M5213BADGE

Development Board for Freescale MCF5213 MCU
Hardware User Manual

Axiom Manufacturing • 2813 Industrial Lane • Garland, TX 75041
Email: Sales@axman.com Web: http://www.axman.com

CONTENTS

Cautionary Notes

1) Electrostatic Discharge (ESD) prevention measures should be applied whenever handling this product. ESD damage is not a warranty repair item.

2) Axiom Manufacturing reserves the right to make changes without further notice to any products to improve reliability, function or design. Axiom Manufacturing does not assume any liability arising out of the application or use of any product or circuit described herein; neither does it convey any license under patent rights or the rights of others.

3) EMC Information on the M5213BADGE board:

a) This product has not been tested for CE or FCC compliance.

b) This product is designed and intended for use as a development platform for hardware or software in an educational / professional laboratory or as a component in a larger system.

c) In a domestic environment this product may cause radio interference in which case the user may be required to take adequate prevention measures.

d) Attaching additional wiring to this product or modifying the products operation from the factory default as shipped may effect its performance and also cause interference with other apparatus in the immediate vicinity. If such interference is detected, suitable mitigating measures should be taken.

Terminology

This development board applies option selection jumpers. Terminology for application of the option jumpers is as follows:

> Jumper on, in, or installed = jumper is a plastic shunt that fits across 2 pins and the shunt is installed so that the 2 pins are connected with the shunt.
>
> Jumper off, out, or idle = jumper or shunt is installed so that only 1 pin holds the shunt, no 2 pins are connected, or jumper is removed. It is recommended that the jumpers be placed idle by installing on 1 pin so they will not be lost.

This development board applies option selections that require a soldering tool to install or remove. This type connection places an equivalent Jumper Installed type option. Applying the connection can be performed by installing a 0 ohm resistor component or small wire between the option pads. See the Options section for more details.

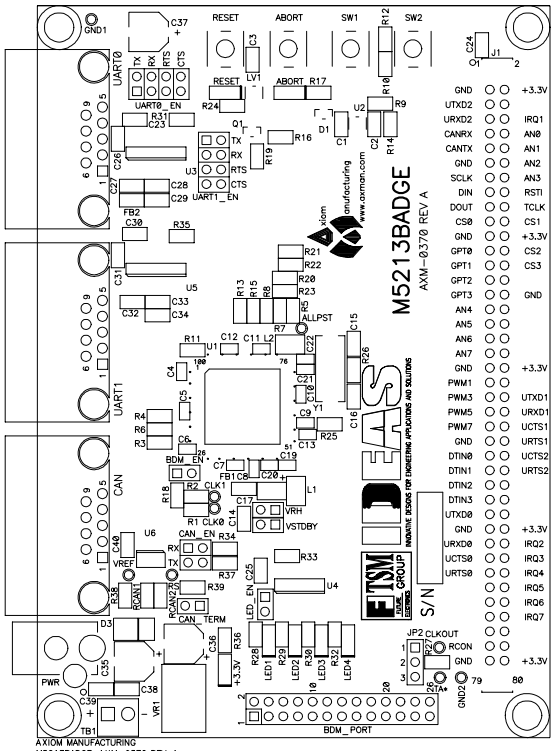
Signal names in this document that are followed by an asterisk (*) denote an active-low signal.

FEATURES

M5213BADGE is a low cost development system for the Freescale MCF5213 ColdFire® microcontroller. Application development is quick and easy with the included DB9 serial cable and dBUG firmware monitor. The BDM port is compatible with standard ColdFire BDM / JTAG interface cables and hosting software, allowing easy application debugging and development with a variety of hardware and software tools.

Features:

- MCF5213 CPU, 100 pin LQFP
  - * 256K Byte Flash (on-chip)
  - * 32K Byte SRAM (on-chip)
  - * DMA Controller w/ four 32-bit Timers
  - * Interrupt Controller
  - * 8 Channel 12-bit A/D
  - * QSPI, IIC, and CAN Serial Ports
  - * 3 UART Serial Ports with DMA capability
  - * Edge / Interrupt Port
  - * 8 PWM timers
  - * 4 16-bit GPT Timers
  - * BDM / JTAG Port
  - * Internal 8MHz Oscillator
  - * 3.3V operation
  - * Up to 80MHz operation
- 8MHz reference crystal, up to 80MHz operation
- MCU port, 80 pin I/O port
- BDM / JTAG Port, 26-pin development port
- UART0 / Terminal Port w/ RS232 DB9-S Connector
- UART1 Port w/ RS232 DB9-S Connector
- CAN port w/ 1Mb CAN transceiver
- RESET switch and indicator
- ABORT (IRQ7) switch
- 4 User Indicators (LEDs)
- 2 User Push Switches
- Regulated +3.3V power supply w/ indicator



M5213BADGE

Specifications:
Board Size 3.5” x 5.0”
Power Input: +5 - +16VDC, 9VDC typical
Current Consumption: 100ma typical @ 9VDC input

The M5213BADGE is provided, operating the Freescale dBUG monitor firmware. The monitor allows serial interface to a PC host for file loading and terminal command line operations. Additional hardware and software development tools are available, but not required.

GETTING STARTED

The M5213BADGE single board computer is a fully assembled, fully functional development board for the Freescale MCF5213 microcontroller. Provided support software for this development board is for Windows 95/98/NT/2000/XP operating systems.

Development board users should also be familiar with the hardware and software operation of the target MCF5213 device. Refer to the microcontroller reference manual, MCF5213RM, for details. The purpose of the development board is to assist the user in quickly developing an application with a known working environment, to provide an evaluation platform, or as a control module for an applied system. Users should be familiar with memory mapping, memory types, and embedded software design for the quickest successful application development.

Software Development

Application development maybe performed by applying the dBUG firmware monitor, or by applying a compatible ColdFire BDM / JTAG cable with supporting host software. The monitor provides an effective and low cost command line debug method.

Software development is best performed with a development tool connected to the BDM port. This provides real-time access to all hardware, peripherals and memory on the board. Development tool software also provides high-level (C/C++) source code debug environment.

The target development environment and procedure for best success is to place software to be tested into RAM memory. Execute software to be tested under dBUG monitor or development tool control. After the software is tested and operational in RAM, it can be ported and programmed into Flash memory. However, note that programming a bootable application into the internal Flash will overwrite the dBUG monitor.

Reference Documentation

The following documents should be referenced when developing with the M5213BADGE. These documents are available on the MCF5213 and M5213BADGE web pages (http://www.freescale.com/coldfire).

M5213BADGEUM – This user manual.
MCF5213RM – MCF5213 Device Reference Manual
CFPRM – ColdFire Programmers Reference Manual with instruction set
M5213BADGE_SCH_A – M5213BADGE Rev. A board schematics

M5213BADGE Startup

Follow these steps to connect and power on the board for the default dBUG monitor operation.

1) Carefully unpack the M5213BADGE and observe ESD preventive measures while handling the M5213BADGE development board.

2) Configure a virtual terminal program (such as HyperTerminal or AXIDE) for a direct connection to the PC COM port to be applied for serial communication with the M5213BADGE. Set the baud rate to 19.2K baud, 8 data bits, 1 stop bit, and no parity. Software XON / XOFF flow control should be enabled for Flash memory support operations. Use the AxIDE '√' tool bar button to configure the COM port on the PC.

3) Connect the M5213BADGE board UART0 / TERMINAL serial port connector to the host PC COM port with the provided 9 pin serial cable.

4) Apply power to the development board by installing a 9VDC wall plug power supply between a wall outlet and the PWR Jack on the board or apply TB1 with a suitable 5 – 16VDC supply. The board voltage indicators should turn on at this time.

5) Observe the terminal window display for the dBUG monitor prompt. The prompt should be similar to the following:

```
External Reset

ColdFire MCF5213 on the M5213EVB
Firmware v4a.1b.1b (Built on May 13 2005 08:04:26)
Copyright 2005 Freescale Semiconductor, Inc.

Enter 'help' for help.

dBUG>
```

6) The board is ready to use now. See the dBUG monitor manual section for additional monitor information. If BDM / JTAG development port interfaced tools are to be applied, see the BDM PORT section of this manual for more details on installation.

M5213BADGE Hardware Configuration and Options

The M5213BADGE board provides a basic development or evaluation platform for the MCF5213 microcontroller. Following are descriptions of the main components and options provided on the board.

MEMORY

The EVB memory is the internal MCF5213 device SRAM and Flash memory. The MCF5213 provides 32K bytes of SRAM and 256K bytes of Flash memory internally. The dBUG monitor occupies the lower 80K bytes of the MCF5213 Flash memory and lower 8K of the SRAM. Refer to the dBUG memory map for default memory locations.

POWER SUPPLY

Unregulated DC input power is applied by external connection to the Power Jack or TB1 terminal block. The EVB 3.3VDC regulator is protected from reverse voltage by diode D3.

Power Jack

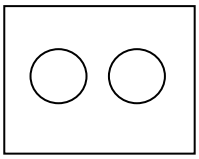
The Power Jack provides the default power input to the board. The jack accepts a standard 2.0 ~ 2.1mm center barrel plug connector (positive voltage center) to provide the +VIN supply of +5 to +16VDC (+9VDC typical).

+Volts, 2mm center

TB1 Power connection

TB1 terminal block provides access to the +VIN and GND (power ground) supplies. The +VIN connection is not switched by the ON-OFF switch or fused between the PWR jack and TB1.

TB1



+VIN - GND

3.3V Indicator

The 3.3V indicator will be ON if voltage is applied to EVB +3.3VDC main supply circuit.

VSTDBY Option

The VSTDBY option installed provides 3.3VDC to the MCF5213 VSTDBY input. To apply another source the option should be opened and alternate supply applied at the option pin 1 position (closest to MCU Port).

VRH Option

The MCF5213 ADC VRH option installed provides 3.3VDDA to the MCF5213 VRH input. To apply another source the option should be opened and alternate supply applied at the option pin 2 position (farthest from MCU Port).

RESET

External reset is provided by the RESET switch, LV1 low voltage detector, or user applied connection to the RSTI* signal on the MCU PORT pin 16. If the main 3.3V supply is below operating level, the LV1 voltage detector will cause the MCF5213 to stay in the RESET condition.

Application of RESET will cause the dBUG monitor or user application to initialize the MCF5213. Previous operating state of the MCF5213 will be lost.

RESET Switch

RESET switch provides for manual application of the MCF5213 RSTI* signal.

RESET Indicator

RESET indicator will be ON for the duration of a valid RSTO* signal. This operation indicates the MCF5213 is in the Reset state.

ABORT Switch

The ABORT switch provides for manual application of the IRQ7 interrupt signal. This operation will allow the dBUG monitor to stop execution of a user program and maintain the CPU operating state for user examination.

SW1 and SW2 Switches

User switches 1 and 2 are available for application as needed. Both switches provide an active low signal when pressed. SW1 applies the MCF5213 IRQ4 signal and SW2 applies the MCF5213 IRQ5 signal.

LED[4:1] User Indicators

Four user indicators are provided for application on the MCF5213 DTIN[3:0] signals (configurable as GPIO). Indicators are buffered so they do not load the MCF5213 I/O port.

Indicator Table

INDICATOR	COLOR	OPERATION	DEFAULT CONDITION
LED1	Green	MCF5213 DTIN0 status, high = ON	ON
LED2	Green	MCF5213 DTIN1 status, high = ON	ON
LED3	Green	MCF5213 DTIN2 status, high = ON	ON
LED4	Green	MCF5213 DTIN3 status, high = ON	ON

LED_EN option

The user LED indicators LED[4:1] must be enabled by the LED_EN option installed.

SYSTEM CLOCK

The MCF5213BADGE default clock source is the 8MHz reference crystal oscillator Y1 with PLL operation enabled. The DBUG monitor configures for a 80Mhz default system clock.

Caution should be applied so that communication with the dBUG monitor is not lost due to clock frequency or serial baud rate change (UART0) by user applications.

CLK0 and CLK1 Test Pads

These test pads provide access to the clock mode pins for the MCF5213. MCF5213 provides several clocking features and the user should refer to the MCF5213RM manual for details. Clock hardware options must be set while the EVB is powered off. Note that for internal and external clocking options the Y1 crystal will need removed and the XTAL pin applied potential.

Clock Type		Clock Mode	
Source	**PLL**	**CLK0**	**CLK1**
Y1 (8MHz)	ON	Out / Idle	Out / Idle
Y1 (8MHz)	OFF	Out / Idle	In
INT OSC	ON	In	Out / Idle
INT OSC	OFF	In	In
EXT CLK	ON	In	Out / Idle
EXT CLK	OFF	In	In

UART0_TERMINAL and UART1 Ports

The UART0_TERMINAL port provides the primary interface to the dBUG monitor with a default baud rate of 19.2K baud, 8 data bits, 1 stop bit, and no parity. Both the UART0 and UART1 ports apply a standard 9-pin serial connector with RS232 type interface to the MCF5213 UART0 or UART1 serial ports. Both ports apply a UARTx_EN option jumper block to enable the MCF5213 UART signals to operate the RS232 ports. A straight through DB9 Male / Female type serial cable can be applied to connect the ports to a standard PC COM port. Following is the DB9S connection reference.

UART0_TERMINAL and UART1 Ports

1	**1**		X	DB9 socket connector with RS232 signal levels.
TXD	**2**	**6**	6	
RXD	**3**	**7**	7 CTS	**1,4,6 connected for status null to host**
4	**4**	**8**	8 RTS	
GND	**5**	**9**	9	

UART0_EN and UART1_EN Options

OPTION Name	UART0_EN MCF5213 signal applied	UART1_EN MCF5213 signal applied
TX	UTXD0 output	UTXD1 output
RX	URXD0 input	URXD1 input
RTS	URTS0* output	URTS1* output
CTS	UCTS0* input	UCTS1* input

CAN Port

The CAN port provides the CAN network port. The CAN_EN option block enables the MCF5213 CAN signals for CAN network operation. The CAN port also has a network termination option, CAN_TERM and biasing components RCAN1 and RCAN2 (not populated).

CAN mode connection will require the cabling to be compatible to the network applied. Following is the DB9S connection reference.

UART2_CAN Port

CAN	DB9		CAN	DB9 socket connector.
	1			
CAN_LO	2	6		
GND	3	7	CAN_HI	
	4	8		
GND	5	9		

CAN Operation

The MCF5213 FlexCAN signals CANRX and CANTX are the secondary operation of the MCF5213 I2C serial port signals SDA and SCL. User must enable the CAN signal operation on the MCF5213 I/O port during the FlexCAN initialization. The FlexCAN transmit and receive signals are connected to the CAN transceiver with the CAN_EN option block. To apply I2C function on these signals the CAN_EN options must be open or idle.

The CAN port provides the physical interface layer for the MCF5213 FlexCAN Controller Area Network version 2.0B peripheral. The FlexCAN transmit and receive signals are connected to a 3.3V CAN transceiver capable of 1M baud communication (SN65HVD230) with the CAN_EN option block. Transceiver differential CAN network signals (CAN_HI and CAN_LO) are provided to the COM_SEL option block for connection by the UART2_CAN port connector.

The CAN transceiver has CAN signal drive control via the **RS** test pad on the development board. The RS signal is provided a 1K Ohm pull-down resistor for the maximum signal rate setting. User may refer to the SN65HVD230 data sheet and apply additional transmit signal control at the RS test pad.

Bias options RCAN1 and RCAN2 (SMT 0805 size, not populated) provide idle bias connections for the CAN network if required by the user.

CAN_TERM

Installs a 62 ohm termination between the CAN_HI and CAN LO signals. Each end of a CAN network must be terminated for proper operation.

CAN_EN option

Enable MCF5213 CAN signals to the CAN transceiver with this option.

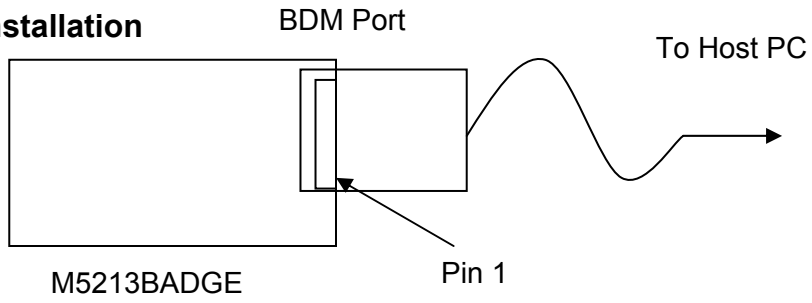
Position	MCF5213 Signal
TX	CANTX
RX	CANRX

M5213BADGE I/O Ports

BDM_PORT

The BDM PORT provides a standard ColdFire BDM / JTAG development port. The **BDM_EN** option provides for the development port mode selection between BDM or JTAG. Option **JP2** provides for a special JTAG mode port configuration that is applied to defeat the MCF5213 flash security (if enabled) for bulk erasing.

Development Cable Installation



BDM_EN Option

The BDM_EN option will select the development port mode at Reset.

Position	Development Port Mode
IN	BDM Mode (Default)
OUT	JTAG Mode

JP2 Option

JP2 provides BDM port signal configuration option for BDM or Special JTAG mode. Special JTAG mode is applied for tools to defeat the MCF5213 flash security.

Position	BDM Port Configuration
1-2	BDM Mode
2-3	Special JTAG

BDM Port Connector

	1	**2**	BKPT*
GND	**3**	**4**	DSCLK
GND	**5**	**6**	TCLK (From CT1 option)
RSTI*	**7**	**8**	DSI
+3.3V	**9**	**10**	DSO
GND	**11**	**12**	PST3
PST2	**13**	**14**	PST1
PST0	**15**	**16**	DDATA3
DDATA2	**17**	**18**	DDATA1
DDATA0	**19**	**20**	GND
	21	**22**	
GND	**23**	**24**	TCLK and CLKOUT Test Pad
+3.3V	**25**	**26**	TA* Test Pad

MCU_PORT

The MCU PORT provides user access to the MCF5213 I/O ports.

GND	1	2	+3.3V
UTXD2	3	4	
URXD2	5	6	IRQ1*
CANRX	7	8	AN0
CANTX	9	10	AN1
GND	11	12	AN2
QSPI_SCLK	13	14	AN3
QSPI_DIN	15	16	RSTI*
QSPI_DOUT	17	18	TCLK
QSPI_CS0	19	20	QSPI_CS1
GND	21	22	+3.3V
GPT0	23	24	QSPI_CS2
GPT1	25	26	QSPI_CS3
GPT2	27	28	
GPT3	29	30	GND
AN4	31	32	
AN5	33	34	
AN6	35	36	
AN7	37	38	
GND	39	40	+3.3V
PWM1	41	42	
PWM3	43	44	UTXD1
PWM5	45	46	URXD1
PWM7	47	48	UCTS1
GND	49	50	URTS1
DTIN0	51	52	UCTS2
DTIN1	53	54	URTS2
DTIN2	55	56	
DTIN3	57	58	
UTXD0	59	60	
GND	61	62	+3.3V
URXD0	63	64	IRQ2*
UCTS0	65	66	IRQ3*
URTS0	67	68	IRQ4*
	69	70	IRQ5*
	71	72	IRQ6*
	73	74	IRQ7*
RSTO*	75	76	
RCON	77	78	
GND	79	80	+3.3V

TROUBLESHOOTING

The M5213BADGE is fully tested and operational before shipping. If it fails to function properly, inspect the board for obvious physical damage first. Verify the communications setup as described under GETTING STARTED.

The most common problems are improperly configured options or communications parameters.

1. Make sure that the RSTI* line is not being held low or the RESET indicator is not on constantly.

2. Ensure that the BKPT* line is not being asserted by an active BDM interface cable applied to the BDM connector.

3. Verify that your COM communications port is working by substituting a known good serial device or by doing a loop back diagnostic. if you applied a different baud rate with the dBUG SET command, make sure the terminal software is set correctly.

4. Verify the power source, +3.3V, indicator is ON? You should measure a minimum of 9 volts between the GND and +V test pad and GND test pad near the power jack with the standard power supply provided.

5. If no power indications or voltage is found, verify the wall plug connections to AC outlet and the PWR jack power connector.

6. Disconnect all external connections to the board except for COM1 to the PC and the wall plug and check operation again.

7. Contact support@axman.com by email for further assistance. Provide board name and describe problem.

dBUG MONITOR OPERATION

dBUG is a firmware resident development environment operated by the ColdFire processor as a primary control program. The monitor provides serial communication for loading and controlling the execution of software under test. User should note that the monitor occupies the first or lower address 80K bytes of the internal Flash memory.

dBUG Communication:

Primary user interface to the dBUG monitor is by command lines that are entered into the serial port. These commands are defined in the following table “dBUG Commands”. For serial communications, dBUG requires eight data bits, no parity, and one stop bit, 8N1 with XON/XOFF soft flow control. The default baud rate is 19200; however, this rate can be changed by the user with a “set” command. The command line prompt is “dBUG> “. Any dBUG command may be entered from this prompt. dBUG does not allow command lines to exceed 80 characters. Wherever possible, dBUG displays data in 80 columns or less. dBUG echoes each character as it is typed, eliminating the need for any “local echo” on the terminal side. In general, dBUG is not case sensitive. Commands may be entered either in upper or lower case, depending upon the user’s equipment and preference. Only symbol names require that the exact case be used.

dBUG System Initialization

The act of powering up the board will initialize the system. The processor is reset and dBUG is invoked. dBUG performs the following configurations of internal resources during the initialization:

Internal memories are mapped as described in the dBUG memory map section below.

The MCF5213 PLL is initialized to multiply the default 8MHz input reference up to 80 MHz.

All volatile code sections and the interrupt vector table are copied from ROM to RAM.

Control is given to the user via the command prompt.

Interrupt Service Support

The Vector Base Register, VBR, is programmed to point to the RAM copy of the vector table at the base of the internal SRAM. To take over an exception vector, the user places the address of the exception handler in the appropriate vector in the vector table.

dBUG Memory Map

The following table shows the dBUG memory map. This information is also provided via the ‘mmap’ dBUG command.

0x00000000 0x00013FFF	Internal 256KByte Flash Memory dBUG protected code 80KBytes
0x00014000 0x0003FFFF	Internal 256KByte Flash Memory User space 176KBytes
0x20000000 0x20001FFF	Internal 32KByte SRAM Memory Vector Table and dBUG protected data 8KBytes
0x20002000 0x20007FFF	Internal 32KByte SRAM Memory User space 24KBytes

Note: Applying BDM / JTAG development port tools does not require following the dBUG memory map.

dBUG Commands

After the system initialization, the dBUG waits for a command-line input from the user terminal. When a proper command is entered, the operation continues in one of the two basic modes. If the command causes execution of the user program, the dBUG firmware may or may not be re-entered, at the discretion of the user's program. For the alternate case, the command will be executed under control of the dBUG firmware, and after command completion, the system returns to command entry mode.

dBUG Command Table

MNEMONIC	SYNTAX	DESCRIPTION
ASM	asm <<addr> <assembly>>	Assemble
BC	bc addr1 addr2 length	Block Compare
BF	bf <width> begin end data <inc>	Block Fill
BM	bm begin end dest	Block Move
BR	br addr <-r> <-c count> <-t trigger>	Breakpoint
BS	bs <width> begin end data	Block Search
DC	dc value	Data Convert
DI	di <addr>	Disassemble
DL	dl <offset>	Download Serial
DLDBUG	dldbug	Download dBUG Update
FL	fl <command> dest <src> size	Flash write or erase
GO	go <addr>	Execute
GT	gt addr	Execute To
HELP	help <command>	Help
IRD	ird <module.register> Internal	Internal Register Display
IRM	irm module.register data	Internal Register Modify
LR	lr <width> addr	Loop Read
LW	lw <width> addr data	Loop Write
MD	md <width> <begin> <end>	Memory Display
MM	mm <width> addr <data>	Memory Modify
MMAP	mmap	Memory Map Display
RD	rd <reg>	Register Display (core)
RM	rm reg data	Register Modify (core)
RESET	reset	Reset
SD	sd	Stack Display (contents)
SET	set <option value>	Set Configurations
SHOW	show <option>	Show Configurations
STEP	step	Step (Over)
SYM	symbol <symb> <-a symb value> <-r symb> <-C| l |s>	Symbol Management
TRACE	trace <num>	Trace (Into)
VER	version	Show dBUG Version

During command execution, additional user input may be required depending on the command function. For commands that accept an optional <width> to modify the memory access size, the valid values are:

• B = 8-bit (byte) access • W = 16-bit (word) access • L = 32-bit (long) access

When no <width> option is provided, the default width is "W", 16-bit.

The core ColdFire register set is maintained by dBUG. These are listed below:

- A0 - A7
- D0 - D7
- PC
- SR

All control registers on ColdFire are not readable by the supervisor-programming model, and thus not accessible via dBUG. User code may change these registers, but caution must be exercised as changes may render dBUG inoperable. A reference to “SP” (stack pointer) actually refers to general purpose address register seven, “A7.”

The commands DI, GO, MD, STEP and TRACE are used repeatedly when debugging. dBUG recognizes this and allows for repeated execution of these commands with minimal typing. After a command is entered, simply press <RETURN> or <ENTER> to invoke the command again. The command is executed as if no command line parameters were provided.

User programs are provided access to various dBUG routines by the “Trap 15 Functions". These functions are discussed at the end of this chapter.

Appendix 1: dBUG Command Set

Note that arguments used in the command examples (such as addresses) are chosen arbitrarily and will not be valid on all ColdFire EVBs.

ASM - Assembler

Usage: ASM <<addr> <assembly>>

The ASM command is a primitive assembler. The <**assembly**> is assembled and the resulting code placed at <**add**r>. This command has an interactive and non-interactive mode of operation.

The value for address <addr> may be an absolute address specified as a hexadecimal value, or a symbol name. The value for stmt must be valid assembler mnemonics for the CPU.

For the interactive mode, the user enters the command and the optional <addr>. If the address is not specified, then the last address is used. The memory contents at the address are disassembled, and the user prompted for the new assembly. If valid, the new assembly is placed into memory, and the address incremented accordingly. If the assembly is not valid, then memory is not modified, and an error message produced. In either case, memory is disassembled and the process repeats.

The user may press the <Enter> or <Return> key to accept the current memory contents and skip to the next instruction, or enter a period (“.”) to quit the interactive mode.

In the non-interactive mode, the user specifies the address and the assembly statement on the command line. The statement is the assembled, and if valid, placed into memory, otherwise an error message is produced.

Examples:

To place a NOP instruction at address 0x2000_2000, the command is:

asm 20002000 nop

To interactively assembly memory at address 0x2000_20000, the command is:

asm 20002000

BC - Block Compare

Usage: BC addr1 addr2 length

The BC command compares two contiguous blocks of memory on a byte by byte basis. The first block starts at address **addr1** and the second starts at address **addr2**, both of **length** bytes.

If the blocks are not identical, the address of the first mismatch is displayed. The value for addresses addr1 and addr2 may be an absolute address specified as a hexadecimal value or a symbol name. The value for length may be a symbol name or a number converted according to the user defined radix (hexadecimal by default).

Example:

To verify that the data starting at 0x20000 and ending at 0x3_0000 is identical to the data starting at 0x8_0000, the command is:

bc 20000 80000 10000

BF - Block Fill

Usage: BF<width> begin end data <inc>

The BF command fills a contiguous block of memory starting at address **begin**, stopping at address **end**, with the value **data**. <**Width**> modifies the size of the data that is written. If no <width> is specified, the default of word sized data is used.

The value for addresses begin and end may be an absolute address specified as a hexadecimal value, or a symbol name. The value for data may be a symbol name, or a number converted according to the user-defined radix, normally hexadecimal.

The optional value <**inc**> can be used to increment (or decrement) the data value during the fill.

This command first aligns the starting address for the data access size, and then increments the address accordingly during the operation. Thus, for the duration of the operation, this command performs properly-aligned memory accesses.

Examples:

To fill a memory block starting at 0x2_0000 and ending at 0x4_0000 with the value 0x1234, the command is:

bf 20000 40000 1234

To fill a block of memory starting at 0x20000 and ending at 0x4_0000 with a byte value of 0xAB, the command is:

bf.b 20000 40000 AB

To zero out the BSS section of the target code (defined by the symbols bss_start and bss_end), the command is:

bf bss_start bss_end 0

To fill a block of memory starting at 0x2_0000 and ending at 0x4_0000 with data that increments by 2 for each <width>, the command is:

bf 20000 40000 0 2

BM - Block Move

Usage: BM begin end dest

The BM command moves a block of memory starting at address **begin** and stopping at address **end** to the new address **dest**. The BM command copies memory as a series of bytes, and does not alter the original block.

The values for addresses begin, end, and dest may be absolute addresses specified as hexadecimal values, or symbol names. If the destination address overlaps the block defined by begin and end, an error message is produced and the command exits.

Examples:

To copy a block of memory starting at 0x4_0000 and ending at 0x7_0000 to the location 0x200000, the command is:

bm 40000 70000 200000

To copy the target code's data section (defined by the symbols data_start and data_end) to 0x200000, the command is:

bm data_start data_end 200000

NOTE: Refer to "upuser" command for copying code/data into Flash memory.

BR - Breakpoints

Usage: BR addr <-r> <-c count> <-t trigger>

The BR command inserts or removes breakpoints at address **addr**. The value for addr may be an absolute address specified as a hexadecimal value, or a symbol name. Count and trigger are numbers converted according to the user-defined radix, normally hexadecimal.

If no argument is provided to the BR command, a listing of all defined breakpoints is displayed.

The **-r** option to the BR command removes a breakpoint defined at address addr. If no address is specified in conjunction with the -r option, then all breakpoints are removed.

Each time a breakpoint is encountered during the execution of target code, its count value is incremented by one. By default, the initial count value for a breakpoint is zero, but the **-c** option allows setting the initial count for the breakpoint.

Each time a breakpoint is encountered during the execution of target code, the count value is compared against the trigger value. If the count value is equal to or greater than the trigger value, a breakpoint is encountered and control returned to **dBUG**. By default, the initial trigger value for a breakpoint is one, but the **-t** option allows setting the initial trigger for the breakpoint.

If no address is specified in conjunction with the -c or -t options, then all breakpoints are initialized to the values specified by the -c or -t option.

Examples:

To set a breakpoint at the C function main() (symbol _main; see “symbol” command), the command is:

```
br _main
```

When the target code is executed and the processor reaches main(), control will be returned to **dBUG**.

To set a breakpoint at the C function bench() and set its trigger value to 3, the command is:

```
br _bench -t 3
```

When the target code is executed, the processor must attempt to execute the function bench() a third time before returning control back to **dBUG**.

To remove all breakpoints, the command is:

```
br -r
```

BS - Block Search

Usage: BS<width> begin end data

The BS command searches a contiguous block of memory starting at address **begin**, stopping at address **end**, for the value **data**. <**Width**> modifies the size of the data that is compared during the search. If no <width> is specified, the default of word sized data is used.

The values for addresses begin and end may be absolute addresses specified as hexadecimal values, or symbol names. The value for data may be a symbol name or a number converted according to the user-defined radix, normally hexadecimal.

This command first aligns the starting address for the data access size, and then increments the address accordingly during the operation. Thus, for the duration of the operation, this command performs properly-aligned memory accesses.

Examples:

To search for the 32-bit value 0x1234_5678 in the memory block starting at 0x4_0000 and ending at 0x7_0000:

bs.l 40000 70000 12345678

This reads the 32-bit word located at 0x0004_0000 and compares it against the 32-bit value 0x1234_5678. If no match is found, then the address is incremented to 0x0004_0004 and the next 32-bit value is read and compared.

To search for the 16-bit value 0x1234 in the memory block starting at 0x0004_0000 and ending at 0x0007_0000:

bs 40000 70000 1234

This reads the 16-bit word located at 0x4_0000 and compares it against the 16-bit value 0x0000_1234. If no match is found, then the address is incremented to 0x0004_0002 and the next 16-bit value is read and compared.

DC - Data Conversion

Usage: DC data

The DC command displays the hexadecimal or decimal value **data** in hexadecimal, binary, and decimal notation.

The value for data may be a symbol name or an absolute value. If an absolute value passed into the DC command is prefixed by ‘0x’, then data is interpreted as a hexadecimal value. Otherwise data is interpreted as a decimal value.

All values are treated as 32-bit quantities.

Examples:

To display the decimal and binary equivalent of 0x1234, the command is:

dc 0x1234

To display the hexadecimal and binary equivalent of 1234, the command is:

dc 1234

DI - Disassemble

Usage: DI <addr>

The DI command disassembles target code pointed to by **addr**. The value for addr may be an absolute address specified as a hexadecimal value, or a symbol name.

Wherever possible, the disassembler will use information from the symbol table to produce a more meaningful disassembly. This is especially useful for branch target addresses and subroutine calls.

The DI command attempts to track the address of the last disassembled opcode. If no address is provided to the DI command, then the DI command uses the address of the last opcode that was disassembled.

The DI command is repeatable.

Examples:

To disassemble code that starts at 0x0004_0000, the command is:

```
di      40000
```

To disassemble code of the C function main(), the command is:

```
di      _main
```

DL - Download Console

Usage: DL <offset>

The DL command performs an S-record download of data obtained from the console or serial port. The value for offset is converted according to the user-defined radix, normally hexadecimal.

If **offset** is provided, then the destination address of each S-record is adjusted by offset.

The DL command checks the destination download address for validity. If the destination is an address outside the defined user space, then an error message is displayed and downloading aborted.

If the destination address is in the user flash memory space, the flash will be programmed but not erased. See the FL command for flash erasing.

If the S-record file contains the entry point address, then the program counter is set to reflect this address.

Examples:

To download an S-record file through the serial port, the command is:

```
dl
```

To download an S-record file through the serial port, and add an offset to the destination address of 0x40000, the command is:

```
dl      0x40000
```

After the DL command is invoked, the user should select file transfer or upload and send the S-record file form the host. The host serial terminal software should apply XON/XOFF flow control to the transfer if the target memory is Flash space to allow flash programming time delays. Alternate method is to download with an offset into SDRAM memory space and then apply the FL command to program the flash memory space.

DLDBUG – Download dBUG (update)

Usage: dldbug

The dldbug command is used to update the dBUG image in Flash memory. When updates to the M5213BADGE dBUG are available, the update S-record may be downloaded into the Flash from the console or serial port similar to the DL command. The user is prompted for verification before performing the operation (note case sensitivity here). XON/XOFF serial flow control must be applied when loading the new S-record. Use this command with extreme caution, as any error can render dBUG useless!

FL – Flash Load or Erase

Usage: FL <command> dest <src> <size>

The FL command is used to erase and write the MCF5213 internal flash, and display flash device information. Erase or write operations must be performed in even page sizes. The write command will erase all of the associated flash pages prior to writing. If the write destination address or byte count range does not provide an even page boundary, dBUG will prompt the user to continue.

The MCF5213 internal flash destination address must be long word (4 byte) aligned and the byte count must be in longword (4 byte) multiples.

To download S-record files directly into the flash, please see the DL command.

Examples:

To view the flash device sector information, the command is:

```
fl
```

To erase 0x10000 (64K) bytes of internal flash starting at 0xF0000000, the command is:

```
fl e F0000000 10000
```

To copy 0x4000 (16K) bytes of data from SRAM (0x20000000) to flash at 0x00020000, the command is:

```
fl w 00020000 20000000 4000
```

GO – Execute user code

Usage: GO <addr>

The GO command executes target code starting at address **addr**. The value for addr may be an absolute address specified as a hexadecimal value, or a symbol name.

If no argument is provided, the GO command begins executing instructions at the current program counter.

When the GO command is executed, all user-defined breakpoints are inserted into the target code, and the context is switched to the target program. Control is only regained when the target code encounters a breakpoint, illegal instruction, or other exception that causes control to be handed back to **dBUG**.

The GO command is repeatable.

Examples:

To execute code at the current program counter, the command is:

```
go
```

To execute code at the C function main(), the command is:

```
go _main
```

To execute code at the address 0x00040000, the command is:

```
go 40000
```

GT - Execute To Address

Usage: GT addr

The GT command inserts a temporary software breakpoint at **add**r and then executes target code starting at the current program counter. The value for addr may be an absolute address specified as a hexadecimal value, or a symbol name.

When the GT command is executed, all breakpoints are inserted into the target code, and the context is switched to the target program. Control is only regained when the target code encounters a breakpoint, illegal instruction, or an exception which causes control to be handed back to **dBUG**.

Examples:

To execute code up to the C function bench(), the command is:

```
gt _bench
```

To execute code up to the address 0x00080004, the command is:

gt 80004

IRD - Internal Register Display

Usage: IRD <module.register>

This command displays the internal registers of the different modules inside the MCF5213. In the command line, module refers to the module name where the register is located and register refers to the specific register to display.

The registers are organized according to the module to which they belong. Refer to the MCF5213 user's manual for information on these modules and the registers they contain.

Example:

ird uart0.umr1

ird uart0

IRM - Internal Register Modify

Usage: IRM module.register data

This command modifies the contents of the internal registers of different modules inside the MCF5213. In the command line, module refers to the module name where the register is located and register refers to the specific register to modify. The data parameter specifies the new value to be written into the register.

The registers are organized according to the module to which they belong. Refer to the MCF5213 user's manual for information on these modules and the registers they contain.

HELP - Help

Usage: HELP <command>

The HELP command displays a brief syntax of the commands available within **dBUG**. In addition, the address of where user code may start is given. If command is provided, then a brief listing of the syntax of the specified command is displayed.

Examples:

To obtain a listing of all the commands available within **dBUG**, the command is:

help

To obtain help on the breakpoint command, the command is:

help br

LR - Loop Read

Usage: LR <width> addr

The LR command continually reads the data at **addr** until a key is pressed. The optional **<width>** specifies the size of the data to be read. If no <width> is specified, the command defaults to reading word sized data.

Example:

To continually read the word data from address 0xFFF2_0000, the command is:

lr FFF20000

LW - Loop Write

Usage: LW <width> addr data

The LW command continually writes data to **addr**. The optional width specifies the size of the access to memory. The default access size is a word.

Examples:

To continually write the data 0x1234_5678 to address 0x0002_0000, the command is:

lw.l 20000 12345678

Note that the following command writes 0x78 into memory:

lw.b 20000 78

MD - Memory Display

Usage: MD <width> <begin> <end>

The MD command displays a contiguous block of memory starting at address begin and stopping at address end. The values for addresses begin and end may be absolute addresses specified as hexadecimal values, or symbol names. Width modifies the size of the data that is displayed. If no **<width>** is specified, the default of word sized data is used.

Memory display starts at the address begin. If no beginning address is provided, the MD command uses the last address that was displayed. If no ending address is provided, then MD will display memory up to an address that is 128 beyond the starting address.

This command first aligns the starting address for the data access size, and then increments the address accordingly during the operation. Thus, for the duration of the operation, this command performs properly-aligned memory accesses.

Examples:

To display memory at address 0x0040_0000, the command is:

md 400000

To display memory in the data section (defined by the symbols data_start and data_end), the command is:

md data_start

To display a range of bytes from 0x00040000 to 0x0005_0000, the command is:

md.b 40000 50000

To display a range of 32-bit values starting at 0x0004_0000 and ending at 0x0005_0000:

md 40000 50000

MM - Memory Modify

Usage: MM<width> addr <data>

The MM command modifies memory at the address **addr**. The value for addr may be an absolute address specified as a hexadecimal value, or a symbol name. Width specifies the size of the data that is modified. If no **<width>** is specified, the default of word sized data is used. The value for **data** may be a symbol name, or a number converted according to the user-defined radix, normally hexadecimal.

If a value for data is provided, then the MM command immediately sets the contents of addr to data.

If no value for data is provided, then the MM command enters into a loop. The loop obtains a value for data, sets the contents of the current address to data, increments the address according to the data size, and repeats. The loop terminates when an invalid entry for the data value is entered, for instance a period '.'.

This command first aligns the starting address for the data access size, and then increments the address accordingly during the operation. Thus, for the duration of the operation, this command performs properly-aligned memory accesses.

Examples:

To set the byte at location 0x0001_0000 to be 0xFF, the command is:

mm.b 10000 FF

To interactively modify memory beginning at 0x0001_0000, the command is:

mm 10000

MMAP - Memory Map Display

Usage: mmap

This command displays the memory map information for the evaluation board. The information displayed includes the type of memory, the start and end address of the memory, and the port size of the memory. The display also includes information on how the Chip-selects are used on the board.

Here is an example of the output from this command:

```
     Type            Start          End
  ---------------------------------------
  SRAM              0x20000000   0x20007FFF
  IPSBAR            0x40000000   0x7FFFFFFF
  Flash             0x00000000   0x0003FFFF

   Protected          Start          End
  ---------------------------------------
  dBUG Code         0x00000000   0x00011FFF
  dBUG Data         0x20000000   0x20001FFF
```

RD - Register Display

Usage: RD <reg>

The RD command displays the register set of the target. If no argument for **reg** is provided, then all registers are displayed. Otherwise, the value for reg is displayed.

dBUG preserves the registers by storing a copy of the register set in a buffer. The RD command displays register values from the register buffer.

Examples:

To display only the program counter:

rd pc

To display all the registers and their values, the command is:

rd

Here is an example of the output from this command:

PC: 00000000 SR: 2000 [t.Sm.000...xnzvc]

An: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 01000000

Dn: 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

RM - Register Modify

Usage: RM reg data

The RM command modifies the contents of the register **reg** to data. The value for reg is the name of the register, and the value for data may be a symbol name, or it is converted according to the user-defined radix, normally hexadecimal.

dBUG preserves the registers by storing a copy of the register set in a buffer. The RM command updates the copy of the register in the buffer. The actual value will not be written to the register until target code is executed.

Example:

To change program counter to contain the value 0x2000_8000, the command is:

rm pc 20008000

RESET - Reset the Board and dBUG

Usage: RESET

The RESET command resets the board and **dBUG** to their initial power-on states.

The RESET command executes the same sequence of code that occurs at power-on. If the RESET command fails to reset the board properly, cycle the power or press the RESET button.

Examples:

To reset the board and clear the **dBUG** data structures, the command is:

reset

SET - Set Configurations

Usage: SET <option value>

The SET command allows the setting of user-configurable options within **dBUG**. With no arguments, SET displays the options and values available. The SHOW command displays the settings in the appropriate format. Note that some configuration items will not take effect until a Reset has occurred. The standard set of options is listed below.

baud - This is the baud rate for the first serial port on the board. All communications between **dBUG** and the user occur using 19200 bps, eight data bits, no parity, and one stop bit, 8N1, with no flow control.

base - This is the default radix for use in converting a number from its ASCII text representation to the internal quantity used by **dBUG**. The default is hexadecimal (base 16), and other choices are binary (base 2), octal (base 8), and decimal (base 10).

Examples:

To set the baud rate of the board to be 38400, the command is:

set baud 38400

NOTE: See the SHOW command for a display containing the correct formatting of these options.

SHOW - Show Configurations

Usage: SHOW <option>

The SHOW command displays the settings of the user-configurable options within **dBUG**. When no option is provided, SHOW displays all options and values.

Examples:

To display the current baud rate of the board, the command is:

show baud

To display all options and settings, the command is:

show

Here is an example of the output from a show command:

```
dBUG> show
        base: 16
        baud: 19200
```

STEP - Step Over

Usage: STEP

The STEP command can be used to “step over” a subroutine call, rather than tracing every instruction in the subroutine. The ST command sets a temporary software breakpoint one instruction beyond the current program counter and then executes the target code.

The STEP command can be used to “step over” BSR and JSR instructions.

The STEP command will work for other instructions as well, but note that if the STEP command is used with an instruction that will not return, i.e. BRA, then the temporary breakpoint may never be encountered and **dBUG** may never regain control.

Examples:

To pass over a subroutine call, the command is:

step

SYMBOL - Symbol Name Management

Usage: SYMBOL <symb> <-a symb value> <-r symb> <-c|l|s>

The SYMBOL command adds or removes **symbol** names from the symbol table. If only a symbol name is provided to the SYMBOL command, then the symbol table is searched for a match on the symbol name and its information displayed.

-a option adds a symbol name and its value into the symbol table.

-r option removes a symbol name from the table.

-c option clears the entire symbol table.

-l option lists the contents of the symbol table.

-s option displays usage information for the symbol table.

Symbol names contained in the symbol table are truncated to 31 characters. Any symbol table lookups, either by the SYMBOL command or by the assembler/disassembler, will only use the first 31 characters. Symbol names are case-sensitive.

Symbols can also be added to the symbol table via in-line assembly labels and Ethernet downloads of ELF formatted files.

Examples:

To define the symbol “main” to have the value 0x0004_0000, the command is:

symbol -a main 40000

To remove the symbol “junk” from the table, the command is:

symbol -r junk

To see how full the symbol table is, the command is:

symbol -s

To display the symbol table, the command is:

symbol –l

TRACE - Trace Into

Usage: TRACE <num>

The TRACE command allows single-instruction execution. If **num** is provided, then num instructions are executed before control is handed back to **dBUG**. The value for num is a decimal number.

The TRACE command sets bits in the processors' supervisor registers to achieve single-instruction execution, and the target code executed. Control returns to **dBUG** after a single-instruction execution of the target code.

This command is repeatable.

Examples:

To trace one instruction at the program counter, the command is:

tr

To trace 20 instructions from the program counter, the command is:

tr 20

VERSION - Display dBUG Version

Usage: VERSION

The VERSION command displays the version information for **dBUG**. The **dBUG** version, build number and build date are all given.

The version number is separated by a decimal, for example, "v 2b.1c.1a".

The version date is the day and time at which the entire **dBUG** monitor was compiled and built.

Examples:

To display the version of the **dBUG** monitor, the command is:

ver

TRAP #15 Functions

An additional utility within the dBUG firmware is a function called the TRAP 15 handler. This function can be called by the user program to utilize various routines within the dBUG, perform a special task, and to return control to the dBUG. This section describes the TRAP 15 handler and how it is used.

There are four TRAP #15 functions. These are: **OUT_CHAR**, **IN_CHAR**, **CHAR_PRESENT**, and **EXIT_TO_dBUG**.

OUT_CHAR

This function (function code 0x0013) sends a character, which is in lower 8 bits of D1, to terminal. Assembly example:

```
/* assume d1 contains the character */
      move.l            #$0013,d0  Selects the function
      TRAP        #15         The character in d1 is sent to terminal
```

C example:

```
void board_out_char (int ch)
{
      /* If your C compiler produces a LINK/UNLK pair for this routine,
       * then use the following code which takes this into account
       */
#if   1
      /* LINK a6,#0     -- produced by C compiler */
      asm (" move.l 8(a6),d1");           /* put 'ch'into d1 */
      asm (" move.l #0x0013,d0");   /* select the function */
      asm (" trap #15");                  /* make the call */
      /* UNLK a6              -- produced by C compiler */
#else
      asm (" move.l 4(sp),d1");           /* put 'ch'into d1 */
      asm (" move.l #0x0013,d0");   /* select the function */
      asm (" trap #15");                  /* make the call */
#endif
}
```

IN_CHAR

This function (function code 0x0010) returns an input character (from terminal) to the caller. The returned character is in D1.

Assembly example:

```
      move.l            #$0010,d0   Select the function
      trap        #15         Make the call, the input character is in d1.
```

C example:

```
int board_in_char (void)
{
      asm (" move.l #0x0010,d0");   /* select the function */
      asm (" trap #15");                  /* make the call */
      asm (" move.l d1,d0");        /* put the character in d0 */
}
```

CHAR_PRESENT

This function (function code 0x0014) checks if an input character is present to receive. A value of zero is returned in D0 when no character is present. A non-zero value in D0 means a character is present.

Assembly example:

```
      move.l      #$0014,d0         Select the function
      trap        #15               Make the call, d0 contains the result
```

C example:

```
int board_char_present (void)
{
      asm (" move.l #0x0014,d0");   /* select the function */
      asm (" trap #15");                  /* make the call */
}
```

EXIT_TO_dBUG

This function (function code 0x0000) transfers the control back to the dBUG, by terminating the user code. The register contents are preserved.

Assembly example:

```
      move.l      #$0000,d0         Select the function
      trap        #15               Make the call,  exit to dBUG.
```

C example:

```
void board_exit_to_dbug (void)
{
      asm (" move.l #0x0000,d0");   /* select the function */
      asm (" trap #15");                  /* exit and transfer to dBUG */
}
```