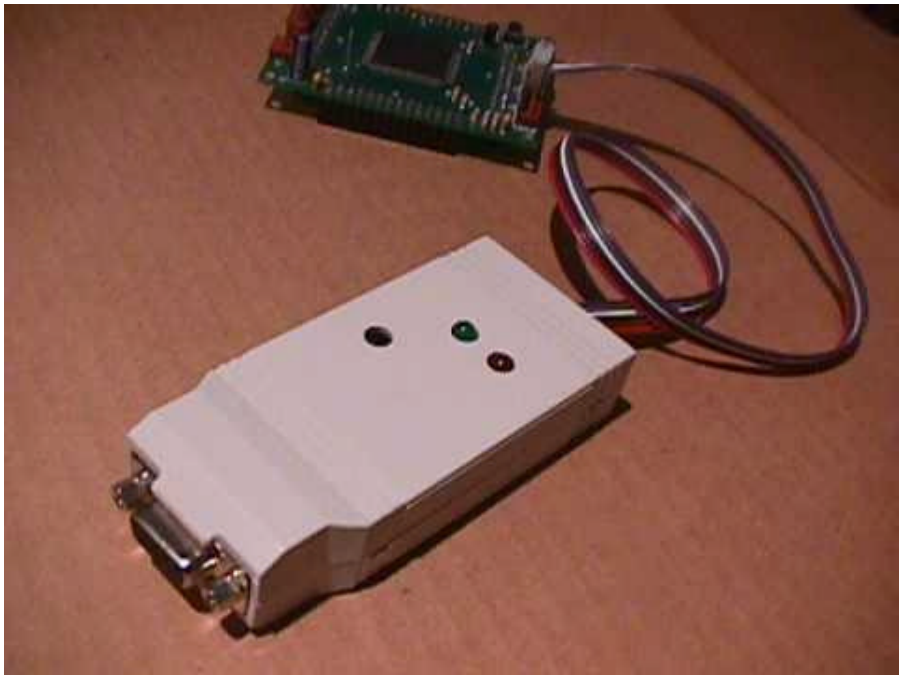
BDM12 Pod
Users Manual

Rev 5F



BDM12 shown with 68HC812A4 board

Kevin W Ross
PO Box 1714
Duvall, WA 98019
kevinro@nwlink.com
(425) 788-5985 (fax)

Printed on Monday, June 11, 2001

Figures, Tables, and Diagrams

Software and Firmware License Agreement

This is a legal agreement between you, the end user, and Kevin W Ross. If you do not agree to the terms of this Agreement, promptly return software, firmware, and product parts and/or accessories in original condition to the place of purchase within 30 days of purchase for a full refund.

1. **GRANT OF LICENSE**. This Kevin W Ross ("**KWR**") Software License Agreement permits you to use one copy of the **KWR** software and firmware product ("**SOFTWARE**") on any single computer, provided the **SOFTWARE** is in use on only one computer at any time and is used in conjunction with original or licensed **KWR** hardware product running original licensed **KWR** firmware.

2. **COPYRIGHT**. The **SOFTWARE** is owned by **KWR** and is protected by United States copyright laws and international treaty provisions. You must treat the **SOFTWARE** like any other copyrighted material (e.g., a book), except that you may make copies for archival purposes only. You may not copy written materials accompanying the **SOFTWARE**.

3. **OTHER RESTRICTIONS**. You may not rent or lease the **SOFTWARE**, but you may transfer your rights under this License on a permanent basis provided that you transfer this License, the **SOFTWARE** and all accompanying written materials, you retain no copies, and the recipient agrees to the terms of this License. If the **SOFTWARE** is an update, any transfer must include the update and all prior versions.

LIMITED WARRANTY

4. **LIMITED WARRANTY**. **KWR** warrants that the **SOFTWARE** will perform substantially in accordance with the accompanying written materials and will be free from defects in materials and workmanship under normal use and service for a period of ninety (90) days from the date of receipt. Any implied warranties on the **SOFTWARE** are limited to 90 days. Some states do not allow limitations on the duration of an implied warranty, so the above limitations may not apply to you. This limited warranty gives you specific legal rights. You may have others, which vary from state to state.

5. **CUSTOMER REMEDIES**. **KWR'**s entire liability and your exclusive remedy shall be, at **KWR'**s option, (a) return of the price paid or b) repair or replacement of the **SOFTWARE** that does not meet **KWR**'s Limited Warranty and that is returned to **KWR**. This Limited Warranty is void if failure of the **SOFTWARE** has resulted from accident, abuse, or misapplication. Any replacement **SOFTWARE** will be warranted for the remainder of the original warrant period or 30 days, whichever is longer.

6. **NO OTHER WARRANTIES**. **KWR** disclaims all other warranties, either express or implied, including but not limited to implied warranties of merchantability and fitness for a particular purpose, with respect to the **SOFTWARE**, the accompanying written materials, and any accompanying hardware. You assume full responsibility for testing for correctness of the output of the **SOFTWARE.**

7. **NO LIABILITY FOR CONSEQUENTIAL DAMAGES** In no event shall **KWR**, its distributors, or its suppliers be liable for any damages whatsoever (including, without limitation damages for loss of business profits, business interruption, loss of business information, or other pecuniary loss) arising out of the use of or inability to use the **SOFTWARE**, even if **KWR** has been advised of the possibility of such damages. The **SOFTWARE** is not designed, intended, or authorized for use in applications in which the failure of the **SOFTWARE** could create a situation where personal injury or death may occur. Should you use the **SOFTWARE** for any such unintended or unauthorized application, you shall indemnify and hold **KWR,** its distributors, and its suppliers harmless against all claims, even if such claim alleges that **KWR** was negligent regarding the design or implementation of the **SOFTWARE.**

Overview

The BDM12 pod is an interface between a host computer and the 1 wire Background Debug Mode (BDM) interface of the Motorola 68HC12. The BDM feature of this chip offers a very powerful debugging aid for the user.

Board Controls

This section will describe the controls available to the user on the BDM pod.

LED Indicators

The pod has two LED indicators. The green indicator shows that the pod is ready for the next command. When the first byte of a command is issued, the pod will turn off the green indicator until the command is complete. This usually happens pretty fast, so you might only see a flicker. During downloads to memory or EEPROM, you may see a definite flashing of the green LED. This is good. During a large memory dump, the green light may turn off. This is normal as the pod dumps large memory buffers using a high speed dump.

The red LED is used to indicate some sort of communication error. The two main reasons for this are communication failure on the RS-232 port, or a bad command code was sent by the host. You need to have your software reset the pod if the red light comes on.

To clear the error manually, you will need to hit the reset switch on the pod.

Reset switch

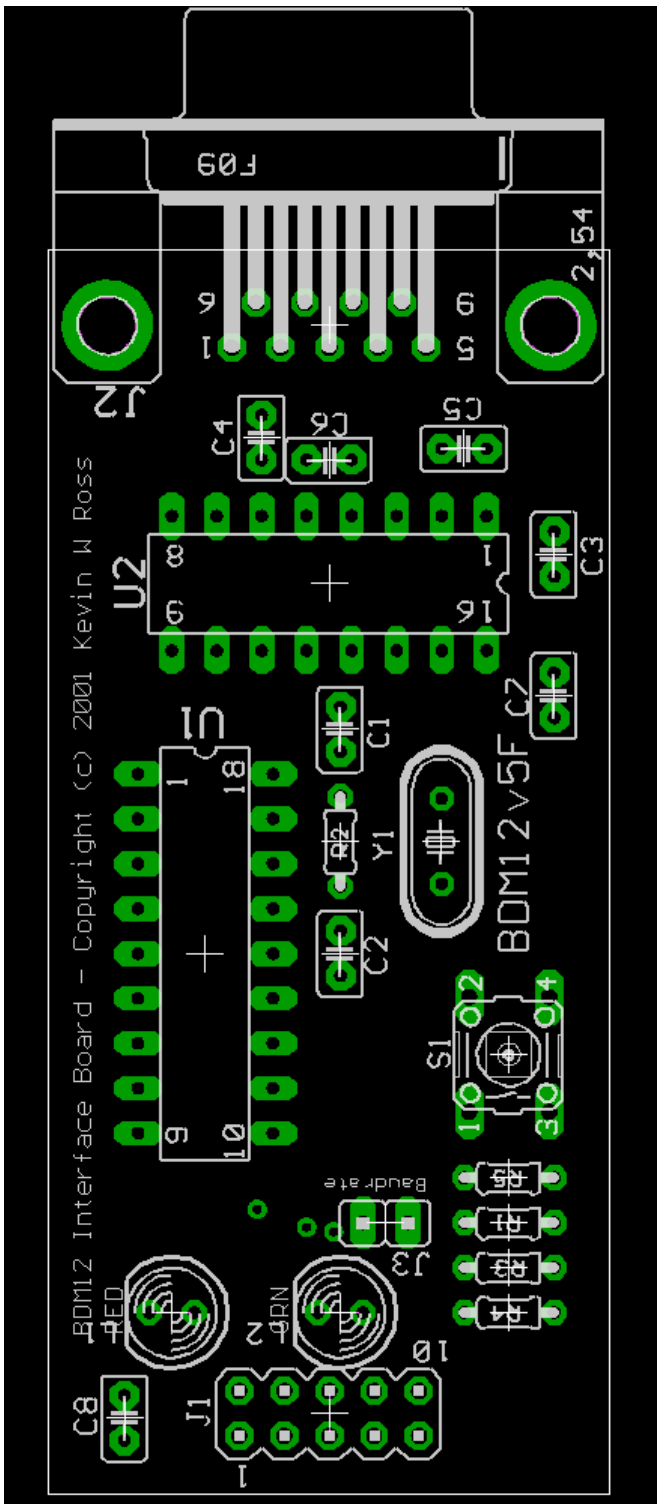
The S1 switch is used to reset the BDM12 pod. Note that this switch does not normally cause a reset on the target 68HC12 system. The switch is located inside the BDM12 pod case, and is a small pushbutton switch. To reset the BDM12, press the switch with a small non-metallic object, such as a pencil, toothpick, or other similar object. In normal operation, you should not need to reset the pod. The debugging software should be able to handle this for you.

Typical use of this switch is to recover from a communication error (red LED) that cannot be cleared in via the debugging software. This is a rare instance.

Assembly Instructions

Bill Of Materials - BDM12v5F		
Part	**Description**	**Case Markings**
U1	PIC16C715	BDM12 vX.X
U1A	18 pin Socket	
U2	MAX232ACPE	MAX232ACPE
C1 C2	22pf Cap	220
C3 C4 C5 C6 C7 C8	0.1µF cap	104
R1 R5	10k 1/8w resistor	BRN BLK ORG
R2	10m 1/8w resistor	BRN BLK BLU
R3 R4	330 1/8w resistor	ORG ORG BRN
Y1	20mhz Crystal	20.00
L1	RED LED	
L2	GREEN LED	
J1	2x5 Header (See Text)	
J2	DB9-F R/A	
J3	Empty jumper	
S1	Reset Switch	
Ribbon Cable		See Text
Plastic Case		

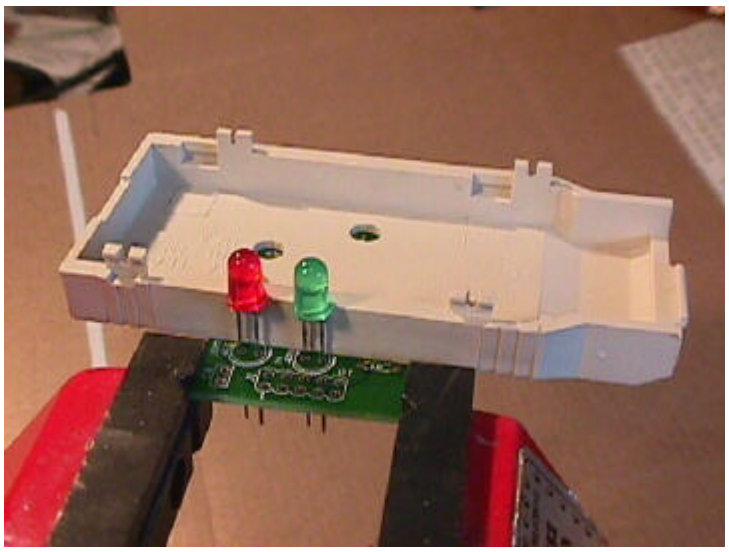
Figure 1 Bill of Materials



Assembly of the BDM12 pod requires a solder iron, rosin core solder, and 15-30 minutes of time. A couple of things to note about the construction. In general, pin 1 on all components has a mark in the silkscreen layout on the board. Some parts it doesn't matter on.

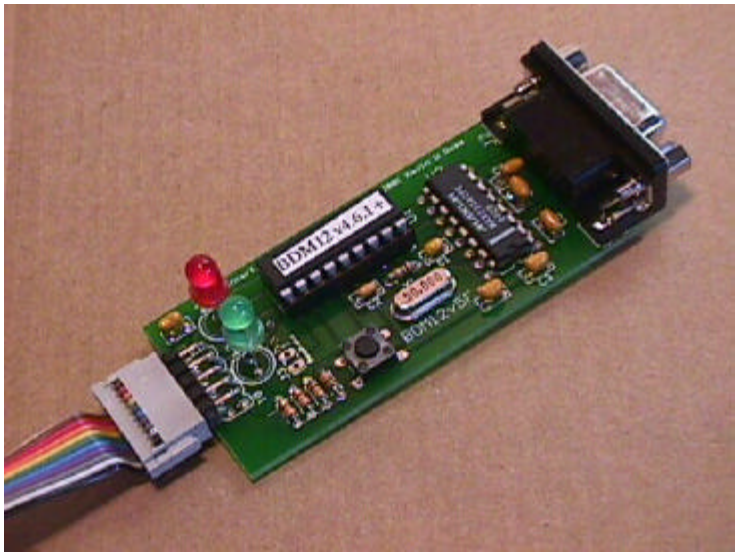
The standard connector defined by Motorola is a 2x3 dual row header, and is the connector found on most target systems. To install a 2x3 dual row header, you should insure that pin 1 of the header is inserted into J1's marked number 1 pin. The dual row header is a right angle type.

Step by Step Instructions



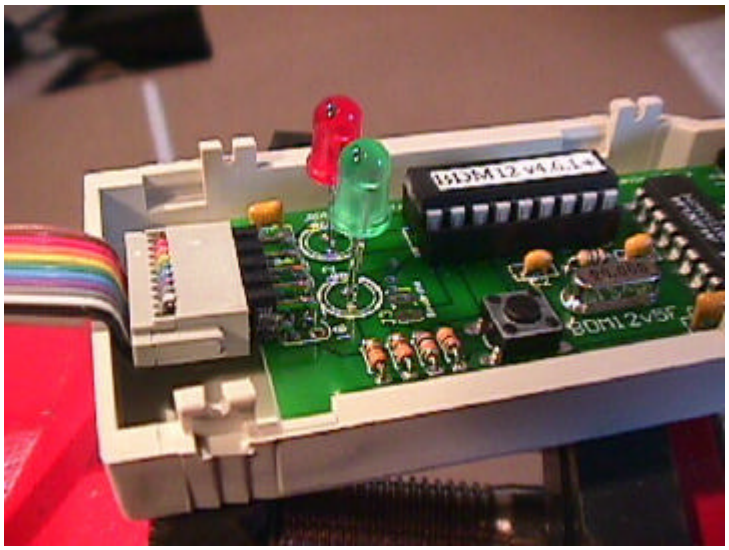
1. The first step is to mount the LED's L1 and L2 into position. The parts need to be mounted so that they are visible through the plastic case. The parts stand off the board by about 3/8" of an inch. A good way to line them up is to place half of the plastic shell across the board, inline across the LED holes. The thickness of the plastic case turns out to be just about the right height for the LED's. Insert L1 (RED) and L2 (GREEN) into their respective postions. Insert the LED so that the flat side matches the flat side on the silkscreen. Using the thickness of the plastic case as a guide, try to make sure that the parts are perpendicular to the board. Then, tack solder one pin on each part on the top side of the board. Don't heat the leads too much as you don't want to melt the plastic case. Once you have tack soldered one pin each, check the parts to see that they are about perpendicular. They don't need to be exact as you will be able to move them slightly while the case is assembled. Once you have what appears to be a good alignment, solder the other pin on each LED. You may wish to solder the bottom side of the board as well. Both pins should now be soldered.

2. Insert the socket for U1. Note where the notch of the socket is, and align it with the diagram. Always use a socket on this part, because firmware updates might require you to change out the part at a later date. After insertion, you can bend a couple of the pins on the backside of the board to hold them in place. Turn the board over and solder pins 1 and 18 (opposite corners). Check to make sure that the part is seated correctly before any other pins are soldered. If seated, then go ahead and solder the part in place.



3. Find R2 (BRN BLK BLU) and insert it into position. Bend the leads to hold it in place
4. Find C1 and C2, which are small capacitors with a 220 marking on their case. Insert and bend the leads to hold in place.
5. Find the crystal Y1 and insert into place, bending the leads slightly to hold it in place.
6. Insert switch S1 into the board. It should hold itself in place.
7. Insert R3 and R4 (ORG ORG BRN) then R1 and R5 (BRN BLK ORG) into the board. Bend slightly to hold the parts in place.
8. Now might be a good time to solder all of the parts into position. When you are done, trim the leads.
9. Insert capacitors C3 C4 C5 C6 C7 and C8 into position. These capacitors have a 104 marking on their case. Bend the leads slightly to hold them into position.
10. Insert U2 into position. This is the MAX232ACPE part. Bend a lead on the backside to hold it into position.

11. Solder and trim these parts.
12. J1 is inserted as a 2x5 right angle header**.**
13. Insert connector J2, the DB-9 female socket, into position. Be sure it is seated. Solder the pins on the backside. In addition, it is suggested that you solder the mounting tabs in the .125" holes. This makes a solid physical connection to the board. You need to fill between the mounting tabs and the hole. This helps hold the connector in place while you are connecting the RS-232 cable.



14.
15. Assemble the ribbon cable. Note that the ribbon connector has a small triangle on the case used to denote pin 1. The ribbon cable should have a 2x3 connector and strain relief on one end, a 2x5 connector and strain relief a few inches back, and a 2x5 connector WITHOUT a strain relief on the other. Pin one should be connected to the brown wire on the ribbon cable. You should align the brown wire with the triangle on the connector..
16. Before final assembly, you need to attach the ribbon cable to connector J1. Once the board is taped down, you will be unable to remove the cable. Pin 1 is marked on the silkscreen. This means that the triangle on the connector will be facing down on the board. Check your alignment carefully.
17. On the bottom half of the plastic case (the one without the holes), you should mount a small square of double sided foam tape to the 'rear' of the case. ONLY EXPOSE 1 SIDE OF THE TAPE UNTIL LATER! The rear of the case is the part farthest from the DB-9 connectors opening. If you dry fit the board first, it will be located directly below J1.This will eventually hold the board in place.
18. The ribbon cable exits the case through a notch in the top half. Try dry fitting the board into place using just half the plastic shell. Without actually closing the case, check to see if the LED's appear to line-up.

19. At this point, you may choose to test the BDM12 using the software procedures below. Otherwise, you can assume it is going to work, and finish the assembly below.
20. Remove the cover to the double sided foam tape on the bottom half of the shell. Position the board in the bottom half of the shell and press firmly into the tape. Now position the top shell over the LED's and snap together.
21. Congratulations, you are done!

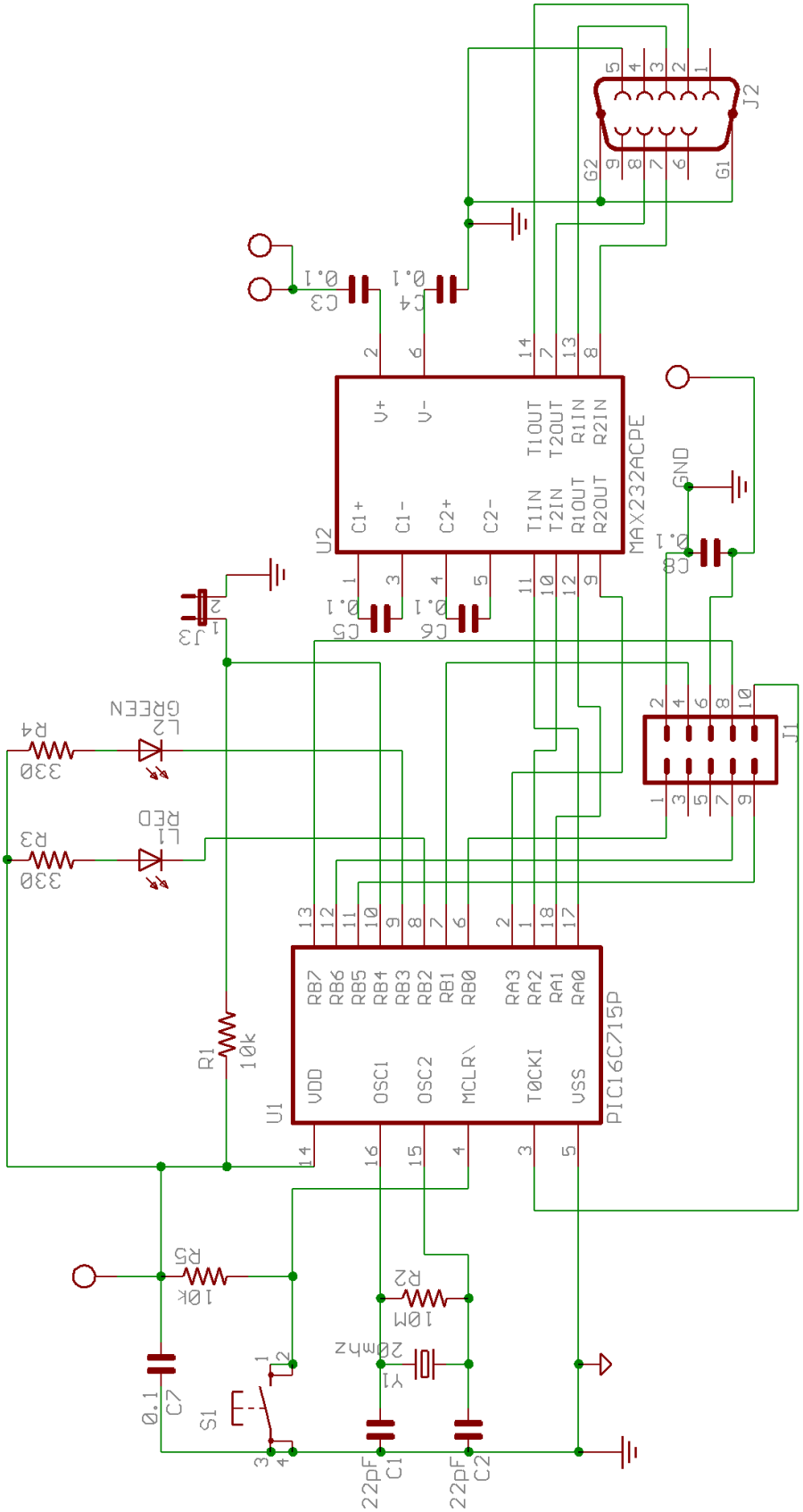


Figure 2

BDM12v5F Schematic

Pod Connections

DB9 RS-232 Connector

The pod requires the use of the CTS and RTS line on the host computer. This is very important, so don't leave it out!

The connection using an standard DB9 RS-232 cable is fairly straight forward. Plug it in to connector J3 and go. You need a DB9 Male to DB9 Female serial cable for your PC. It is recommended that you use a straight 9 conductor cable. Most computer stores sell these cables. For a bare minimum, you need to connect pins 2, 3, 5, 7, and 8. These wires are required for the pod to operate correctly. The CTS and RTS lines are used by the pod and the PC host debugging software for flow control.

Target connections

The BDM12 Pod is designed to be connected to the target system using the 'standard' 6 pin (2x3 pin configuration) header. The interface between the pod and the target system is implemented using a color coded ribbon cable. The ribbon cable is terminated with a header in a 2x5 pin configuration shown in the diagram. There are a few key points you should understand about this configuration.

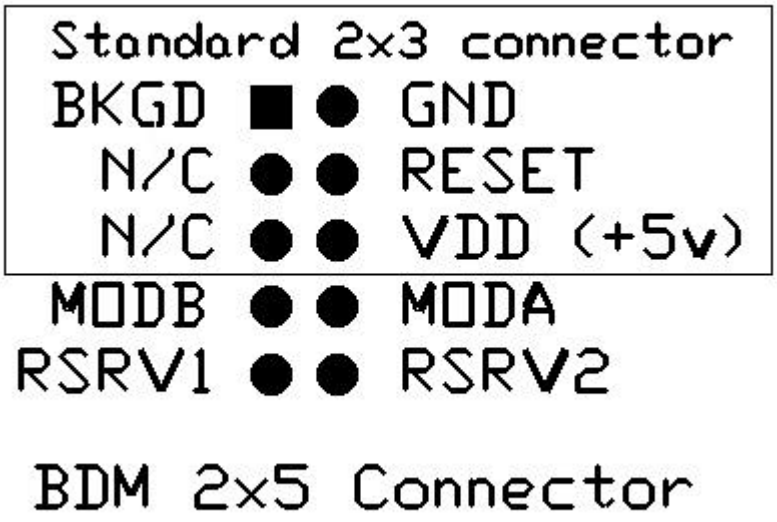


Figure 3 Standard Connector Wiring Diagram

1. The 'standard' 6 pin (2x3) connector is a subset of this 2x5 connector. Pin 1, which is the BKGD pin, through pin 6 (VDD) . This design allows for use of either the 2x5 'extended' connector, or the 2x3 'standard' connector.
2. The BDM12 Pod is powered via the standard connector. The BDM12 expects to find +5 volts and GND power on this connector.
3. The BDM12 Pod draws an average of 25mA of power (50mA maximum).
4. The BKGD pin should be pulled high on the target system using a pull up resistor. 10k is a recommended value for the pull up resistor.
5. The lines RSRV1 and RSRV2 are reserved for future use. They should be left unconnected.

Electrical Considerations for Target System

This section is going to discuss electrical considerations for designing a target system for use with the BDM12 debugging pod.

The BDM12 pod operates by controlling 4 control lines that are connected to the target system. These lines are the BKGD line, the RESET line, MODA, and MODB. Each of these lines are connected to output pins of the BDM12 pods Microcontroller (a PIC16C61). The BDM12 expects the lines to be pulled up or down as appropriate on the target system, and provides no other pull mechanism.

The BKGD line should always be pulled high on a target system. This enables the BDM12 pod to communicate correctly with the BDM module on the chip. If the line is not pulled high, then communication errors are quite frequent. A suggested circuit is shown here (the pin number is for the 812A4).

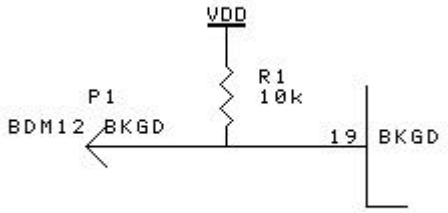


Figure 4 BKGD Pull-up configuration

The RESET line is another line that is controlled by the BDM12 pod. It is typical to pull the reset pin high with a 10k resistor. It is also common to have a Low Voltage Reset onboard to insure the integrity of the system in the event of power fluctuations.

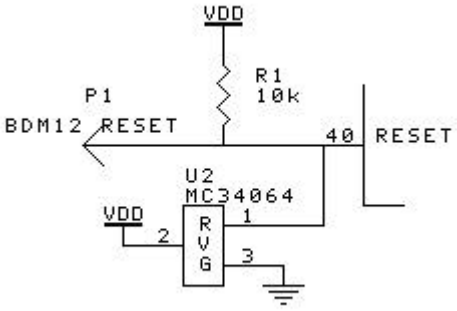


Figure 5 RESET Pull-up configuration

MODA and MODB select how the chip is initialized during startup. In conjunction with the BKGD pin, these two pins determine which bus mode the chip is going to operate in. These pins should be pulled high or low depending on the runtime state of your target system. They MUST use current limiting resistors if you are planning on connecting the MODA/MODB leads. Not using current limiting resistors can cause permanent damage to the BDM12 pod and/or your target system. For example, the following circuit sets up the chip to start in Expanded Narrow mode.

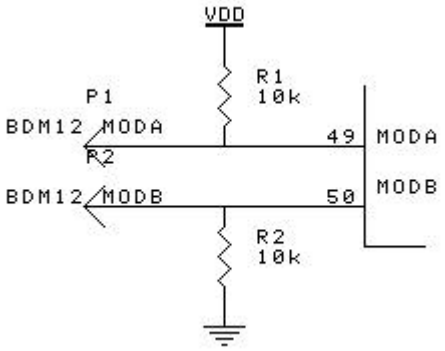


Figure 6 MODA/MODB Pull-up/down configuration

Another point to watch out for is the fact that MODA and MODB are also part of PORTE. They are PORTE pins 5 and 6 respectively. If your system is using MODA/MODB as output pins, then you probably don't want to connect the BDM12 pod using the MODA/MODB pins. You could end up with PE5 or PE6 going low, and the BDM12 pod driving them high, which could cause a short.

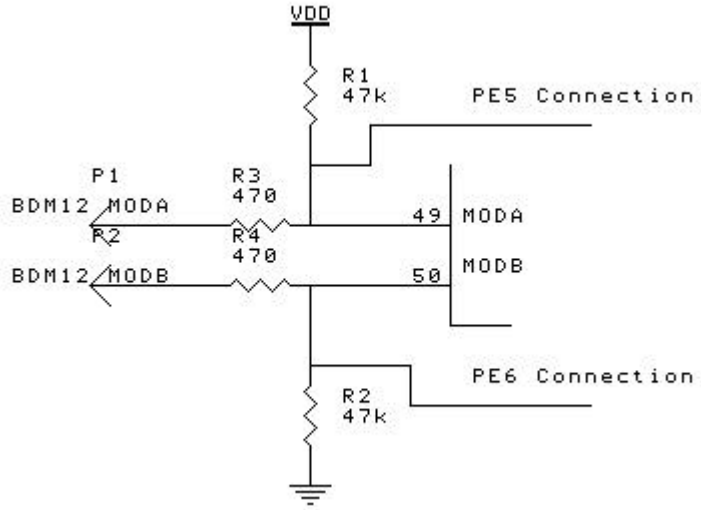


Figure 7 MODA/MODB PE5/PE6 Pull up/down configuration

To enable the use of PE5 and PE6, you need to add a series resistor to between the BDM12 pod and these pins. Note in the diagram that R1 and R2 are 47k, while R3 and R4 are 470 ohms. The resistor network formed here will insure that when MODA/MODB are pulled low by the BDM12, the voltage seen at the pins will be extremely close to zero. Your design needs to take into consideration the effects of having PE5/PE6 driven high or low by the BDM12 pod.

Operational Procedures

This section will describe some common usage scenarios for the BDM12 pod.

Basic setup and operation

The basic operation of the pod is very straightforward.

1) Connect the BDM12 Pod to the host computer (PC) using the DB9 connector.

2) Connect the BDM12 Pod to the target computer using the 2x3 ‘standard’ connector Be sure to align pin 1 correctly. If you constructed and inserted the cable correctly, it is marked with the triangle.
3) Apply power to the target system.
4) The BDM12 Pod should have a green LED lite, indicating that the pod is ready for use.
5) Start your debugging software such as the DB12 debugger.

If the target system is already running, and you don’t care if it resets, you can connect the BDM12 pod to a powered target. Chances are pretty strong that the target will reset during connection.

Expanded Memory Configurations

Note: As of the writing of this document, the db12.exe debugger does not make use of the MODA/MODB lines. However, other debuggers, such as the Stingray Debugger from Softtools (www.softtools.com***) does support MODA/MODB***

If your target system is using expanded memory, and you wish to be able to have the system reset the chip properly, then you will need to connect the MODA and MODB lines to the target system. The standard 2x3 header defined by Motorola did not have the MODA and MODB pins connected. Therefore, the standard connector doesn’t support systems with expanded memory.

The BDM12 pod handles this in two ways. There is a 2x5 (10 pin total) header that has the MODA/MODB control lines. You can connect these to the MODA/MODB pins on the target CPU. Your running system should pull these lines either low or high, depending the configuration, with a resistor. A 10k or 47k pull up/down resistor is recommended on your target system. See the section on Electrical Considerations.

Installing Software

The BDM12 is provided with software on a diskette. To install the software, insert the diskette into a drive and run setup.exe from that drive. Follow the instructions provided by the install program. It will create an icon for you to click on to start the db12 software.

Basic operation of DB12

Steps for setting configuration

The following checklist will help insure that you have correctly setup your BDM12 pod for use.

1. Determine the speed of your target CPU. The pod supports 8mhz, 4mhz, 2mhz, and 1mhz operation. The speed is determined by dividing the crystal frequency by 2. This result is called the ECLOCK speed. If you are running at a crystal that isn't listed, you might try the value closest to your target CPU speed. There is some leeway in the timing for the Motorola background debug interface. A custom version of the pod can be created to match the speed of the system you are developing for a nominal fee. Contact me if you are in need of this service.
2. Connect a DB-9 cable to the pod and to the RS-232 port of your computer, and J4 to the target system. Power up your target system. **Note: The pod powers itself by drawing power from the target system on connector J4.**
3. You should see the green LED light up on the pod board. If it does not, check the power connection on the target system.
4. On your host PC, startup the db12 debugger software. You can use the -c command line switch to change COM ports, and the -s command line switch to set the COM speed. DB12 will attempt to communicate with the pod. You should see a prompt with a 'R>' or a 'S>'. Watch for error messages in from the DB12 software.
5. If you see the green LED, but do not get a prompt, check your RS-232 cable and COM port settings.
6. If you get a red LED, then try hitting the reset switch. If it still comes up red, the two possibilities are a bad RS-232 cable, or a bad pod CPU. If the red light goes out, and the green light comes on, the pod is operating correctly.
7. If you are getting red lights after entering commands into db12.exe, it is possible you did not set the baud rates correctly. Check to insure that the command line switch matches the configuration switch on the pod board.
8. Issue an 'status' command by entering it into the DB12 command prompt followed by Enter. This should dump a single byte value. The value should be 40 or C0.
9. If the 'status' command returns FF, then the target BDM line did not respond correctly. This can be caused by many different things, including a repeated watchdog timeout. Try issuing a 'reset' command, which should reset the chip in special mode. This should bring the chip up into Background mode, and you can repeat step 10 and see a 40 or a C0.

The DB12 software

db12.exe is a simple debugging program that allows you to disassemble memory, load S19 files, single step, and modify RAM and registers. The following table shows the current command set. Updates to the debugger program are occasionally sent out via electronic mail. The program provides very basic help in the form of a help command.

DB12 Debugger Command Summary

b		Break into background mode. This stops the CPU execution and awaits further commands.
config		Dumps the current configuration to the screen
config autoload	“on” | “off”	Turns auto configuration file loading on or off. When turned ON, db12 will always load the last configuration file used. To set the configuration, use the config load, config save, or config file commands.
config autosave	“on” | “off”	Controls automatic saving of the configuration file. When db12 exits, and autosave is on, the current configuration is saved into the current configuration file.
config file	<filename>	Sets the configuration file name
config baudrate	<baud rate>	Sets the baud rate for communicating with the BDM12 pod. Default is 115200. <baud rate> is specified as a decimal number
config comport	<num>	Sets the comport number (decimal value) that the BDM12 pod is connected to.
config cputype	“812a4” | “912b32”	Sets the CPU type for the target system. Some features are different between chip versions, such as the flash programming, break point support, etc. Defaults to812a4. This command does NOT alter any of the memory map settings.
config freq	<frequency>	Sets the frequency of the crystal installed on the target system. This number is specified in decimal, and defaults to 16000000. To make life easier, you can specify just the MHZ value. For example, config freq 16 is equivalent to config freq 16000000 This setting is important on systems with Flash EEPROM, because the timing used in the downloader program are different based on the frequency of the target systems clock.
config reset	“812a4” | “912b32”	Config reset sets the target system CPU type, and also sets the appropriate memory maps information for the chip. This command is assumes that the target system runs in single chip mode with a standard memory map.
config verify	“on” | “off”	Sets memory verify after write on or off. This tells the db12 software if it should verify all memory writes. Useful to turn off if you are changing memory variables that the software running on the target might change immediately.
config regbase	<addr>	Sets the base address of the register page. Defaults to zero
eeprom		Dumps the current status of the eeprom
eeprom autobulk	“on” | “off”	Controls if the download code always performs a bulk erase of the EEPROM before downloading code.
eeprom protect	“on” | “off”	Sets the eeprom protection bits on or off
eeprom erase	<addr> | “all”	Erases the EEPROM block starting at specified address. ‘all’ specifies the erasing of all EEPROM blocks. All is used in the event that the device supports more than one EEPROM range
eeprom set	<start> <stop>	Sets an EEPROM address range from <start> up to an including <stop>. Use this command if you have moved the EEPROM in memory
eeprom clear	<addr>	Removes an EEPROM address range from the memory map. <addr> is the starting address of the EEPROM range to remove
eeprom show		Print a list of EEPROM ranges currently in memory
d		Dump memory using last address and size. This is a shorthand to continue the previous ‘dump’ command
db	[<range>]#	Dump BYTES

dw	[<range>]	Dump WORDS
eb	<addr> <byte>	Edit byte of memory. *
er	<reg> <value>	Edit register contents. Valid registers include A,B,D,X,Y,PC,SP,CR
ew	<addr> <word>	Edit word of memory. *
firm		Enable Firmware Commands
fload	<filename.s19>	Loads FLASH EEPROM on a 68HC912B32
flash		Prints information about the Flash EEPROM array
flash protect	"on" | "off"	Sets the Flash EEPROM protection bits
flash erase	<addr>	Erases the entire Flash EEPROM array starting at address <addr>
flash load	<filename.s19>	Loads an S19 file into Flash memory. Same as fload
g	[<addr>]	Go at current PC or address
help		Displays this help message
load	<file>	Load S19 file to memory
log		Shows current logfile status
log append	<filename>	Opens in append mode the log file. All output from the debugger is copied to both the console and the file.
log close		Flushes then closes the logfile
log open	<filename>	Opens a new logfile (or replaces an existing file with the same name)
quit		Exit Debugger
r		Dumps Registers. The CPU must be stopped (see the break command) in order to display registers.
reset		Reset chip into Special mode
run	<filename>	Runs a command script file. Accepts command input from a file. Allows for simple macros to be stored in separate files. Commands echoed from the file will be prepended with a '>>' string to indicate that the commands originated in a script file.
save	<range> <file>	Save memory to S19 file
status		Dumps the STATUS byte for the BDM firmware. Used to check for active BDM mode.
t	<count>	Single step <count> times (default is 1). Single steps, then dumps the registers. If you wish to stop the trace, press any key, like the space bar.
tq	<addr>	Trace Quietly to <addr>. Causes the debugger to single step quietly until the PC == <addr>. Used to quickly step through or over routines. Pressing any key, such as the space bar, will cause the tq command to stop early. Prints a report of the number of instructions executed when completed or stopped.
tt	<addr>	Trace To <addr> Causes the debugger to single step until the PC == <addr>. Used to quickly step through or over routines while still getting a register dump after each instruction. Pressing any key, such as the space bar, will cause the tq command to stop early. Prints a report of the number of instructions executed when completed or stopped.
u	[<range>]	Unassemble range

[#] A Range consists of two WORD sized HEX addresses separated by a space. Example: db f000 f0ff

* To edit the contents of EEPROM, you will need to manually change the EEPROT register. By default, the EEPROT register is at $00F1. Write a $00 to this byte value to enable the changing of EEPROM memory.

DB12 accepts commands from the console, and outputs its data to a command line window. There are some parameters that can be set on the command line when DB12 starts:

db12 [options]
where [options] include:
-c <num> Set com port to <num> (eg '-c 1')
-s <speed> Set baud rate to <num> (eg '-s 115200')
-f <filename> Set the configuration filename

Defaults settings are -c 1 -s 115200

Here are some random points about using DB12

- You can set your favorite command line parameters in an environment variable called DB12. For example: `set DB12=-c 2 -s 115200`
- The prompt provides useful information about the state of the running target. A prompt of 'R>' means that the target CPU is running its program. A prompt of 'S>' means the target is stopped (or is in active BDM mode). Some commands only work with the CPU stopped, such as dumping registers.
- Loading S19 files generally stops the CPU. The 'load' command will reset the chip into special single chip mode before downloading.
- The 'fload' command downloads code to the Flash EEPROM of the 68HC912B32. This command works by resetting the chip, then downloading a special program to the RAM locations 0800-0BFF. The special program is then run in RAM, and does the actual work of programming the FLASH. As a result, the contents of RAM are erased during Flash EEPROM programming. Also note that writing to RAM during Flash EEPROM programming will more than likely crash the target.

Standard Operating Procedures

Downloading an S19 file

This section assumes you have successfully created an S19 file that is appropriate for the target system you are working with.

1) Power up your target system with the BDM12 pod connected. Start the DB12 software, insuring that the communications parameters match the defaults for your BDM12.
2) Type the command 'reset'. This will reset your target system, and bring it up in special mode.
3) If you are loading RAM and EEPROM, then use the 'load' command. If you are loading the Flash EEPROM of the 68HC912B32, then use the flash load command. To program the Flash EEPROM, you must apply the 12 volt Flash programming voltage. Also, if you are programming Flash, check to see that the crystal speed is set correctly using the '**config**' command. If you are using a different speed, the '**config freq <speed>**' will change this appropriately.
4) Once loaded, you can type the 'reset' command again, and you should see the first instruction of the reset routine.

Using the Translator mode

Translator mode is intended to be used via a host based program. It will accept and transfer all of the commands shown in the BDM section of the HC12 reference manual. In addition, there are a few commands that are processed internally by the pod. They are shown below:

Command	Opcode	Data	Description
Sync code	$00		Ignored by the pod. Used to reset and sync the translator interface after an error is detected. Sending 6 of these will turn off the red LED indicator.
Reset CPU	$01		Resets the HC12 by pulling the RESET line low, the

			BKGD pin low, raising the RESET line, then raising the BKGD pin. HC12 should come up in special mode.
Reset LOW	$02		Sets the RESET line to low and holds it there.
Reset HIGH	$03		Sets the RESET line to HIGH and holds it there.
Enter Debug Mode	$04		Switches pod protocol to debugger mode. Compliments the '#' command in debugger mode.
EEPROM Write	$05	<addr><word>	Performs WORD sized write to EEPROM. Address MUST be aligned.
Set register base	$06	<byte>	Allows the setting of the register base MSB. The register page can be mapped to various addresses. If the registers have been moved to an address not starting at 0, then the MSB of the address should be written via this mechanism. The pod will use this byte as the high 8 bits of register addresses for programming the EEPROM
Bulk Erase EEPROM	$07	<addr>	Performs a bulk erase of EEPROM starting at address

Translator Commands

Determining the status of the target/Dumping registers

When you are debugging an HC12 system, the core CPU runs independently from the debugger interface. This is a great feature, but can lead to confusion when working with the debugger. Some commands require the CPU to stopped by the debugger, or in Background Mode, before they give meaningful results. For example, dumping the registers doesn't give meaningful results when the CPU is running. This is because the debugger reads each register independently from memory. If the CPU is running, you will not get an consistent result because the values of the registers are changing rapidly while the CPU is running.

To determine which state the CPU is in, you need to dump the status register. The 'status' command is provided, and will print out a byte hex value for the STATUS byte as documented in the HC12 manual. Bit 6 (of bits 0-7) indicates when the CPU is waiting for debugger input. If you look at the definition of this byte value, you will see that the upper values of the STATUS byte (0xFF01) have a few key bits set. The lower bits don't matter as much. Some result values from the 'status' command are shown below for the upper nibble:

STATUS BYTE	DESCRIPTION
Cx	Firmware commands enabled, BDM active and waiting for command. Register dump works.
8x	Firmware commands enabled, CPU running/BDM not active. Needs break command to dump registers
4x	Firmware commands disabled, BDM active and waiting for command.
FF	Usually means the CPU isn't responding to the BDM commands. You may need to perform a reset command 'Z'. This is also may indicate a communication error between the BDM12 interface and the target system.

BDM12 Firmware

Programmers Specification

This section will provide more details about how to write software to communicate and operate the BDM12 interface.

Determining the version and capabilities of a BDM12 pod

There are two basic flavors of BDM12 in existence at the time of this document being written. The original version used a simple CTS line handshake (CtsOnlyControl) to control the flow of data from the host PC. However, the CtsOnlyControl mechanism started to fail on some newer machines running Windows/95 or Windows/98. The reason appears to be an incompatibility with some of the newer UART chips which do not recognize some transitions in the CTS line.

To fix this problem, a new handshake has been implemented that uses both the CTS and RTS lines (CtsRtsControl). The exact handshake for both versions are describe in a moment. This section will help you determine which version you actually have.

The determination is based on what happens if the RTS line is raised and lowered without sending data. The older CtsOnlyControl BDM12 pod will hold the CTS line high while it is ready to receive data. The newer CtsRtsControl version of the BDM12 will only raise the CTS line if the RTS line is high. If the RTS line goes low, then the CTS line will follow it.

To determine which pod you have, you simply raise the RTS line, and check for the CTS line. If the CTS line when the RTS line is high, then you at least have something attached to the port. To determine if it is a newer version, lower the RTS line. If the CTS line lowers, then it is a newer pod. If the CTS line stays high, it is the older version.

The following code will help you accomplish this task:

```
//+--------------------------------------------------------------------
// BDM12::SetRTSLine(BOOL fState)
//
// Sets the RTS line to the state in fState
//
//+--------------------------------------------------------------------

int BDM12::SetRTSLine(BOOL fState)
{
  int res = 0;

#ifdef WIN32
  if(!EscapeCommFunction(m_hCommDev,fState?SETRTS:CLRRTS))
  {
      res = GetLastError();
  }
#endif

#ifdef LINUX
  int tiocm = TIOCM_RTS;
  res = ioctl(m_hCommDev,fState?TIOCMBIS:TIOCMBIC,&tiocm);
#endif

  return res;
```

```
}
//+----------------------------------------------------------------------
//
//  Method:     BDM12::CheckCTSLine
//
//  Synopsis:   Checks to see if the CTS line is set. Platform dependent
//
//  Arguments:  (none)
//
//  Returns:    CTS_LINE_SET, CTS_LINE_CLEAR, or an error code
//
//----------------------------------------------------------------------
int BDM12::CheckCTSLine()
{
    int res = 0;
#ifdef WIN32
    BOOL fGetState;
    DWORD dwModemState;
    //
    // Check the CTS line.
    //
    fGetState = GetCommModemStatus(m_hCommDev,&dwModemState);
    if (!fGetState)
    {
      // Error occurred getting modem status
      res = GetLastError();
    }
    else
    {
      res = (dwModemState & MS_CTS_ON)?CTS_LINE_SET:CTS_LINE_CLEAR;
    }
#endif

#ifdef LINUX
    struct serial_icounter_struct sic;
    unsigned s;
    int ncts;

    res = ioctl(m_hCommDev,TIOCMGET,&s);
    if(res < 0)
    {
      return errno;
    }

    res = (s & TIOCM_CTS)?CTS_LINE_SET:CTS_LINE_CLEAR;

#endif
    return(res);
}

// Determine if this is a newer version by checking the
// CTS/RTS interaction. On the newer BDM12, the pod watches the state of
// the RTS line. If the line is not asserted,then CTS will not be
// asserted. If the line IS asserted, the CTS will be asserted.
//
// Start by insuring that CTS is asserted. In either case, it should
// be if RTS is asserted

int iState;
SetRTSLine(TRUE);

m_ucVersion = 0;
```

```
if(CheckCTSLine() == CTS_LINE_SET)
{
    // It is ON when it should be. There may be a pod attached.
    // See if the CTS will turn off
    SetRTSLine(FALSE);
    if((iState = CheckCTSLine()) == CTS_LINE_CLEAR)
    {
        // This is a pod that does the new handshake
        m_fCtsRtsControl = 1;
    }
    else if(iState == CTS_LINE_SET)
    {
        // This pod doesn't do the new handshake
        m_fCtsRtsControl = 0;
    }
    else
    {
        // Some sort of error occurred! Handle it appropriately
    }
}
else
{
    // It is quite possible that nothing is attached!
}
SetRTSLine(FALSE);
```

The code isn't too complicated. The main thing to be determined is if the CtsRts control handshake

Communication with the BDM12 CtsOnlyControl (v4.4 or lower)

The BDM12 (v4.4) pod uses the simple CtsOnlyControl handshake to insure that data is not lost. The BDM12 will assert the CTS line on the RS-232 port when it is able to receive data. When the pod is busy, it will drop the CTS line. Therefore, a serial routine that communicates with the BDM12 pod will need to monitor this line, and only send data when the CTS line is asserted. Note that the BDM12 pod will only lower the CTS line in response to input from the host PC. Therefore, no race condition exists between the checking of the CTS line and the actual sending of the data. Once the CTS line is asserted, the pod will wait for the next byte before lowering it again. This makes the programming much easier. The following code example is a routine that writes to the BDM12 pod using Win32 interfaces:

```
int BDM12::Write_To_BDM(UINT uByteCount, BYTE *pbData)
{
   DWORD uWritten;
   int res;
   UINT i;
   DWORD dwStartTime;
   DWORD dwCurrentTime;
   DWORD dwTimeSpan;
   DWORD dwModemState;

   UINT c;

   for(c=0;c<uByteCount;c++)
   {
         dwStartTime = GetMillisecondCount();
         dwCurrentTime = dwStartTime;
         int iLoop = 2;
```

```
            //
            // Wait for the CTS line to be asserted. This while loop will insure that we
            // don't wait for an unreasonable amount of time, which is currently
            // 100 milliseconds
            //
            while(((dwTimeSpan = dwCurrentTime - dwStartTime) < 100) || iLoop)
            {
               res = CheckCTSLine();

               if(iLoop)
               {
                     iLoop--;
               }

               if(res == CTS_LINE_SET)
               {
                     break;
               }
               if(res != CTS_LINE_CLEAR)
               {
                     //
                     // Something went wrong reading port
                     //
                     m_pMC->ErrorMessage("CTS not set error %x\n",res);
                     return res;
               }

               if(m_fDebugOutput)
               {
                     m_pMC->DebugMessage("*");
               }
               m_dwCtsWaitLoops++;
               dwCurrentTime = GetMillisecondCount();
            }
            if(res != CTS_LINE_SET)
            {
               //
               // The CTS line was never asserted, therefore something is wrong
               // with the communication
               //
               m_pMC->ErrorMessage("CTS not asserted by BDM interface.\nCheck cable and BDM12
power");

               return(ERROR_CTS_FAILURE);
            }
            //
            // Finally, write the byte. If you start seeing error 1121 (0x461), then it
            // probably means that there is a byte already being sent. This is the problem
            // that all of the following code should have fixed.
            //
            res = WriteToComm(&pbData[c],1,&uWritten);

            if(res)
```

```
        {
           m_pMC->ErrorMessage("Write to BDM failed with error 0x%x",res);
           return res;
        }
        if(uWritten != 1)
        {
           m_pMC->ErrorMessage("Write to BDM failed to complete all bytes.\nCheck interface or
cable");
           return ERROR_INCOMPLETE_WRITE;
        }

        dwStartTime = GetMillisecondCount();
        dwCurrentTime = dwStartTime;
        iLoop = 2;

        //
        // Monitor the CTS line to insure it goes low. There is a problem on Windows/95
        // with some serial port implementations that causes problems with detecting the
        // CTS line. We need to insure it goes low, otherwise we will overrun the pods
        // input buffer by sending another character before it gets a chance to deal with
        // it. In the event that more than 2 milliseconds have passed, it probably means
        // that this process was task switched. The byte should be long gone by now.
        //
        DWORD lc = 0;
        while(((dwTimeSpan = GetMillisecondCount() - dwStartTime) < 2) || iLoop)
        {
           res = CheckCTSLine();

           if(iLoop)
           {
                iLoop--;
           }

           if(res == CTS_LINE_CLEAR)
           {
                break;
           }
           if(res != CTS_LINE_SET)
           {
                //
                // Something went wrong reading port
                //
                m_pMC->ErrorMessage("CheckCTSLine for clear error %x\n",res);
           }
           m_dwCtsChangeLoops++;
        }
        if(dwTimeSpan >= 2)
        {
           m_dwCtsChangeMissed++;
        }
        m_dwBytesSent++;
   }
   return 0;
}
```

Communication with the BDM12 CtsOnlyControl (v4.5 or higher)

The newer CtsRtsControl handshake uses the CTS and the RTS lines. The CTS line is asserted by the pod while the RTS line is asserted by the host system. In pseudo code, the steps to write a byte to the pod are pretty simple:

1) Assert the RTS line. In response, the pod, when ready, will raise the CTS line.
2) Wait until the CTS line goes high.
3) Send the byte
4) The pod receives the byte. It will lower the CTS line, and is now waiting for the RTS line to drop
5) The PC checks the CTS line. If it has been lowered, then the RTS line is dropped
6) The pod, seeing the RTS line drop, continues execution

While that seems like a time consuming set of events, in reality, it really doesn't take much time nor does it slow the communications with the pod down. On many systems, especially those which show this CTS line incompatibility, this new handshake actually speeds up the communication with the pod.

The following code shows how to write to the pod using the CtsRtsControl handshake.

```
int BDM12::Write_To_BDM2(UINT uByteCount, BYTE *pbData)
{
   UINT uWritten;
   int res;
   UINT i;
   DWORD dwStartTime;
   DWORD dwCurrentTime;
   DWORD dwTimeSpan;
   int iCTSLine;

   for(i=0;i<uByteCount;i++)
   {
         //
         // Assert RTS. This should cause the pod to raise its CTS line when
         // it is able to accept a byte.
         //
      SetRTSLine(TRUE);

         //
         // Wait for CTS to assert. This is done in response to the RTS line
         // being asserted To insure that we don't just go away and hang, a
         // timeout period is recognized.
         //
         // As is the problem with timeouts on
         // a multi-tasking system, you need to insure a busy system doesn't
         // cause errors. Therefore, a separate loop count insures we get
         // through the loop at least twice. Since the pod runs asynchronously,
         // it should be done before the timeout duration occurs regardless of
         // the CPU load.
         //

         dwStartTime = GetMillisecondCount();
         dwCurrentTime = dwStartTime;

         int iLoop = 2;
```

```
        while(((dwTimeSpan = dwCurrentTime - dwStartTime) < 100) || iLoop)
        {
          iCTSLine = CheckCTSLine();
          if(iCTSLine == CTS_LINE_SET)
            {
              break;
            }
          else if(iCTSLine != CTS_LINE_CLEAR)
            {
                    m_pMC->ErrorMessage("WriteToBDM2 CheckCTSLine error 0x%x",iCTSLine);
            }
            dwCurrentTime = GetMillisecondCount();
            if(iLoop)
            {
                    --iLoop;
            }
        }
        if(iCTSLine != CTS_LINE_SET)
        {
            m_pMC->ErrorMessage("WriteToBDM2 CTS not asserted\n");
            return(ERROR_CTS_FAILURE);
        }

        //
        // Write Byte using the system API
        //
        res = WriteToComm(&pbData[i],1,&uWritten);

        if(res)
        {

            m_pMC->ErrorMessage("Write to BDM failed with error 0x%x",res);
            return res;
        }
        if(uWritten != 1)
        {
            m_pMC->ErrorMessage("Write to BDM failed to complete all bytes.\nCheck interface or
cable");
            return ERROR_INCOMPLETE_WRITE;
        }
        //
        // Wait for CTS to low, using the same timeout mechanism as above.
        //
        dwCurrentTime = GetMillisecondCount();
        dwStartTime = dwCurrentTime;
        iLoop = 2;

        while(((dwTimeSpan = dwCurrentTime - dwStartTime) < 100) || iLoop)
        {
          iCTSLine = CheckCTSLine();
          if(iCTSLine == CTS_LINE_CLEAR)
            {
              break;
            }
          else if(iCTSLine != CTS_LINE_SET)
```

```
            {
                m_pMC->ErrorMessage("WriteToBDM2 CheckCTSLine error 0x%x",iCTSLine);
                break;
            }
            dwCurrentTime = GetMillisecondCount();
            if(iLoop)
            {
                --iLoop;
            }
        }

        // Lower RTS. The pod will wait for this to happen before continuing
        SetRTSLine(FALSE);

        if(iCTSLine != CTS_LINE_CLEAR)
        {
            m_pMC->ErrorMessage("WriteToBDM2 CTS not cleared\n");
            return(ERROR_CTS_FAILURE);
        }
    }
    return 0;
}
```

Software Commands and Packets

The BDM12 pod, when working in translator mode, is designed to accept commands from the RS-232 line, pass them to the target system using the Background Debug Mode of the `HC12, and depending on the particular packet sent, pass the results back to the PC. In addition, there are several commands that affect the operation of the BDM12 pod itself. They are listed in this section along with their programming parameters. Most of the commands are single byte commands, which require no additional arguments.

Sync $00

The Sync command is used to sync the BDM12 pod and the host PC. If the pod is currently in debugger mode, then the Sync command will cause the pod to enter translator mode. If the pod is currently in an error state, then the sync command will take it out of error mode. There are no data arguments.

Reset CPU $01

The Reset CPU command instructs the BDM12 pod to reset the HC12 by lowering the RESET line on the target, lowering the BKGD line on the target, waiting 10ms, raising the RESET line on the target, then raising the BKGD line on the target. The HC12 is should be reset into special mode, which has the BDM interface active. There are no data arguments.

Reset LOW $02

The Reset LOW command will lower the target systems RESET line, and leaves it there. This is useful if you need the ability to hold the line low while some other operation is underway. There are no data arguments for this command. (See the Extended Command IOCTL as well)

Reset High $03

The Reset HIGH command will raise the target systems RESET line and leaves it there. This is useful if you need the ability to hold the line high while some other operation is underway. There are no data arguments for this command. (See the Extended Command IOCTL as well)

Enter Debug Mode $04

(Firmware 4.4 and below only) The Enter Debug Mode command is used to have the BDM12 debugger enter the onboard mini-debugger. This feature was removed in firmware v4.5 and above in favor of extending the firmware commands.

Extended Command $04

(Firmware 4.5 and above only!) Extended Commands are prefaced by the $04 command. There are a number of extended commands currently available for use.

Extended Code	**Operands**	**Operational Description**
$00 VersionNum	<none>	Returns a byte with the firmware version number
$01 RegDump	<none>	Full register dump. Returns a dump of all registers
$02 TraceTo	<Flags> <Addr>	TraceTo single steps through the code to a specific address
$03 MemDump	<addr> <wcnt>	Dumps wcnt WORDS of memory starting at <addr>
$04 SetParam	<Flags> <addr> <addr>	Performs a memory move operation from <addr> to <addr>
$05 IOCTL	<byte><wcnt><addr>	Fill wcnt bytes of memory starting at addr with byte
$06 MemPut	<addr> <wcnt> [<words>]	Fill a memory range starting at <addr> with word sized data.
$07 ExtendedSpeed	<byte> <word>	Sets the pod to extended speeds. See text (v 4.7 or above)

Each of these extended commands is discussed in more detail below. Before that discussion, you need to know about the format of the ReplyPacket. A ReplyPacket is a data structure used for the output of several commands. The ReplyPacket is a byte packed data structure that contains a variable number of fields. The ReplyPacket starts with a header byte that denotes what is in the packet. This header byte is present in all packets, and is the minimum packet you can expect to see. The possible flags include:

STATUS_BYTE	0x01	Packet contains the status byte from the BDM memory of the target. Used to determine the current state of the target
REGISTERS	0x02	Packet contains a register dump. The registers are dumped in the order specified below in the RegDump command
PC	0x04	The Program Counter (PC) register is in the packet.
PACKET_DWORD	0x40	A 32-bit value is in the packet. Used to pass back data such as the instruction count. Interpretation of this value depends on the specific call.
PACKET_DONE	0x80	This denotes that an operation has completed, and that this is the last packet. Used to determine when an operation such as a TraceTo operation has completed.

These flags are not exclusive, meaning that zero, one, or multiple fields may be included in the ReplyPacket. The order of the data items is indicated by its numeric value. STATUS_BYTE will be first,

REGISTERS second, PC third, etc. If a flag is not set in the header, then it doesn't exist in the data packet.

Command: GetVersion

OpCode: $04 $00

Synopsis: Returns the firmware version of the BDM pod

Arguments: None

Return Bytes: On completion of this command, a single byte BYTE(bVersion) is returned. The high bit ($80) determines if the hardware and firmware supports the extended control lines. If set, then the firmware has the MODA/MODB/RSRV1/RSRV2 lines available for control. Otherwise, the only IOCTL lines available are the BDM/RESET/REDLED/GRNLED lines.

The low 7 bits denote the firmware version. Versions numbers have a major and minor component. The low 4 bites are the minor number, and the high 3 bits are the major number. Therefore, a version number of 0x46 is considered to be version 4.6

Command: RegDump

OpCode: $04 $01
Synopsis: Dumps the contents of all registers, including the BDM Status register.
Arguments: None
Return Bytes: On completion of this command, the following are returned to the serial port. This particular command is also used by several other commands, so the format of the return packet is used often.

Byte#	Description
1	HI(PC)
2	LO(PC)
3	HI(D) *[A]*
4	LO(D) *[B]*
5	HI(X)
6	LO(X)
7	HI(Y)
8	LO(Y)
9	HI(SP)
10	LO(SP)
11	Condition Code Register CCR

Command: TraceTo

OpCode: $04 $02
Synopsis: Single steps instructions up to but not including the specified address. This command allows the user to quickly step through code to a specific address.
Arguments: BYTE(Flags) Allows specification of various features

$01 Output a register dump after each instruction traced
$02 Output an instruction count on exit
$04 Addr represents a count of trace instructions, rather than an address. Used to execute a specific number of traces or stepped instructions, rather than tracing to an address.
$08 Outputs just the PC after each instruction traced.

WORD(Addr) Address to trace to

Description: TraceTo is used to quickly step to a specific address. This is very useful when you wish to quickly step over a routine, or when you wish to count the number of instructions executed in a routine. On parts with Flash Memory for code space, this function is a quick and useful way to get around the limited number of breakpoint registers.

This command has an optional flag that allows for the output of a full register dump after each instruction. This is useful to get a trace log of the operation of the program.

If the $04 bit is set, then the WORD argument Addr is used as a trace count rather than an address. The routine will execute Addr number of traces.

If the $08 bit is set, and the $01 bit is cleared, then just the PC is dumped after each trace operation.

When the command is completed, a final register dump is generated regardless of the optional flag.

The RTS line should be clear for the duration of the trace. Setting the RTS flag will interrupt the command at the next instruction boundary, and will cause a final register dump and the instruction count to be generated.

Return Bytes: If the $01 bit is set in the Flags, then after each instruction traced, a RegDump will occur.

At the end of the trace sequence, the following return bytes are issued:

0-11	See the RegDump command)
12-15	32-bit DWORD with the instruction count. This is an unsigned number **dumped in LSB order**. This word is optional, depending on the $02 bit in the Flags.

Command: MemDump

OpCode: $04 $03
Synopsis: Dumps a range of memory WORDS. Used for reading memory quickly
Arguments: WORD(wAddress) Starting address
WORD(wCount) Number of words to dump
Description: This function dumps wCount worth of words starting at wAddress. There are restrictions on the arguments.

wAddress MUST be word aligned (ie an even address).
wCount MUST be greater than zero
Retured Bytes: A series of <wCount> WORDS is returned

Command: SetParam

OpCode: $04 $04
Synopsis: Sets operating parameters for the BDM12 pod
Arguments: BYTE(bECLOCK) Determines the ECLOCK speed for the pod to operate at
BYTE(bPaceCount) Specifies a delay for the continuous output of trace information
BYTE(bResetCount) Specifies a delay between raising of the RESET line and the BKGD pin.
Description: This command is used to set some operating parameters for the BDM12 pod. Newer versions of the BDM12 debugging pod do not have the configuration switches of the older versions. To set target dependent information, such as the speed of the target, the BDM12 pod needs to have SetParam called.

The definitions for bECLOCK are as follows. All other values are invalid.

ECLOCK_1MHZ	0x00	
ECLOCK_2MHZ	0x01	
ECLOCK_4MHZ	0x02	
ECLOCK_8MHZ	0x03	
ECLOCK_EXTENDED	0x04	(v4.7 or greater)

To properly use the ECLOCK_EXTENDED value, you should set the appropriate values using the ExtendedSpeed command (opcode $07). Once the ExtendedSpeed is processed, then you may send the SetParam with ECLOCK_EXTENDED as the ECLOCK field. See the ExtendedSpeed command for more details.

bPaceCount slows the speed of trace information being dumped by adding a delay loop

between outputs. The delay is between 0 and 255 milliseconds. The actual delay is approximate within a 50 microseconds or so.

bResetCount adds an additional period of time between the RESET line going high and the BKGD pin going high. This is used in cases where the target requires additional time for the reset, such as a LVR reset part that has a built in timer. The values are between 0 and 255 milliseconds, and again are approximate within about 50 microseconds or so.

Returns: Nothing

Command: IOCTL (firmware v46 or greater)

OpCode: $04 $05
Synopsis: Allows for external control of the control lines on the BDM12 pod
Arguments: BYTE(bLineControl)
Description: The BDM12 pod has several control lines, such as the BKGD (label BDM) line and RESET line. The IOCTL command allows an external program to set or reset these lines.

The bit values allowed are as follows.

BDM_IOCTL_BDM	0x01
BDM_IOCTL_RESET	0x02
BDM_IOCTL_RED	0x04
BDM_IOCTL_GRN	0x08

The following lines are only available on pods with extended control lines.

BDM_IOCTL_RSRV2	0x10
BDM_IOCTL_RSRV1	0x20
BDM_IOCTL_MODB	0x40
BDM_IOCTL_MODA	0x80

The IOCTL call does a set of all the pins. You MUST send the entire state each time.

Command: MemPut (firmware v46 or greater)

OpCode: $04 $06
Synopsis: Writes a range of memory WORDS. Used for writing memory quickly
Arguments: WORD(wAddress) Starting address
WORD(wCount) Number of words to put
[<WORD>] A series of wCount <WORDS>
Description: The MemPut command is used to quickly write to memory of the target. The memory is written using a series of WRITE_WORD commands on the pod. The speed advantage is derived by not having to send the command and address codes for each WORD write.

The word wAddress MUST be word aligned (even)
The word wCount MUST be > 0
Returns: Nothing

Command: Extended Speed (firmware v47 or greater)

OpCode: $04 $07
Synopsis: Sets the target CPU speed to a non-standard speed

Arguments: BYTE(bEClockScalar) Scales the ECLOCK
WORD(w128EClocks) Delay count for 128 EClocks

Description: The Extended Speed packet allows the pod to operate with target CPU's with ECLOCK ratings other than the standard 8mhz, 4mhz, 2mhz, or 1mhz. This is done by using a variable length loop for timing on the BDM12 pod. To pull this off, two values need to be calculated by the driving program.

The first value is the EclockScalar. This value is used to determine the length of each ECLOCK. The value sent is used as a loop counter for each BDM bit transferred. To calculate the appropriate EclockScalar, you need to know the approximate ECLOCK of the target system, and run it through the following formula:

First, calculate the ECLOCK_PERIOD_NS variable, which is the ECLOCK period in nanoseconds:

ECLOCK_PERIOD_NS = 1000000000 / ECLOCK_FREQUENCY

Then calculate the EclockScalar according the following

EclockScalar = (ECLOCK scalar formula)

The second value is a 16 bit value that is used to calculate a 128 ECLOCK delay. This delay is used by the pod to adhere to the 128 ECLOCK delay required by the BDM interface of the target. It allows the target time to process some commands.

To calculate this 16-bit value, you need to use the ECLOCK period in the following formula:

w128ECLocks= ((ECLOCK_PERIOD_NS * 128) - 1400) / 800

The 128 is for 128 ECLOCKS. The 1400 is the internal loop overhead in the BDM12 pod, and the 800 is the overhead per count in the clock. Rounding up is safer than rounding down.

The word wAddress MUST be word aligned (even)
The word wCount MUST be > 0

Returns: Nothing

EEPROM Write $05

The EEPROM Write command performs a WORD sized write to EEPROM. There are two data arguments, the address and the word. A data packet for this command should look like:

$05	HIBYTE(Addr)	LOBYTE(Addr)	HIBYTE(Data)	LOWBYTE(Data)

Requirements: Addr MUST be WORD aligned.

The result of this command is that the BDM12 pod will set the appropriate registers to perform a WORD aligned write of EEPROM memory. This command takes up to approximately 22ms to complete, and the CTS line will be held low during this time. The BDM12 pod will use an intelligent programming algorithm to determine if the WORD must be erased first. This will affect the programming time.

Note that the PC software is responsible for setting the EEPROM protection bits in the register base address + $00F1 EEPROT registers.

Set Register Base $06

Set Register Base is used to inform the pod when the register base has been moved. The HC12 allows the user to specify at which address the registers are mapped. The registers may be mapped to any 2 page boundary. This command takes one byte argument, which is the high byte of the register base address. The default is for this value to be zero, which is the chips default. The following is the data packet format for this command.

$06	High Byte of Register Base

The BDM12 pod uses some of the registers on the target system to control things like EEPROM writes. If the user has moved the register base, and this command is not issued, then writing to EEPROM will not function correctly.

Bulk Erase EEPROM $07

This command will erase the contents of the EEPROM. The Bulk Erase EEPROM command takes an address as an argument, and uses this as the address to write into the EEPROM latch during the bulk erase step. The following is the data packet format for this command.

$07	HIBYTE(Addr)	LOBYTE(Addr)

BDM Commands

The following table is based on information found in the Motorola 68HC12A4 manual. Rather than duplicate the same information, the following table presents the input and output packets for each command. Refer to the 68HC12 manuals for the functional descriptions of each command. All values are in hex

BDM Hardware Commands

Command	Packet In	Packet Out
BACKGROUND	$90	none
READ_BD_BYTE	$E4 <16-bit address>	<16-bit data out>
READ_BD_WORD	$EC <16-bit address>	<16-bit data out>
READ_BYTE	$E0 <16-bit address>	<16-bit data out>
READ_WORD	$E8 <16-bit address>	<16-bit data out>
WRITE_BD_BYTE	$C4 <16-bit address> <byte> <byte>	none
WRITE_BD_WORD	$CC <16-bit address> <16-bit data in>	none
WRITE_BYTE	$C0 <16-bit address> <byte> <byte>	none
WRITE_WORD	$C8 <16-bit address> <16-bit data in>	none

BDM Firmware Commands

Command	Packet In	Packet Out

READ_NEXT	$62	<16-bit data out>
READ_PC	$63	<16-bit data out>
READ_D	$64	<16-bit data out>
READ_X	$65	<16-bit data out>
READ_Y	$66	<16-bit data out>
READ_SP	$67	<16-bit data out>
WRITE_NEXT	$42 <16-bit data in>	none
WRITE_PC	$43 <16-bit data in>	none
WRITE_D	$44 <16-bit data in>	none
WRITE_X	$45 <16-bit data in>	none
WRITE_Y	$46 <16-bit data in>	none
WRITE_SP	$47 <16-bit data in>	none
GO	$08	none
TRACE1	$10	none
TAGGO	$18	none

1. Always write the most significant byte first. For example, the for a 16-bit address such as $FF01, the bytes in the serial stream are sent $FF followed by $01.
2. For byte sized data operands, the location of the data is determined by the address. If the address of the byte is even, then it is the first byte (high or MSB). If the address is odd, then it is the second byte (low, or LSB). This is true on both input and output. On byte sized writes, it is easy to send the same byte in the high and low fields, then you don't have to think about it.
3. For word sized operations, the address MUST be word aligned. This means the address is always even.