

A Background Debugging Mode Driver Package for Modular Microcontrollers

By Scott Howard

INTRODUCTION

Motorola's 16-bit and 32-bit modular microcontroller devices feature a unique mode of operation called background debugging mode (BDM). When enabled, this mode allows an external host processor to control a target microcontroller unit (MCU) and access both memory and I/O devices via a simple serial interface. BDM can be very useful during initial debugging of control system hardware and software, and can also simplify production-line testing and configuration of an end product.

This application note shows how to enable and control BDM using an IBM-compatible personal computer. Driver routines and demonstration programs written in C are included to assist the user in quickly and easily implementing a custom test jig or debug facility. The only additional hardware required is a simple hardware interface attached to the PC-compatible parallel printer port.

BDM OVERVIEW

There are several considerations that affect the design of microcontroller emulation and debugging tools. Conventional emulators use buffering circuitry to separate the processor from the prototype under development, and connect to a target system using a long in-circuit emulation cable. The buffers and cable have a significant impact on MCU bus timing, making truly transparent emulation very difficult and expensive to achieve. This is especially true when the MCU is surface-mounted on a circuit board, since emulation cables and connectors for surface-mount packages tend to be expensive. In addition, accesses to on-chip memory and peripherals cannot easily be monitored in this way, particularly when the MCU operates as a single-chip system with no external bus.

Background debugging mode was conceived as a way to eliminate these problems. BDM uses a small amount of on-chip support logic, some additional microcode in the CPU module, and a dedicated serial port. By routing background mode interface signals to a small, inexpensive header on the target prototype, the designer can enable a host debug computer to take complete control of the target system – stopping it, examining and changing registers and memory locations, single-stepping, resetting, and restarting it – without affecting normal operation in any way. None of the user resources of the chip or the target system, such as timers, on-chip memory, or I/O pins are required to support debugging mode. The microcontroller runs at its full rated speed with no timing restrictions imposed by an external emulator.

BDM is implemented in both the CPU16 and the CPU32 processor modules of the Motorola modular microcontroller family. Thus any MCU that incorporates one of these processor modules includes the BDM interface. This includes all M68HC16 devices and all M68300 devices starting with MC68330.

BDM is fully documented in Section 10 of the *CPU16 Reference Manual* (CPU16RM/AD), and in Section 7 of the *CPU32 Reference Manual* (CPU32RM/AD).



BDM OPERATION

Background debug mode is a special mode of operation of the CPU module. Three pins provide the hardware interface to the MCU ($\overline{\text{BKPT}}/\text{DSCLK}$, $\overline{\text{IPIPE0}}/\text{DSO}$, and $\overline{\text{IPIPE1}}/\text{DSI}$ on devices that incorporate the CPU16, $\overline{\text{BKPT}}/\text{DSCLK}$, $\overline{\text{IPIPE}}/\text{DSO}$, and $\overline{\text{IFETCH}}/\text{DSI}$ on devices that incorporate the CPU32). BDM must be enabled by holding the $\overline{\text{BKPT}}$ pin low on the rising edge of $\overline{\text{RESET}}$. A fourth pin, FREEZE, indicates whether or not the MCU is in BDM.

During normal operation, the CPU fetches instructions from system memory and executes them, the FREEZE signal is held low, and the pipeline state signals ($\overline{\text{IPIPE0}}$ and $\overline{\text{IPIPE1}}$ or $\overline{\text{IPIPE}}$ and $\overline{\text{IFETCH}}$) allow external hardware to track the operation of the instruction prefetch queue. When BDM has been enabled, driving the $\overline{\text{BKPT}}$ pin low during normal instruction execution causes the CPU to finish executing the current instruction, then enter background debugging mode. In BDM, FREEZE is driven high, and the BDM pins change function to become a synchronous serial port. The BDM port is very similar in operation to a Motorola serial peripheral interface (SPI) slave device; the $\overline{\text{BKPT}}$ pin becomes the development system clock (DSCLK) pin. The DSCLK signal clocks data transfers between the host and the target MCU. $\overline{\text{IPIPE0}}$ (or $\overline{\text{IPIPE}}$) becomes a development system output (DSO) pin, which carries data from the target MCU to the host processor. $\overline{\text{IPIPE1}}$ (or $\overline{\text{IFETCH}}$) becomes a development system input (DSI) pin, which carries data from the host processor to the target MCU.

The BDM serial interface transfers data in 17-bit words, most significant bit first. Each bit is transferred on the rising edge of DSCLK, which must be generated by the host system. The MSB returned from the target MCU is a status bit. The bit normally has a value of zero, but is set when an error or abnormal condition is detected. The other 16 bits transfer data and/or commands between the host system and the MCU.

The CPU receives debug instructions via the interface port, executes them, and returns results (if any) to the host system via the interface port. The debugging instructions are relatively simple, but provide all the primitive operations required in a debugging environment.

To efficiently implement BDM in a product, Motorola recommends that a 10-pin header with the configuration shown in Table 1 be used to connect the MCU to an external host. This configuration is used by Motorola development tools and is also supported by several third-party tool vendors:

Table 1 BDM Connector Standard Pinout

Pin Number	Pin Name	Description
1	DS	Data strobe from target MCU. Not used in current interface circuitry
2	BERR	Bus error input to target. Allows development system to force bus error when target MCU accesses invalid memory
3	V_{SS}	Ground reference from target
4	BKPT/DSCLK	Breakpoint input to target in normal mode; development serial clock in BDM. Must be held low on rising edge of reset to enable BDM
5	V_{SS}	Ground reference from target
6	$\overline{\text{FRZ}}$	Freeze signal from target. High level indicates target is in BDM
7	RESET	Reset signal to/from target. Must be held low to force hardware reset
8	$\overline{\text{IFETCH}}/\text{DSI}$ (CPU32) $\overline{\text{IPIPE0}}/\text{DSI}$ (CPU16)	Used to track instruction pipe in normal mode. Serial data input to target from MCU in BDM
9	V_{CC}	+5V supply from target. BDM interface circuit draws power from this supply and also monitors 'target powered/not powered' status
10	$\overline{\text{IPIPE}}/\text{DSO}$ (CPU32) $\overline{\text{IPIPE1}}/\text{DSO}$ (CPU16)	Tracks instruction pipe in normal mode. Serial data output from target MCU in BDM

INTERFACING TO AN IBM COMPATIBLE PC

The BDM interface requires synchronous timing, and other MCU status and control signals, such as $\overline{\text{RESET}}$ and $\overline{\text{BERR}}$, must also be monitored during BDM. These factors make a direct RS-232 BDM interface impractical. Instead, the IBM PC parallel printer port can be used to interface directly to the logic-level signals of the target microcontroller. Software provides the synchronous timing required.

A hardware interface circuit must be used to reduce errors due to noise, and also to buffer microcontroller signals so that they are not unduly loaded by the PC and cable. The code provided in this application note supports two different interface circuits.

The first interface circuit is incorporated in the Motorola In-Circuit Debugger (ICD). The ICD product includes an interface circuit board and a debug monitor program that runs on a host IBM-compatible computer. ICD are available for CPU16 microcontrollers (Motorola part no. M68ICD16) and for CPU32 microcontrollers (Motorola part no. M68ICD32).

The second interface circuit is the public-domain circuit shown in Figure 1. The circuit uses two 74HC series logic devices and a minimum of passive components, making it inexpensive and easy to construct.

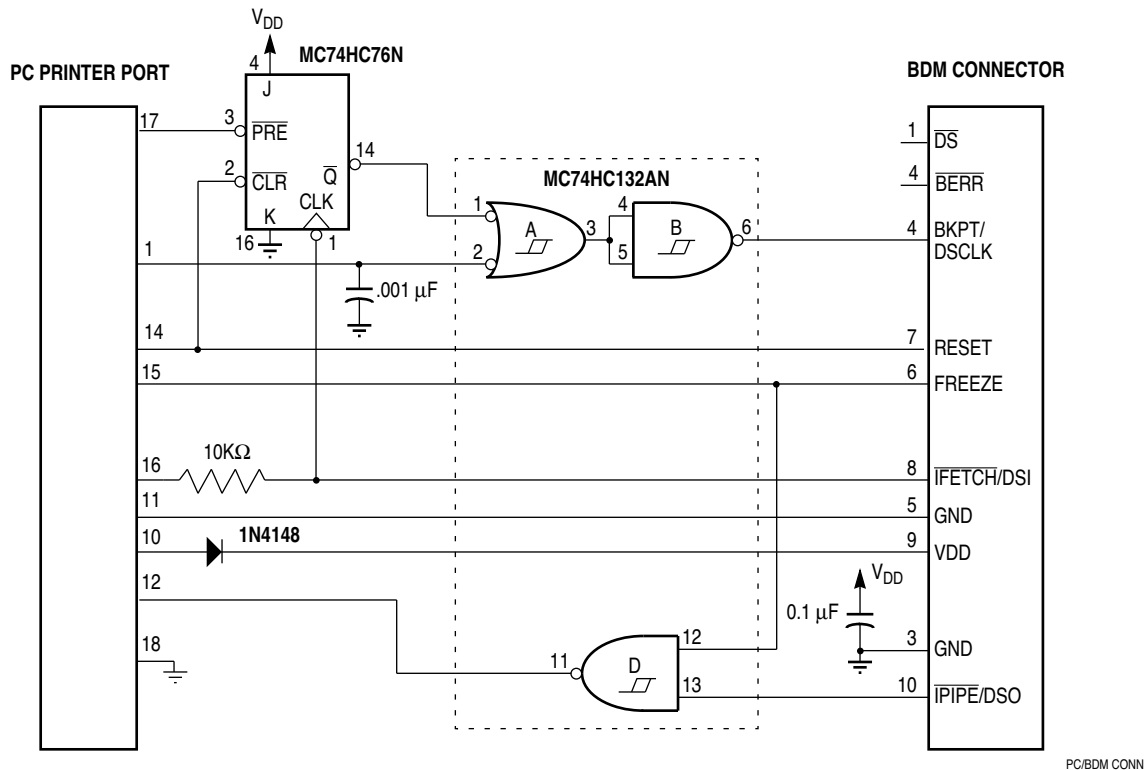


Figure 1 Public Domain BDM Interface Circuit

SOFTWARE DRIVERS

The software drivers presented in this application note are written in C. The drivers are designed to be linked into a user's debug or production test program. They perform the following functions for both CPU16 and CPU32 targets:

- Initialize and de-initialize the parallel printer port interface
- Provide basic control functions to reset, stop, and start code execution on the target MCU, and to monitor target status
- Read and write CPU registers, including system registers supported by the target MCU
- Read and write memory and memory-mapped I/O, in byte, word, and long word sizes
- Take advantage of memory block read and write operations supported by the target MCU
- Read S-records from an IBM PC disk file, and write the contents to target system memory

A typical application would begin by calling `Init()` to initialize the driver package. It is a good practice to apply a hardware reset to the target system after it is connected to the PC and powered up, since BDM is not active unless the target is reset with the `BKPT` pin held low. This can be achieved by calling either `ResetChip()`, which resets the target and allows it to run, or by calling `RestartChip()`, which resets the target and forces it into background mode immediately after the initial program counter and stack pointer fetches.

The user's debug software can get the current status of the target by calling `GetStatus()`. This function call returns an integer value in which each bit indicates the state of one signal on the target. The type of interface hardware used between the PC and the target determines which signals are available. Table 2 shows these status bits:

Table 2 Status Bits Returned from `GetStatus()`

Bit Position	Symbolic Name (in 'trgtstat.h')	Description	Public-Domain Interface	ICD Interface
Bit 0	TARGETRESET	1 if target in reset state ($\overline{\text{RESET}} = 0$)	Yes	No
Bit 2	TARGETSTOPPED	1 if target in background mode ($\text{FREEZE} = 1$)	Yes	Yes
Bit 3	TARGETPOWER	1 if no power applied to target	Yes	No
Bit 4	TARGETNC	1 if BDM interface cable is disconnected	Yes	No

There are several other control functions as well. `StopChip()` puts the target into background mode (if it is not already) by holding the `BKPT` pin low until `FREEZE` is asserted. `StepChip()` single-steps the target by allowing one instruction to be fetched and executed before `BKPT` is asserted again. `RunChip()` resumes full-speed execution, either at a programmer-defined address or at the location pointed to by the MCU program counter.

Reading and writing CPU registers using function calls `GetReg()` and `PutReg()` is straightforward. These functions have an integer parameter that identifies whether the register is to be read or written. The symbolic names for these register identifiers are defined in the include file `regnames.h`.

Reading and writing memory is only slightly more complex. Memory can be accessed in byte, word, or long word format. The different access sizes are handled by function calls `GetByte()`, `GetWord()`, `GetLong()`, and `PutByte()`, `PutWord()`, and `PutLong()`. Memory accesses also use a memory space attribute, represented by the function code value that is driven out on the `FC[2:0]` pins during the MCU memory access bus cycle. The memory space can be one of User Data (function code 1), User Code (function code 2), Supervisor Data (function code 5), or Supervisor Code (function code 6). Memory space is selected by calling function `SetFC()`, passing the desired function code as a parameter. M68HC16 microcontrollers always operate in supervisor mode, so for these, only function codes 5 and 6 are supported.

All functions operate whether the target MCU is running or stopped. For example, if the `GetReg ()` function is called when the target is already in background mode, the BDM driver software simply fetches the contents of the specified target register and returns it to the calling routine. However, if the target MCU is executing code, the target MCU is temporarily halted, the specified register is retrieved, and the target MCU is then immediately restarted. This provides some interesting capabilities. For instance, it is possible to monitor RAM locations as the processor executes code, or to profile MCU code execution by periodically sampling the value in the program counter.

ERROR HANDLING

All of the BDM drivers employ some level of error handling and retry. However, there are cases where the driver cannot perform the requested function, such as trying to read a target register when the interface cable is disconnected. Returning an error code to the calling routine is not a good solution, since in that case an error flag must be tested each time the interface function is called. Instead, when one of these cases is encountered, the driver package calls a user-defined function called `DriverError ()`. This function is passed through an integer error code which identifies the error condition.

Several actions are possible with `DriverError ()`. The easiest implementation is to simply report the error and then return control to the operating system. This is probably not satisfactory for most applications, however, and a program which does not handle errors more gracefully is frustrating to the user. Another approach is to use `setjmp ()` in the main loop of the program, and design `DriverError ()` to report the problem, and then call `longjmp ()` to return to the mainline routine.

ACCESSING THE BDM DRIVER PACKAGE

The source code for this driver package is too large to be reproduced in this application note. However, a current version of the source code is maintained on Motorola's Freeware Bulletin Board System.

The Freeware BBS can be accessed by modem at (512) 891-3733. Configure the interface for eight data bits, no parity, and one stop bit. The BBS is compatible with any V.32 or Bell 212 modem operating at data rates from 1200 to 9600 baud. Log on and check the MCU332 file area for an archive named `BDMVx-y.ZIP`, where 'x' and 'y' represent the latest version number of the driver package.

FUNCTION REFERENCE

The driver package is contained in several files, listed in Table 3. To produce an executable program, the user's program must be compiled and linked along with either `bdm-icd.c` or `bdm-pd.c` depending on which interface hardware (ICD or Public-Domain, respectively) is used, and either `bdmcpu32.c` or `bdmcpu16.c` depending upon whether the target system is an M68300 or M68HC16 device. The file `bdm-util.c` contains routines to read Motorola S-records from disk files and write the S-record contents to target memory, and must be compiled and linked only if these S-record routines are needed in the user's program.

For convenience, the archive on the BBS includes pre-compiled small-model object files for Turbo C and Borland C. The programs use some Borland compiler-specific features, such as library calls to perform port I/O and directives to insert assembly-language code into the program; however, most PC compilers provide these features, and conversion to another compiler will probably require only minor editing, compilation, and retesting.

Table 3 Source Files in the BDM Archive

Source File	Optional?	Description
<code>bdmcpu16.c</code> and <code>bdmcpu32.c</code>	No	Contain high-level driver functions for the CPU16 and CPU32 targets, respectively. One driver must be compiled and linked into user application to provide access to background mode interface.
<code>bdm-icd.c</code> and <code>bdm-pd.c</code>	No	Contain the low-level control routines for the ICD card and the public domain circuit, respectively. One of these files must be compiled and linked into user application to provide access to background mode interface.
<code>bdmcalls.h</code>	No	Defines high-level function calls for use by external code; should be included in any user program.
<code>bdmerror.h</code>	No	Define error codes passed to <code>DriverError ()</code> user function. Should be included in user program where <code>DriverError ()</code> is defined.
<code>bdm-util.c</code>	Yes	Contains the utility functions to read S-records from disk files and write the contents of S-records into target memory. Only required if program must read S-record files from disk and/or write S-records to target memory.
<code>bdm-util.h</code>	Yes	Prototypes utility function calls (in <code>bdm-util.c</code>) for S-record access. Should be included into user code if program must read S-record files from disk and/or write S-records to target memory.
<code>regs-16.h</code> and <code>regs-32.h</code>	Yes	Define symbolic names for register identifiers passed to <code>GetReg ()</code> and <code>PutReg ()</code> . Only required if user calls <code>GetReg ()</code> and <code>PutReg ()</code> to read and write target CPU registers.
<code>trgtstat.h</code>	Yes	Defines bit values returned by <code>GetStatus ()</code> and <code>GetStatusMask ()</code> functions. Only required if user calls one of these functions.
<code>textio.h</code>	Yes	This header file allows the user to select, using a <code>#define</code> constant, whether the text output routines in <code>do_load ()</code> use Borland's <code>cprintf/cputs/putch</code> or the ANSI standard <code>printf/puts/putchar</code> calls. It may also be included into user code for text output to the PC's screen.

Table 4 contains a detailed BDM driver function reference.

Table 4 BDM Driver Package Function Reference

Prototype	Description	Return Value
<pre>#include "bdmcalls.h" unsigned ValidPorts (void);</pre>	Checks the PC's BIOS data tables for the number of parallel printer ports installed in the PC; returns a bit vector representing which parallel ports (LPT1, LPT2, and LPT3) are installed.	The return value has one bit set for each parallel port (LPT1-LPT3) which is in- stalled in the PC, i.e. bit 0 is set if LPT1 is installed, bit 1 for LPT2, and bit 2 for LPT3.
<pre>#include "bdmcalls.h" int Init (unsigned Port, unsigned Speed);</pre>	The desired port (1 through 3, for LPT1 through LPT3 respectively) is checked for validity, and if available will be initialized to the desired bit rate to communicate with the BDM interface circuit.	Returns a non-zero value if the initialization failed for any reason, such as printer port not installed.
<pre>#include "bdmcalls.h" void SetFC (void);</pre>	Sets the function code to be used when the BDM drivers access target memory. The valid function codes are 1 (user data), 2 (user program), 5 (supervisor data), and 6 (supervisor code). The default is 5 (supervisor data). If an invalid code is passed, the function code will remain at its previous setting. Note: for M68HC16 microcontrollers, only 5 and 6 are valid.	Nothing.
<pre>#include "bdmcalls.h" void DeInit (void);</pre>	Restores the parallel port hardware to its original state before <code>Init ()</code> was called. This function should be called by user code before the program exits to DOS.	Nothing.
<pre>#include "bdmcalls.h" unsigned GetStatus (void);</pre>	Returns a bit vector representing the current status of the target MCU. Some error conditions are not detectable by both hardware interfaces. Therefore, only those errors which can be detected by the interface in use will actually be reported.	The following bits are defined in "trgtstat.h", which is included by "bdmcalls.h". The state is true when the corresponding bit is set to 1: TARGETRESET: Target is in reset state TARGETHALT: Target is halted TARGETSTOPPED: Target is in FREEZE state TARGETINVALID: Target interface is not initialized
<pre>#include "bdmcalls.h" unsignedGetStatusMask (void);</pre>	Returns a bit vector representing which bits are valid in the return value of <code>GetStatus ()</code> .	The following bits are defined in "trgtstat.h", which is included by "bdmcalls.h". Any bit set to 1 indicates that the <code>GetStatus ()</code> function is capable of detecting and returning the corresponding status: TARGETRESET: Target is in reset state TARGETHALT: Target is halted TARGETSTOPPED: Target is in FREEZE state TARGETINVALID: Target interface is not initialized
<pre>#include "bdmcalls.h" int StopChip (void);</pre>	Stops (places into FREEZE state) target MCU.	Non-zero if the target was running when <code>StopChip ()</code> was called; otherwise zero.

Table 4 BDM Driver Package Function Reference (Continued)

Prototype	Description	Return Value
#include "bdmcalls.h" void StepChip (void);	Single-steps target MCU by one instruction.	Nothing.
#include "bdmcalls.h" void ResetChip (void);	Forces hardware reset on target MCU; leaves MCU in running state.	Nothing.
#include "bdmcalls.h" void RestartChip (void);	Resets target MCU, and stops execution on first instruction fetch (after PC and SP are fetched from vector table). If target does not stop, and the ICD interface circuit is used, a bus error will be asserted on the target MCU to bring it into BDM; else if the public-domain circuit is used, DriverError () will be called with the appropriate error.	Nothing.
#include "bdmcalls.h" void RunChip (LONG where);	Starts execution of target MCU at address specified by 'where'. If 'where' is 0, execution starts at address contained in Program Counter of target MCU.	Nothing.
#include "bdmcalls.h" #include "regs-16.h" (or) #include "regs-32.h" LONG GetReg (unsigned which);	Retrieves and returns the contents of the target CPU register indicated by 'which'. Symbolic values for the register codes are defined in the file 'regs-16.h' (CPU16 targets) and 'regs-32.h' (CPU32 targets).	The data contained in the specified register.
#include "bdmcalls.h" #include "regs-16.h" (or) #include "regs-32.h" void PutReg (unsigned which, LONG data);	Writes the value 'data' into the target CPU register indicated by 'which'. Symbolic values for the register codes are defined in the file 'regs-16.h' (CPU16 targets) and 'regs-32.h' (CPU32 targets).	Nothing.
#include "bdmcalls.h" LONG GetByte (LONG Where);	Performs a byte read on the specified memory location and returns value retrieved.	Returns contents of byte-addressed memory at location 'Where'. Values returned are 0-0xff.
#include "bdmcalls.h" LONG GetWord (LONG Where);	Performs a word read on the specified memory location and returns value retrieved.	Returns contents of word-addressed memory at location 'Where'. Values returned are 0-0xffff.
#include "bdmcalls.h" LONG GetLong (LONG Where);	Performs a longword read on the specified memory location and returns value retrieved.	Returns contents of longword-addressed memory at location 'Where'.
#include "bdmcalls.h" LONG DumpByte (LONG Where);	Performs a byte read on the specified memory location and returns value retrieved. On CPU16 targets, this call is identical to GetByte (). On CPU32 targets, a special BDM command is used which is faster than a normal memory access. GetByte (), GetWord (), or GetLong () should be called first, to set the initial address for memory subsequent memory writes; then any number of DumpByte (), DumpWord (), or DumpLong () calls may be made to read from following locations of memory.	Returns contents of byte-addressed memory at location 'where'. Values returned are 0-0xff.

Table 4 BDM Driver Package Function Reference (Continued)

Prototype	Description	Return Value
<pre>#include "bdmcalls.h" LONG DumpWord (LONG Where);</pre>	<p>Performs a word read on the specified memory location and returns value retrieved.</p> <p>On CPU16 targets, this call is identical to <code>GetWord ()</code>. On CPU32 targets, a special BDM command is used which is faster than a normal memory access. <code>GetByte ()</code>, <code>GetWord ()</code>, or <code>GetLong ()</code> should be called first, to set the initial address for memory subsequent memory writes; then any number of <code>DumpByte ()</code>, <code>DumpWord ()</code>, or <code>DumpLong ()</code> calls may be made to read from following locations of memory.</p>	<p>Returns contents of word-addressed memory at location 'where'. Values returned are 0-0xffff.</p>
<pre>#include "bdmcalls.h" LONG DumpLong (LONG Where);</pre>	<p>Performs a longword read on the specified memory location and returns value retrieved.</p> <p>On CPU16 targets, this call is identical to <code>GetLong ()</code>. On CPU32 targets, a special BDM command is used which is faster than a normal memory access.</p> <p><code>GetByte ()</code>, <code>GetWord ()</code>, or <code>GetLong ()</code> should be called first, to set the initial address for memory subsequent memory writes; then any number of <code>DumpByte ()</code>, <code>DumpWord ()</code>, or <code>DumpLong ()</code> calls may be made to read from following locations of memory.</p>	<p>Returns contents of longword-addressed memory at location 'where'</p>
<pre>#include "bdmcalls.h" void PutByte (LONG Where, BYTE Data);</pre>	<p>Performs a byte write at location 'where', writing the value 'Data'.</p>	<p>Nothing.</p>
<pre>#include "bdmcalls.h" void PutWord (LONG Where, WORD Data);</pre>	<p>Performs a word write at location 'where', writing the value 'Data'</p>	<p>Nothing.</p>
<pre>#include "bdmcalls.h" void PutLong (LONG Where, LONG Data);</pre>	<p>Performs a longword write at location 'where', writing the value 'Data'.</p>	<p>Nothing.</p>
<pre>#include "bdmcalls.h" void FillByte (LONG Where, BYTE Data);</pre>	<p>Performs a byte write at location 'where', writing the value 'Data'.</p> <p>On CPU16 targets, this call is identical to <code>PutByte ()</code>. On CPU32 targets, a special BDM command is used which is faster than a normal memory access.</p> <p><code>PutByte ()</code>, <code>PutWord ()</code>, or <code>PutLong ()</code> should be called first, to set the initial address for memory subsequent memory writes; then any number of <code>FillByte ()</code>, <code>FillWord ()</code>, or <code>FillLong ()</code> calls may be made to write to following locations of memory</p>	<p>Nothing.</p>
<pre>#include "bdmcalls.h" void FillWord (LONG Where, WORD Data);</pre>	<p>Performs a word write at location 'where', writing the value 'Data'.</p> <p>On CPU16 targets, this call is identical to <code>PutWord ()</code>. On CPU32 targets, a special BDM command is used which is faster than a normal memory access.</p> <p><code>PutByte ()</code>, <code>PutWord ()</code>, or <code>PutLong ()</code> should be called first, to set the initial address for memory subsequent memory writes; then any number of <code>FillByte ()</code>, <code>FillWord ()</code>, or <code>FillLong ()</code> calls may be made to write to following locations of memory</p>	<p>Nothing.</p>

Table 4 BDM Driver Package Function Reference (Continued)

Prototype	Description	Return Value
<pre>#include "bdmcalls.h" void FillLong (LONG Where, LONG Data);</pre>	<p>Performs a longword write at location 'where', writing the value 'Data'.</p> <p>On CPU16 targets, this call is identical to PutLong (). On CPU32 targets, a special BDM command is used which is faster than a normal memory access. PutByte (), PutWord (), or PutLong () should be called first, to set the initial address for memory subsequent memory writes; then any number of FillByte (), FillWord (), or FillLong () calls may be made to write to following locations of memory</p>	<p>Nothing.</p>
<pre>#include"bdm-calls.h" #include "bdm-util.h" #include <stdio.h> int do_srec (srecord *where, FILE *infile);</pre>	<p>Reads one S-record from the file addressed by 'infile' and writes the contents into the srecord structure '*where' in memory. The 'srecord' structure is defined in the include file "bdm-util.h".</p> <p>When do_srec () returns indicating either no error or SREC_S9 (end of file record), the structure '*where' contains the following information:</p> <p>where->rectype holds the contents of the S-record type field, one of the following values: 0, 1, 2, 3, 7, 8, or 9. (Note that a type 0 field is a header field and should be ignored; types 7, 8, and 9 are end-of-file records, causing do_srec to return SREC_S9 instead of 0.)</p> <p>where->reclen holds the contents of the record length field of the S-record.</p> <p>where->address holds the contents of the address field of the S-record.</p> <p>where->bytes [] holds the contents (if any) of the data field of the S-record. The number of valid bytes will be: where->reclen, minus the size of the address field (2 for types 0, 1, and 9; 3 for types 2 and 8; 4 for types 3 and 7), minus one.</p>	<p>Returns a status code representing the success or failure of the read. The codes are defined in the include file "bdm-util.h".</p> <p>0: S-record was read OK -memory structure contains valid data. SREC_EOF: file ended before end-of-file record found SREC_FORMAT: file does not match defined S-record format. SREC_CHECKSUM: checksum error in S-record. SREC_S9: this S-record is the end-of-file record as defined in the Motorola S-record spec. This status is not an error, but rather an indication that the last record was read from the file, and the user program should not continue to call do_srec ().</p>
<pre>#include"bdm-calls.h" #include "bdm-util.h" voidput_srec (srecord *data, LONG load_offset);</pre>	<p>Writes the data contained in data->bytes to memory in the target system, at the address specified by data->address plus load_offset. Type 0 records are ignored; type 7, 8, and 9 records cause the global variable ExecuteAddress to be set to the value data->address plus load_offset.</p>	<p>Nothing.</p>
<pre>#include"bdm-calls.h" #include "bdm-util.h" int do_load (LONG load_offset, char *in- filename);</pre>	<p>Opens the disk file named by 'infilename', reads S-records from the file on at a time using do_srec (), and writes the contents of the S-records to target memory using put_srec ().</p>	<p>Returns the same error codes as do_srec (), with one addition: if the file specified by 'infilename' cannot be opened, do_load returns the value -1.</p>

AN EXAMPLE PROGRAM

The BDM archive contains a file called `test.c`, which implements a simple monitor. The monitor allows a user to examine any 256-byte range of target memory, display CPU registers, restart the target from power-on reset, single-step the target, hardware reset the target, and load S-record files on disk into target memory. A simple `DriverError ()` routine reports errors in the target interface, then returns control to the user.

To produce an executable program for a CPU32 target, `test.c` must be compiled and linked with `bdmcpu32.c`, `test32.c`, `bdm-util.c`, and either `bdm-icd.c` or `bdm-pd.c`, depending on the interface hardware used. For a CPU16 target, `test.c` must be compiled and linked with `bdmcpu16.c`, `test16.c`, `bdm-util.c`, and either the `bdm-icd.c` or `bdm-pd.c`. `Test16.c` and `Test32.c` contain the function `DumpRegisters ()`, which displays target CPU register content, and is therefore specific to the type of CPU being controlled.

Test.c Source Code

```
/* test.c - test background mode drivers
 * this program is a very simple monitor program which will allow the user
 * examine and change memory and registers, start/stop/reset/single step, etc.
 * Functions which are common to both CPU32 and CPU16 targets are in this file
 * test16.c and test32.c contain the code specific to the target
 */
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <setjmp.h>
#include "bdmcalls.h"
#include "trgtstat.h"
#include "bdmerror.h"
#include "bdm-util.h"
#include "textio.h"

#define MaxBuffer 128

extern void DumpRegisters (void);
extern char *TargetName;
int GetCommandString (char *where, int count);
char ReturningToDOS [] = "Returning to DOS" NEWLINE;
jmp_buf Saviour;

/* DriverError is a function called by background mode drivers when error
 * detected
 * this version simply prints an error corresponding to the error code and exits
 */

static char *ErrorStrings [] =
{
NEWLINE "Unknown Error Encountered - Check Communications Speed" NEWLINE ,
NEWLINE "Power Failed on Target MCU" NEWLINE,
NEWLINE "Cable Disconnected on Target MCU" NEWLINE,
NEWLINE "No Response from Target MCU" NEWLINE,
NEWLINE "Can't Clock Target MCU while in Reset" NEWLINE ,
NEWLINE "Specified Parallel Port Not Available on this PC" NEWLINE
};

int ErrorCode, ErrorRW;
LONG ErrorAddress;

void DriverError (int ErrCode, int LastRW, LONG LastAddress)
{
    ErrorCode = ErrCode;
    ErrorRW = LastRW;
    ErrorAddress = LastAddress;
    longjmp (Saviour, 1);
}
```

```

}
void ReportError (void)
{
    if (ErrorCode == BDM_FAULT_BERR)
        printf (NEWLINE "Bus error %s address %lx" NEWLINE ,
                ErrorRW ? "reading" : "writing",
                ErrorAddress);
    else puts (ErrorStrings [ErrorCode]);
}

int main ()
{
    unsigned CharCount, Counter, Counter1, SR, GotOne = 0, Ports = ValidPorts ();
    int Stopped, Speed = -1, SelectedPort = 0;
    char *ptr, Buffer [MaxBuffer], QuitFlag = '\0';
    LONG LongTemp, MemDump = 0;
    BYTE ByteTemp;

    clrscr ();
    printf ("Test Program for %s Background Mode Drivers" NEWLINE , TargetName);
    printf ("Available LPT Ports: <");
    for (SR = 1, Counter = 1; Counter <= 16; Counter++, SR <= 1)
    {
        if (SR & Ports)
        {
            if (GotOne) putchar (',' );
            GotOne = 1;
            printf ("%d", Counter);
        }
    }
    if (!GotOne) puts ("none");
    puts ( ">" NEWLINE );
    if (!Ports)
    {
        printf ("I need at least one parallel printer port to function!" NEWLINE
"%s", ReturningToDOS);
        exit (1);
    }
    for (;;)
    {
        puts ("Please select Printer Port:");
        CharCount = GetCommandString (Buffer, MaxBuffer);
        if (!CharCount || !sscanf (Buffer, "%d", &SelectedPort)) return 0;
        do
        {
            puts ("Please select Clock Speed (0-100): ");
            CharCount = GetCommandString (Buffer, MaxBuffer);
            if (!CharCount || !sscanf (Buffer, "%d", &Speed)) return 0;
        }
        while (Speed < 0 || Speed > 100);
        if (setjmp (Saviour)) ReportError ();
        else if (Init (SelectedPort, Speed))
            printf ("Can't initialize port %d at speed %d" NEWLINE,
                SelectedPort, Speed);
        else break;
    }
    printf ("Port %d Initialized at Speed %d" NEWLINE , SelectedPort, Speed);
    puts ("Resetting Target MCU to enable BDM" NEWLINE );
    ResetChip ();
    while (!QuitFlag)
    {
        if (setjmp (Saviour)) ReportError ();
        puts ("Enter Command DHMRSQ('H' for Help):");
        GetCommandString (Buffer, MaxBuffer);
        for (ptr = Buffer ;isspace (*ptr); ptr++) ;
        switch (toupper (*ptr))
        {
            case 'B':

```

```

RestartChip ();

case 'D':
    DumpRegisters ();
    break;
case 'H':
    printf ("Help for TEST (%s Target)" NEWLINE
           "B: Begin Program Execution from Reset" NEWLINE
           "D: Dump Target MCU Registers" NEWLINE
           "H: Print This Help Summary" NEWLINE
           "L: Load S-Record File into Target" NEWLINE
           "M: Memory Hex/ASCII Display" NEWLINE
           "R: Hardware Reset Target MCU" NEWLINE
           "S: Single Step Target MCU" NEWLINE
           "Q: Quit back to DOS" NEWLINE , TargetName);
    break;

case 'L':
    printf ("S-Record File to Load: ");
    if (!GetCommandString (Buffer, MaxBuffer)) break;
    PrintEachRecord = ".";
    do_load (0, Buffer);
    break;

case 'M':
    printf ("Memory Dump start address in hex:");
    GetCommandString (Buffer, MaxBuffer);
    if (!sscanf (Buffer, "%lx", &LongTemp))
        LongTemp = MemDump;
    Stopped = StopChip ();
    set_fc ();
    for (Counter = 16; Counter; Counter--)
    {
        printf ("%08lX ", LongTemp);
        for (Counter1 = 16; Counter1; Counter1--)
            printf ("%02X ", (BYTE) GetByte (LongTemp++));
        LongTemp -= 16;
        putchar (' ');
        for (Counter1 = 16; Counter1; Counter1--)
        {
            ByteTemp = GetByte (LongTemp++);
            putchar (isprint (ByteTemp) ? ByteTemp : '.');
        }
        puts (NEWLINE);
    }
    MemDump = LongTemp;
    restore_fc ();
    if (Stopped) RunChip (0);
    break;

case 'R':
    printf ("Resetting Target MCU" NEWLINE );
    ResetChip ();
    break;


case 'S':
    printf ("Single Step" NEWLINE );
    StepChip ();
    DumpRegisters ();
    break;

case 'Q':
    QuitFlag = 1;
}
}
DeInit ();
puts (ReturningToDOS);
return 0;

```

```
}  
  
/* GetCommandString returns keyboard entry in buffer, to max length specified */  
  
int GetCommandString (char *where, int count)  
{  
    char TempBuffer [MaxBuffer+2];  
  
    TempBuffer [0] = count;  
    strcpy (where, gets (TempBuffer));  
    puts (NEWLINE);  
    return TempBuffer [1];  
}
```

NOTES

Motorola reserves the right to make changes without further notice to any products herein. Motorola makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Motorola assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters can and do vary in different applications. All operating parameters, including "Typicals" must be validated for each customer application by customer's technical experts. Motorola does not convey any license under its patent rights nor the rights of others. Motorola products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Motorola product could create a situation where personal injury or death may occur. Should Buyer purchase or use Motorola products for any such unintended or unauthorized application, Buyer shall indemnify and hold Motorola and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Motorola was negligent regarding the design or manufacture of the part.  **MOTOROLA** is a registered trademark of Motorola, Inc. Motorola, Inc. is an Equal Opportunity/Affirmative Action Employer.

TO OBTAIN ADDITIONAL PRODUCT INFORMATION:

USA/EUROPE: Motorola Literature Distribution;
P.O. Box 20912; Phoenix, Arizona 85036. 1-800-441-2447

JAPAN: Nippon Motorola Ltd.; Tatsumi-SPD-JLDC, Toshikatsu Otsuki,
6F Seibu-Butsuryu-Center, 3-14-2 Tatsumi Koto-Ku, Tokyo 135, Japan. 03-3521-8315

HONG KONG: Motorola Semiconductors H.K. Ltd.; 8B Tai Ping Industrial Park,
51 Ting Kok Road, Tai Po, N.T., Hong Kong. 852-26629298

MFAX: RMFAX0@email.sps.mot.com - TOUCHTONE (602) 244-6609

INTERNET: <http://www.mot.com>



MOTOROLA