, and you may wish to increase it using one of the predetermined layouts available under the View menu item.

Connecting the Target Board

Connect the target MCU target board to one of the PC serial ports. The minimum hardware configuration is shown in [Appendix B - Hardware](#). A remote reset capability is a luxury, and not necessary to get up and running, although some form of on-board reset button will still be required. Note that the MCU must be wired to come out of reset in Special Bootstrap Mode, i.e. the MODA and MODB pins must be at logic zero during reset. If you use a USB-to-serial adaptor, then read [USB-to-Serial Adapters](#) on page 13 as not all such adaptors are suitable.

Configuring JBug11

- On the main menu, click 'File' and then 'New Project...'. The Settings dialog will open at the 'General' tab.
- Select the MCU which are using from the drop-down list. This will automatically update the Talker, Overlay and Memory information.
- Move to the 'COM Port' tab in Settings. Select the COM port to which you connected your target board.
- Select the frequency for the crystal on the target board, and accept the default baud rates offered when the drop-down list closes. If the crystal you are using does not appear in the list, refer to [Settings>COM Port](#).
- If your [Hardware](#) allows remote resetting, tick the checkbox 'PC controls reset' and select the control pin to use and the pulse polarity.

- Close the Settings dialog by clicking 'OK'.

Loading the Talker

- Click the 'Connect' speedbutton at the left hand end of the row of buttons.
- Reboot the MCU. Either using the reset button on the target board, or, if the appropriate hardware is in place, by clicking the 'Reset' speedbutton.
- JBug11 should now send the talker. If it does so, a listing of the talker will appear in the output window, the status line should read 'Stopped' and 'RAM talker loaded'. If a 'Comms Error' dialog appears, then there is a communication problem. If this persists, consider sending an [Error Report](#), see page 98.

Using Commands

The monitoring and de-bugging functions of JBug11 are invoked by typing in [Commands](#) into the [Command Edit Box](#), see page 29. Some important commands, such as those to load a file to the MCU, are also duplicated in the [Actions](#) menu. See the Quick Tour below for some initial commands to try.

Beginner's Advice

If you are quite new to programming for the HC11 series, I cannot do better than repeat the advice given in the Starter Package manual for the Technological Arts micro controller boards: '..start with something that works, and then add new features incrementally'. The very simple 'Looptest' program in the \Samples\ folder is a possible starting point.

Happy Debugging!

QUICK TOUR

This is a quick introduction to the capabilities of JBug11.

1. Listing Memory

If you are able to list (or dump) memory contents then at least you know that the PC is interfacing correctly with the talker on the MCU. Type the following example of the [List Memory](#) command (page 68) on the command line, and press <Enter>:

```
L 0
```

Note the space between the 'L' and the zero. This should produce a listing of the first sixteen addresses on the MCU. Why sixteen? - because this is the default for the number of locations to list. It may be changed on the [Settings>General](#) tab. If you type:

```
L 0 2
```

then you will get a listing of the first three locations only. Note that address arguments supplied to JBug11 commands always specify an inclusive range.

2. Reading a control register

Try using the [Register Display and Change](#) command (page 73). Type on the command line:

```
R CONFIG
```

This should produce something like:

```
103F CONFIG [$0D, #13] 0 0 0 0 NOSEC NOCOP ROMON EEON
```

Note that the individual bits are named (where appropriate) and displayed in bold if set.

3. Loading an S19 file

Try loading an S19 file to EEPROM memory. The 'Samples' sub-folder of the JBug11 installation folder contains a file named 'Looptest.asm' which has been assembled to start at various addresses, including \$B600 in the file 'Looptest_B600.rec'. If you have free EEPROM at \$B600 then you can proceed to load Looptest_B600.rec immediately, otherwise choose another loading address or re-assemble 'Looptest.asm'. As this is a write to Eeprom, it may be necessary to alter the BPROT register to a suitable value. If the chip you are using has a BPROT register, give the command:

```
R BPROT=10
```

To load the file, click the 'Load S19 to MCU' speedbutton, and navigate to the 'Samples' sub-folder. Open the file Looptest_B600.rec. The output window should display the messages:

```
LD "[JBug11]\Samples\Looptest_B600.rec"  
Loading overlay for on-chip EEPROM  
Writing EEPROM          B600..B630  
Unloading overlay  
Loading complete
```

4. Unassembling Memory

Confirm that you have loaded the 'Looptest' file satisfactorily by unassembling it. Type (upper or lower case):

```
U B600 B630
```

Compare the listing in the output window with that in the LOOPTEST_B600.LST file (also in the \Samples\ subfolder).

5. Unassembling with Labels

Open [Settings>Debug](#), and check 'As last loaded S19 file' in the Symbol Table panel. Close Settings by clicking the 'OK' button. Recall the previous unasassembly command by hitting the up arrow once. Hit <enter>. Now the listing should include the symbolic label information for 'Looptest'.

You may view the currently-loaded symbol table by clicking 'Symbol Table' in the View menu. The symbol table display may be adjusted for height, and moved to some other part of the screen - its layout will be remembered in the Windows Registry for the next time it is used.

6. Verifying Memory

You can also verify a loaded program by using the [Verify](#) command (page 83). Click on Actions>Verify S19... (in the main menu), and select Looptest_B600.rec again in the file-open dialog. This should produce the message 'MCU memory verifies OK' in the output window.

7. Running a Program

Type:

```
G B600 <enter>
```

The status line will change to 'Running' and the Output Window will display information on the instruction and registers at the starting point, \$B600 (if enabled in [Settings>Debug](#)). The instruction at the starting point, \$B600, also appears in the 'Start / Break Points' display in green and in brackets.

The Looptest program ends with an endless loop, so to stop the program, give the [Stop](#) command (page 76):

```
S
```

The output window will then display information on the stopping point, \$B615.

8. Tracing a Program

Type:

```
T B600 <enter>
```

Depending on the selections in Settings>Debug, the output window will display information on the instruction and registers at the starting point, \$B600, and at the break point (\$B603). The status line should show 'Tracing - stopped at breakpoint'. The [Start / Break Points](#) display on the main form shows an abbreviated form of the instruction at the point where you began tracing, in green, and for the breakpoint, highlighted in red because the program has halted there. The 'L/U' edit box shows the default starting address for list or unassemble operations - it is updated during tracing to reflect the program breakpoint. Note also that the command edit box is pre-filled with 'T' so that simply pressing <enter> will trace the next instruction. Note that tracing in EEPROM is a bit slower than tracing in RAM because of the extra work involved in changing EEPROM bytes.

9. Watching a Variable

Add an address to monitor during tracing. Right-click in the Watch Window (main form, lower right-hand corner) and select 'Add...'. The [Add to Watch](#) dialog opens. In the Looptest program, the counter is stored in \$00C0 so type this address in the 'User Defined' panel (in lower or upper case), then click 'Add & Close'. Now each time you repeat the trace command, the current value of address \$00C0 is updated in the watch window.

If you have arranged for the automatic loading of a symbol file according to the last S19 file loaded, then you may open the [Symbol Table/Register Display](#), select 'Counter' from the Symbol Display, and click 'Add to Watch'.

10. Erasing

Try erasing the EEPROM containing Looptest. Type:

EBULK

JBug11 will reply with a short confirmatory dialog, answer OK and JBug11 should zap your program. (If you are still tracing, JBug11 will object, and you will need to issue the 'Stop' command before giving the 'EBULK' one). You can check by using the [Verify Erase](#) command (page 84):

VE B600 B628

***** This concludes the quick tour. *****

APPENDIX H - DISTRIBUTION FILES

The installation file Install-JBug11-xyz.exe sets up the following files, in the sub-directories listed below. Note that the JBug11 printed manual itself is an Adobe Acrobat (.pdf) file which must be downloaded from the web site separately.

Primary installation folder:

ReadMe.txt	General set-up instructions and late-breaking information
JBug11.exe	The main monitor/debugging program

InstallLib.dll	
UnInstall.exe	Files connected with installing and uninstalling

Sub-folder ..\Help\

JBug11.HLP	
JBug11.CNT	On-line help files.

Sub-folder ..\MCU\

MCUData.cfg	Default configuration information for MCU's
-------------	---

Sub-folder ..\Opcodes\

HC11_Opcodes.csv	Opcode and instruction information. Applies to all chips
------------------	--

Sub-folder ..\Overlays\

Ovly_Eeprom_A.rec	Talker overlay file for writing to EEPROM on A1 and A8 series chips
Ovly_Eeprom_E.rec	Talker overlay file for writing to EEPROM on E1,E2,E20 and E9 chips
Ovly_Eeprom_F1.rec	Talker overlay file for writing to EEPROM on F1 chips
Ovly_Eeprom_K.rec	Talker overlay file for writing to EEPROM on K chips

Ovly_Eprom_E.rec	Talker overlay file for programming EPROM on 711E9 chips
Ovly_Eprom_E20.rec	Talker overlay file for programming EPROM on 711E20 chips
Ovly_Eprom_K4.rec	Talker overlay file for programming EPROM on 711K4 chips

Ovly_Eeprom_A.asm	
Ovly_Eeprom_E.asm	
Ovly_Eeprom_F1.asm	
Ovly_Eeprom_K.asm	
Ovly_Eprom_E.asm	
Ovly_Eprom_E20.asm	
Ovly_Eprom_K4.asm	Assembly language source files for the above.

Ovly_Page_AT28C256.asm	
Ovly_Page_AT29C256.asm	Sample talker overlay file for external memory that requires writing within a page at a time, such as EEPROM or FLASH. This file will need recompiling to suit the memory that you have

available. The source code is provided. See [Writing External Memory](#) on [page 23](#).

Ovly_25LC640_RWMem.rec
Ovly_25LC640_RWReg.rec

Talker overlay files for reading and writing to indirectly addressed (SPI) Microchip© 25LC640 serial eeprom such as is found on the BotBoard+

Ovly_25LC640_RWMem.asm
Ovly_25LC640_RWReg.asm

Source files for the above which may also be used as a starting point for writing overlays to suit other kinds of indirectly-addressed memory

Ovly_Eprom_711D3.asm
Ovly_Eprom_711D3.rec

Source and overlay for programming the EPROM on a 711D3. The overlay should be reassembled with a different time delay factor if crystals other than 8 MHz are used.

Sub-folder ..\Projects

Default location for project files. The distribution includes three pre-written project file, as follows:

BotBoard+.jbp
MicroStamp11.jbp
MicroStamp11_Turbo.jbp

Starter project files for the Technological Arts MicroStamp11 boards

Sub-folder ..\Registers

Regs_HC11A.csv

Control register information for A series chips

Regs_HC11D3.csv
Regs_HC711D3.csv

Control register information for D series chips

Regs_HC11E9.csv
Regs_HC11F1.csv

Control register information for E series chips
Control register information for the F1 chip

Regs_HC811E2.csv
Regs_HC711E20.csv

Control register information for E2 series chips
Control register information for E20 series chips

Regs_HC711K4.csv
Regs_HC711KS2.csv

Control register information for K series chips

Regs_25LC640.csv

Memory access registers on the 25LC640. Provided for use with the BotBoard+

Sub-folder ..\Samples

Looptest.asm

Elementary HC11 program to demonstrate the tracing of branching instructions

Looptest_0000.rec

S19 output of above program, assembled to start at \$0000

Looptest_8000.rec	S19 output of above program, assembled to start at \$8000
Looptest_B600.rec	S19 output of above program, assembled to start at \$B600
LOOPTEST_0000.LST	Assembly listings of the above files
LOOPTEST_8000.LST	
LOOPTEST_B600.LST	
Looptest_0000.sym	Symbol table files for the above
Looptest_8000.sym	
Looptest_B600.sym	
Sample 1.mcr	Simple demonstration macro library
TermDemo.asm	HC11 assembly language program to demonstrate the use of the terminal window. See Terminal Demonstration Program.
User SWI Test.asm	Program to demonstrate tracing through a user-placed SWI.
ISR_Demo.asm	Program to demonstrate setting a breakpoint, and tracing, within an interrupt service routine
Illop_Demo.asm	Program to demonstrate tracing through an illegal opcode.
Sub-folder ..\Talkers\	
JBug_Talk.asm	Assembly language source used to generate talkers. This file may be re-assembled for any A, E or F1 chip for either a .BOO or .XOO type talker - see Talkers.
Talk_A.BOO	Talker image file for MC68HC11A series devices, using the SCI interrupt (also the 811E2)
Talk_A.XOO	
Talk_E.BOO	Talker image files for MC68HC11E series devices
Talk_E.XOO	
Talk_E2.BOO	Talker image files for MC68HC811E2 series devices. Actually these are identical with the files for the A series chips, being suitable for MCU's with only 256 bytes of RAM
Talk_E2.XOO	
Talk_F1.BOO	Talker image file for MC68HC11F1 devices, using the SCI interrupt
Talk_K.BOO	Talker image files for MC68HC(7)11K4/S2 devices
Talk_K.XOO	
Talk_A.map	
Talk_E.map	
Talk_E2.map	

Talk_F1.map	
Talk_K.map	Address mapping files for the above talkers
Talk_K.asm	Assembly language source used to generate K series talkers
Talk_AW.map	Map file for an AI William's style talker - see Tracing in EEPROM on page 25
Talk_Eeprom.asm	Assembly language listing of a talker which may be loaded to EEPROM as an EEPROM-resident talker. Will need re-assembling to suit your memory layout. Don't forget to then adjust the map file also.
Talk_Eeprom_E2.rec	
Talk_Eeprom_E2.map	Sample EEPROM-resident talker and map file suitable for uploading to the EEPROM of an E2 chip (EEPROM starting at \$E800)
Sub-folder ../Talkers/Talk_D\	Subfolder containing 'D' series talker files, with particular emphasis on the MicroStamp11. These are:
Talk1_D.asm	
Talk2_D.asm	General assembly language listings suitable for modification to suit your particular D-series board.
Read_Me.txt	Important information for users of the Technological Arts MicroStamp11 target boards.
Talk1_D_MS11.asm	
Talk2_D_MS11.asm	
Talk1_D_MS11.BOO	
Talk2_D_MS11.rec	
Talk_D_MS11.MAP	Talker and map files for the MicroStamp11
Talk_711D3.BOO	
Talk_711D3.map	
Talk_711D3.asm	Talker, map file and source for programming EPROM on the 711D3. Use in conjunction with Ovly_Eprom_711D3.rec

APPENDIX J - JBUG11 REVISION HISTORY & KNOWN BUGS

Revision History

Version 5.0.0 (April 22, 2006)	First publishing of version 5
Version 5.0.1 (July 1, 2006)	<ol style="list-style-type: none"> 1. Bug fix: Could not open S19 format files that contained S0 (comment) lines - program went into an endless loop 2. Modify Memory command now launched by M as well as MM 3. Project file format slightly revised (previous files still accepted) 4. This revision appears to work OK with Win 98
Version 5.0.2 (September, 2006)	<ol style="list-style-type: none"> 1. Automatic selection of the correct files in 'Settings' improved. 2. Automatic selection of baud rates 3. 'Ignore echo errors on write' added back in 4. Pre-written project files available for the MicroStamp11 5. Reporting of set BPRT bits in BPROT improved. 6. Version 5 Manual issued.
Version 5.0.3 (March, 2007)	<ol style="list-style-type: none"> 1. Bug fix: Serial port communication rate incorrectly initialized when using a talker in external memory 2. Bug fix: Disassembly of BRCLR and similar instructions was incorrect 3. Bug fix: Play Macro failed because of spurious inclusion of an '&' character 4. Supplied K talker file rewritten to use a comms baud rate of 9600 with an 8 MHz crystal
Version 5.0.4 (March 2007)	<ol style="list-style-type: none"> 1. Bug fix: CCR (condition code register) failed to display, although program worked OK
Version 5.1.0 (April 2007)	<ol style="list-style-type: none"> 1. Code added to allow writing to indirectly-addressed memory, such as an SPI (I²C) serial Eeprom
Version 5.1.1 (August 2007)	<ol style="list-style-type: none"> 1. Bug fix: Macro names were wrongly inserted by the drop-down menu adjacent to the 'Play' button. 2. 'CONFIG' command improved, and associated help topic revised.
Version 5.1.2 (September 2007)	<ol style="list-style-type: none"> 1. Bug fix: Progress bar failed to work correctly if an overlay was needed to load a S19 file
Version 5.1.3 (May 2008)	<ol style="list-style-type: none"> 1. Bug fix: Binary files could not be loaded to EEPROM 2. Bug fix: Duplicate and Fill commands could misbehave in the 'Local' mode (minor problem)
Version 5.1.4 (May 2008)	<ol style="list-style-type: none"> 1. Bug fix: Parameter substitution in macros
Version 5.1.5 (February 2009)	<ol style="list-style-type: none"> 1. Boot Script may now be initiated on opening the COM port (suits setups where the talker is in ROM) 2. Bug fix: Register information corrected for E9 chips
Version 5.2.0 (March 2009)	<ol style="list-style-type: none"> 1. External EEPROM writing capability extended 2. Improved reporting of errors in S19 record files 3. Color may be used to help differentiate 'set' bits in the output window display of the Register Display and Change command. 4. Revised action of the <Backspace> key in the terminal 5. Bug fix: It was possible for the terminal windows to fail to recognize newline character strings sent by the MCU
Version 5.2.1 (October 2009)	<ol style="list-style-type: none"> 1. New talker and overlay file for EPROM programming the MC68HC711D3

Known Bugs:

- Does not display correctly on operating systems, such as Microsoft Vista®, when using displays set to 120 dpi or more. Display is effectively unusable. The same sort of problem may occur on earlier operating systems if large font sizes are selected in 'Display Properties> Appearance'.

APPENDIX K - CHANGES: Version 4 to Version 5

Users of JBug11 version 4xx will find many changes in version 5. This appendix addresses some "how do I do that?" questions:

1. Alter HPRI0 on boot

Version 5 does not have check and edit boxes for altering HPRI0 - these are replaced by a more flexible system which allows any number of registers to be changed after booting. This is the [Boot Script](#) facility (page 87).

To alter HPRI0 to, say, \$E5 on boot, you should add the line:

```
R HPRI0=E5
```

to the Boot Script edit box on the Settings>Macros tab, and select the 'After Booting' radio button. The import utility (see following paragraph) will automatically make the necessary transfer of data from version 4xx to 5xx.

2. Save/Recall project configurations

Version 4 could store up to ten sets of configuration information in the Windows Registry which were accessed by the Save/Recall dialog. Version 5 stores this information in plain text files, called Project Files. There is no limit to the number of projects with this system (until your hard disk is full), and backing up and briefcase working are greatly simplified. To recover your version 4 projects, use the File>Import... menu item which will open the [Import](#) dialog (page 12).

3. Alter CONFIG

Version 5 has the specialized [Altering CONFIG](#) command (page 59) to change the CONFIG register where it is implemented as an EEPROM byte. Using the 'R' command will only attempt to change the RAM latched copy of CONFIG.

4. Add items to the Watch Window

The speedbuttons have disappeared from above the Watch Window in version 5. To add an item to the window, either:

- Use the View menu to open the symbol/register display, or
- Use the right-click context menu in the Watch Window

5. Modify the CPU inherent registers

Besides using the Register Display and Change command, ACCA, ACCB, IX, IY, SP and CCR may be modified before running a program by directly editing their values in the register display.

6. Use a menu item to upload a file

The menu options for loading S19 files, etc. used to be under 'File' in the Main Menu. These are now under 'Actions'.

7. Save the Output Window to a text file

Right-click in the Output Window, choose 'Select All', right click again, choose 'Copy'. This will copy the contents to the clipboard from where it may be pasted into any text editor.

8. Use command-line switches

Command-line switches, which affect the program configuration at launch, are not supported in version 5.

9. Load an S19 file on boot

Add the appropriate Load command to the Boot Script, see Load Memory. Again, the import utility handles this automatically.

10. Load a macro on boot

Go to the Settings>Macros tab and tick the 'Associate current macro with Project' check box. Make sure that the macro library you wish to have open automatically is the one currently visible in the macro editor. This macro library file will then be re-opened automatically when JBug11 next starts up, and will therefore be available at re-booting.

11. Use local commands only

JBug11 version 4xx had an 'Allow local commands only' check box which, if ticked, would give an error message whenever a command was given which did not have the 'local' specifier (e.g. 'LD' instead of 'LDL'). This has been dropped in Version 5, because the local variant of the commands may always be used when disconnected (COM port closed).

12. Launch the calculator

The calculator is referred to as the [Base Converter](#) in version 5xx. Use the menu item on the View menu to launch it, or type Ctrl+K as before.